




## 6 Booting System

This chapter describes three assembly language files in the `boot/` directory, as shown in Listing 6-1. As mentioned in the previous chapter, these three files are all assembly language programs, but use two different syntax formats. `bootsect.S` and `setup.S` are real-mode 16-bit code programs that use Intel's assembly language syntax and require the 8086 assembly compiler and linker `as86` and `ld86`. However, `head.s` uses an AT&T assembly syntax format and runs in protected mode, which needs to be compiled with GNU's `as` (gas) assembler.

The main reason why Linus Torvalds used two assemblers at the time was that for Intel x86 processors, the GNU assembly compiler in 1991 only supported i386 and later 32-bit CPU code instructions. It is not supported to generate 16-bit code programs that run in real mode. Until 1994, the GNU `as` assembler began to support the `.code16` directive for compiling 16-bit code (See the "Writing 16-Bit Codes" of the "80386 Related Features" section in the GNU Assembler manual.). Starting with kernel 2.4.X, the `bootsect.S` and `setup.S` programs began to be uniformly written using GNU `as`.

List 6-1 linux/boot/ directory

	Filename	Size	Last Modified Time(GMT)	Description
	<a href="#">bootsect.S</a>	7574 bytes	1992-01-14 15:45:22	
	<a href="#">head.s</a>	5938 bytes	1992-01-11 04:50:17	
	<a href="#">setup.S</a>	12144 bytes	1992-01-11 18:10:18	

In addition to knowing some of the 8086 assembly language knowledge, reading these codes requires an understanding of some PC architectures with Intel 80X86 microprocessors and the basic principles of programming in 32-bit protected mode. So you should have a basic understanding of the previous chapters before start reading the source code. When we read the code, we will explain the specific problems encountered in detail.

### 6.1 Main Functions

Let us first describe the main execution process of the Linux operating system booting part. When the PC power is turned on, the 80x86 CPU will automatically enter the real mode. The program code is automatically executed starting at address `0xFFFF0`. This address is usually the address in the ROM-BIOS. The PC's BIOS will perform hardware detection and diagnostic operations of the system and begin to initialize the interrupt vector at physical address 0. Thereafter, it reads the first sector of the bootable device (disk boot sector, 512 bytes) into the absolute address `0x7C00` of the memory and jumps to this location to start the boot process. The boot device is usually a floppy drive or a hard drive. The description here is simple, but it is enough to understand the beginning of the kernel initialization work.

The first part of Linux is written in 8086 assembly language (`boot/bootsect.S`) and stored in the first sector of the boot device. It will be loaded by the BIOS to the absolute address of the memory `0x7C00` (31KB). When executed, it will move itself to the absolute address `0x90000` (576KB) in memory, and read the 2KB byte code

(boot/setup.S) in the boot device to memory 0x90200. The rest of the kernel (system module) is read in and loaded to the beginning of the memory address 0x10000 (64KB). Therefore, the process of sequential execution from the power-on of the machine is shown in Figure 6-1.

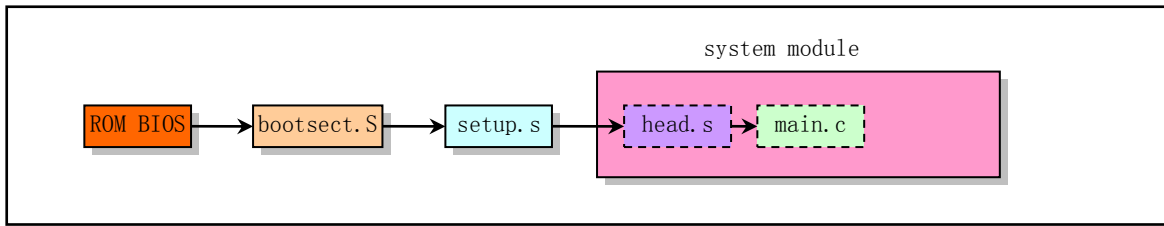


Figure 6-1 The order of execution from system power up

Because the length of the system module at that time will not exceed 0x80000 bytes (ie 512KB), therefore, when the bootsect program reads the system module into the physical address 0x10000 start position, it will not overwrite the bootsect and setup modules starting at 0x90000 (576KB). The subsequent setup program will also move the system module to the physical memory start location, so that the address of the code in the system module is equal to the actual physical address, which is convenient for operating the kernel code and data. Figure 6-2 clearly shows the dynamic location of these programs or modules in memory when the Linux system starts. In the figure, each vertical bar represents the image location map of each program in memory at a certain moment. The message "Loading..." will be displayed during system loading. Control is then passed to the code in boot/setup.S, which is another real-mode assembly language program.

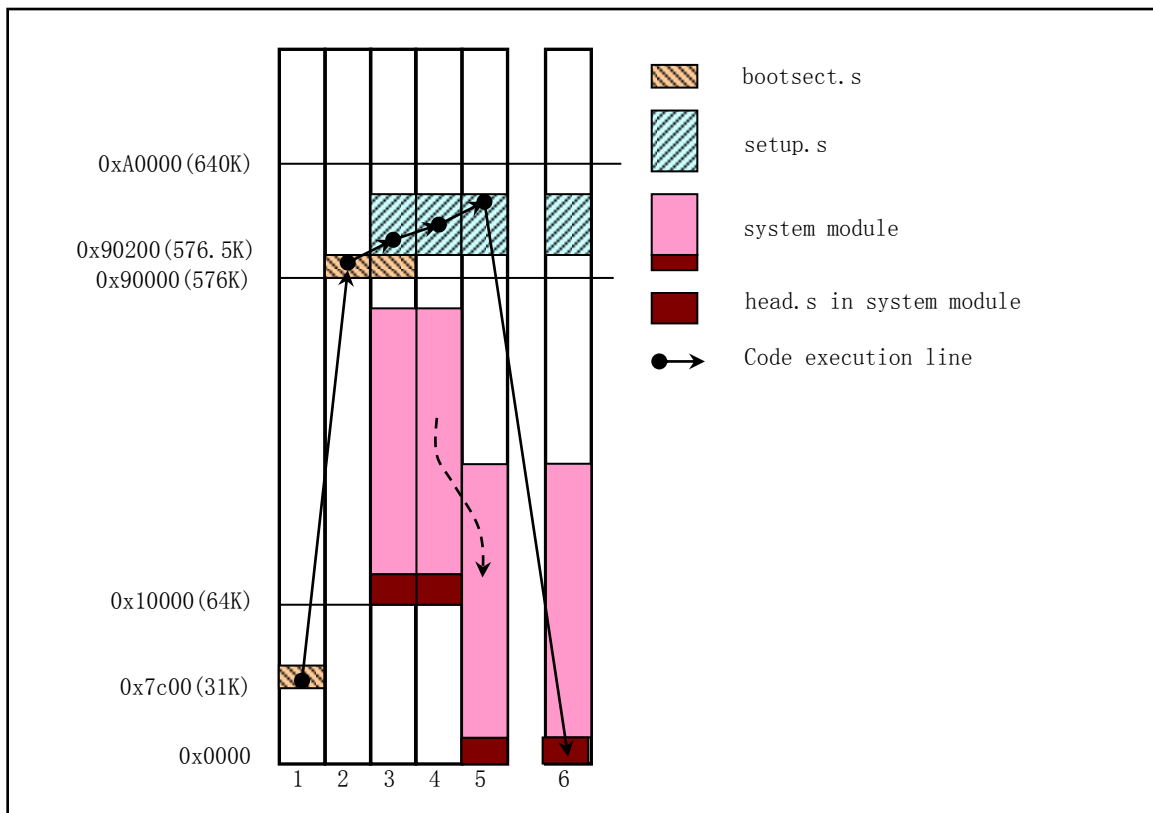


Figure 6-2 Dynamic location of the kernel in memory

The boot part identifies certain features of the host and the type of VGA card, and if required, it asks the user to select a display mode for the console. Then move the entire system from address 0x10000 to 0x0000, enter protected mode and jump to the rest of the system (at 0x0000). At this point, the boot settings for all 32-bit modes have been completed: IDT, GDT, and LDT are loaded, the processor and coprocessor are also confirmed, and paging is also set. Finally, the `main()` code in `init/main.c` is called. The source code for the above operation is in `boot/head.s`, which is probably the most tricky code in the entire kernel. Note that if something goes wrong in any of the previous steps, the computer will deadlock. It can't be handled before the operating system is fully operational.

Someone may ask, why does the bootsect code not load the system module directly to the beginning of the physical address 0x0000, but to move it again in the setup program? This is because the subsequent setup code also needs to use the interrupt call function provided by the ROM BIOS to get some parameters about the machine configuration (such as display card mode, hard disk parameter table, etc.). However, when the BIOS is initialized, an interrupt vector table of size 0x400 bytes (1KB) is placed at the beginning of the physical memory. Directly placing the system module at this location will cause the BIOS interrupt vector table to be overwritten. Therefore, the bootloader needs to move the system module to this area after using the BIOS interrupt call.

In addition, loading the above kernel modules only in memory is not enough for the Linux system to run. As a fully operational Linux system, you also need a basic file system support, the root file system. The Linux 0.12 kernel only supports the MINIX 1.0 file system. The root file system usually exists on another floppy disk or in a hard disk partition. In order to inform the kernel where the root file system is stored, the default block device number `ROOT_DEV` where the root file system is located is given on line 44 of the `bootsect.S` program. See the comments in the program for the meaning of the block device number. The specified device number in the 509, 510 (0x1fc--0x1fd) byte of the boot sector is used when the kernel is initialized. The swap device number `SWAP_DEV` is given on line 45 of the `bootsect.S`, which indicates the external device number used as the virtual storage swap space.

## 6.2 bootsect.S

### 6.2.1 Functional Description

The `bootsect.S` code is a disk boot block program that resides in the first sector of the disk (boot sector, 0 track (cylinder), 0 head, 1st sector). After the PC is powered on and the ROM BIOS self-tests, the ROM BIOS loads the boot sector code `bootsect` to the memory address 0x7C00 and executes it. During the execution of the `bootsect` code, it moves itself to the beginning of the memory absolute address 0x90000 and continues execution. The main function of this program is to first load the four-sector setup module (compiled from `setup.s`) starting from the second sector of the disk into memory immediately after the `bootsect` (0x90200). Then use BIOS interrupt 0x13 to take the parameters of the current boot disk in the disk parameter table, and then display the "Loading system..." string on the screen. Then load the system module behind the setup module on the disk to the beginning of the memory 0x10000. The device number of the root file system is then determined. If not specified, the type of disk (is 1.44M A disk?) is discriminated according to the saved number of sectors per track of the boot disk, and the device number is stored in `root_dev` (at position 508 of the boot block). Finally, long jumps to the beginning of the setup program (0x90200) to execute `setup`. On the disk, the location and size of the boot block, setup module, and system module are shown in Figure 6-3.

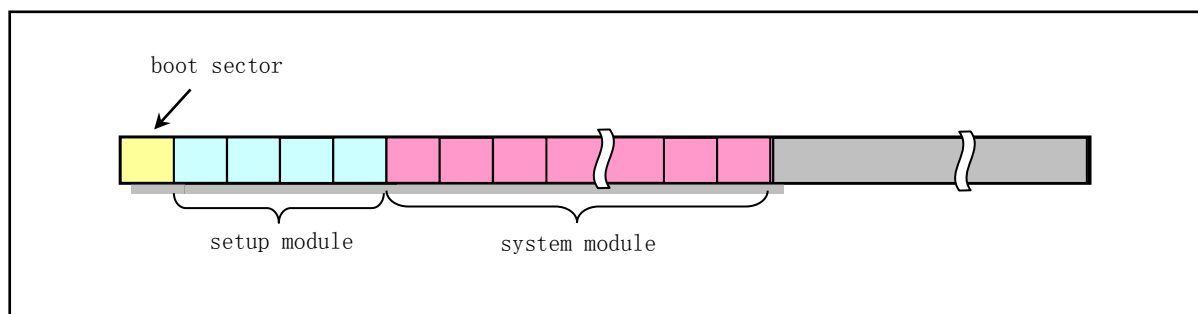


Figure 6-3 Distribution of Linux 0.12 kernel on 1.44MB disk

The figure shows the distribution of sectors occupied by the Linux 0.12 kernel on a 1.44MB disk. There are 80 tracks (cylinders) on each side of the 1.44MB disk platter, each with 18 sectors and a total of 2880 sectors. The boot program code occupies the first sector, the setup module occupies the next four sectors, and the 0.12 kernel system module occupies approximately the next 260 sectors. There are still more than 2610 sectors left unused. These remaining unused space can be used to store a basic root file system, creating an integrated disk that can be used to run the system using a single disk. This will be covered in more detail in the chapter on block device drivers.

In addition, the file name of this program is different from other gas assembly language programs. Its suffix is the uppercase 'S'. Using such a suffix allows gas to use the preprocessing functions of the GNU compiler, so you can include statements such as "#include" and "#if" in assembly language programs. The program uses the uppercase suffix mainly to use the "#include" statement in the program to include the constants defined in the linux/config.h header file. See line 6 of the program.

Note again that the text or statements with line number in the source file are original, and statements without line number are the comments of the author.

## 6.2.2 Code Comments

### Program 6-1 linux/boot/bootsect.S

```

! Here the '!' or ';' at the beginning of a line indicates a comment statement.
1 !
2 ! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
3 ! 0x3000 is 0x30000 bytes = 196kB, more than enough for current
4 ! versions of linux
! SYS_SIZE is the size of the system module to be loaded. The unit of size is paragraph,
! 16-byte per paragraph in X86 memory segmentation scope. 0x3000 is 0x30000 bytes=196KB.
! If 1KB=1024 bytes, it is 192 KB. This space size is sufficient for the current kernel
! version. When the it is 0x8000, it means that the kernel is at most 512KB. Since the
! memory of the bootsect and setup code is stored at 0x90000, the value must not exceed
! 0x9000 (indicating 584KB).
5 !
! File linux/config.h defines some constant symbols used by the kernel and the default
! hard disk parameter block used by Linus himself. For example, the following constants
! are defined:
! DEF_SYSSIZE = 0x3000 - The default system module size (in paragraph);
! DEF_INITSEG = 0x9000 - The default destination position to move to;
! DEF_SETUPSEG = 0x9020 - The default location for setup code;
! DEF_SYSSEG = 0x1000 - The default location for system module from disk.

```

```
6 #include <linux/config.h>
7 SYSSIZE = DEF_SYSSIZE
8 !
9 !      bootsect.s          (C) 1991 Linus Torvalds
10 !      modified by Drew Eckhardt
11 !
12 ! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
13 ! itself out of the way to address 0x90000, and jumps there.
14 !
15 ! It then loads 'setup' directly after itself (0x90200), and the system
16 ! at 0x10000, using BIOS interrupts.
17 !
18 ! NOTE! currently system is at most 8*65536 bytes long. This should be no
19 ! problem, even in the future. I want to keep it simple. This 512 kB
20 ! kernel size should be enough, especially as this doesn't contain the
21 ! buffer cache as in minix
22 !
23 ! The loader has been made as simple as possible, and continuous
24 ! read errors will result in a unbreakable loop. Reboot by hand. It
25 ! loads pretty fast by getting whole sectors at a time whenever possible.
26 !
27 ! The directive (pseudo-operator) .globl or .global is used to define whether the
28 ! subsequent identifier is external or global and is forced to be introduced even if
29 ! it is not used. .text, .data, and .bss are used to define the current code section,
30 ! data section, and uninitialized data section, respectively.
31 ! When link multiple objects, the ld86 combines (merges) the corresponding sections
32 ! in each object module according to their categories. Here, all three sections are
33 ! defined in the same address range, so the program is not actually segmented.
34 ! In addition, string followed by a colon is a label, such as the 'begtext:'.
35 .globl begtext, begdata, begbss, endtext, enddata, endbss
36 .text                      ! text section.
37 begtext:
38 .data                      ! data section
39 begdata:
40 .bss                      ! uninitialized data section
41 begbss:
42 .text
43 !
44 ! The equal sign '=' or symbol 'EQU' is used to define the value of identifier or label.
45 SETUPLEN = 4              ! nr of setup-sectors
46                          ! sectors occupied by setup code.
47 BOOTSEG  = 0x07c0         ! original address of boot-sector
48 INITSEG  = DEF_INITSEG    ! we move boot here - out of the way
49                          ! Move to 0x90000 - avoid location used by system;
50 SETUPSEG = DEF_SETUPSEG   ! setup starts here
51                          ! setup code start from 0x90200
52 SYSSEG   = DEF_SYSSEG     ! system loaded at 0x10000 (65536).
53 ENDSEG   = SYSSEG + SYSSIZE ! where to stop loading
54
55 ! ROOT_DEV & SWAP_DEV are now written by "build".
56 ! The root fs device number ROOT_DEV and swap device SWAP_DEV are now written by build.
57 ! Device number 0x306 specifies that the root fs device is the first partition of the
58 ! second hard disk. Linus installed the Linux 0.11 system on the second hard disk, so
```

```

! ROOT_DEV is set to 0x306. When compiling this kernel, you can modify the device number
! according to the location of the device where your root fs is located. For example, if
! your root file system is on the first partition of the first hard disk, then the value
! should be 0x0301, that is (0x01, 0x03). This device number is the old-fashioned drive
! naming method of Linux, until kernel 0.95. The specific values of the hard disk device
! number are as follows:
! Device nr = major device nr * 256 + minor device nr, (or dev_no = (major<<8) + minor)
! (Major nr: 1-memory, 2-disk, 3-drive, 4-ttyx, 5-tty, 6-parallel port, 7-unnamed pipe)
! 0x300 - /dev/hd0 - Represents the entire 1st hard drive;
! 0x301 - /dev/hd1 - The 1st partition of the 1st disk;
! ...
! 0x304 - /dev/hd4 - The 4th partition of the 1st disk;
! 0x305 - /dev/hd5 - Represents the entire 2nd hard drive;
! 0x306 - /dev/hd6 - The 1st partition of the 2nd disk;
! ...
! 0x309 - /dev/hd9 - The 4th partition of the 2nd disk;
!
44 ROOT_DEV = 0                ! The root fs device uses the same boot device;
45 SWAP_DEV = 0                ! The swap device uses the same boot device;
46
! The directive entry forces the linker to include the specified identifier or label in
! the generated executable (a.out). Here is the starting point for program execution.
! The function of 49 -- 58 lines is to move itself (bootsect) from the current segment
! position 0x07c0 (31KB) to 0x9000 (576KB), a total of 256 words (512 bytes), and then
! jump to the label go of the moved code, that is, the next statement of the program.
47 entry start                ! Tell the linker, program starts at the label start.
48 start:
49     mov     ax, #BOOTSEG     ! Set the ds segment register to 0x7C0;
50     mov     ds, ax
51     mov     ax, #INITSEG     ! Set the es segment register to 0x9000;
52     mov     es, ax
53     mov     cx, #256         ! Set move count = 256 words (512 bytes);
54     sub     si, si           ! Source address ds:si = 0x07C0:0x0000
55     sub     di, di           ! Destination address es:di = 0x9000:0x0000
56     rep     rep              ! counting down, until cx = 0.
57     movsw                    ! Move cx words from memory [si] to [di].
! Jump Intersegment. INITSEG - the segment to jump to, go - the offset within the segment.
58     jmp     go, INITSEG
59
! Starting from below, the CPU executes in code that has moved to 0x90000 position.
! This code sets several segment registers, including the stack registers ss and sp.
! The stack pointer sp can be pointed as long as it points farther than the 512-byte
! offset (ie address 0x90200). Because the setup program needs to be placed from the
! 0x90200 address ( 4 sectors in size), so the sp should point to a position greater
! than (0x200 + 0x200 * 4 + stack size). Here sp is set to 0x9ff00 - 12 (parameter table
! size), ie sp = 0xfef4. A self-built drive parameter list will be stored above this
! location, as explained below. In fact, when the BIOS loads the boot sector to 0x7c00
! and gives execution to the bootloader, ss = 0x00, sp = 0xffff.
! In addition, the desired function of the push instruction on line 65 is to temporarily
! save the segment on the stack, and then wait for the following to determine the number
! of track sectors before popping the stack, and assign to the segment registers fs and
! gs (line 109). However, since the two statements on lines 67 and 68 modify the position
! of the stack. Therefore, this design is wrong unless the stack is restored to its

```

```

! original location before the stack pop operation is performed. So there is a bug here.
! One of the corrections is to remove line 65 and change line 109 to "mov ax, cs".
60 go:      mov     ax,cs                ! Set ds, es, and ss to segment after moved (0x9000).
61         mov     dx,#0xfef4          ! arbitrary value >>512 - disk parm size
62
63         mov     ds,ax
64         mov     es,ax
65         push    ax                  ! Temp save segment (0x9000) for 109 lines. (slipper!)
66
67         mov     ss,ax               ! put stack at 0x9ff00 - 12.
68         mov     sp,dx
69 /*
70 *      Many BIOS's default disk parameter tables will not
71 *      recognize multi-sector reads beyond the maximum sector number
72 *      specified in the default diskette parameter tables - this may
73 *      mean 7 sectors in some cases.
74 *
75 *      Since single sector reads are slow and out of the question,
76 *      we must take care of this by creating new parameter tables
77 *      (for the first disk) in RAM. We will set the maximum sector
78 *      count to 18 - the most we will encounter on an HD 1.44.
79 *
80 *      High doesn't hurt. Low does.
81 *
82 *      Segments are as follows: ds=es=ss=cs - INITSEG,
83 *      fs = 0, gs = parameter table segment
84 */
85 ! The interrupt 0x1E set by BIOS is not actually an interrupt. The location corresponding
! to the interrupt vector is the address of floppy drive parameter table. This interrupt
! vector is located at memory 0x1E * 4 = 0x78. This code first copies the original floppy
! disk parameter table from memory 0x0000:0x0078 to 0x9000:0xfef4, and then modifies the
! maximum number of sectors per track at offset 4 in the table to 18. Table size is 12 bytes.
86
87         push    #0                  ! set segment reg fs = 0.
88         pop     fs
89         mov     bx,#0x78            ! fs:bx is parameter table address
! The following instruction indicates that the operand of the next statement is in fs
! segment, and it only affects its next statement. Here, the table address pointed to by
! fs:bx is placed in the register pair gs:si as the original address, and register pair
! es:di = 0x9000:0xfef4 is used as the destination address.
90         seg     fs
91         lgs     si,(bx)             ! gs:si is source
92
93         mov     di,dx               ! es:di is destination ! dx = 0xfef4, set on line 61.
94         mov     cx,#6              ! copy 12 bytes
95         cld                        ! Clear direction flag. Increment pointer when copying.
96
97         rep                                ! Copy the 12-byte table to 0x9000:0xfef4.
98         seg     gs
99         movw
100
101         mov     di,dx               ! Es:di points to new table, then modifies the table.
102         movb    4(di),*18          ! patch sector count

```

```

103
104     seg fs                      ! Let interrupt vector 0x1E point to the new table.
105     mov     (bx),di
106     seg fs
107     mov     2(bx),es
108
109     ! ax is segment value (0x9000) saved on line 65. Set fs=gs=0x9000 to restore original segment.
110     pop     ax
111     mov     fs,ax
112     mov     gs,ax
113
114     ! BIOS INT 0x13 function 0 is used to reset floppy disk controller, that is to forces
115     ! controller to recalibrate drive heads (seek to track 0).
116
117     xor     ah,ah                ! reset FDC
118     xor     dl,dl                ! dl = drive, here set to first disk drive.
119     int     0x13
120
121     ! load the setup-sectors directly after the bootblock.
122     ! Note that 'es' is already set up.
123
124     ! The purpose of line 121--137 is to use the BIOS INT 0x13 function 2 (read disk sectors)
125     ! to read setup module from the beginning of the second sector on disk to the memory at
126     ! 0x90200, total of 4 sectors. If an error occurs during a read operation, the location of
127     ! the error sector is displayed, then the drive is reset and retried without a retreat.
128     ! The INT 0x13 (read sector) parameters are set as follows:
129     ! ah = 0x02 - read disk sector;  al = nr of sectors to read;
130     ! ch = low 8 bits of cylinder nr; cl = sector nr(bit 0-5) high 2 bits cylinder(bit 6-7);
131     ! dh = head number;              dl = drive number(bit 7 set for hard disk);
132     ! return:
133     ! If error, flag CF is set, and ah contains error code.
134     ! es:bx ->point to data buffer;
135
136 load_setup:
137     xor     dx, dx                ! drive 0, head 0
138     mov     cx,#0x0002            ! sector 2, track 0
139     mov     bx,#0x0200            ! address = 512, in INITSEG
140     mov     ax,#0x0200+SETUPLEN  ! service 2, nr of sectors
141     int     0x13                  ! read it
142     jnc     ok_load_setup         ! ok - continue
143
144     push    ax                    ! dump error code
145     call    print_nl              ! print next line.
146     mov     bp, sp                ! ss:bp point to chars (word)
147     call    print_hex             ! display hex value.
148     pop     ax
149
150     xor     dl, dl                ! reset FDC !
151     xor     ah, ah
152     int     0x13
153     j       load_setup            ! j = jmp
154
155 ok_load_setup:
156
157 ! Get disk drive parameters, specifically nr of sectors/track

```



```

! The following code uses INT 0x13 function 8 to take the parameters of the disk drive.
! In fact, the number of sectors per track is taken and stored at the location sectors.
! The INT 0x13 (get disk drive parameters) parameters are set as follows:
! ah = 0x08      dl = drive number (bit 7 set for hard disk);
! Return:
! CF is set if an error occurs, and ah = status code.
! ah = 0,  al = 0,          bl = drive type (AT/PS2);
! ch = low 8 bits of max cylinder nr;
! cl = max sector number (bit 5-0), high 2 bits of max cylinder number (bit 7-6);
! dh = max head number;    dl = number of drives;
! es:di -> drive parameter table (floppy only).

```

[142](#)[143](#)

```
    xor    dl, dl
```

[144](#)

```
    mov    ah, #0x08      ! AH=8 is get drive parameters
```

[145](#)

```
    int    0x13
```

[146](#)

```
    xor    ch, ch
```

```

! The following instruction indicates that the operand of the next statement is in cs
! segment. It only affects its next statement. In fact, since the code and data are all
! set in the same segment, the values of the segment registers cs and ds, es are the same.
! Therefore, the instruction may not be used here.

```

[147](#)

```
    seg cs
```

```

! The next sentence saves the number of sectors per track. For a floppy disk (dl=0), its
! maximum track number will not exceed 256, and ch is enough to represent it, so bits 6-7
! of cl must be zero. Also, since 146 lines have been set to ch=0, at this time, cx is the
! number of sectors per track.

```

[148](#)

```
    mov    sectors, cx
```

[149](#)

```
    mov    ax, #INITSEG
```

[150](#)

```
    mov    es, ax      ! Because the interrupt changed es, here restore it.
```

[151](#)[152](#)

```

! Print some inane message
! Using BIOS INT 0x10 function 0x03 and 0x13 to display the message: "'Loading' + cr + lf",
! which displays a total of 9 chars, including carriage return and line feed control chars.
! The BIOS INT 0x10 (read cursor location) parameters are set as follows:
! ah = 0x03, read cursor position and size;
! bh = page number;
! Return:
! ch = start scan line; cl = end scan line;
! dh = row (0x00 is top); dl = colum(0x00 is left);
!
! The BIOS INT 0x10 (write string) parameters are set as follows:
! ah = 0x13, write string;
! al = write mode. 0x01-use attributes in bl, cursor stop at end of string.
! bh = page number; bl = attributes;  dh, dl = row, colum at which to start writing.
! cx = number of characters in string.
! es:bp -> string to write.
! Return: Nothing.

```

[153](#)[154](#)

```
    mov    ah, #0x03      ! read cursor pos
```

[155](#)

```
    xor    bh, bh
```

[156](#)

```
    int    0x10      ! position: dh - row(0--24), dl - colum(0--79).
```

[157](#)[158](#)

```
    mov    cx, #9      ! Total 9 charachers.
```

```

159      mov     bx,#0x0007      ! page 0, attribute 7 (normal)
160      mov     bp,#msg1        ! es:bp point to message.
161      mov     ax,#0x1301      ! write string, move cursor
162      int     0x10
163
164 ! ok, we've written the message, now
165 ! we want to load the system (at 0x10000)
166
167      mov     ax,#SYSSEG
168      mov     es,ax           ! segment of 0x010000
169      call    read_it         ! load system module, es is parameter.
170      call    kill_motor      ! stop motor to know the drive status.
171      call    print_nl
172
173 ! After that we check which root-device to use. If the device is
174 ! defined (!= 0), nothing is done and the given device is used.
175 ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
176 ! on the number of sectors that the BIOS reports currently.
177 !
178 ! The meanings of the above two device files are as follows:
179 ! In Linux, the floppy drive's major number is 2 (see comment on line 43), the
180 ! minor device number = type*4 + nr, where nr is 0-3 for floppy drive A, B, C or D
181 ! respectively; type is floppy drive Type (2--1.2MB or 7--1.44MB, etc.).
182 ! Since 7*4 + 0 = 28, /dev/PS0(2,28) refers to 1.44MB A drive with device number 0x021c.
183 ! Similarly, /dev/at0 (2,8) refers to 1.2MB A drive with device number 0x0208.
184
185 ! The root_dev is defined at the location of the boot sector 508, 509 bytes, referring
186 ! to the device number of the root file system.
187 ! This value needs to be modified based on the drive where your root fs is located.
188 ! For example, if your root fs is on the 1st partition of the 1st hard disk, then the
189 ! value should be 0x0301, that is (0x01, 0x03). If the root fs is on the second 1.44MB
190 ! floppy disk, then the value should be 0x021D, which is (0x1D, 0x02).
191 ! When compiling the kernel, you can specify your own values in the Makefile. The kernel
192 ! image file creation program tools/build will also use the value you specify to set the
193 ! device number of your root file system.
194
195      seg cs
196      mov     ax,root_dev
197      or      ax,ax           ! root_dev is defined (not 0) ?
198      jne     root_defined
199
200 ! The following statements use the number of sectors per track saved in 'sectors' in
201 ! line 148 above to determine the disk type. If sectors = 15, it means 1.2MB drive; if
202 ! sectors = 18, it means 1.44MB floppy drive. Because it is a bootable drive, it's
203 ! definitely an A drive.
204
205      seg cs
206      mov     bx,sectors
207      mov     ax,#0x0208      ! /dev/ps0 - 1.2Mb
208      cmp     bx,#15          ! sectors = 15 ?
209      je      root_defined
210      mov     ax,#0x021c      ! /dev/PS0 - 1.44Mb
211      cmp     bx,#18
212      je      root_defined
213      undef_root:            ! If not, then an infinite loop (dead).

```

```
191         jmp undef_root
192 root_defined:
193         seg cs
194         mov     root_dev,ax          ! Save the checked device number to root_dev.
195
196 ! after that (everything loaded), we jump to
197 ! the setup-routine loaded directly after
198 ! the bootblock:
199 ! The inter segment jump instruction, jump to 0x9020:0000 to execute setup code.
200         jmp     0,SETUPSEG          ! At this point, the bootsect code ends!!!

! Here are a few subroutines. read_it is used to read the system module on the disk.
! Kill_moter is used to close the floppy drive motor. There are also some screen display
! subroutines.
201
202 ! This routine loads the system at address 0x10000, making sure
203 ! no 64kB boundaries are crossed. We try to load it as fast as
204 ! possible, loading whole tracks whenever we can.
205 !
206 ! in:  es - starting address segment (normally 0x1000)
207 !
208 ! The following directive .word defines a 2-byte target, which is equivalent to the
209 ! variables defined in the C language program and the amount of memory space occupied.
210 ! The value '1+SETUPLEN' indicates that one boot sector has been read at the beginning,
211 ! plus the number of sectors occupied by the setup code (SETUPLEN = 4).
212 read_it: .word 1+SETUPLEN          ! sectors read of current track
213 head:    .word 0                  ! current head
214 track:    .word 0                  ! current track
215
216 ! First check the input segment. The data read from disk must be stored at the beginning
217 ! of the boundary of memory address 64KB, otherwise it will enter an infinite loop.
218 ! Register bx is the starting position for storing data in the current segment.
219 ! The test instruction on line 214 is bitwise logical with two operands. If the bits
220 ! corresponding to both operands are 1, the resulting value is 1, otherwise 0. The result
221 ! of this operation only affects flags (zero flag ZF, etc.). For example, if AX=0x1000,
222 ! the test result is (0x1000 & 0x0fff) = 0x0000, and the ZF flag is set. At this point,
223 ! the next instruction jne condition does not hold.
224
225         mov ax,es
226         test ax,#0x0fff
227 die:     jne die                  ! es must be at 64kB boundary
228         xor bx,bx                  ! bx is starting address within segment
229 rp_read:
230 ! Then check if all the data has been read. Compare whether the currently read segment is
231 ! the segment where the end of the system data is located (#ENDSEG). If not, jump to the
232 ! label ok1_read below to continue reading data. Otherwise return.
233
234         mov ax,es
235         cmp ax,#ENDSEG            ! have we loaded all yet?
236         jb ok1_read
237         ret
238 ok1_read:
239 ! Next, calculate and verify the number of sectors that the current track needs to read,
```

```

! and put it in ax register. The method is as follows:
! According to the number of sectors that have not been read on the current track and the
! offset position of the data bytes in the segment, it is calculated whether the total
! number of bytes read will exceed the limit of 64 KB segment length if all unread sectors
! are read. If it is exceeded, the number of sectors that need to be read this time is
! calculated based on the maximum number of bytes that can be read (64KB - offset ).
223     seg cs
224     mov ax, sectors      ! get nr of sectors per track.
225     sub ax, sread       ! Subtract the nr of sectors the track has been read.
226     mov cx, ax          ! cx = ax = the nr of unread sectors on the track.
227     shl cx, #9          ! cx = cx * 512 + current offset (bx).
228     add cx, bx          !   = total nr of bytes read after the operation.
229     jnc ok2_read        ! If it does not exceed 64KB, jump to ok2_read.
230     je ok2_read
! Add the data of the unread sectors on the track. If the result exceeds 64KB, then
! calculate the maximum number of bytes that can be read at this time: (64KB - offset).
! Then convert to the number of sectors to be read. Where 0 minus a certain number is the
! complement number of 64KB.
231     xor ax, ax
232     sub ax, bx
233     shr ax, #9
234 ok2_read:
! Read the data on the track to es:bx based on the specified start sector (cl) and number
! of sectors (al). The number of sectors that have been read on the current track is then
! counted and compared to the maximum number of sectors in the track. If it is less than
! 'sectors', there is still sectors unread on the track, so jump to ok3_read to continue.
235     call read_track     ! Reads data for given number of sectors on the track.
236     mov cx, ax          ! cx = the nr of sectors read by this time.
237     add ax, sread       ! plus the nr of sectors that have been read.
238     seg cs
239     cmp ax, sectors     ! If there are unread sectors on the track, jump to ok3_read
240     jne ok3_read
! If all sectors of the current head of the track have been read, the data on the next
! head of the track (head 1) is read. If it is already done, go to the next track.
241     mov ax, #1
242     sub ax, head        ! check current head no.
243     jne ok4_read       ! If it is head 0, then go get sectors on head 1.
244     inc track
245 ok4_read:
246     mov head, ax        ! store current head no to head.
247     xor ax, ax          ! Clear the number of sectors read.
248 ok3_read:
! If there are still unread sectors on the current track, first save the number of sectors
! read and then adjust the starting position where the data is stored. If it is less than
! the 64KB boundary value, jump to rp_read (line 217) and continue reading data.
249     mov sread, ax       ! save the nr of sectors read on the track.
250     shl cx, #9          ! nr of sectors read * 512 bytes.
251     add bx, cx          ! Adjust the starting position of the data.
252     jnc rp_read
! Otherwise it indicates that 64KB of data has been read. At this point, adjust the
! current segment to prepare for reading the next segment.
253     mov ax, es
254     add ah, #0x10        ! Adjust segment base address to point to next 64KB.

```

```
255      mov es,ax
256      xor bx,bx                ! clear offse value.
257      jmp rp_read
258
! Read_track subroutine. Read the data of the specified start sector and the number of
! sectors on the track to the beginning of es:bx. See the description of the BIOS disk
! read interrupt int 0x13, ah=2 under line 119.
! al - sectors to be read; es:bx - data buffer.
259 read_track:
! First call BIOS INT 0x10, function 0x0e (write characters by telex), the cursor moves
! forward one position. Show a dot '.'.
260      pusha                    ! push all registers.
261      pusha
262      mov ax, #0xe2e           ! loading... message 2e = .
263      mov bx, #7               ! character foreground color attribute.
264      int 0x10
265      popa
266
! Then the track sector read operation is formally performed.
267      mov dx,track             ! current track.
268      mov cx,sread             ! sectors already read on the current track.
269      inc cx                   ! cl = start reading sector nr.
270      mov ch,dl                ! ch = current head nr.
271      mov dx,head              ! get current head nr.
272      mov dh,dl                ! dh = head nr, dl = drive (0 for A drive)
273      and dx,#0x0100           ! head nr is no more than 1.
274      mov ah,#2                ! ah = 2, read sectors.
275
276      push dx                  ! save for error dump
277      push cx
278      push bx
279      push ax
280
281      int 0x13
282      jc bad_rt                ! if error, jump to bad_rt
283      add sp,#8                ! if ok, discard status info.
284      popa
285      ret
286
! Error reading disk. The error message is displayed first, then the drive reset operation
! (disk interrupt function number 0) is executed, and then jump to the read_track to try again.
287 bad_rt: push ax               ! save error code
288      call print_all           ! ah = error, al = read
289
290
291      xor ah,ah
292      xor dl,dl
293      int 0x13
294
295
296      add sp, #10              ! Discard the info saved for the error condition.
297      popa
298      jmp read_track
```

```
299
300 /*
301 *    print_all is for debugging purposes.
302 *    It will print out all of the registers.  The assumption is that this is
303 *    called from a routine, with a stack frame like
304 *    dx
305 *    cx
306 *    bx
307 *    ax
308 *    error
309 *    ret <- sp
310 *
311 */
312
313 print_all:
314     mov cx, #5                ! error code + 4 registers
315     mov bp, sp                ! Save current stack pointer sp.
316
317 print_loop:
318     push cx                    ! save count left
319     call print_nl              ! nl for readability
320     jae no_reg                 ! see if register name is needed
321                                ! if CF=0, registers are not displayed and jump.
    Corresponding to the stack register order, display their names: "AX : " etc.
322     mov ax, #0xe05 + 0x41 - 1 ! ah = function 0x0e; al = char (0x05 + 0x41 -1)
323     sub al, cl
324     int 0x10
325
326     mov al, #0x58              ! X
327     int 0x10
328
329     mov al, #0x3a              ! :
330     int 0x10
331
    ! Display the contents of stack pointed by bp. originally bp points to return address.
332 no_reg:
333     add bp, #2                 ! next register
334     call print_hex             ! print it
335     pop cx
336     loop print_loop
337     ret
338
    ! Call BIOS INT 0x10 to display carriage return and line feed control chars.
339 print_nl:
340     mov ax, #0xe0d             ! CR
341     int 0x10
342     mov al, #0xa               ! LF
343     int 0x10
344     ret
345
346 /*
347 *    print_hex is for debugging purposes, and prints the word
348 *    pointed to by ss:bp in hexadecimal.
```

```

349 */
350     ! Call BIOS INT 0x10 to display the word pointed to by ss:bp in 4 hexadecimals.
351 print_hex:
352     mov     cx, #4             ! 4 hex digits
353     mov     dx, (bp)          ! load word into dx
354 print_digit:
355     ! The high byte is displayed first, so rotate dx by 4 to move high 4 bits to dx lower 4 bits.
356     rol     dx, #4            ! rotate so that lowest 4 bits are used
357     mov     ah, #0xe
358     mov     al, dl             ! mask off so we have only next nibble
359     and     al, #0xf           ! put in al, and get lower 4 bits only.
360     ! Add '0' ASCII code 0x30 to convert the value to a char. If value in al exceeds 0x39, it
361     ! means that the value displayed exceeds number 9, so it needs to be represented by 'A'--'F'.
362     add     al, #0x30          ! convert to 0 based digit, '0'
363     cmp     al, #0x39          ! check for overflow
364     jbe     good_digit
365     add     al, #0x41 - 0x30 - 0xa ! 'A' - '0' - 0xa
366 good_digit:
367     int     0x10
368     loop    print_digit       ! cx--. If cx>0, the next value is displayed.
369     ret
370 /*
371 * This procedure turns off the floppy drive motor, so
372 * that we enter the kernel in a known state, and
373 * don't have to worry about it later.
374 */
375 ! The value 0x3f2 on line 377 below is a port address of the floppy disk controller and
376 ! is referred to as a digital output register (DOR) port. It is an 8-bit register with
377 ! bits 7 - 4 for controlling the start and stop of four floppy drives (D--A). Bits 3 - 2
378 ! are used to enable/disable DMA and interrupt requests and to start/reset the floppy
379 ! disk controller FDC. Bit 1 - Bit 0 is used to select the floppy drive for the selected
380 ! operation. The value of 0 set in al on line 378 is used to select the A drive, turn
381 ! off the FDC, disable the DMA and interrupt requests, and turn off the motor. See the
382 ! instructions behind the kernel/blk_drv/floppy.c program for more information on floppy
383 ! control card programming.
384 kill_motor:
385     push dx
386     mov dx, #0x3f2             ! floppy controller port DOR.
387     xor al, al                 ! A drive, close FDC, disable DMA & int, close moter
388     outb                                ! output al to port dx.
389     pop dx
390     ret
391 sectors:
392     .word 0                    ! store nr of sectors per track.
393 msg1:
394     .byte 13, 10               ! message to display, total 9 chars.
395     .ascii "Loading"

```

```
389 ! Start at address 506 (0x1FA), so root_dev is in 2 bytes starting at 508 of boot sector.
390 .org 506
391 swap_dev:
392     .word SWAP_DEV
393 root_dev:
394     .word ROOT_DEV

    ! 0xAA55 is a flag for the boot disk to have a valid boot sector for use by the BIOS
    ! program to load the boot sector. It must be the last two bytes of the boot sector.
395 boot_flag:
396     .word 0xAA55
397
398 .text
399 endtext:
400 .data
401 enddata:
402 .bss
403 endbss:
404
```

---

### 6.2.3 Reference information

For the description of the bootsect.S program, a large amount of documents can be found online. Among them, Alessandro Rubini's article "Tour of the Linux kernel source" describes the kernel boot process more comprehensively. Since this program runs in CPU real mode, it will be easier to understand. If you still have difficulty reading at this time, then I suggest you review the 80x86 assembly and its hardware first, and then continue reading this book. For the newly developed Linux kernel, this program has not changed much, and basically maintains the appearance of the 0.12 version of the bootsect file.

#### 6.2.3.1 Linux 0.12 Hard Disk Device Number

In Linux systems, various devices are accessed by device number, or referred to as logical device number. The device number consists of the major device number and the minor device number. The major device number specifies the type of device, while the minor device number specifies the specific device object. The major and minor device numbers are represented by 1 byte, that is, each device number has 2 bytes. The hard disk involved in the bootsect program is a block device whose major device number is 3. The major device numbers of basic devices in Linux systems are:

- 1 - memory;
- 2 - floppy disk;
- 3 - hard disk;
- 4 - ttyx;
- 5 - tty;
- 6 - parallel port;
- 7 - Unnamed pipe.

Since there can be 1--4 partitions in a traditional hard disk, the hard disk also uses the minor device number to specify the partition. The logical device number of the hard disk is composed of the following:



device number = major device number <<8 + minor device number.

-----

All logical device numbers for both hard disks are shown in Table 6–1. 0x0300 and 0x0305 do not correspond to any one partition, but represent the entire hard disk. Also note that since the Linux kernel version 0.95 has not used this cumbersome naming method, it uses the same naming method as it is now.

Table 6-1 Hard disk logical device number

Device nr	Device file	Description
0x0300	/dev/hd0	Represents the entire first hard driv
0x0301	/dev/hd1	The first partition of the first disk
0x0302	/dev/hd2	The second partition of the first disk
0x0303	/dev/hd3	The third partition of the first disk
0x0304	/dev/hd4	The fourth partition of the first disk
0x0305	/dev/hd5	Represents the entire second hard driv
0x0306	/dev/hd6	The first partition of the second disk
0x0307	/dev/hd7	The second partition of the second disk
0x0308	/dev/hd8	The third partition of the second disk
0x0309	/dev/hd9	The fourth partition of the second disk

### 6.2.3.2 Booting from hard disk

The bootsect program gives the default method and process for booting a Linux system from a floppy disk. If you want to boot your system from a hard drive, you usually need to use a different multi-OS bootloader to boot the system, such as multiple operating system bootloaders such as Shoelace, LILO, or Grub. At this point, the operations that bootsect.S needs to perform will be completed by these programs, and the bootsect program will not be executed. Because if you boot from the hard disk, usually the kernel image file will be stored in the root file system of an active partition of the hard disk. So you need to know where the kernel image file is in the file system and what file system it is, that is, your boot sector program needs to be able to recognize and access the file system and read the kernel image file from it.

The basic process of booting from a hard disk is: after the system is powered on, the first sector of the bootable hard disk (MBR - Master Boot Record) will be loaded into the memory 0x7c00 by the BIOS and execution will begin. The program will first move itself down to memory 0x600, then load into memory 0x7c00 according to the first sector (boot sector) in the active partition specified by the partition table in the MBR, and then start execution.

For the Linux 0.12 system, the kernel image file is independent of the root file system. If you use this method directly to boot the system from the hard disk, you will encounter the problem that the root file system cannot coexist with the kernel image file. There are two possible solutions. One way is to set up a small-capacity active partition to hold the kernel image file, and the corresponding root file system is placed in another partition. Although this will use one more primary partitions on the hard disk, it should be able to boot the system from the hard disk with minimal modifications to the bootsect.S program. Another approach is to combine the kernel image file with the root file system in a partition, that is, the kernel image file is placed in some sectors at the beginning of the partition, and the root file system is stored from a subsequent specified sector. Both methods require some modifications to the code. Readers can refer to the last chapter to use the bochs simulation software to do some experiments.

## 6.3 setup.S

### 6.3.1 Function Descriptions

setup.S is an operating system loader. Its main function is to use the ROM BIOS interrupt to read the machine configuration data, and save the data to the beginning of 0x90000 (covering the place where the bootsect program is located). The parameters obtained and the stored memory locations are shown in Table 6-2. These parameters will be used by the relevant programs in the kernel. For example, the console.c and tty\_io.c programs in the character device driver set.

Table 6-2 Parameters read and stored by the setup program

Address	Size (bytes)	Name	Description
0x90000	2	Cursor Location	Colum (0x00-left), Row (0x00-top most)
0x90002	2	Extended Memory	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	Char Rows ??	
0x9000A	1	Display memory	0x00-64k, 0x01-128k, 0x02-192k, 0x03=256k
0x9000B	1	Display status	0x00-Color, I/O=0x3dX; 0x01-Mono, I/O=0x3bX
0x9000C	2	Property Paras	Property parameters of display adapter.
0x9000E	1	Screen rows	Screen current display rows.
0x9000F	1	Screen columns	Screen current display columns.
...			
0x90080	16	Hd Paras Table	Hard disk parameter table for the first one.
0x90090	16	Hd Paras Table	Hd parameter table for the second one (zero if none).
0x901FC	2	Root devie no	Root file system device number (set in bootsec.s)

Then the setup program moves the system module from 0x10000-0x8ffff to the absolute address 0x00000 (At the time, it was considered that the length of the kernel system module system would not exceed this value: 512 KB). Next, load the interrupt descriptor table register (IDTR) and the global descriptor table register (GDTR), turn on the A20 address line, reconfigure the two interrupt control chips 8259A, and reconfigure the hardware interrupt number to 0x20 - 0x2f. Finally, the CPU's control register CR0 (also called the machine status word) is set, enters the 32-bit protected mode, and jumps to the head.s program at the forefront of the system module to continue running.

In order to enable head.s to run in 32-bit protected mode, the interrupt descriptor table (IDT) and the global descriptor table (GDT) are temporarily set in the program, and the descriptor of the current kernel code and data segments are set in the GDT. In the head.s program below, these descriptor tables are also reconfigured according to the needs of the kernel.

First, let's review the format of the segment descriptor, the structure of the descriptor table, and the format of the segment selector. The format of the code segment and data segment descriptor used in the Linux kernel is shown in Figure 6-4. For the detailed meaning of each field, please refer to the description in Chapter 4.

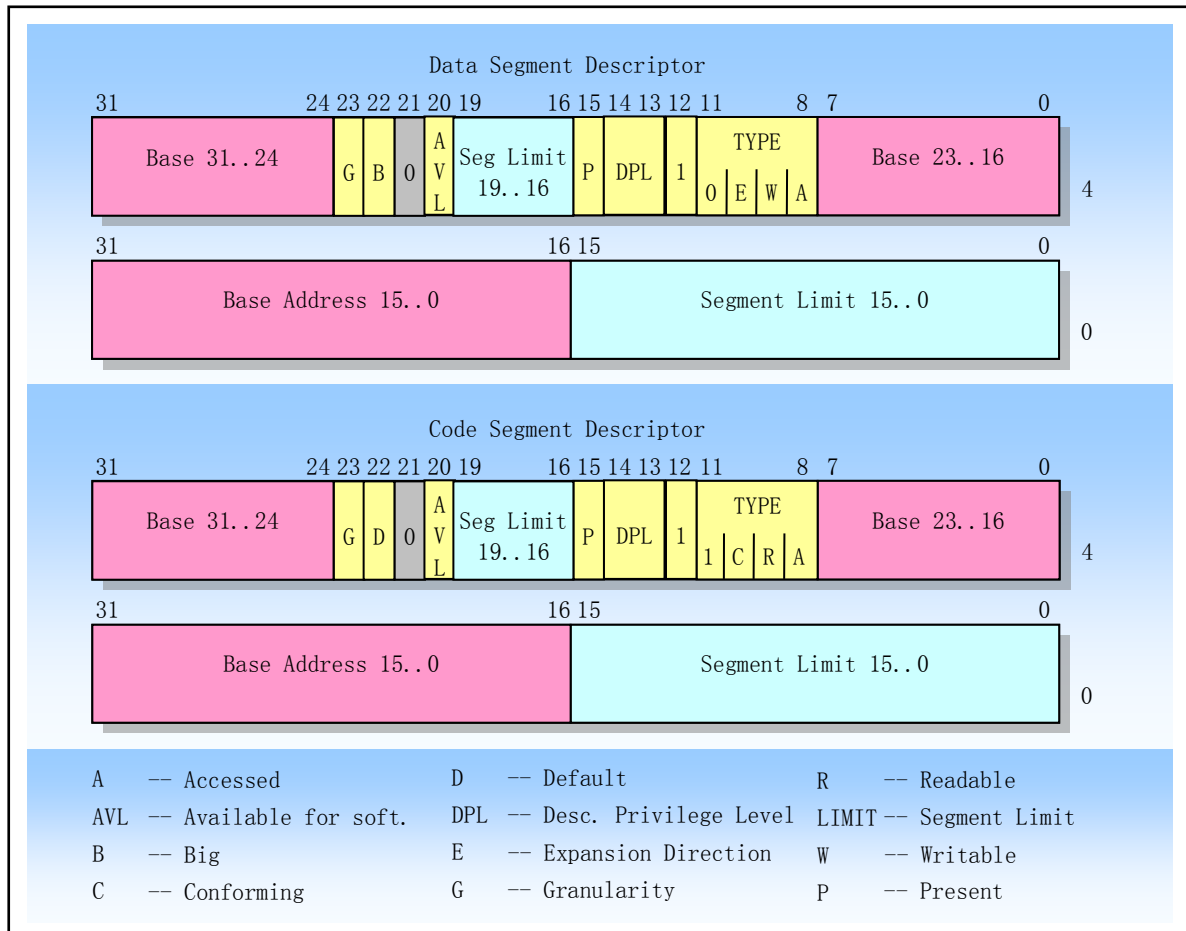


Figure 6-4 Descriptor format for code and data segments

The segment descriptor is stored in the descriptor table. The descriptor table is actually an array of descriptor items in memory. There are two types of descriptor tables: Global descriptor table (GDT) and Local descriptor table (LDT). The processor locates the GDT table and the current LDT table by using the GDTR and LDTR registers. These two registers hold the base address of the descriptor table and the length of the table in a linear address. The instructions LGDT and SGDT are used to access the GDTR register; the instructions LLDT and SLDT are used to access the LDTR register. The LGDT uses a 6-byte operand in memory to load the GDTR register. The first two bytes represent the length of the descriptor table, and the last four bytes are the base address of the descriptor table. Note, however, that the operand used by the LLDT instruction to access the LDTR register is a 2-byte operand representing the selector of a descriptor entry in the global descriptor table GDT. The descriptor item in the GDT table corresponding to the selector should correspond to a local descriptor table.

For example, the GDT descriptor item set by the setup.S program (see lines 567-578). The value of the code segment descriptor is 0x00C09A000000007FF (ie: 0x07FF, 0x0000, 0x9A00, 0x00C0). Indicates that the limit length of the code segment is 8MB ( $= (0x7FF + 1) * 4KB$ , where 1 is added because the limit length value is counted from 0), the base address of the segment in the linear address space is 0, and the segment type value 0x9A indicates that the segment exists in memory, the privilege level of the segment is 0, the segment type is a readable executable code segment, the segment code is 32 bits, and the granularity of the segment is 4 KB. The value of the data segment descriptor is 0x00C092000000007FF (ie: 0x07FF, 0x0000, 0x9200, 0x00C0), which means that the limit length of the data segment is 8MB, the base address of the segment in the linear address

space is 0, the segment type value 0x92 indicates that the segment exists in the memory, and the privilege level of the segment is 0. The type is a readable and writable data segment, the segment code is 32 bits and the segment granularity is 4 KB.

Here are some more explanations for the selector. The selector part is used to specify a segment descriptor, which is done by specifying a descriptor table and indexing one of the descriptor items. Figure 6-5 shows the format of the selector.

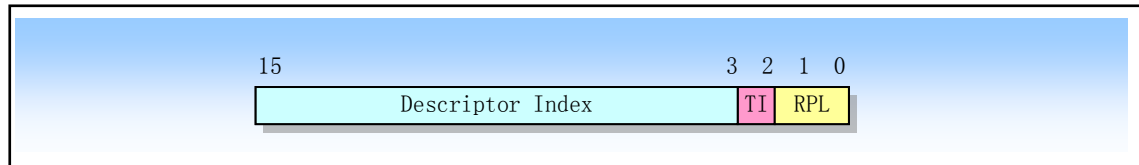


Figure 6-5 Segment selector format

The index is used to select one of the 8192 ( $2^{13}$ ) descriptors in the specified descriptor table. The processor multiplies the index by 8, and adds the base address of the descriptor table to access the segment descriptor specified in the table. The Table Indicator (TI) is used to specify the descriptor table referenced by the selector. A value of 0 indicates that the GDT table is specified, and a value of 1 indicates that the current LDT table is specified. The Requestor's Privilege Level (RPL) is used to protect the mechanism.

Since the first item of the GDT table (index value 0) is not used, a selector with an index value of 0 and a table indicator value of 0 (that is, a selector pointing to the first item of the GDT) can be used as a null selector. When a segment register (cannot be CS or SS) loads a null selector, the processor does not generate an exception. However, an exception is generated if the segment register is used to access memory. This feature is useful for applications that initialize unused segment registers so that their unexpected references can produce a specified exception.

Before entering protected mode, we must first set up the segment descriptor table that will be used, such as the global descriptor table GDT. The instruction LGDT is then used to inform the CPU of the base address of the descriptor table (the base address of the GDT table is stored in the GDTR register). Then set the protection mode flag of the machine status word to enter the 32-bit protection mode.

In addition, the line 215-566 of the setup.S is used to identify the type of display card used in the machine. If the system uses a VGA display card, then we check to see if the display card supports an extended display mode (or display mode) of more than 25 lines x 80 columns. The so-called display mode refers to the method in which the ROM BIOS interrupts the function 0 (ah=0x00) of the INT 0x10 setting screen display information. The input parameter value in al register is the display mode or display mode number we want to set. Usually we refer to several display modes that can be set when IBM PC was first released as the standard display mode, and some modes added later are called extended display mode. For example, in addition to supporting the standard display mode, the ATI display card also supports extended display mode 0x23 and 0x33, that is, it can also display information on the screen using two display modes of 132 columns x 25 rows and 132 columns x 44 rows. When VGA and SVGA are just present, these extended display modes are supported by the BIOS on the display card. If a known type of display card is identified, the program will provide the user with an opportunity to select a resolution.

Since this part of the program involves port information unique to each of the many display cards, this fragment program is complicated. Fortunately, this part of the code has little to do with the kernel's operating

principle, so you can skip it. If you want to understand this code thoroughly, then you should refer to Richard F. Ferraro's book "Programmer's Guide to the EGA, VGA, and Super VGA Cards", or refer to the classic VGA programming material that can be downloaded online:"VGADOC4". This part of the program was programmed by Mats Andersson (d88-man@nada.kth.se), and now Linus has forgotten who is d88-man :-).

### 6.3.2 Code Comments

---

#### Program 6-2 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
15 ! NOTE! These had better be the same as in bootsect.s!
16 ! config.h defines: DEF_INITSEG = 0x9000; DEF_SYSSEG = 0x1000; DEF_SETUPSEG = 0x9020
17 #include <linux/config.h>
18
19 INITSEG = DEF_INITSEG      ! we move boot here - out of the way
20 SYSSEG  = DEF_SYSSEG      ! system loaded at 0x10000 (65536).
21 SETUPSEG = DEF_SETUPSEG   ! this is the current segment
22
23 .globl begtext, begdata, begbss, endtext, enddata, endbss
24 .text
25 begtext:
26 .data
27 begdata:
28 .bss
29 begbss:
30 .text
31 entry start
32 start:
33
34 ! ok, the read went well so we get current cursor position and save it for
35 ! posterity.
36
37      mov     ax,#INITSEG      ! Set ds to INITSEG (0x9000)
38      mov     ds,ax
39
40 ! Get memory size (extended mem, kB)
41 ! Use BIOS INT 0x15 function 0x88 to get extended memory size and store it at 0x90002.
42 ! Return:
```

```

! ax = the extended memory from 0x100000(1MB) in KB. if error CF is set, ax = error code.
41
42     mov     ah,#0x88
43     int     0x15
44     mov     [2],ax           ! store at 0x90002
45
46 ! check for EGA/VGA and some config parameters
! Use BIOS INT 0x10 function 0x12 (video subsystem configuration) to get EGA configuration info.
! ah = 0x12;    bl = 0x10 - return video configuration information.
! Return:
! bh = video state (0x00 - color, I/O port =0x3dX;    0x01 - mono, I/O port =0x3bX).
! bl = installed memory (0x00 - 64k; 0x01 - 128k; 0x02 - 192k; 0x03 = 256k).
! cx = adapter features and settings (see the description of INT 0x10 after this list).
47
48     mov     ah,#0x12
49     mov     bl,#0x10
50     int     0x10
51     mov     [8],ax           ! 0x90008 = ??
52     mov     [10],bx          ! 0x9000A = installed mem, 0x9000B = display state.
53     mov     [12],cx          ! 0x9000C = adapter features and settings.

! Detect screen rows and columns. If adapter is a VGA card, the user is requested
! to select the row and save it to 0x9000E.
54     mov     ax,#0x5019       ! set default row and columns in ax(ah = 80, al = 25).
55     cmp     bl,#0x10         ! If bl is 0x10, it means not a VGA card, jump.
56     je      novga
57     call    chsvga           ! get card manufacturer & type, modify row col(line 215)
58 novga: mov     [14],ax        ! Save screen row & column values (0x9000E, 0x9000F).

! Use BIOS INT 0x10 function 0x03 to get cursor position, and save it in 0x90000 (2 bytes).
! ah = 0x03 get cursor position; bh = page number.
! Return:
! ch = start scan line;cl = end scan line;
! dh = row (0x00 is top); dl = column(0x00 is left);
59     mov     ah,#0x03         ! read cursor pos
60     xor     bh,bh
61     int     0x10             ! save it in known place, con_init fetches
62     mov     [0],dx           ! it from 0x90000.
63
64 ! Get video-card data:
! Use BIOS INT 0x10, function 0x0f to get the current display mode and status.
! ah = 0x0f - get current video mode and state
! Return:
! ah = nr of screen columns; al = display mode; bh = current display page.
! 0x90004(word) - store current page; 0x90006 - display mode; 0x90007 - screen columns
65
66     mov     ah,#0x0f
67     int     0x10
68     mov     [4],bx           ! bh = display page
69     mov     [6],ax           ! al = video mode, ah = window width
70
71 ! Get hd0 data
! The address of the first hard disk parameter table is actually the vector value of

```

! interrupt 0x41! And the second hard disk parameter table is next to the first one. The  
! vector value of the interrupt 0x46 also points to the second hard disk parameter table.  
! The table size is 16 bytes. The following part copies the BIOS's two parameter tables to  
! the new location: the first table is stored at 0x90080, the second is stored at 0x90090.  
! For the description of the hard disk parameter table, see Table 6-4 in section 6.3.3.

[72](#)

! The 75th line reads a long pointer value from memory and places it in the ds and si  
! registers. Here, the 4 bytes stored at the memory 4 \* 0x41 (= 0x104) are read. These 4  
! bytes are the start address of the hard disk parameter table.

[73](#)

mov ax, #0x0000

[74](#)

mov ds, ax

[75](#)

lds si, [4\*0x41] ! Get INT 0x41 vector, addr of hd0 para table -> ds:si

[76](#)

mov ax, #INITSEG

[77](#)

mov es, ax

[78](#)

mov di, #0x0080 ! Destination of replication: 0x9000:0x0080 -> es:di

[79](#)

mov cx, #0x10 ! move total 16 bytes.

[80](#)

rep

[81](#)

movsb

[82](#)

[83](#) ! Get hdl data

[84](#)

[85](#)

mov ax, #0x0000

[86](#)

mov ds, ax

[87](#)

lds si, [4\*0x46] ! INT 0x46 vector value -> ds:si

[88](#)

mov ax, #INITSEG

[89](#)

mov es, ax

[90](#)

mov di, #0x0090 ! 0x9000:0x0090 -> es:di

[91](#)

mov cx, #0x10

[92](#)

rep

[93](#)

movsb

[94](#)

[95](#) ! Check that there IS a hdl :-)

! Check if the machine has a second hard drive. If not, clear the second table.

! Use the ROM BIOS INT 0x13 function 0x15 to retrieve the disk type.

! ah = 0x15 - get disk type.

! dl = drive number (0x8X for hard dirve, 0x80 - drive 0, 0x81 - drive 1)

! Return:

! ah = type code (00 - no drive; 01 - floppy, no change detection;

! 02 - floppy (or other removable), change detection; 03 - hard disk).

! cx:dx - number of 512-byte sectors.

! CF set on error, ah = status.

[96](#)

[97](#)

mov ax, #0x01500

[98](#)

mov dl, #0x81

[99](#)

int 0x13

[100](#)

jc no\_disk1

[101](#)

cmp ah, #3 ! hard drive? (type == 3)?

[102](#)

je is\_disk1

[103](#) no\_disk1:

[104](#)

mov ax, #INITSEG ! no 2nd hard drive, clean 2nd parameter table.

[105](#)

mov es, ax

[106](#)

mov di, #0x0090

[107](#)

mov cx, #0x10

```

108      mov     ax,#0x00
109      rep
110      stosb
111 is_disk1:
112
113 ! now we want to move to protected mode ...
114
115      cli                      ! no interrupts allowed !
116
117 ! first we move the system to it's rightful place
! The purpose of the following program is to move the entire system module to the 0x00000
! position, that is, move memory block (512KB) (0x10000 - 0x8ffff) to low end of memory.
118
119      mov     ax,#0x0000
120      cld                      ! 'direction'=0, movs moves forward
121 do_move:
122      mov     es,ax            ! destination segment ! es:di (0x0:0x0 initially)
123      add     ax,#0x1000
124      cmp     ax,#0x9000      ! Has the last seg (64KB from 0x8000 seg) code moved?
125      jz      end_move        ! yes, jump.
126      mov     ds,ax            ! source segment ! ds:si (0x1000:0x0 initially)
127      sub     di,di
128      sub     si,si
129      mov     cx,#0x8000      ! move 0x8000 words (64K bytes)
130      rep
131      movsw
132      jmp     do_move
133
134 ! then we load the segment descriptors
! From here on, you will encounter 32-bit protected mode operation. See Chapter 4 for
! information on this. Before running into protected mode, we need to first set up the
! segment descriptor table to be used. Here you need to set the global descriptor table
! GDT and the interrupt descriptor table IDT.
!
! The instruction, LIDT, is used to load the interrupt descriptor table register. Its
! operand (idt_48) has 6 bytes. The first 2 bytes (bytes 0-1) are the size of descriptor
! table; the last 4 bytes (bytes 2-5) are the 32-bit linear base of the descriptor table.
! See the following 580--486 lines. Each 8-byte entry in the IDT table indicates the code
! information that needs to be called when an interrupt occurs. It is somewhat similar to
! the interrupt vector, but contains more information.
!
! The LGDT instruction is used to load the global descriptor table register with the same
! operand format as LIDT instruction. Each descriptor item (8 bytes) in the GDT describes
! the information of the data segment and code segment (block) in the protected mode. This
! includes the segment's maximum limit (16 bits), linear base address (32 bits), privilege
! level, in memory flag, read and write permissions, and other flags. See line 567--578.
135
136 end_move:
137      mov     ax,#SETUPSEG     ! right, forgot this at first. didn't work :-)
138      mov     ds,ax            ! ds point to this code segment (setup)
139      lidt    idt_48           ! load idt with 0,0
140      lgdt    gdt_48           ! load gdt with whatever appropriate
141

```



```

142 ! that was painless, now we enable A20
! In order to access and use more than 1MB of physical memory, we need to first enable the
! A20 address line. See the description of the A20 line after this programs. As for whether
! the machine actually enable the A20 line, we also need to test it after entering the
! protection mode (after more than 1MB memory can be accessed). This work is placed in the
! head.S program (32-36 lines).
143
144         call    empty_8042        ! Test 8042 status reg, wait for input buffer be empty.
! A write cmd can be run only if input buffer is empty.
145         mov     al,#0xD1          ! command write ! 0xD1 cmd code, write to P2 of 8042.
146         out     #0x64,al          ! Bit 1 of P2 is used for strobing of A20 line.
147         call    empty_8042        ! Waiting buffer to be empty, to see if cmd is accepted.
148         mov     al,#0xDF          ! A20 on          ! parameters for the A20 line.
149         out     #0x60,al          ! write to port 0x60.
150         call    empty_8042        ! if input buffer is empty, then A20 line enabled.
151
152 ! well, that went ok, I hope. Now we have to reprogram the interrupts :-(
153 ! we put them right after the intel-reserved hardware interrupts, at
154 ! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
155 ! messed this up with the original PC, and they haven't been able to
156 ! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
157 ! which is used for the internal hardware interrupts as well. We just
158 ! have to reprogram the 8259's, and it isn't fun.
159
! The PC uses two programmable interrupt controller chips 8259A. For the programming
! method of 8259A, please refer to the introduction after this program. The two words
! (0x00eb) defined on line 162 are two relative jump instructions that are directly
! represented by machine code, acting as a delay.
!
! 0xeb is the opcode of the direct near jump instruction with a relative offset value of
! 1 byte. The CPU creates a new effective address by adding this relative offset to the
! EIP register. The number of CPU clock cycles spent on execution is 7 to 10. 0x00eb
! indicates an instruction whose jump offset is 0, so the next instruction is executed
! directly. These two instructions provide a total delay time of 14-20 CPU clock cycles.
! Because there is no mnemonic for the corresponding instruction in as86, Linus uses
! machine code directly to represent this instruction in some assembly files. In addition,
! the number of clock cycles per NOP instruction is 3, so 6 to 7 NOP instructions are
! required to achieve the same delay effect.
!
! The 8259A chip master port is 0x20-0x21, and the slave port is 0xA0-0xA1. The output
! value 0x11 indicates the start of the initialization command, which is the ICW1 command
! word, indicating the edge trigger, multiple 8259 cascades, and requires the ICW4 command
! word to be sent last.
160         mov     al,#0x11          ! initialization sequence
161         out     #0x20,al          ! send it to 8259A-1
162         .word   0x00eb,0x00eb    ! jmp $+2, jmp $+2      ! '$' is current address.
163         out     #0xA0,al          ! and to 8259A-2
164         .word   0x00eb,0x00eb
! The Linux system hardware interrupt number is set to start at 0x20.
165         mov     al,#0x20          ! start of hardware int's (0x20)
166         out     #0x21,al          ! send ICW2 cmd to master chip.
167         .word   0x00eb,0x00eb
168         mov     al,#0x28          ! start of hardware int's 2 (0x28)

```

```
169      out      #0xA1,al          ! send ICW2 cmd to slave chip.
170      .word    0x00eb,0x00eb
171      mov      al,#0x04          ! 8259-1 is master
172      out      #0x21,al          ! ICW3 cmd, chain pin IR2 to pin INT of slave chip.
173      .word    0x00eb,0x00eb
174      mov      al,#0x02          ! 8259-2 is slave
175      out      #0xA1,al          ! ICW3 cmd, chain pin INT to pin IR2 on master chip.
176      .word    0x00eb,0x00eb
      ! 8086 mode. It means normal EOI, unbuffered mode, need to send instructions to reset.
      ! Initialization is over, chip ready.
177      mov      al,#0x01          ! 8086 mode for both
178      out      #0x21,al          ! ICW4 cmd (8086 mode).
179      .word    0x00eb,0x00eb
180      out      #0xA1,al          ! send ICW4 to slave chip.
181      .word    0x00eb,0x00eb
182      mov      al,#0xFF          ! mask off all interrupts for now
183      out      #0x21,al
184      .word    0x00eb,0x00eb
185      out      #0xA1,al
186
187 ! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
188 ! need no steenking BIOS anyway (except for the initial loading :-).
189 ! The BIOS-routine wants lots of unnecessary data, and it's less
190 ! "interesting" anyway. This is how REAL programmers do it.
191 !
192 ! Well, now's the time to actually move into protected mode. To make
193 ! things as simple as possible, we do no register set-up or anything,
194 ! we let the gnu-compiled 32-bit programs do that. We just jump to
195 ! absolute address 0x00000, in 32-bit protected mode.
196
      ! Below we set and enter the 32-bit protected mode to run. First load the machine status
      ! word (LMSW, also known as Control Register CR0), whose bit 0 is set to cause the CPU to
      ! switch to protected mode and run in privilege level 0, ie the current privilege level
      ! CPL=0. The segment register still points to the same linear address as in real-address
      ! mode (the linear address is the same as the physical memory address in real-address mode).
      ! After setting this bit, the next instruction must be an inter-segment jump instruction
      ! to flush the current instruction queue of the CPU.
      !
      ! Because the CPU reads the instruction from memory and decodes it before executing an
      ! instruction. Therefore, those pre-fetched instructions that belong to the real mode
      ! after entering the protection mode become no longer valid. An inter-segment jump
      ! instruction will flush the current instruction queue of the CPU, that is, discard these
      ! invalid information. In addition, in the Intel company's manual, it is recommended that
      ! the CPU of 80386 or above should use the instruction "mov cr0, ax" to switch to the
      ! protection mode. The lmsw instruction is only for compatibility with previous 286 CPUs.

197      mov      ax,#0x0001        ! protected mode (PE) bit
198      lmsw     ax                ! This is it!
199      jmp     0,8                ! jmp offset 0 of segment 8 (cs)
```

```
! We have moved the system module to the beginning of 0x00000, so the offset address in
! the previous sentence is 0. The value 8 is already a segment selector in protected mode,
! which is used to select the descriptor table and its entry, and the required privilege
```

! level. The segment selector 8 (0b0000, 0000, 0000, 1000) indicates that the privilege  
! level 0 is requested, and the second descriptor item in the GDT table is used. This entry  
! indicates that the base address is 0 (see line 571), so the jump instruction here will  
! execute the code in system module.

200

201 ! This routine checks that the keyboard command queue is empty  
202 ! No timeout is used - if this hangs there is something wrong with  
203 ! the machine, and we probably couldn't proceed anyway.

!

! A write cmd can be executed only when input buffer is empty (status register bit 1 = 0).

204 empty\_8042:

205 .word 0x00eb, 0x00eb  
206 in al, #0x64 ! 8042 status port  
207 test al, #2 ! is input buffer full?  
208 jnz empty\_8042 ! yes - loop  
209 ret

210

! Note that the following 215--566 lines of code involve a lot of graphics card hardware  
! information, so it is more complicated. However, since this code has little to do with  
! the kernel, you can skip it first.

211 ! Routine trying to recognize type of SVGA-board present (if any)

212 ! and if it recognize one gives the choices of resolution it offers.

213 ! If one is found the resolution chosen is given by al, ah (rows, cols).

!

! The following code first displays msg1 on line 588-589, then loops through the keyboard  
! controller output buffer, waiting for user to press the button. If user presses Enter  
! key, it checks the SVGA mode of the system and returns the maximum row and column values  
! in AL and AH. Otherwise, the default of AL=25 rows, AH=80 columns is set and returned.

214

215 chsvga: cld  
216 push ds ! Save ds, will be popped out on line 231 (or 490 or 492).  
217 push cs ! ds = cs  
218 pop ds  
219 mov ax, #0xc000  
220 mov es, ax ! es points to 0xc000 seg. it's BIOS area on VGA card.  
221 lea si, msg1 ! ds:si points to null terminated message msg1.  
222 call prtstr ! displays msg1.

! First of all, please note that the scan code generated when a button is pressed is called  
! make code. The scan code generated by releasing a pressed button is called break code.  
! The following code read keyboard controller output buffer to get scan code or command.  
! If the received scan code is smaller than 0x82, then it's a make code, because 0x82 is  
! the minimum value of break code, and less than 0x82 means that the button has not been  
! released. If the scan code is greater than 0xe0, it indicates that the extended scan  
! code prefix is received. If the break code is 0x9c, it means user pressed/released the  
! Enter key. The program then jumps to check if the system has SVGA mode. Otherwise, the  
! return row and column are set to AL=25 rows and AH=80 columns by default.

223 nokey: in al, #0x60 ! read in scan code from the controller buffer.  
224 cmp al, #0x82 ! compare with minimum break code 0x82  
225 jb nokey ! if less than it, no key is released yet.  
226 cmp al, #0xe0  
227 ja nokey ! if great than 0xe0, it's a prefix of code.  
228 cmp al, #0x9c

```

229     je      svga          ! if break code is 0x9c, enter key is pressed/released.
230     mov     ax,#0x5019    ! otherwise set al = 25, ah = 80
231     pop     ds
232     ret

```

! The following is based on the feature data string at the specified location in the ROM  
! BIOS on the VGA card or the features supported to determine what brand of display card  
! is installed on machine. The program supports a total of 10 display card extensions.  
! Note that on line 220 the program has pointed es to the BIOS segment 0xc000 on the VGA  
! card (see Chapter 2).

! First let's check if the display card is an ATI adapter.  
! We point ds:si to the ATI card feature data string on line 595, and let es:si point to  
! the specified location (offset 0x31) in the VGA BIOS. The feature string has a total of  
! 9 characters ("761295520"), and we loop through the feature string. If they are the same,  
! the VGA card in the machine is ATI brand. So let ds:si point to the row and column mode  
! value dscati (line 615) that the display card can set, let di point to the number modes  
! that can be set, and jump to the label selmod (438 lines) for further settings.

```

233 svga:  lea     si,idati      ! Check ATI 'clues'
234        mov     di,#0x31      ! the feature data is at 0xc000:0x0031
235        mov     cx,#0x09      ! 9 bytes
236        repe
237        cmpsb                ! If 9 bytes are the same, means we have an ATI card.
238        jne     noati

```

! Ok, we know the adapter's brand. Now let si points to the ATI display card optional  
! row value table (dscati), di points to the extended mode number and extended mode number  
! list (moati), then jumps to selmod (438 lines) to continue processing.

```

239        lea     si,dscati
240        lea     di,moati
241        lea     cx,selmod
242        jmp     cx

```

! Now let's test to see if it is the Ahead brand's display card.  
! First, write the main enable register index 0x0f to be accessed to the EGA/VGA pattern  
! index register 0x3ce, and write the open extension register flag value 0x20 to the 0x3cf  
! port (in this case, corresponding to the main enable register). The main enable register  
! value is then read through the 0x3cf port to check if the enable extended register flag  
! can be set. If it can, it is a Ahead brand card. Note that when the word is output,  
! al -> port n, ah -> port n+1.

```

243 noati:  mov     ax,#0x200f    ! Check Ahead 'clues'
244        mov     dx,#0x3ce      ! data port 0x0f -> 0x3ce port
245        out     dx,ax          ! set extend reg flag: 0x20 -> 0x3cf port
246        inc     dx             ! then check the flag to see if it has been set.
247        in      al,dx
248        cmp     al,#0x20       ! if it's 0x20, an Ahead A adapter found.
249        je      isahed         ! if it's 0x20, its an Ahead B adapter.
250        cmp     al,#0x21       ! if not a Ahead adapter, jump.
251        jne     noahed

```

! Ok, we know the adapter's brand. Now let si points to the Ahead display card optional  
! row value table (dscahead), di points to the extended mode number and extended mode  
! number list (moahead), then jumps to selmod (438 lines) to continue processing.

```

252 isahed: lea     si,dscahead
253        lea     di,moahead

```

```

254      lea    cx,selmod
255      jmp    cx

! Now let's check if it is a graphics card produced by Chips & Tech.
! The VGA enable register entry mode flag (bit 4) is set via port 0x3c3 (0x94 or 0x46e8),
! and the display card chipset identification value is then read from port 0x104. If the
! id is 0xA5, it means that it is a display card produced by Chips & Tech.
256 noahed: mov    dx,#0x3c3      ! Check Chips & Tech. 'clues'
257      in     al,dx            ! read enable reg from port 0x3c3, add setup flag(bit 4).
258      or     al,#0x10
259      out    dx,al
260      mov    dx,#0x104        ! read chip id from global id port 0x104, stored in bl.
261      in     al,dx
262      mov    bl,al
263      mov    dx,#0x3c3        ! reset setup flag to port 0x3c3.
264      in     al,dx
265      and    al,#0xef
266      out    dx,al
267      cmp    bl,[idcandt]     ! compare bl and id(0xA5) in idcandt( line 596).
268      jne    nocant

! Ok, we know the adapter brand is Chips & Tech. Now let si points to the card optional
! row value table (dsccandt), di points to the extended mode number and extended mode
! number list (mocandt), then jumps to selmod (438 lines) to continue processing.
269      lea    si,dsccandt
270      lea    di,mocandt
271      lea    cx,selmod
272      jmp    cx

! Now let us check if the card is a Cirrus display card.
! The detection method is to use the contents of the CRT controller index number 0x1f
! register to try to disable the extended function. This register is called the Eagle ID
! register. The value of the high and low nibbles is exchanged and written to the 6th
! index register of port 0x3c4. This operation should disable the extended function of the
! Cirrus display card. If it is not prohibited, it means that it is not a Cirrus display
! card. Because the content read from the 0x1f eagle register indexed by port 0x3d4 is the
! value after the XOR operation of the memory start address high byte register content
! corresponding to the eagle value and the 0x0c index number. Therefore, before reading
! the contents of 0x1f, we need to save the contents of the memory start high byte register
! and then clear it, and restore it after checking. In addition, writing the escaped Eagle
! ID value to the No. 6 sequence/extension register of the 0x3c4 port index will re-enable
! the extension.
273 nocant: mov    dx,#0x3d4      ! Check Cirrus 'clues'
274      mov    al,#0x0c          ! write reg index 0x0c to port 0x3d4 to get mem addr.
275      out    dx,al            !
276      inc    dx                ! read high byte of mem addr from port 0x3d5 to bl.
277      in     al,dx
278      mov    bl,al
279      xor    al,al
280      out    dx,al
281      dec    dx                ! write reg index 0x1f to port 0x3d4 to get Eagle ID.
282      mov    al,#0x1f
283      out    dx,al
284      inc    dx

```

```
285      in      al,dx          ! get Eagle ID from port 0x3d5, and store to bh.
286      mov     bh,al          ! swap nibbles and store to cl. left shift to ch.
287      xor     ah,ah          ! then put number 6 to cl.
288      shl     al,#4
289      mov     cx,ax
290      mov     al,bh
291      shr     al,#4
292      add     cx,ax
293      shl     cx,#8
294      add     cx,#6
! Finally, the cx value is stored in ax. At this time, ah is the "Eagle ID" value after
! transposition, and al is index number 6, which corresponds to the sequencing/extension
! register. Writing the ah to the 0x3c4 port indexing sequence/extension register should
! cause the Cirrus graphics card to disable extensions.
295      mov     ax,cx
296      mov     dx,#0x3c4
297      out     dx,ax
298      inc     dx
! If the extension is really disabled, then the value read in should be 0. If not, it
! means that it is not a Cirrus display card.
299      in      al,dx
300      and     al,al
301      jnz     nocirr
! Execution to this point indicates that the card in the machine may be a Cirrus display
! card. Then use the original value of "Eagle ID" saved in bh (line 286) to re-enable the
! Cirrus card extension function. The return value read should be 1. If not, it is still
! not a Cirrus display card.
302      mov     al,bh          !
303      out     dx,al          !
304      in      al,dx          !
305      cmp     al,#0x01
306      jne     nocirr
! Ok, now we know that the graphics card is a Cirrus brand. So first call the rst3d4
! subroutine to restore the CRT controller's display start address high byte register
! contents, then let si point to the brand display card's optional row value table
! (dsccirrus), di points to the extended mode number and the extended mode number list
! (mocirrus), then jump to selmod (line 438) to continue setting operation.
307      call    rst3d4
308      lea     si,dsccirrus
309      lea     di,mocirrus
310      lea     cx,selmod
311      jmp     cx
! This subroutine restores the display start address high byte register contents of the
! CRT controller using the value stored in bl (line 278).
312 rst3d4: mov     dx,#0x3d4
313      mov     al,bl
314      xor     ah,ah
315      shl     ax,#8
316      add     ax,#0x0c
317      out     dx,ax          ! note, word output, al -> 0x3d4, ah -> 0x3d5.
318      ret
```

! Now check if the Everex graphics card is in the system. The method is to call Everex's

```

! extended video BIOS function with interrupt 0x10 function 0x70 (ax =0x7000, bx=0x0000).
! For an Everex type display card, the interrupt call should return to the simulation
! state, ie the following return information:
! al = 0x70, if it is a Trident-based Everex display card;
! cl = type: 00-mono; 01-CGA; 02-EGA; 03-digital multi-freq; 04-PS/2; 05-IBM 8514; 06-SVGA.
! ch = attr: Bit7-6 :00-256K, 01-512K, 10-1MB, 11-2MB; Bit4-Enable VGA protect; Bit0-6845Simu.
! dx = board model: Bit 15-4: board type id; bit 3-0: board correction id.
!      0x2360-Ultragraphics II; 0x6200-Vision VGA; 0x6730-EVGA; 0x6780-Viewpoint.
! di = The video BIOS version number represented in BCD code.
319 nocirr: call    rst3d4          ! Check Everex 'clues'
320          mov     ax, #0x7000    ! int 0x10 with ax = 0x7000, bx=0x0000.
321          xor     bx, bx
322          int     0x10
323          cmp     al, #0x70      ! al should contain 0x70 for Everex card.
324          jne     noevrx
325          shr     dx, #4         ! ignore board fix number(bit3-0).
326          cmp     dx, #0x678     ! if board type is 0x678, its a Trident card.
327          je      istrid
328          cmp     dx, #0x236     ! if board type is 0x236, also a Trident card.
329          je      istrid
! Ok, now we know that the card is a Everex brand. So first we let si point to the card's
! optional row value table (dsceverex), di points to the extended mode number and the
! extended mode number list (moeverex), then jump to selmod (line 438) to continue setting .
330          lea     si, dsceverex
331          lea     di, moeverex
332          lea     cx, selmod
333          jmp     cx
334 istrid: lea     cx, ev2tri      ! Everex card with a Trident type, jump to ev2tri
335          jmp     cx

! Now check if it is a Genoa display card. The way is to check the feature number string
! (0x77, 0x00, 0x66, 0x99) in its video BIOS. Note that at this time es has been set to
! the segment 0xc000 where the ROM BIOS is located on the VGA card.
336 noevrx: lea     si, idgenoa    ! Check Genoa 'clues'
337          xor     ax, ax        ! ds:si points to feature data.
338          seg es
339          mov     al, [0x37]     ! get feature data from VGA card at 0x37.
340          mov     di, ax        ! es:di point to 0x37.
341          mov     cx, #0x04
342          dec     si
343          dec     di
344 l1:      inc     si             ! compare the 4 feature bytes.
345          inc     di
346          mov     al, (si)
347          seg es
348          and     al, (di)
349          cmp     al, (si)
350          loope   l1
351          cmp     cx, #0x00
352          jne     nogen
! Ok, now we know that the card is a Genoa card. So we let si point to the card's
! optional row value table (dscgenoa), di points to the extended mode number and the
! extended modes list (mogenoa), then jump to selmod (line 438) to continue setting .

```

```
353     lea     si,dscgenoa
354     lea     di,mogenoa
355     lea     cx,selmod
356     jmp     cx
```

! Now check if it is a Paradise display card. The same is true for comparing the feature  
! strings ("VGA=") in the BIOS on the display card.

```
357 nogen: lea     si,idparadise    ! Check Paradise 'clues'
358       mov     di,#0x7d          ! es:di point to 0xc000:0x007d.
359       mov     cx,#0x04          ! there should be 4 bytes: "VGA="
360       repe
361       cmpsb
362       jne     nopara
```

! Ok, we know the card is a Paradise card. So we let si point to the card's optional  
! row value table (dscparadise), di points to the extended mode number and the extended  
! modes list (moparadise), then jump to selmod (line 438) to continue setting.

```
363     lea     si,dscparadise
364     lea     di,moparadise
365     lea     cx,selmod
366     jmp     cx
```

! Now check if it is a Trident (TVGA) card. Bits 3--0 of the TVGA display card expansion  
! mode control register 1 (0x0e of the 0x3c4 port index) are 64K memory page values. This  
! field value has a property: when writing, we need to first XOR the value with 0x02 and  
! then write; when reading the value, no XOR operation is required. That is, the value  
! before XOR should be the same as the value read after writing. The following code uses  
! this feature to check if it is a Trident display card.

```
367 nopara: mov     dx,#0x3c4        ! Check Trident 'clues'
368       mov     al,#0x0e          ! output index 0x0e (mode ctrl reg 1) to port 0x3c4
369       out     dx,al             ! read original value from port 0x3c5 and store to al
370       inc     dx
371       in      al,dx
372       xchg    ah,al
```

! Then we write 0x00 to this register and read its value ->al. Writing 0x00 is equivalent  
! to the value written after "original value" 0x02 or 0x02, so if it is a Trident card,  
! the value read after this should be 0x02. After swapping, a = the value of the original  
! mode control register 1, ah = the last read value.

```
373     mov     al,#0x00
374     out     dx,al
375     in      al,dx
376     xchg    al,ah
377     mov     bl,al               ! Strange thing ... in the book this wasn't
378     and     bl,#0x02           ! necessary but it worked on my card which
379     jz      setb2              ! is a trident. Without it the screen goes
380     and     al,#0xfd           ! blurred ...
381     jmp     clrb2              !
382 setb2: or     al,#0x02          !
383 clrb2: out     dx,al
384     and     ah,#0x0f           ! get page number field (bit 3-0) (line 375)
385     cmp     ah,#0x02           ! if equal 0x02, it's a Trident card.
386     jne     notrid
```

! Ok, we know the card is a Trident card. So we let si point to the card's optional



! row value table (dsctrident), di points to the extended mode number and the extended  
! modes list (motrident), then jump to selmod (line 438) to continue setting.

```
387 ev2tri: lea    si,dsctrident
388         lea    di,motrident
389         lea    cx,selmod
390         jmp    cx
```

! Now check if it is a Tseng card (ET4000AX or ET4000/W32). The method is to perform read  
! and write operations on the Segment Select register corresponding to the 0x3cd port. The  
! upper 4 bits (bits 7--4) of the register are the 64KB segment number (Bank number) to be  
! read, and the lower 4 bits (bits 3--0) are the segment numbers specified for writing. If  
! the value of the specified segment select register is 0x55 (indicating reading and  
! writing the sixth 64KB segment), then for the Tseng display card, writing the value to  
! the register should be still 0x55.

```
391 notrid: mov    dx,#0x3cd      ! Check Tseng 'clues'
392         in     al,dx          ! Could things be this simple ! :-)
393         mov    bl,al          ! read original seg selector data from 0x3cd to bl.
394         mov    al,#0x55       ! write to it with value 0x55, and read again to ah.
395         out    dx,al
396         in     al,dx
397         mov    ah,al
398         mov    al,bl          ! restore original data.
399         out    dx,al
400         cmp    ah,#0x55       ! if read value equal to write, it's a Tseng card.
401         jne    notsen
```

! Ok, we know the card is a Tseng card. So we let si point to the card's optional  
! row value table (dsctseng), di points to the extended mode number and the extended  
! modes list (motseng), then jump to selmod (line 438) to continue setting.

```
402         lea    si,dsctseng
403         lea    di,motseng
404         lea    cx,selmod
405         jmp    cx
```

! Check if it is a Video7 display card. Port 0x3c2 is the mixed output register write port  
! and 0x3cc is the mixed output register read port. Bit 0 of this register is a mono/color  
! flag. If it is 0, it means mono, otherwise it is color. The way to determine whether the  
! Video7 card is used is to use the CRT control extension identification register (index  
! number is 0x1f). The value of this register is actually the result of the XOR operation  
! of the memory address high byte register (index number 0x0c) and value 0xea. Therefore,  
! we only need to write a specific value to the memory start address high byte register,  
! and then read the identification value from the identification register to check.

```
406 notsen: mov    dx,#0x3cc      ! Check Video7 'clues'
407         in     al,dx
408         mov    dx,#0x3b4       ! set dx to mono control index register port 0x3b4.
409         and    al,#0x01        ! If bit0 of mixed output reg is 0(mono), jump directly.
410         jz     even7           ! Otherwise set dx to color control index reg port 0x3d4.
411         mov    dx,#0x3d4
412 even7:  mov    al,#0x0c         ! Set index to 0x0c for the mem address high byte reg.
413         out    dx,al
414         inc    dx
415         in     al,dx            ! read high byte reg of vmem address , save to bl.
416         mov    bl,al
417         mov    al,#0x55        ! then write 0x55 to high byte reg and read it out.
```

```

418      out    dx,al
419      in     al,dx
! Then select the Video7 display card identification register whose index number is 0x1f
! through the CRT index register port 0x3b4 or 0x3d4. The contents of this register are
! actually the result of the XOR of the memory start address and the value 0xea.
420      dec    dx
421      mov    al,#0x1f
422      out    dx,al
423      inc    dx
424      in     al,dx          ! read Video7 card id register value and save it in bh.
425      mov    bh,al
426      dec    dx          ! select addr high byte reg to restore its original value.
427      mov    al,#0x0c
428      out    dx,al
429      inc    dx
430      mov    al,bh
431      out    dx,al
! Then we will verify that the "Video7 display card identification register value is the
! result value of the memory memory start address high byte and 0xea after XOR operation".
! Therefore, the result of the XOR operation of 0x55 and 0xea should be equal to the test
! value of the identification register. If it is not a Video7 card, set the default display
! row and column value (492 lines). Otherwise it is the Video7 card. So let si point to
! the display card row value table (dscvideo7), let di point to the number of extended
! mode and mode number list (movideo7).
432      mov    al,#0x55
433      xor    al,#0xea
434      cmp    al,bh
435      jne    novid7
436      lea    si,dscvideo7
437      lea    di,movideo7

```

! Through the inspection and analysis of the above display card and the Video7 display card here, we can see that the inspection process is usually divided into three basic steps. The first is to read and save the original value of the register that is needed for the test, then use the specific test value for the write and read operations, and finally restore the original register value and make a judgment on the check result.

!

! The following is based on the above-mentioned code to determine the type of the display card and the related extended mode information (list of row and column values pointed to by si; di points to the number of extended modes and the list of mode numbers), prompting the user to select an available display mode and setting it to display mode accordingly. Finally, the subroutine returns the screen row and column values currently set by the system (ah = columns; al = rows). For example, if the system is an ATI graphics card, the following message will appear on the screen:

! Mode: COLSxROWS:

! 0. 132 x 25

! 1. 132 x 44

! Choose mode by pressing the corresponding number.

!

! The following code will display the null-terminated string "Mode: COLSxROWS:" on the screen.

```

438 selmod: push    si
439          lea     si,msg2
440          call    prtstr

```

```

441      xor     cx,cx
442      mov     cl,(di)          ! cl is the extended modes of the checked card.
443      pop     si
444      push    si
445      push    cx
! Then, the extended mode rows and columns selectable by the current display card are
! displayed for user to select.
446 tbl:  pop     bx              ! bx = total extend modes number.
447      push    bx
448      mov     al,bl
449      sub     al,cl
450      call    dprnt            ! display the value in decimal format.
451      call    spcing           ! a dot, and tehn 4 spaces.
452      lodsw                     ! load row & column pointed to by si in ax, then si++.
453      xchg    al,ah            ! swap, al = columns.
454      call    dprnt            ! display column number.
455      xchg    ah,al            ! al = rows.
456      push    ax
457      mov     al,#0x78         ! show "x"
458      call    prnt1
459      pop     ax               ! al= row number
460      call    dprnt            ! display row number.
461      call    docr             ! cr,lf
462      loop    tbl              ! display next row colums, mode number decreased by 1.
! Then display prompt string "Choose mode by pressing the corresponding number."
463      pop     cx               ! cl = total extend modes number.
464      call    docr
465      lea     si,msg3           ! "Choose mode by pressing the corresponding number."
466      call    prtstr

```

! Then, the scan code of the user button is read from the keyboard port, the row and  
! column mode number selected by the user is determined according to the scan code, and  
! the corresponding display mode is set by using the ROM BIOS INT 0x10 function 0x00.  
! The "mode number + 0x80" on line 468 is the break scan codes of the number key -1  
! pressed. For the 0--9 number keys, their break codes are:

```

!      0 - 0x8B; 1 - 0x82; 2 - 0x83; 3 - 0x84; 4 - 0x85;
!      5 - 0x86; 6 - 0x87; 7 - 0x88; 8 - 0x89; 9 - 0x8A

```

! Therefore, if the read break code is less than 0x82, it means that it's not a numeric  
! key; if the scan code is equal to 0x8B, it means that the user pressed number 0 key.

```

467      pop     si              ! pop up original row & column table pointer.
468      add     cl,#0x80         ! cl + 0x80 = the break code for the "number key -1".
469 nonum: in     al,#0x60         ! Quick and dirty...
470      cmp     al,#0x82         ! less than 0x82 ? ignore it.
471      jb      nonum
472      cmp     al,#0x8b         ! scan code = 0x8b? it's number key 0.
473      je      zero
474      cmp     al,cl            ! great than the number of modes?
475      ja      nonum           ! non number key pressed.
476      jmp     nozero

```

! Next, the break scan code is converted into a corresponding digital key value, and then  
! the corresponding mode number is selected from the mode number and the mode number list  
! by using the value. Then call the ROM BIOS interrupt INT 0x10 function 0 to set the

! screen to the mode specified by the mode number. Finally, use the mode number to select  
! from the display card row and column table and return the corresponding row and column  
! values in ax.

```

477 zero:  sub    al,#0x0a      ! al = 0x8b - 0x0a = 0x81
478 nozero: sub    al,#0x80     ! subtract 0x80 to obtain the mode selected by user.
479         dec    al           ! count from 0
480         xor    ah,ah        ! set display mode
481         add    di,ax
482         inc    di           ! di points to the mode number (skip the first).
483         push   ax
484         mov    al,(di)       ! mode number -> al, call int to set mode.
485         int    0x10
486         pop    ax
487         shl    ax,#1        ! mode nr x 2: pointer into the row & column table.
488         add    si,ax
489         lodsw
490         pop    ds           ! restore ds saved on line 216. return value in ax.
491         ret

```

! If none of the graphics cards tested above, then we have to use the default 80 x 25  
! standard row and column values.

```

492 novid7: pop    ds           ! Here could be code to support standard 80x50,80x30
493         mov    ax,#0x5019
494         ret

```

495

496 ! Routine that 'tabs' to next col.

497

! display a dot '.' and four spaces

```

498 spcing: mov    al,#0x2e      ! a dot '.'
499         call   prntl
500         mov    al,#0x20
501         call   prntl
502         mov    al,#0x20
503         call   prntl
504         mov    al,#0x20
505         call   prntl
506         mov    al,#0x20
507         call   prntl
508         ret

```

509

510 ! Routine to print asciiz-string at DS:SI

511

```

512 prtstr: lodsb
513         and    al,al
514         jz     fin
515         call   prntl         ! print a char in al
516         jmp    prtstr
517 fin:    ret

```

518

519 ! Routine to print a decimal value on screen, the value to be

520 ! printed is put in al (i.e 0-255).

521

```

522 dprnt:  push   ax

```

```
523      push    cx
524      mov     ah, #0x00
525      mov     cl, #0x0a
526      idiv    cl
527      cmp     al, #0x09
528      jbe     lt100
529      call    dprnt
530      jmp     skip10
531 lt100: add     al, #0x30
532      call    prnt1
533 skip10: mov     al, ah
534      add     al, #0x30
535      call    prnt1
536      pop     cx
537      pop     ax
538      ret
539
540 ! Part of above routine, this one just prints ascii al
541 ! This subroutine uses the interrupt 0x10 function 0x0E to write a character on the screen
542 ! by telex. The cursor will automatically move to the next position. If a line is written,
543 ! the cursor will move to the beginning of the next line. If the last line of a screen has
544 ! been written, the entire screen will scroll up one line. The characters 0x07 (BEL), 0x08 (BS),
545 ! 0x0A (LF), and 0x0D (CR) are not displayed as commands.
546 ! Input: AL -- character; BH -- page number; BL -- foreground color (in graphic mode).
547
548 prnt1: push    ax
549      push    cx
550      mov     bh, #0x00      ! page number.
551      mov     cx, #0x01
552      mov     ah, #0x0e
553      int     0x10
554      pop     cx
555      pop     ax
556      ret
557
558 ! Prints <CR> + <LF>
559
560 docr:  push    ax
561      push    cx
562      mov     bh, #0x00
563      mov     ah, #0x0e
564      mov     al, #0x0a
565      mov     cx, #0x01
566      int     0x10
567      mov     al, #0x0d
568      int     0x10
569      pop     cx
570      pop     ax
571      ret
```

! Start here is the global descriptor table GDT. It consists of multiple 8-byte long  
! descriptor entries. Three descriptor items are given here. The first entry is useless

```

! (568 lines), but it must exist. The second item is the system code segment descriptor
! (lines 570-573), and the third is the system data segment descriptor (lines 575-578).
567 gdt:
568     .word    0,0,0,0          ! dummy
569
! The offset here in GDT is 0x08. It happens to be the value of kernel code selector.
570     .word    0x07FF          ! 8Mb - limit=2047 (0--2047, so 2048*4096=8Mb)
571     .word    0x0000          ! base address=0
572     .word    0x9A00          ! code read/exec
573     .word    0x00C0          ! granularity=4096, 386
574
! The offset here in GDT is 0x10. It happens to be the value of kernel data selector.
575     .word    0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
576     .word    0x0000          ! base address=0
577     .word    0x9200          ! data read/write
578     .word    0x00C0          ! granularity=4096, 386
579
! Below is the 6-byte operand required by the instruction lidt that loads the interrupt
! descriptor table register. The first 2 bytes are the limit of the IDT table, and the
! last 4 bytes are the 32-bit base address of the idt table in the linear address space.
! The CPU requires that the IDT table be set before entering the protection mode, so here
! an empty table of length 0 is set first.
580 idt_48:
581     .word    0              ! idt limit=0
582     .word    0,0           ! idt base=0L
583
! This is the 6-byte operand required by the instruction lgdt to load the global
! descriptor table register. The first 2 bytes are the limit length of the gdt table, and
! the last 4 bytes are the linear base address of the gdt table. The global table size is
! set to 2KB (0x7ff). Since every 8 bytes constitutes a segment descriptor item, there are
! a total of 256 entries in the table.
! The 4-byte linear base address is 0x0009<<16 + 0x0200 + gdt, which is 0x90200 + gdt.
! (The symbol gdt is the offset address of the global table in this block, see line 205)
584 gdt_48:
585     .word    0x800          ! gdt limit=2048, 256 GDT entries
586     .word    512+gdt,0x9    ! gdt base = 0X9xxxx
587
588 msg1:  .ascii  "Press <RETURN> to see SVGA-modes available or any other key to continue."
589         db      0x0d, 0x0a, 0x0a, 0x00
590 msg2:  .ascii  "Mode: COLSxROWS:"
591         db      0x0d, 0x0a, 0x0a, 0x00
592 msg3:  .ascii  "Choose mode by pressing the corresponding number."
593         db      0x0d, 0x0a, 0x00
594
! Below are the feature data strings for the four display cards.
595 idati:  .ascii  "761295520"
596 idcandt: .byte  0xa5          ! idcandt means "ID of Chip AND Tech."
597 idgenoa: .byte  0x77, 0x00, 0x66, 0x99
598 idparadise: .ascii "VGA="
599
! The following is a list of the number of extended modes and corresponding mode numbers
! that can be used by various display cards. The first byte of each line is the number of

```

```
! modes, and the subsequent values are the mode numbers that can be used by interrupt 0x10
! function 0 (AH=0). For example, from line 602, for the ATI brand card, two extended
! modes can be used in addition to the standard mode: 0x23 and 0x33.
```

```
600 ! Manufacturer:   Numofmodes:   Mode:
```

```
601
```

```
602 moati:           .byte    0x02,    0x23, 0x33
```

```
603 moahead:         .byte    0x05,    0x22, 0x23, 0x24, 0x2f, 0x34
```

```
604 mocandt:         .byte    0x02,    0x60, 0x61
```

```
605 mocirrus:        .byte    0x04,    0x1f, 0x20, 0x22, 0x31
```

```
606 moeverex:        .byte    0x0a,    0x03, 0x04, 0x07, 0x08, 0x0a, 0x0b, 0x16, 0x18, 0x21, 0x40
```

```
607 mogenoa:         .byte    0x0a,    0x58, 0x5a, 0x60, 0x61, 0x62, 0x63, 0x64, 0x72, 0x74, 0x78
```

```
608 moparadise:      .byte    0x02,    0x55, 0x54
```

```
609 motrident:       .byte    0x07,    0x50, 0x51, 0x52, 0x57, 0x58, 0x59, 0x5a
```

```
610 motseng:         .byte    0x05,    0x26, 0x2a, 0x23, 0x24, 0x22
```

```
611 movideo7:        .byte    0x06,    0x40, 0x43, 0x44, 0x41, 0x42, 0x45
```

```
612
```

```
! Below is a list of columns and rows for the modes that can be used with various brands
! of VGA cards. For example, line 615 indicates that the column and row values of the
! two extension modes for ATI card are 132 x 25 and 132 x 44, respectively.
```

```
613 !               msb = Cols   lsb = Rows:
```

```
614
```

```
615 dscati:           .word    0x8419, 0x842c
```

```
616 dscahead:        .word    0x842c, 0x8419, 0x841c, 0xa032, 0x5042
```

```
617 dsccandt:        .word    0x8419, 0x8432
```

```
618 dsccirrus:        .word    0x8419, 0x842c, 0x841e, 0x6425
```

```
619 dsceverex:        .word    0x5022, 0x503c, 0x642b, 0x644b, 0x8419, 0x842c, 0x501e, 0x641b, 0xa040,
0x841e
```

```
620 dscgenoa:         .word    0x5020, 0x642a, 0x8419, 0x841d, 0x8420, 0x842c, 0x843c, 0x503c, 0x5042,
0x644b
```

```
621 dscparadise:      .word    0x8419, 0x842b
```

```
622 dsctrident:       .word    0x501e, 0x502b, 0x503c, 0x8419, 0x841e, 0x842b, 0x843c
```

```
623 dsctseng:         .word    0x503c, 0x6428, 0x8419, 0x841c, 0x842c
```

```
624 dscevideo7:       .word    0x502b, 0x503c, 0x643c, 0x8419, 0x842c, 0x841c
```

```
625
```

```
626 .text
```

```
627 endtext:
```

```
628 .data
```

```
629 enddata:
```

```
630 .bss
```

```
631 endbss:
```

---

## 6.3.3 Reference information

In order to get the basic parameters of the machine and display messages of the boot process to user, this program calls interrupt services in the BIOS multiple times and starts to involve some access operations to the hardware ports. The following briefly describes several BIOS interrupt services used and explains the cause of the A20 address line problem. Finally we also mentioned the issues of the 80X86 CPU 32-bit protection mode operation.

### 6.3.3.1 Current memory image

After the execution of the setup.s program, the system module is moved to the beginning of the physical memory at address 0x00000, and from the location 0x90000, some basic system parameters that the kernel will use are stored, as shown in Figure 6-6.

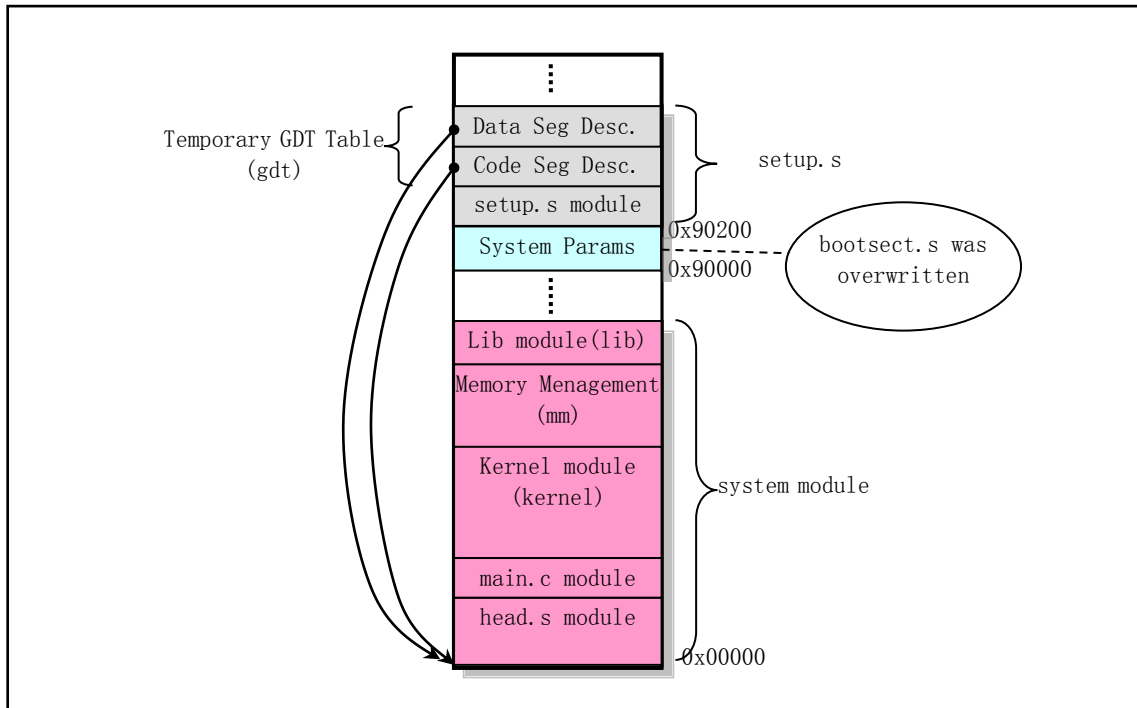


Figure 6-6 Diagram of memory map after setup.s ends

At this point, there are three descriptors in the temporary global table GDT. The first one is NULL not used, the other two are code and data segment descriptors. They all point to the beginning of the system module, which is the physical memory address 0x00000. Thus, when the last instruction 'jmp 0,8' (line 193) is executed, it will jump to the beginning of the head.s program to continue execution. The '8' in this instruction is the value of the segment selector, which is used to specify the descriptor item to be used. This is the code segment descriptor in the GDT. '0' is the offset in the code segment specified by the descriptor.

### 6.3.3.2 BIOS Video Interrupt 0x10

This section describes the ROM BIOS video interrupt service function used in the above program. See Table 6-3 for a description of the function of obtaining display card information (other auxiliary function selection). Other display service functions are given in the program comments.

Table 6-3 Obtain display card information (function: ah = 0x12, bl = 0x10)

Input/Return	Register	Description
Input Information	ah	Function No. = 0x12, Obtain display card information.
	bl	Sub-Function No. = 0x10
Return Information	bh	Video Status: 0x00 - Color mode (the video hardware I/O port base address is 0x3DX); 0x01 - Mono mode (the video hardware I/O port base address is 0x3BX); (where the X value in the port address can be 0 -- F)
	bl	Installed video memory size: 00 = 64K, 01 = 128K, 02 = 192K, 03 = 256K
	ch	Feature connector bit information: Bits 0-1      Feature line 1-0, Status 2; Bits 2-3      Feature line 1-0, Status 1;



		Bits 4-7 Not used ( set to 0)
	cl	Video switch settings: Bits 0-3 correspond to switches 1-4. Bits 4-7 is not used. Original EGA/VGA switch settings: 0x00 MDA/HGC;                      0x01-0x03 MDA/HGC; 0x04 CGA 40x25;                      0x05 CGA 80x25; 0x06 EGA + 40x25;                      0x07-0x09 EGA + 80x25; 0x0A EGA + 80x25Mono;                      0x0B EGA + 80x25Mono.

### 6.3.3.3 Hard Drive Basic Parameter Table ("INT 0x41")

In the ROM BIOS interrupt vector table, the interrupt vector location of INT 0x41 (4 \* 0x41 = 0x0000:0x0104) stores not the address of the interrupt program, but the address of the basic parameter table of the first hard disk. For the BIOS of an IBM PC fully compatible machine, the address stored here is F000h:E401h. The basic parameter table address for the second hard disk is stored at the INT 0x46 interrupt vector location.

Table 6-4 Hard disk basic parameter table

Offset	Size	Name	Description
0x00	word	cyl	Number of cylinders
0x02	byte	head	Number of heads
0x03	word		Start cylinder to reducing the write current (only for PC/XT, others are 0)
0x05	word	wpcom	Pre-compensation cylinder number before start writing (multiplied by 4)
0x07	byte		Maximum ECC burst size (only for PC/XT, others are 0)
0x08	byte	ctl	Control byte (driver step selection): Bit 0 - Not used (0);                      Bit 1 - Reserved (0) (Close IRQ) Bit 2 - Allow reset;                      Bit 3 - Set if number of heads great than 8 Bit 4 - Not used (0);                      Bit 5 - Set if there is bad map at cylinder number +1 Bit 6 - Disable ECC retry;                      Bit 7 - Disable Access retry.
0x09	byte		Standard timeout value (only for PC/XT, others are 0)
0x0A	byte		Format timeout value (only for PC/XT, others are 0)
0x0B	byte		Detect drive timeout value (only for PC/XT, others are 0)
0x0C	word	lzone	Head landing (stop) cylinder number
0x0E	byte	sect	Number of sectors per track
0x0F	byte		Reserved.

### 6.3.3.4 A20 address line problem

In August 1981, IBM's original personal computer IBM PC used a 16-bit Intel 8088 CPU. The CPU has a 16-bit internal (8-bit external) data bus and a 20-bit address bus width. Therefore, there are only 20 address lines (A0 – A19) in this PC, and the CPU can address only up to 1MB of memory range. At the time when the popular machine memory capacity was only a few tens of KB and several hundred KB, 20 address lines were enough to address the memory. The highest address that it can address is 0xffff:0xffff, which is 0x10ffef. For memory addresses that exceed 0x100000 (1MB), the CPU will default wrap around to the 0x0ffef position.

When IBM introduced the new PC/AT model in 1985, it used the Intel 80286 CPU. It has 24 address lines,

can address up to 16MB of memory, and has a real-mode of operation that is fully compatible with the 8088. However, when the addressing value exceeds 1MB, it cannot implement addressing surround like the 8088 CPU. But at the time there were programs that were designed to work with this address wrapping mechanism. Therefore, in order to achieve full compatibility with the original PC, IBM invented the use of a switch to enable or disable the 0x100000 address bit. Since there was just a free port pin (output port P2, pin P21) on the keyboard controller 8042 at the time, this pin was used as an AND gate to control this address bit. This signal is called A20. If it is zero, then bits 20 and above are cleared, thus achieved the compatibility of memory addressing. IBM's 80X86-based machines since then have also inherited this feature. For details on the keyboard controller 8042 chip, see the description after the kernel/chr\_drv/keyboard.S program.

For compatibility, the A20 address line is disabled by default when the machine is booted, so the operating system of the 32-bit machine must use the appropriate method to enable it. However, due to the different chipsets used by various compatible machines, it is very troublesome to do this. Therefore, it is usually necessary to choose among several control methods.

A common method of controlling the A20 signal line is to set the port value of the keyboard controller. The typical control method is used by the setup.s program (lines 138-144). For other compatible microcomputers, other methods can be used to control the A20 line. Some operating systems use the A20's enable and disable as part of the standard process of converting between real mode and protected mode of operation. Since the controller of the keyboard is very slow, it is not possible to operate the A20 line using the keyboard controller. To this end, an A20 fast door option (Fast Gate A20) was introduced. It uses I/O port 0x92 to handle the A20 signal line, eliminating the need for slow keyboard controller operation. For systems without a keyboard controller, only the 0x92 port can be used for control. However, the port may also be used by devices on other compatible microcomputers (such as display chips), resulting in system error operation. Another way is to open the A20 signal line by reading the 0xee port, and writing the port will disable the A20 signal line.

### **6.3.3.5 Programming method of 8259A interrupt controller**

In Chapter 2 we have outlined the basic workings of the interrupt mechanism and the hardware interrupt subsystem used in PC/AT compatible computers. Here we first introduce the working principle of the 8259A chip, and then explain in detail the programming method of the 8259A chip and how the Linux kernel works.

#### **1. 8059A chip working principle**

As mentioned earlier, a cascade of two 8259A programmable controller (PIC) chips is used in the PC/AT series compatible machines to manage a total of 15 interrupt vectors, as shown in Figure 2-20. It is connected from the INT pin of slave chip to the IR2 pin of the master chip. The port base address of the master 8259A is 0x20, and the slave chip is 0xA0. The logic block diagram of an 8259A chip is shown in Figure 6-7.

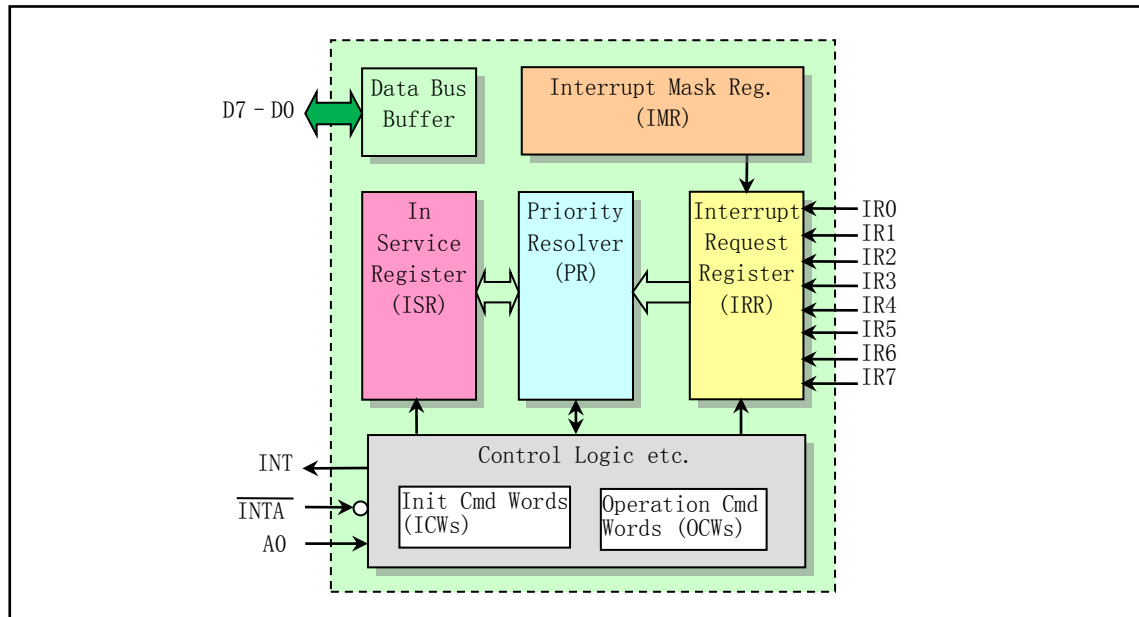


Figure 6-7 Programmable Interrupt Controller 8259A Chip Diagram

In the figure, the Interrupt Request Register (IRR) is used to store all the requested service interrupt levels on the interrupt request input pin. The 8 bits (D7-D0) of the register correspond to the pins IR7-IR0. The Interrupt Mask Register (IMR) is used to store the bits corresponding to the masked interrupt request line. The 8 bits of the register also correspond to 8 interrupt levels. Which bit is set to 1 masks which level of interrupt request. That is, the IMR processes the IRR, each bit of which corresponds to each request bit of the IRR. Masking high priority input lines does not affect the input of low priority interrupt request lines. The priority resolver (PR) is used to determine the priority of the bits set in the IRR, and the highest priority interrupt request is strobed into the in-service register (ISR). The ISR holds an interrupt request that is receiving service. The register set in the control logic block is used to accept two types of commands generated by the CPU. Before the 8259A can operate normally, the contents of the Initialization Command Word (ICW) registers must be set first. In the course of its work, you can use the Operation Command Words (OCW) registers to set and manage the 8259A's working mode at any time. The A0 line is used to select the register for the operation. In the PC/AT microcomputer system, when the A0 line is 0, the port address of the chip is 0x20 (master chip) and 0xA0 (slave chip), and when A0=1, the port is 0x21 and 0xA1.

The interrupt request lines from each device are connected to the IR0-IR7 interrupt request pin of the 8259A. When one or more interrupt request signals arrive on these pins, the corresponding bit in the interrupt request register IRR is set and latched. At this time, if the corresponding bit in the interrupt mask register IMR is set, the corresponding interrupt request will not be sent to the priority parser. After an unmasked interrupt request is sent to the priority resolver, the highest priority interrupt request is selected. At this point, the 8259A will send an INT signal to the CPU, and the CPU will return an INTA to the 8259A to respond to the interrupt signal after executing the current instruction. After receiving the response signal, the 8259A saves the selected highest priority interrupt request to the service register ISR, that is, the bit corresponding to the interrupt request level in the ISR is set. At the same time, the corresponding bit in the interrupt request register IRR is reset, indicating that the interrupt request is starting to be processed.

After that, the CPU will send a second INTA pulse signal to the 8259A, which is used to inform the 8259A to send the interrupt number. Therefore, during the pulse signal, 8259A will send an 8-bit data representing the interrupt number to the data bus for reading by the CPU.

At this point, the CPU interrupt period ends. If the 8259A is using the Automatic End of Interrupt (AEOI) mode, the current service interrupt bit in the service register ISR at the end of the second INTA pulse is reset. Otherwise, if the 8259A is in the non-automatic end mode, then at the end of the interrupt service routine the program will need to send an End of Interrupt (EOI) command to the 8259A to reset the bits in the ISR. If the interrupt request comes from the second 8259A chip that is connected, then the EOI command needs to be sent to both chips. After that, 8259A will check the next highest priority interrupt and repeat the above process. Below we first give the programming method of the initialization command word and the operation command word, and then further explain some of the operation methods used therein.

## 2. Initialization command word programming

The programmable controller 8259A mainly has four working modes: (1) full nested mode; (2) rotating priority mode; (3) special mask mode and (4) program polled mode. By programming the 8259A, we can choose the current working mode of the 8259A. Programming is divided into two phases. The first is to write the programming of the 4 initialization command words (ICW1 - ICW4) register of each 8259A chip before the 8259A works; the second is to program the 8 operation command words (OCW1 - OCW3) of the 8259A at any time during the work. After initialization, the contents of the operation command word can be written to the 8259A at any time. Below we explain the programming operation of the 8259A initialization command word.

The programming operation flow of the initialization command word is shown in Figure 6-8. As can be seen from the figure, the settings for ICW1 and ICW2 are required. ICW3 needs to be set only when the system includes multiple 8259A chips and is connected. This needs to be clearly stated in the settings of ICW1. In addition, whether you need to set ICW4 also needs to be specified in ICW1.

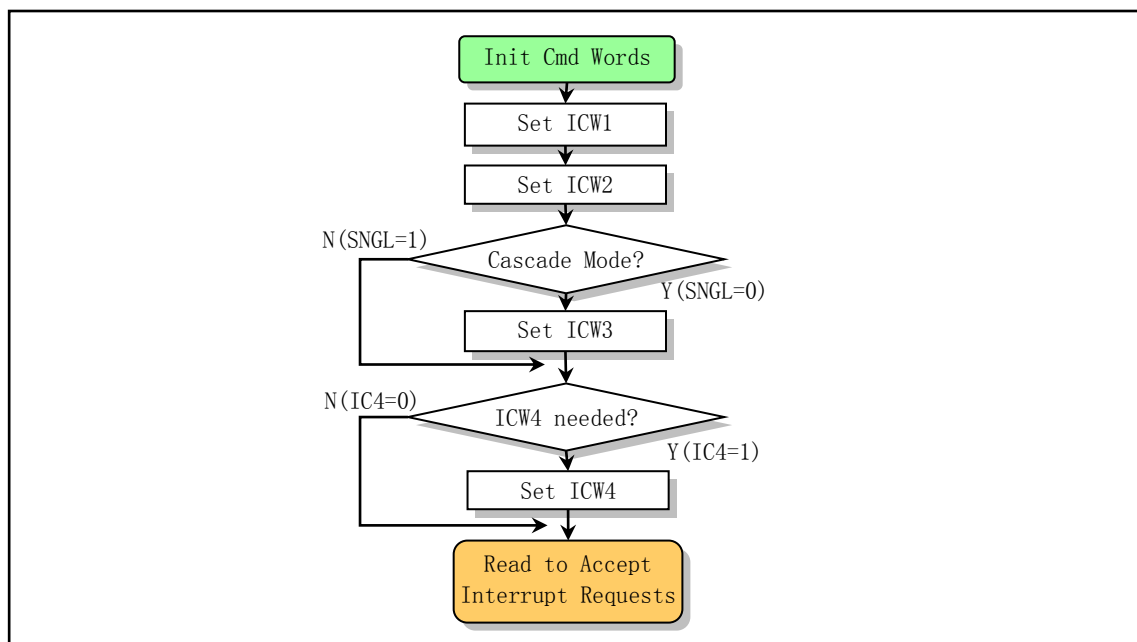


Figure 6-8 8259A Initialization Sequence

(1) ICW1 When the transmitted byte 5th bit (D4) = 1 and the address line A0 = 0, it indicates that ICW1 is programmed. At this time, for the multi-chip cascading case of the PC/AT microcomputer system, the port address of the 8259A main chip is 0x20, and the port address of the slave chip is 0xA0. The format of ICW1 is shown in Table 6-5.

Table 6-5 Interrupt initialization command word ICW1

Bit	Name	Description
D7	A7	A7-A5 indicates the page start address used in the MCS80/85 for the interrupt service process. They are combined with A15-A8 in ICW2. These are not used in 8086/88.
D6	A6	
D5	A5	
D4	1	Always 1
D3	LTIM	1 - Level triggered interrupt mode; 0 - Edge triggered mode.
D2	ADI	The MCS80/85 used for the CALL instruction address interval. Not used in 8086/88.
D1	SNGL	1 - Single 8259A; 0 - Cascade mode.
D0	IC4	1 - requires ICW4; 0 - not required.

In the Linux 0.12 kernel, ICW1 is set to 0x11. It indicates that the interrupt request is edge triggered, multiple slices of 8259A are cascaded, and finally ICW4 needs to be sent.

(2) ICW2 This initialization command word is used to set the upper 5 bits of the interrupt number sent by the chip. After the ICW1 is set, the interrupt number indicates that ICW2 is set when A0=1. At this time, for the multi-chip cascading of the PC/AT microcomputer system, the port address of the 8259A main chip is 0x21, and the port address of the slave chip is 0xA1. The ICW2 format is shown in Table 6-6.

Table 6-6 Interrupt initialization command word ICW2

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	A15/T7	A14/T6	A13/T5	A12/T4	A11/T3	A10	A9	A8

In the MCS80/85 system, the A15-A8 indicated by bits D7-D0 and the A7-A5 set by ICW1 form the interrupt service program page address. In a system or compatible system using the 8086/88 processor, T7-T3 is the upper 5 bits of the interrupt number, and the lower 3 bits automatically set by the 8259A chip form an 8-bit interrupt number. When the 8259A receives the second interrupt response pulse INTA, it will be sent to the data bus for the CPU to read.

The Linux 0.12 system sets the ICW2 of the main slice to 0x20, indicating that the main chip interrupt request is 0 level - the corresponding interrupt number range of the 7th level is 0x20-0x27. The ICW2 of the slave slice is set to 0x28, indicating that the interrupt number range corresponding to the 8-level to 15-level slave interrupt request is 0x28-0x2f.

(3) ICW3 This command word is used to load an 8-bit slave register when multiple 8259A chips are cascaded. The port address is the same as above. The ICW3 format is shown in Table 6-7.

Table 6-7 Interrupt initialization command word ICW3

	A0	D7	D6	D5	D4	D3	D2	D1	D0
Master	1	S7	S6	S5	S4	S3	S2	S1	S0
Slave	1	0	0	0	0	0	ID2	ID1	ID0

The master chip bits S7\_S0 correspond to the cascaded slaves. Which bit is 1 means that the signal on the interrupt request pin IR of the master is from the slave, otherwise the corresponding IR pin is not connected to the slave. The slave chip bits of ID2\_ID0 correspond to the identification numbers of the slave chips, that is, the interrupt level connected to the master chip. When a slave receives a cascading line (CAS2 - CAS0) input value equal to its own ID2 - ID0, it means that the slave is selected. At this point, the slave should send the interrupt number of the interrupt request currently selected from the slave chip to the data bus.

The Linux 0.12 kernel sets the ICW3 of the 8259A main chip to 0x04, that is, S2=1, and the remaining bits are 0. Indicates that the IR2 pin of the master chip is connected to a slave chip. The ICW3 of the slave chip is set to 0x02, that is, its identification number is 2. Represents the IR2 pin from the slave chip connected to the main chip. Therefore, the order of the interrupt priorities is the highest at level 0, followed by the level 8-15 on the chip, and finally the level 3-7.

(4) ICW4 When IC0 bit 0 (IC4) is set, it indicates that ICW4 is required. Address line A0=1. The port address is the same as above. The ICW4 format is shown in Table 6-8.

Table 6-8 Interrupt initialization command word ICW4

Bit(s)	Name	Description
D7-5		Always 0
D4	SFNM	1 - special fully nested mode; 0 - not a special fully nested mode.
D3	BUF	1 - buffer mode; 0 - unbuffered mode.
D2	M/S	1 - Buffered mode /Slave; 0 - Buffered mode /Master.
D1	AEOI	1 - Auto End of Interrupt mode; 0 - Normal End of Interrupt mode.
D0	μ PM	1 - 8086/88 processor system; 0 - MCS80/85 system.

The value of the ICW4 command word sent to the 8259A master chip and the slave chip by the Linux 0.12 core is 0x01. Indicates that the 8259A chip is set to a normal fully nested, unbuffered, non-automatic end interrupt mode and is used in the 8086 and its compatible systems.

### 3. Operation command word programming

After setting the initialization command word register to the 8259A, the chip is ready to receive interrupt request signal from the device. However, during the 8259A operation, we can also use the operation command word OCW1-OCW3 to monitor the working status of the 8259A, or change the working mode of the 8259A set at the time of initialization.

(1) OCW1 This operation command word is used to read/write the interrupt mask register IMR. Address line A0 needs to be 1. The port address description is the same as above. The OCW1 format is shown in Table 6-9.

Table 6-9 Interrupt operation command word OCW1

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

Bits D7-D0 correspond to 8 interrupt requests, 7 levels - 0 level mask bits M7 - M0. If M=1, the corresponding interrupt request level is masked; if M=0, the corresponding interrupt request level is allowed. In addition, masking high priority does not affect other low priority interrupt requests.

During the Linux 0.12 kernel initialization process, the code uses the operation command word to modify

the relevant interrupt request mask bit after setting the relevant device driver. For example, at the end of the floppy disk driver initialization, in order to allow the floppy device to issue an interrupt request, port 0x21 is read to obtain the current mask byte of the 8259A chip. Then, with AND  $\sim 0x40$  operation, the mask bit of the interrupt request 6 connected to the corresponding floppy disk controller is reset. Finally write back to the interrupt mask register. See line 461 of the kernel/blk\_drv/floppy.c program.

(2) OCW2 is used to send EOI commands or set the automatic rotate mode for interrupt priority. When the bit D4D3 = 00, the address line A0 = 0 indicates that the OCW2 is programmed. The format of the operation command word OCW2 is shown in Table 6–10.

Table 6-10 Interrupt operation command word OCW2

Bit(s)	Name	Description
D7	R	Priority rotation state.
D6	SL	Priority setting flag.
D5	EOI	Non-Automatic End of Interrupt flag.
D4-3		Always 0
D2	L2	L2 -- L0 - 3 bits form the level number, corresponding to the interrupt request level IRQ0--IRQ7 (or IRQ8-IRQ15).
D1	L1	
D0	L0	

The roles and meanings of the combination of bits D7-D5 are shown in Table 6–11. Those with an \* can specify the priority to reset the ISR by setting L2--L0, or select the special rotate priority to become the current lowest priority.

Table 6-11 OCW2 bit D7--D5 combination meaning

R(D7)	SL(D6)	EOI(D5)	Description	Type
0	0	1	Non-specific EOI command (fully nested mode).	End of Interrupt
0	1	1	Specific EOI command (not fully nested).	
1	0	1	Rotate on non-specific EOI command.	Automatic Rotation
1	0	0	Rotate in Automatic EOI mode (Set).	
0	0	0	Rotate in Automatic EOI mode (Clean).	
1	1	1	Rotate on Specific EOI command.	Specific rotation
1	1	0	Set priority command.	
0	1	0	No operation.	

The Linux 0.12 kernel uses only the operational command word to send an end interrupt EOI command to the 8259A before the end of the interrupt processing. The OCW2 value used is 0x20, indicating a non-special end interrupt EOI command in full nested mode.

(3) OCW3 is used to set the special mask mode and read register status (IRR and ISR). When D4D3=01 and address line A0=0, it means that OCW3 is programmed (read/write). However, this operation command word is not used in the Linux 0.12 kernel. The format of OCW3 is shown in Table 6–12.

Table 6-12 Interrupt operation command word OCW3

Bit	Name	Description
D7		Always 0
D6	ESMM	Operate in a special mask mode: D6 -- D5: 11 - Set special mask; 10 - Reset special mask; 00,01 - No action.
D5	SMM	
D4		Always 0
D3		Always 1
D2	P	1 - Poll command; 0 - No poll command.
D1	RR	Read register status command on the next RD pulse: D1 -- D0: 11 - Read In Service Reg. ISR; 10 - Read Interrupt Reg. IRR
D0	RIS	

#### 4. 8259A operation mode description

In the programming process of the 8259A initialization command word and the operation command word, some working methods are mentioned. The following is a detailed description of several common ways to better understand how the 8259A chip operates.

##### (1) Full nested mode

After initialization, the operation automatically enters this fully nested mode unless the operation command word has been used to change the way the 8259A works. In this mode, the order of interrupt request priority is from level 0 to level 7 (level 0 has the highest). When the CPU responds to an interrupt, the highest priority interrupt request is determined and the interrupt request's interrupt number is placed on the data bus. In addition, the corresponding bit in the interrupt service register ISR will be set and the set state of the bit will remain until the end of the interrupt EOI command is sent before returning from the interrupt service procedure. If the automatic interrupt end AEOI bit is set in the ICW4, the bit in the ISR will be reset at the end edge of the second interrupt response pulse INTA issued by the CPU. During the ISR with a set bit, all interrupt requests of the same priority and low priority will be temporarily disabled, but higher priority interrupt requests are allowed to respond and be processed. Furthermore, the corresponding bits of the interrupt mask register IMR can mask 8-level interrupt requests, respectively, but masking any one of the interrupt requests does not affect the operation of other interrupt requests. Finally, after initializing the command word programming, the 8259A pin IR0 has the highest priority, while IR7 has the lowest priority. The Linux 0.12 kernel code works with the machine's 8259A chip set in this mode.

##### (2) End of Interrupt (EOI) method

As described above, the bit corresponding to the interrupt request being processed in the service register ISR can be reset in two ways. One is that when the automatic interrupt end bit AEOI in ICW4 is set, it is reset by the end edge of the second interrupt response pulse INTA issued by the CPU. This method is called the Automatic End of Interrupt (AEOI) method. The second is to send an end interrupt EOI command to reset interrupt before returning from the interrupt service process. This method is called the End of Program Interrupt (EOI) method. In cascading system, the slave interrupt service routine needs to send two EOI commands, one for the slave chip and one for the master chip.

There are two ways in which a program can issue an EOI command. One is called a special EOI command, and the other is called a non-special EOI command. The special EOI command is used in non-fully nested mode and can be used to specify the interrupt level bits for the specific reset of the EOI command. That is, when sending a special EOI command to the chip, it is necessary to specify the priority in the reset ISR. The special EOI command is sent using OCW2, the upper 3 bits are 011, and the lowest 3 bits are used to specify the priority. This special EOI command is used in current Linux systems. The non-special EOI command for the



fully nested mode automatically resets the highest priority bit currently in the service register ISR. Because in the fully nested mode, the highest priority bit in the ISR is definitely the priority of the last response and service. It is also sent using OCW2, but the highest 3 bits need to be 001. This non-special EOI command is used in the Linux 0.12 system discussed in this book.

### (3) Special full nested mode

The special full nesting mode (D4=1) set in ICW4 is mainly used in large cascading systems, and the priority in each slave chip needs to be saved. This approach is similar to the normal full nesting approach described above, with the following two exceptions:

A. When an interrupt request from a slave chip is being serviced, the slave chip is not excluded by the priority of the master chip. Therefore, other higher priority interrupt requests issued from the chip will be recognized by the master chip, and the master chip will immediately issue an interrupt to the CPU. In the above conventional full nesting mode, when a slave interrupt request is being serviced, the slave chip is masked by the master chip. Therefore, a higher priority interrupt request issued from the slave chip cannot be processed.

B. When exiting the interrupt service routine, the program must check if the current interrupt service is the only interrupt request issued from the slave chip. The method of checking is to first issue a non-special interrupt EOI command to the slave chip and then read the value of its service register ISR. Check if the value is 0 at this time. If it is 0, it means that a non-special EOI command can be sent to the main chip. If it is not 0, there is no need to send an EOI command to the main chip.

### (4) Cascade mode method

The 8259A can be easily connected into a master and a number of slave chips. If you use 8 slave chips, you can control up to 64 interrupt priorities. The master chip controls the slaves through three cascaded lines. These three cascaded lines are equivalent to the chip selection signal from the chip. In the cascade mode, the interrupt output of the slave chip is connected to the interrupt request input pin of the master chip. When an interrupt request line from the chip is processed and responded, the master chip selects the slave chip to place the corresponding interrupt number on the data bus.

In a cascading system, each 8259A chip must be initialized independently and can operate in different ways. In addition, the initialization command word ICW3 of the master chip and the slave chip are separately programmed. It is also necessary to send 2 interrupt end EOI commands during operation, one for the main chip and the other for the slave chip.

### (5) Automatic rotation priority mode

When we are managing devices with the same priority, we can use OCW2 to set the 8259A chip to automatic rotation priority mode. That is, after a device receives a service, its priority automatically becomes the lowest. The priorities are cyclically changed in sequence. The most unfavorable situation is that when an interrupt request comes in, it needs to wait for 7 devices before it can get the service.

### (6) Interrupt mask mode

The interrupt mask register, IMR, controls the masking of each interrupt request. The 8259A can be set to two mask methods. For general normal masking, use OCW1 to set the IMR. The IMR bits (D7--D0) are applied to the respective interrupt request pins IR7-IR0. Masking an interrupt request does not affect other priority interrupt requests. This normal masking mode causes the 8259A to mask all low priority interrupt requests for an interrupt request during the response and service (before the EOI command is sent). However, in some applications it may be necessary for interrupt service process to dynamically change the system's priority. In order to solve this problem, the special masking method was introduced in the 8259A. We need to first set this mode (D6, D5 bits) using OCW3. In this special masking mode, the masking information set by OCW1 causes all unmasked priority interrupts to be responded to during an interrupt.

#### (7) Read register status

There are three registers (IMR, IRR, and ISR) in the 8259A that allow the CPU to read its status. The current mask information in the IMR can be obtained by directly reading OCW1. Before reading the IRR or ISR, you need first use OCW3 to output reading command of IRR or ISR before you can read them.

## 6.4 head.s

### 6.4.1 Function description

The head.s program is linked to the system module along with the target files of other programs in the kernel after being compiled to an object file, and is located at the beginning of the system module. This is why it is called a head program. The system module will be placed in the sector starting after the setup module on the disk, starting from the sixth sector on the disk. Under normal circumstances, the system module of the Linux 0.12 kernel is about 120 KB in size, so it accounts for about 240 sectors on the disk.

From here on, the kernel is completely running in protected mode. The heads.s assembly file is different from the previous assembly syntax. It uses AT&T's assembly language format and requires GNU's gas and gld to compile and link. So please note that the direction of the assignment in the code is from left to right.

This program is actually at the beginning of the absolute address 0 of the memory. 这个程序的功能比较单一。First, it loads each data segment register, re-establishes the interrupt descriptor table IDT, a total of 256 items (boot/head.s, 78), and makes each entry point to a dummy interrupt subroutine ignore\_int that only reports errors. This dummy interrupt vector points to a default "ignore interrupt" process (boot/head.s, 150). The message "Unknown interrupt" is displayed when an interrupt occurs and the interrupt vector has not been setup. All 256 items are set here to prevent the occurrence of a general protection fault (Exception 13). Otherwise, if the IDT is set to less than 256 entries, the CPU will generate a general protection fault (Exception 13) when the descriptor entry specified by a required interrupt is greater than the maximum descriptor entry set. In addition, if there is a problem with the hardware and the device vector is not placed on the data bus, the CPU will usually read all 1 (0xff) as a vector from the data bus, so it will read the 256th item in the IDT table. Therefore, it also causes general protection errors if the vector is not set.

For some interrupts that need to be used in the system, the kernel will setup the interrupt descriptor items for these interrupts during the process of their initialization (init/main.c), and point them to the corresponding actual interrupt handler procedures. Usually, the exception interrupt handler (int0 -- int 31) is re-installed in the initialization function of traps.c (kernel/traps.c, line 185), and the system call interrupt int 0x80 is installed in the scheduler initialization function (kernel/sched.c, line 417).

Each descriptor item in the interrupt descriptor table IDT also occupies 8 bytes, and its format is shown in Figure 6-9. Where P is the segment presence flag; DPL is the priority of the descriptor.

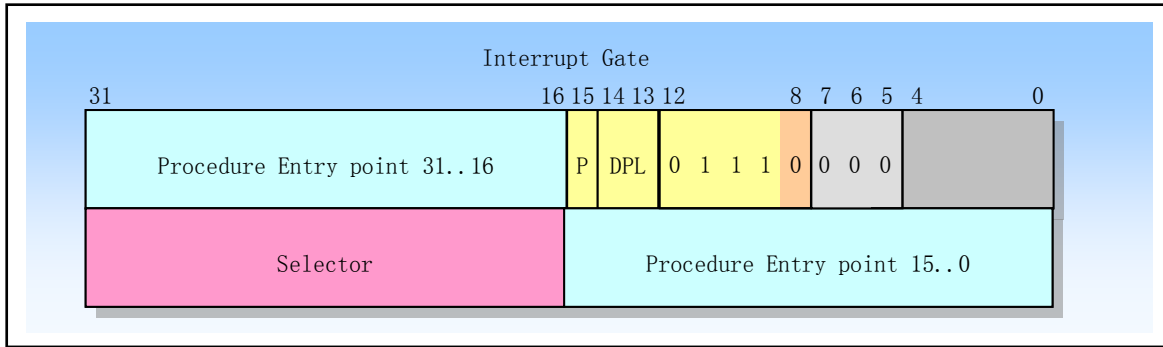


Figure 6-9 Interrupt Gate Descriptor Format in IDT

In the head.s program, the segment selector field in the interrupt gate descriptor is set to 0x0008, which indicates that the dummy interrupt service routine ignore\_int is in the kernel code. and the offset is set to the offset of ignore\_int interrupt service handler in the head.s program. Since the head.s program is moved to the beginning of memory address 0, the offset of the interrupt service handler is also the offset in the kernel code. Since the kernel code segment is always in memory and the privilege level is 0 (ie, P=1, DPL=00), it can be seen from the figure that the value of byte 5 and byte 4 of the interrupt gate descriptor should be 0x8E00.

After setting the interrupt descriptor table, the program rebuilds the global segment descriptor table GDT. In fact, descriptors in the newly created GDT table is not much different from that in the original GDT table. Except for some differences in the limit (originally 8MB, now 16MB), the other content is exactly the same. So we can also set the segment limit of the descriptor directly to 16MB in setup.s, and then directly move the original GDT table to the appropriate location in the memory. So the main reason for re-create a new GDT here is to put the GDT table in a reasonable place in the kernel space. The previously set GDT table is at location 0x902XX in memory. This place will be used as part of the memory cache after the kernel is initialized.

Then it is detected whether the A20 address line is turned on. The method is to compare the contents starting at the memory address 0 with the contents beginning from address 1 MB. If the A20 line is not enabled, the CPU will cyclically access the contents at address (address MOD 1MB) when accessing more than 1MB of physical memory, that is, the same as accessing the corresponding byte starting from address 0. If the program detects that it is not open, it enters an infinite loop. Otherwise the program will continue to test whether the PC contains a math coprocessor chip (80287, 80387 or its compatible chip) and set the corresponding flag in the control register CR0.

Then head.s code sets the paging mechanism for managing memory, placing the page directory table at the beginning of the physical address 0 (also the memory area this program is located, so the code area that has finished execution will be overwritten ). Immediately afterwards, four page tables with a total addressable 16 MB of memory are placed, and their entries are set separately. The page directory entry and page table entry format are shown in Figure 6-10. Where P is page exists flag; R/W is read/write flag; U/S is user/super user flag; A is page visited flag; D is page content modified flag; and the leftmost 20 bits are the upper 20 bits of the page address in the memory corresponding to the page entry.

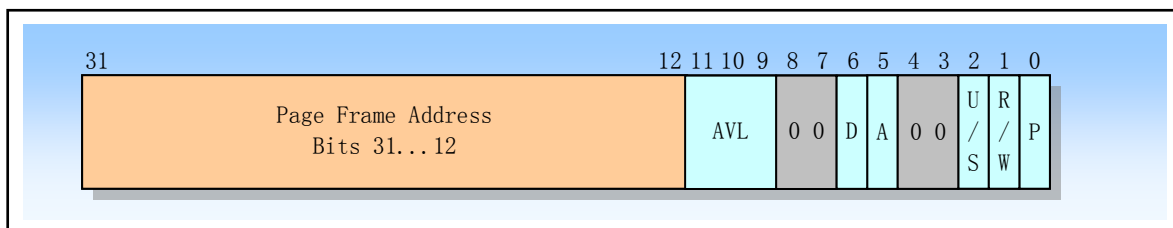


Figure 6-10 Entry structure of page directory and page table

Here, the attribute flag of each entry is set to 0x07 (P=1, U/S=1, R/W=1), indicating that the page exists and the user can read and write. The reason for setting the kernel page table attribute in this way is that both the segmentation and the paging management have protection methods. The protection flags (U/S, R/W) set in the page directory and page table entries need to be combined with the privilege level (PL) protection in the segment descriptor. But the PL in the segment descriptor plays a major role. The CPU will first check the segment protection and then check the page protection. If the current privilege level CPL < 3 (for example, 0), the CPU is running as a supervisor. At this point all pages can be accessed, and free to read and write. If CPL = 3, the CPU is running as user (user). At this point only the pages belonging to user (U/S=1) are accessible, and only pages marked as readable and writable (W/R = 1) are writable. At this time, the page belonging to the super user (U/S=0) can neither be written nor read. Because the kernel code is a bit special, it contains the code and data for task 0 and task 1. So setting the page property to 0x7 here will ensure that the two tasks can be executed in user mode, but they cannot access kernel resources arbitrarily.

Finally, the head.s program uses the return instruction to pop out the entry address of the /init/main.c program pre-placed on the stack, and transfer the execution rights to the main() code.

## 6.4.2 Code Comments

Program 6-3 linux/boot/head.s

---

```
1 /*
2  * 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:                # The page directory will be stored here.

```

# Note again!! This is already in 32-bit mode, so 0x10 is now a selector for a descriptor  
# and the instruction loads the contents of the corresponding descriptor into the segment  
# register. Here, the meaning of 0x10 is: request privilege level RPL is 0 (bit 0-1 = 0),  
# select global descriptor table GDT (bit 2 = 0), and select the second item in the table  
# (bits 3-15 = 2). It just points to the data segment descriptor item in the table (see  
# the 575--578 lines in setup.s for the specific value of the descriptor).  
#  
# The following code means: set ds, es, fs, gs to the kernel data segment constructed in  
# setup.s with selector=0x10 (corresponding to item 3 of the global descriptor table),  
# and place the stack in the user\_stack array area pointed to by stack\_start. Then use  
# the new interrupt descriptor table (line 232) and the global segment description table  
# (lines 234--238) defined later in this program. The initial content in the new GDT table

# is basically the same as in setup.s, and only the segment length is changed from 8MB to 16MB.  
 # Stack\_start is defined in kernel/sched.c, lines 82-87. It is a long pointer to the end  
 # of the user\_stack array. The code on line 23 sets the stack used here, which we now call  
 # the system stack. But after moving to task 0 execution (137 lines in init/main.c), the  
 # stack is used as the user stack for task 0 and task 1.

```

17 startup_32:                # set each data segment registers.
18     movl $0x10,%eax         # direct operand starts with '$', otherwise it's an address.
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, set system stack.
24     call setup_idt          # line 67--93
25     call setup_gdt          # line 95--107
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             # GDT changed, all segs need to be reloaded.
30     mov %ax,%gs

```

# Since the segment length in the descriptor has been changed from 8MB to 16MB (see  
 # setup.s line 567-578 and lines 235-236 later in this program), therefore, the load  
 # operation must be performed on all segment registers again. In addition, by using the  
 # bochs simulation software to track the code, if the CS is not loaded again, the limit  
 # length in the invisible portion of CS is still 8 MB when executing to line 26. It seems  
 # that the CS should be reloaded here. However, since the code segment descriptor only  
 # changes the segment length, the rest is exactly the same, so the 8MB limit length does  
 # not cause problems during the kernel initialization phase. In addition, the inter-segment  
 # jump instruction will reload CS during the kernel execution process, so not loading it  
 # here will not cause future kernel errors.  
 # In response to this problem, the current kernel has added a long jump instruction after  
 # the 25th line: 'ljmp \$(\_\_KERNEL\_CS), \$1f', jump to line 26 to ensure that the CS is  
 # indeed reloaded.

```

31     lss _stack_start,%esp

```

# Lines 32-36 are used to test if the A20 address line is enabled. The method used is to  
 # write any value to the memory address beginning at 0x000000, and then see if the value  
 # at the cooresponding address 0x100000 (1M) contains the same value. If they are always  
 # the same, they will continue to compare, that is, infinite loops and crashes. This means  
 # that the address A20 line is not strobed, and the kernel cannot use more than 1MB of  
 # memory.

#  
 # '1:' on line 33 is a label consisting of a local symbol. At this point the symbol  
 # represents the current value of the Active Location Counter and can be used as the  
 # operand of the instruction. Local symbols are used to help compilers and programmers  
 # temporarily use some names. There are a total of 10 local labels that can be reused  
 # throughout the program. These labels are referenced using the names '0', '1', ..., '9'.  
 # In order to define a local symbol, the label is written in the form 'N:' (where N  
 # represents a number). In order to reference this previously defined label, it needs to  
 # be written as 'Nb'. In order to reference the next definition of a local label, it needs  
 # to be written as 'Nf'. Above 'b' means "backwards" and 'f' means "forwards". At some

# point in the assembly file, we can refer up to 10 labels backwards/forward.

```

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 backward label 1.
                                   # '5f' means forward 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 bit 16 of the CR0 control register in the 486 CPU mentioned in the previous comment
# is the write-protection flag (WP), which is used to prohibit the super-user-level
# program from writing to the general user read-only page. This flag is mainly used by
# the operating system to implement a copy-on-write method when creating a new process.
#
# The following code (lines 43-65) is used to check if the math coprocessor chip is
# present. The method is to modify the control register CR0 and execute a coprocessor
# instruction assuming a coprocessor is present. If something goes wrong, the coprocessor
# chip does not exist. The coprocessor emulation bit EM (bit 2) in CR0 needs to be set and
# the coprocessor presence flag MP (bit 1) is reset.

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   # line 135
50
51 /*
52  * We depend on ET to be correct. This checks for 287/387.
53  */

# The following fninit and fstsw are instructions for the math coprocessor (80287/80387).
# finit issues an initialization command to the coprocessor, which places the coprocessor
# in a known state that is not affected by previous operations, sets its control word to
# the default value, clears the status word and all floating point stack registers. This
# non-waiting form of fninit also causes the coprocessor to terminate execution of any
# previous arithmetic operations currently in progress. The fstsw instruction gets the
# status word of the coprocessor. If there is a coprocessor in the system, then the status
# low byte must be 0 after the fninit instruction is executed.

54 check_x87:
55      fninit
56      fstsw %ax               # get status word -> ax
57      cmpb $0,%al            # status word should be 0 after fninit if has a math.
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

```

# .align is an assembly indicator. Its meaning refers to the storage boundary alignment adjustment. Here, '2' indicates that the offset position of the subsequent code or data is adjusted to the position where the last 2 bits of the address value are zero ( $2^2$ ), that is, the memory address is aligned in a 4-byte manner. (But now GNU as is writing the aligned value directly instead of the power of 2). The purpose of using this directive to achieve memory alignment is to increase the speed and efficiency of 32-bit CPU access to code or data in memory.

# The following two byte are the machine code of 80287 instruction fsetpm. Its role is to set 80287 to protection mode. The 80387 does not require this instruction and will treat it as a nop.

```

63 .align 2
64 1:      .byte 0xDB,0xE4          /* fsetpm for 287, ignored by 387 */
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 */

```

# Each item in the interrupt descriptor table (IDT) is composed of 8 bytes, but its format is different from that in the GDT table, and is called a gate descriptor. Its 0-1, 6-7 bytes are offsets, 2-3 bytes are selectors, and 4-5 bytes are some flags.

# This code first sets the default interrupt descriptor value of 8 bytes in EDX and EAX, and then places the descriptor in each item of the idt table, a total of 256 items. EAX contains a descriptor lower 4 bytes and EDX contains a height of 4 bytes. During the subsequent initialization process, the kernel replaces the current default settings with those really useful interrupt descriptor entries.

```

78 setup_idt:
79      lea ignore_int,%edx          # effective addr of ignore_int -> edx
80      movl $0x00080000,%eax        # store selector 0x0008 to high word of eax
81      movw %dx,%ax                /* selector = 0x0008 = cs */
82      movw $0x8E00,%dx            # store offset low 16 bits to low word of eax.
83                                  /* interrupt gate - dpl=0, present */
84                                  # edx contains high 4 bytes of gate descriptor.
85      lea _idt,%edi               # _idt is IDT table address (offset).
86      mov $256,%ecx
87 rp_sidt:
88      movl %eax, (%edi)            # store the dummy descriptor into IDT table
89      movl %edx, 4(%edi)           # store eax to [edi+4]
90      addl $8,%edi                # edi point to the next item by plus 8.
91      dec %ecx
92      jne rp_sidt

```

```
92      lidt idt_descr          # Load the interrupt descriptor table register.
93      ret
94
95 /*
96 *  setup_gdt
97 *
98 *  This routines sets up a new gdt and loads it.
99 *  Only two entries are currently built, the same
100 *  ones that were built in init.s. The routine
101 *  is VERY complicated at two whole lines, so this
102 *  rather long comment is certainly needed :-).
103 *  This routine will beoverwritten by the page tables.
104 */
105 setup_gdt:
106     lgdt gdt_descr          # contents on lines 234-238.
107     ret
108
109 /*
110 *  I put the kernel page tables right after the page directory,
111 *  using 4 of them to span 16 Mb of physical memory. People with
112 *  more than 16MB will have to expand this.
113 */
114
115 # Each page table is 4KB in size (1 page memory), and each page table entry requires 4
116 # bytes, so a page table can store a total of 1024 entries. If a page table entry
117 # addresses a 4KB address space, a page table can address 4 MB of physical memory. The
118 # format of the page table entry is: the first 0-11 bits of the item store some flags,
119 # such as whether it is in memory (P bit 0), read and write permission (R/W bit 1), normal
120 # user or super user (U/S, Bit 2), whether it has been modified or dirty (D bit 6), etc.;
121 # bits 12-31 are page frame address, which are used to indicate the physical start address
122 # of a page.
123
124 .org 0x1000      # here begins the first page table. page directory stored at 0.
125 pg0:
126
127 .org 0x2000
128 pg1:
129
130 .org 0x3000
131 pg2:
132
133 .org 0x4000
134 pg3:
135
136 .org 0x5000      # the code or data below starts from offset 0x5000.
137 /*
138 *  tmp_floppy_area is used by the floppy-driver when DMA cannot
139 *  reach to a buffer-block. It needs to be aligned, so that it isn't
140 *  on a 64kB border.
141 */
142 _tmp_floppy_area:
143     .fill 1024, 1, 0      # 1024 bytes filled with 0.
144
```



# The following push operations are used to prepare for the jump to the main() function in the init/main.c. The instruction on line 139 pushes the return address (label L6) on the stack, while line 140 pushes the address of the main() function code. When head.s finally executes the ret instruction on line 218, it will pop up the address of main() and transfer control to the init/main.c program. See the description of C function call mechanism in Chapter 3.

# The first three pop-up 0 values represent the main function arguments envp, argv pointer, and argc, respectively, but main() does not use them. The 139-line push operation simulates the return address of the call to main(). So if the main program really exits, it will return to the label L6 to continue, that is, performs the infinite loop. Line 140 pushes the address of main() onto the stack, so that when the 'ret' instruction is executed after setting the paging processing (setup\_paging), the address of the main() is popped out of the stack and the main() is executed.

```

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            # '_main' is the internal representation of main().
141     jmp setup_paging        # jump to line 198.
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"    # 定义字符串“未知中断(回车换行)”。
149     .align 2                        # alignment with 4 bytes in memory.
150 ignore_int:
151     pushl %eax
152     pushl %ecx
153     pushl %edx
154     push %ds                  # ds, es, fs, gs still occupy 2 words each when on stack.
155     push %es
156     push %fs
157     movl $0x10, %eax          # set selector (ds, es, fs points data descriptor in gdt)
158     mov %ax, %ds
159     mov %ax, %es
160     mov %ax, %fs
161
162 # Put printk() function's parameter pointer onto the stack. Note that if '$' is not added
163 # before int_msg, it means that the long word ('Unkn') at the int_msg symbol is pushed
164 # onto the stack. This function is in /kernel/printk.c. '_printk' is the internal
165 # representation in the printk compiled module.
166     pushl $int_msg
167     call _printk
168     popl %eax                 # /kernel/printk.c
169     pop %fs
170     pop %es
171     pop %ds
172     popl %edx
173     popl %ecx
174     popl %eax
175     iret                     # iret pop out CFLAGS too.

```

```
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  */
# The meaning of the second paragraph of the original comment above means that more than
# 1MB of memory space in the machine is mainly used for the main memory area. This main
# memory area is managed by the mm module, which involves page mapping operations. All
# other functions in the kernel are the general (ordinary) functions referred to here. To
# use the page in the main memory area, you need to use the get_free_page() function to
# get it. Because the memory pages in the main memory area are shared resources, there
# must be a program for unified management to avoid resource contention.
#
# The 1-page page directory table and the 4-page page table are stored at the physical
# address 0x0 of the memory. The page directory table is common to all processes in the
# system, and the 4-page page table here is kernel-specific. The initial 16MB linear
# address space is mapped one by one to the physical memory space. For newly created
# processes, the system will store the page table for its application in the main memory
# area. In addition.

197 .align 2                                # align memory boundaries in 4-byte.
198 setup_paging:
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                         # eax -> [es:edi], edi increased by 4

# The following statements sets the items in the page directory. Since the kernel uses a
# total of 4 page tables, only 4 items need to be set. The structure of the page directory
# entry is the same as the structure of the entries in the page table, with 4 bytes being
# one item. See the descriptions under line 113 above.
# For example, "$pg0+7" means: 0x00001007, which is the first item in the page directory,
```

# then the address of the first page table = 0x00001007 & 0xfffff000 = 0x1000; the  
 # attribute flag of the first page table = 0x00001007 & 0x00000fff = 0x07, indicating that  
 # the page exists and the user can read and 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   /* ----- " " ----- */
```

# The following 6 lines of code fill in the contents of all the items in the four page  
 # tables, the total number of items: 4 (page table) \* 1024 (item / page table) = 4096  
 # items (0 - 0xffff), that is, it can map physical memory 4096 \* 4Kb = 16Mb. The content of  
 # each item is: the physical memory address mapped by the current item + page flags (all 7).  
 # The method used to fill in is to fill in the reverse order from the last item in the  
 # last page table. The position of the last item in each page table is 1023\*4 = 4092, so  
 # the last item on the last page is \$pg3 + 4092.

```
207      movl $pg3+4092,%edi      # edi -> points to the last item in the last page
208      movl $0xffff007,%eax     /* 16Mb - 4096 + 7 (r/w user,p) */
                                     # the last item map to physical mem addr 0xffff000 + 7
209      std                      # set direction flag, edi is decreased (by 4 bytes).
210 1:      stosl                 /* fill pages backwards - more efficient :-) */
211      subl $0x1000,%eax        # each time an item is filled,addr is reduced by 0x1000.
212      jge 1b                  # if less than 0, all filled.
# Now set the page directory base register cr3, it contains the physical address of the
# page directory table. Then set to start using paging (the PG flag of cr0, bit 31).
213      xorl %eax,%eax          /* pg_dir is at 0x0000 */
214      movl %eax,%cr3          /* cr3 - page directory start */
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 */
```

# After changing the paging flag, it is required to use the branch instruction to refresh  
 # the prefetch instruction queue. The ret instruction is used here. Another function of  
 # this return instruction is to pop the address of the main() pushed by instructions on  
 # line 140, and jump to the /init/main.c program to run. This program really ends here.

```
219
220 .align 2                      # align memory boundaries in 4-byte.
221 .word 0                      # skip a word, so that line 224 are 4-byte aligned.
```

# The following is the 6-byte operand required by the instruction LIDT to load the  
 # interrupt descriptor table register. The first 2 bytes are the limit of the IDT table,  
 # and last 4 bytes are the 32-bit base address of the IDT table in linear address space.

```
222 idt_descr:
223      .word 256*8-1            # idt contains 256 entries
224      .long _idt
225 .align 2
226 .word 0
```

# The 6-byte operand required by the LGDT instruction of the global descriptor table  
 # register is loaded below. The first 2 bytes are the limit of the GDT, and the last 4  
 # bytes are the linear base address of the GDT. Here the global table size is set to 2KB

```

# bytes (ie 0x7ff). Since each descriptor item has 8 bytes, there are a total of 256
# entries in the table. The symbol _gdt is the offset position of the global table in the
# program, see line 234.
227 gdt_descr:
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              # align memory boundaries by 8 (2^3) bytes.
232 _idt: .fill 256,8,0      # idt is uninitialized
233
# Global descriptor table GDT. The first four items are: empty item (not used), kernel
# code segment descriptor, kernel data segment descriptor, system call segment descriptor.
# The system call segment descriptor is not used. Linus might have wanted to put the
# system call code in this separate segment. A space of 252 items is reserved later for
# placing the local descriptor table (LDT) of the newly created task and the descriptor of
# the task state segment TSS.
# (0-nul, 1-cs, 2-ds, 3-syscall, 4-TSS0, 5-LDT0, 6-TSS1, 7-LDT1, 8-TSS2 etc...)

234 _gdt: .quad 0x0000000000000000    /* NULL descriptor */
235     .quad 0x00c09a0000000fff      /* 16Mb */    # 0x08, kernel code seg.
236     .quad 0x00c0920000000fff      /* 16Mb */    # 0x10, kernel data seg.
237     .quad 0x0000000000000000      /* TEMPORARY - don't use */
238     .fill 252,8,0                /* space for LDT's and TSS's etc */

```

## 6.4.3 Reference Information

### 6.4.3.1 Memory map after the end of head.s program execution

After the execution of head.s program, the kernel code has officially completed the settings of the memory page directory and the page tables, and re-created the interrupt descriptor table IDT and the global descriptor table GDT. In addition, the program also opened a 1KB byte buffer for the floppy disk driver. At this time the detailed image of the system module in memory is shown in Figure 6-11.

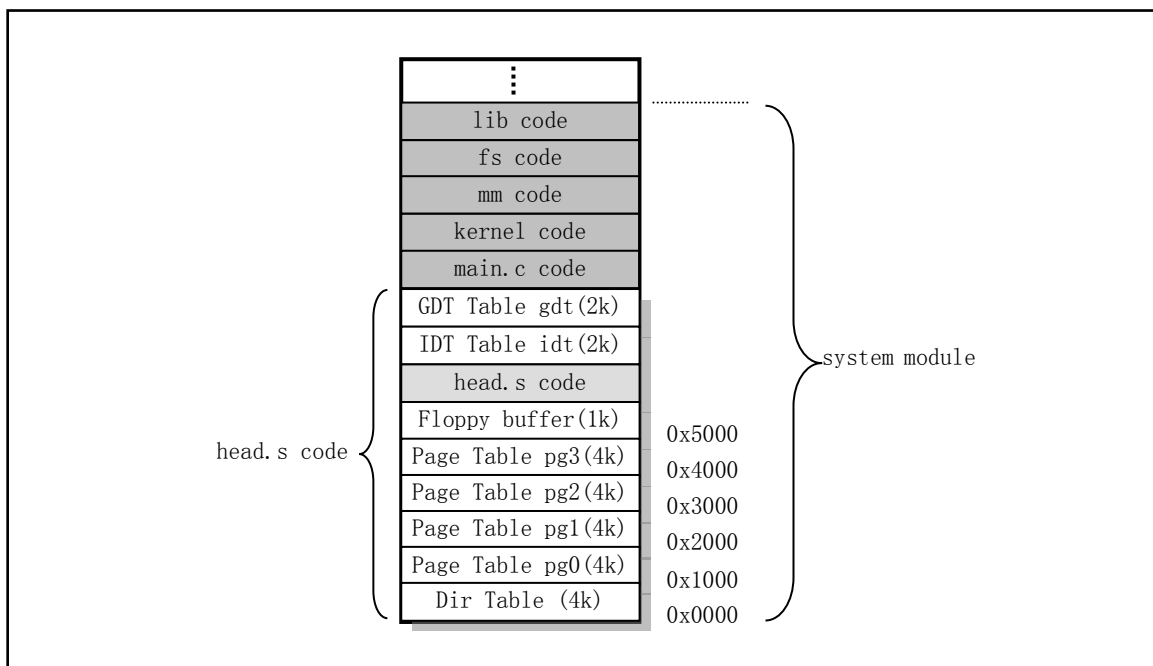


Figure 6-11 名 Map of the system module in memory

### 6.4.3.2 Intel 32-bit protection operation mechanism

The key to understanding this program is to know the operating mechanism of the Intel 80X386 32-bit protection mode. In order to be compatible with the 8086 CPU, the 80X86 protected mode was designed to be more complicated. See Chapter 4 for a detailed description of how the protected mode operates. Here we make a brief introduction to the protection mode by comparing the real mode and the protection mode.

When the CPU is running in real mode, the segment register is used to place the base address of a memory segment (for example, 0x9000). The size of the memory segment is fixed at 64KB. Up to 64KB of memory can be addressed in this segment. However, when entering the protection mode, the segment register does not contain segment base address in memory, but the selector of the descriptor item corresponding to the segment in the descriptor table. The 8-byte size descriptor contains the 'segment' base address and segment length of the segment linear address, as well as other bits describing the segment feature. Therefore, the memory location addressed at this time can be specified by the base address of the segment plus the current offset value. Of course, the actual physical memory address that is addressed needs to be transformed by the memory paging mechanism. In short, the memory addressing mode in 32-bit protected mode requires one more procedure, which is determined by using descriptors in the descriptor table and memory paging management.

Descriptor tables are divided into three types for different purposes: Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), and Local Descriptor Table (LDT). When the CPU is running in protected mode, there can only be one GDT and IDT at a time, and their table base addresses are specified by the registers GDTR and IDTR respectively. The number of local descriptor tables can be zero or up to 8191, as determined by the number of unused items in the GDT table and the specific system being designed. At some point, the base address of the current LDT table is specified by the contents of the LDTR register, and the contents of the LDTR are loaded using a descriptor in the GDT, that is, the LDT is also specified by the descriptor in the GDT.

In general, the kernel uses one LDT for each task (process). At runtime, the program can use the descriptors in the GDT as well as the descriptors in the LDT of the current task. For the Linux 0.12 kernel, there are 64 tasks that can be executed at the same time, so there are up to 64 descriptor entries for the LDT table in the GDT table.

The structure of the interrupt descriptor table IDT is similar to that of the GDT, which is located just in front of the GDT table in the Linux kernel. It contains a total of 256 8-byte descriptors. However, the format of each descriptor item is different from that of GDT, which stores the offset (0-1, 6-7 bytes) of the corresponding interrupt handler procedure, the selector of the segment (2-3 bytes), and Some flags (4-5 bytes).

Figure 6-12 is a schematic diagram of the descriptor table used in the Linux kernel. In the figure, each task occupies two descriptor items in the GDT. The LDT0 descriptor entry in the GDT table is the descriptor of the local descriptor table of the first task (process), and TSS0 is the descriptor of the task state segment (TSS) of the first task. Each LDT contains three descriptors, the first of which is not used, the second is the descriptor of the task code segment, and the third is the descriptor of the task data segment and the stack segment. When the DS segment register is the data segment selector of the first task, DS:ESI points to a certain data in the task data segment.

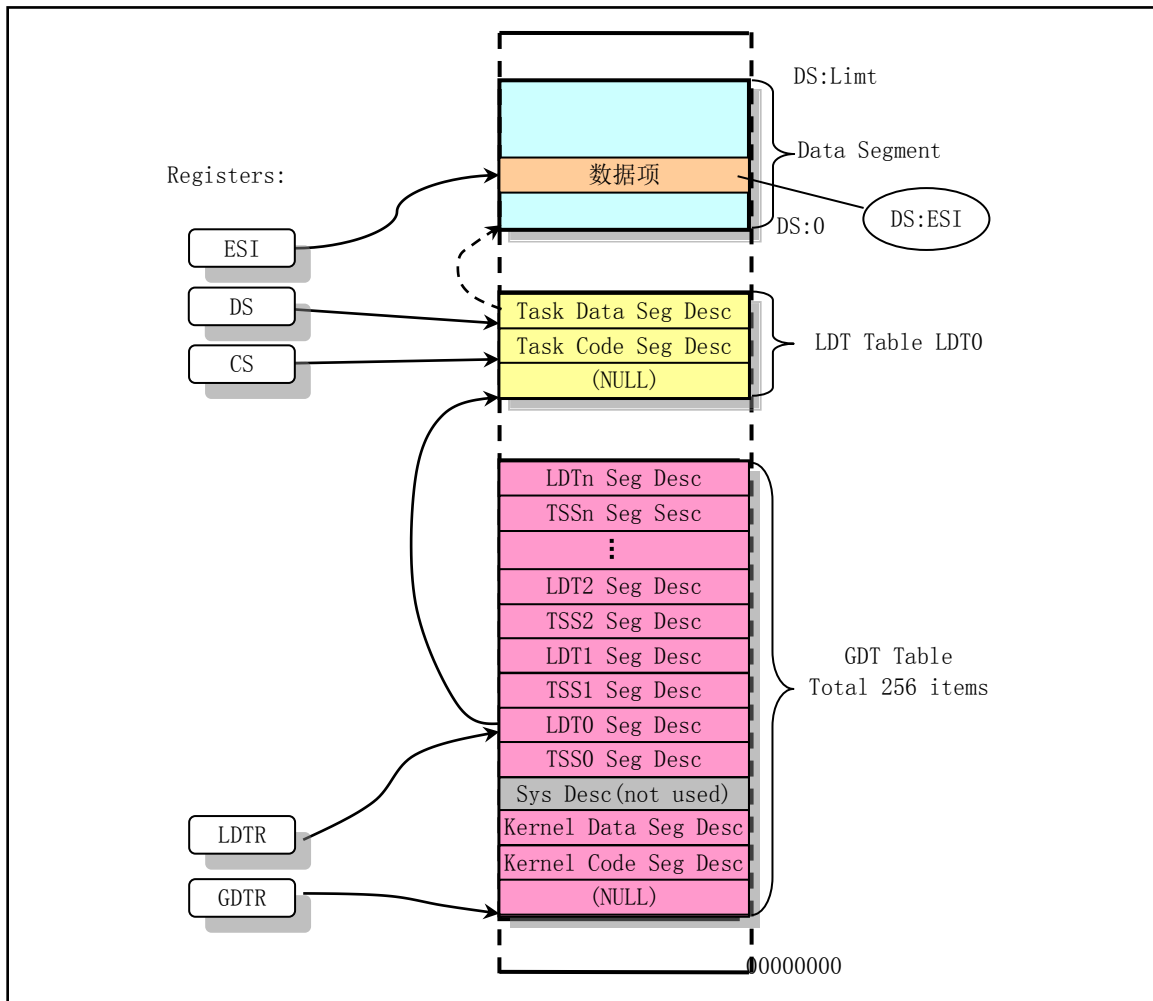


Figure 6-12 Descriptor table used by the Linux kernel

### 6.4.3.3 align directive

We have already explained the align directive when we introduced the assembler in Chapter 3. Here we will summarize it. The use of the directive `.align` is to instruct the compiler to populate the location counter (like the instruction counter) to a specified memory boundary at compile time. The goal is to increase the speed and efficiency with which the CPU can access code or data in memory. Its full format is:

```
.align val1, val2, val3
```

The first parameter value `val1` is the required alignment ; the second is specified by the padding byte. The padding value can be omitted, and if omitted, the compiler is padded with a value of 0. The third optional parameter value `val3` is used to indicate the maximum number that can be used for padding or skipping. If the boundary alignment exceeds the maximum number of bytes specified by `val3`, then no alignment is done at all. If you need to omit the second parameter `val2` but still need to use the third parameter `val3`, you only need to put two commas.

For programs that now use the ELF object format, the first parameter, `val1`, is the number of bytes that need to be aligned. For example, `'align 8'` means to adjust the position counter until it points to a multiple of 8 boundaries. If it is already on the multiple of 8, then the compiler does not have to change. However, for the

system using the a.out object format here, the first parameter `val1` is the number of the lower 0 bits, that is, the power of 2 ( $2^{\text{val1}}$ ). For example, `'align 3'` in the previous program `head.s` means that the position counter needs to be on the multiple of 8 boundaries. Again, if it is already on the multiple of 8 boundaries, then the directive does nothing. GNU `as(gas)` treats these two target formats differently because `Gas` is formed to mimic the behavior of the assembler that comes with various architecture systems.

## 6.5 Summary

The boot loader `bootsect.S` loads the `setup.s` code and the system module into memory, and moves its self and `setup.s` code to physical memory `0x90000` and `0x90200` respectively, and then delegates the execution to the setup program. The header of the system module contains the `header.s` code.

The main function of the setup program is to use the ROM BIOS interrupt program to get some basic parameters of the machine, and save it in the memory block starting at `0x90000` for later programs. At the same time, move the system module down to the beginning of the physical address `0x00000`, so the `head.s` code in the system is at the beginning of `0x00000`. The descriptor table base address is then loaded into the descriptor table register to prepare for operation in 32-bit protected mode. Next, re-build the interrupt control hardware. Finally, set the machine control register `CR0` and jump to the `head.s` code of the system module to start the CPU in 32-bit protected mode.

The main function of the `Head.s` program is to initially initialize the 256-item descriptor in the interrupt descriptor table, check if the `A20` address line is already open, and test whether the system contains a math coprocessor. Then initialize the memory page directory table to prepare for the paging management of the memory. Finally, jump to the initialization program `init/main.c` in the system module to continue execution.

The main content of the next chapter is to describe in detail the function of the `init/main.c` program.





## 7 Initialization program (init)

There is only one main.c file in the init/ directory of the kernel source. The system module will pass execution rights to main.c after executed the code of boot/head.s. Although the program is not long, it includes all the work of kernel initialization. Therefore, when reading the kernel source, you need to refer to the initialization part of many other programs. If you can fully understand all the functions called here, then after reading this chapter you should have a general understanding of how the Linux kernel works.

Starting from this chapter, we will encounter a large number of C language programs, so readers should already have a certain C language knowledge. The best reference book on C should be the "C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie. The understanding of pointers and arrays in Chapter 5 of the book can be said to be the key to understanding the C language. In addition, you need to have the GNU gcc manual as a reference, because gcc's extended features, such as inline functions, inline assembly statements, etc., are used in many places in the kernel source code.

When annotating a C language program, we use `/*` as the starting symbol of the comment statement in order to distinguish it from the original comment in the program. The translation of the original comments uses the same comment mark. For the header file (`*.h`) included in the program, only the meaning of the summary is given. The specific detailed comments on the header file will be given in the corresponding section of the comment header file.

### 7.1 main.c

#### 7.1.1 Function description

The main.c program first uses the machine parameters obtained from the previous setup.s to set the system's root file device number and some memory global variables. These memory variables indicate the starting address of the main memory area, the amount of memory the system has, and the end address of the cache memory. If a virtual disk (RAMDISK) is also defined, the main memory area will be appropriately reduced. A schematic diagram of the entire memory space is shown in Figure 7-1. In the figure, the cache portion also needs to deduct the portions occupied by video memory and its BIOS of the display card. The high-speed cache is used for temporarily storing data from or to a block device such as a disk, and takes 1K (1024) bytes as a block unit. The main memory area is managed and allocated by the memory management module mm through the paging mechanism, and 4K bytes is a memory page unit. The kernel program has free access to the data in the cache, but it needs to pass mm to use the allocated memory page.

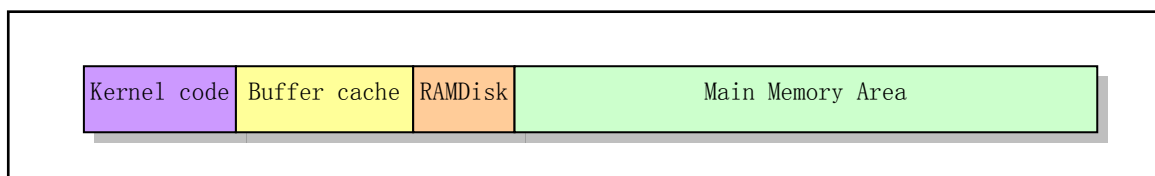


Figure 7-1 Diagram of memory function partitioning in the system

Then, the kernel code performs various aspects of hardware initialization work. This includes trap gates, block devices, character devices, and ttys, as well as manually setting up the first task (task 0). After all initialization work is completed, the kernel sets the interrupt enable flag to turn on the interrupt and switches to task 0 to run. At this point, it can be said that the kernel has basically completed all the setup work. The kernel then creates several initial tasks through task 0, runs the shell program and displays the command line prompt, so that the Linux system is in normal operation.

Note that when reading these initialization subroutines, it is best to go deeper into the program being called. If you really can't understand it, just put it aside and continue to look at the next initialization function. After some understanding, continue to study the places that have not been read.

### 7.1.1.1 Kernel initialization flow diagram

After the entire kernel has finished initializing, the kernel switches execution control to user mode (task 0), that is, the CPU switches from privilege level 0 to privilege level 3. At this point, the main program runs in task 0. Then, the system first calls the process creation function `fork()` to create a child process (init process) for running `init()`. The entire initialization process of the system is shown in Figure 7-2.

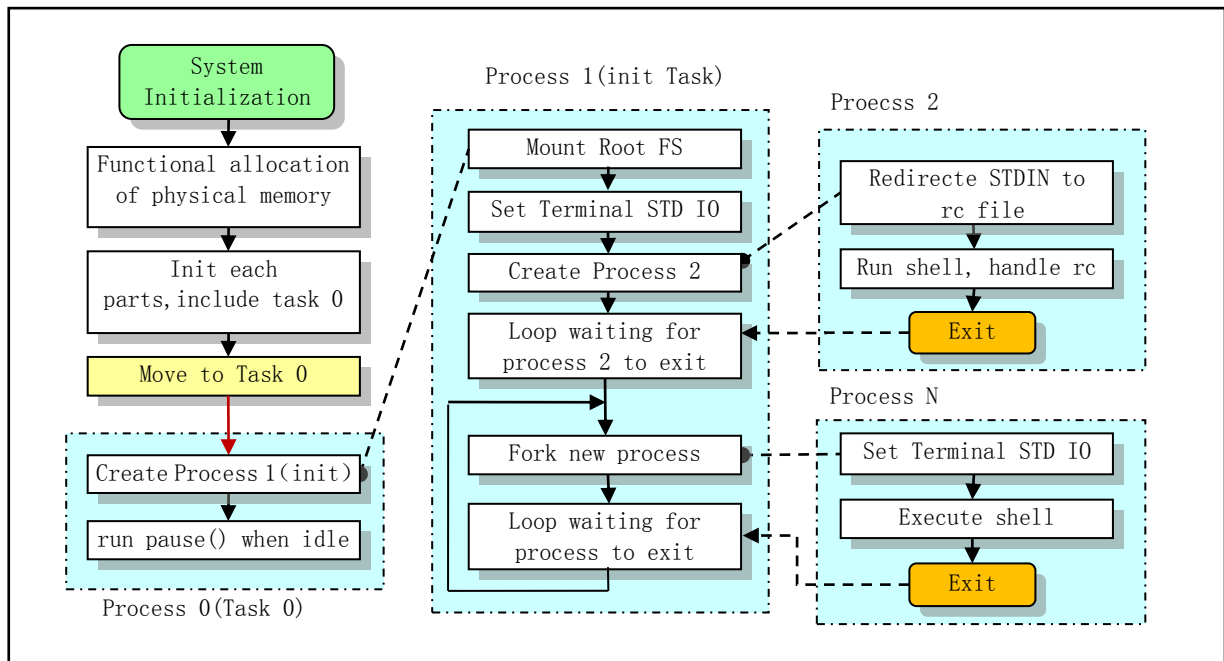


Figure 7-2 Kernel initialization process flow diagram

As can be seen from the figure, the `main.c` program first determines how to allocate the system physical memory, and then calls the initialization functions of each part of the kernel to initialize the memory management, interrupt processing, block device and character device, process management, and hard disk and floppy disk hardware. After these operations are completed, the various parts of the system are already operational. The program then moves itself "manually" to task 0 (process 0) and uses the `fork()` call to create process 1 (init process) for the first time and calls the `init()` function there. In this function the program will continue to initialize the application environment and execute the shell login program. The original process 0 is scheduled to execute when the system is idle, so process 0 is also commonly referred to as the idle process. At this point, process 0 only executes the `pause()` system call and calls the scheduler function.

The function of the `init()` can be divided into four parts: (1) to install the root file system; (2) to display system information; (3) to run the system initial resource configuration file `rc`; (4) to execute the user login shell program.

1. Install root file system

The code first calls the syscall `setup()` to collect the hard disk device partition table information and install the root file system. Before installing the root file system, the system will first determine if you need to create a virtual disk first. If the size of the virtual disk is set when the kernel is compiled, and a memory has been opened for use as a virtual disk during the kernel initialization process, the kernel will first try to load the root file system into the virtual extent of the memory.

2. Display system information

Then `init()` opens a terminal device `tty0` and copies its file descriptor or handler to produce standard input `stdin`, standard output `stdout`, and error output `stderr` device. The kernel then uses these file descriptors to display some system information on the terminal, such as the total number of buffer blocks in the cache, the total number of bytes of free memory in the main memory area, and so on.

3. Run resource configuration file

Next, `init()` creates a new process (Process 2) and performs some initial configuration operations in it to establish a user interaction environment. That is, before the user can use the shell command line environment, the kernel uses the `/bin/sh` program to run the commands set in the configuration file `etc/rc`. The role of the `rc` file is similar to the `AUTOEXEC.BAT` file on the root directory of the DOS operating system. This code first directs the standard input `stdin` to the `etc/rc` file by closing file descriptor 0 and immediately opening the file `/etc/rc` so that all standard input data will be read from the file. The kernel then executes `/bin/sh` in a non-interactive manner to implement the running of commands in the `/etc/rc` file. When the command in the file is executed, `/bin/sh` will exit immediately, so process 2 will end.

4. Execute user login shell

The last part of the `init()` function is used to create a new session for the user in the new process and run the user login shell `/bin/sh`. When the system executes the program in process 2, the parent process (`init` process) waits for its end. With the exit of process 2, the parent process enters an infinite loop. In this loop, the parent process will generate a new process again, then create a new session in the process, and execute the program `/bin/sh` again for login to create a user interaction shell environment. The parent process then continues to wait for the child process. Although the login shell is the same program `/bin/sh` as the previous non-interactive shell, the command line arguments used (`argv[]`) are different. The first character of the 0th command line argument of the login shell must be a minus sign `'-'`. This particular flag tells `/bin/sh` that it is not a normal run, but runs `/bin/sh` as the login shell. From this point on, the user can use the Linux command line environment normally, and the parent process will enter the wait state. Thereafter, if the user executes the `exit` or `logout` command on the command line, after displaying a information of about current login shell exit, the system will repeat the process of creating the login shell process again in this infinite loop.

The last two parts of the `init()` function running in task 1 should actually be the functions of the independent environment initialization program `init`. See the description of this after the program list.

### 7.1.1.2 Operations of the initial user stack

Since the process of creating a new process is implemented by completely copying the parent process code

and data segment, when the new process init is created using `fork()` for the first time, in order to ensure that there is no redundant information of process 0 in the user process stack of the new process, Process 0 should not use its user-mode stack until the first new process (Process 1) is created, that is, Task 0 is not required to call the function. Therefore, after the `main.c` program moves to the execution of task 0, the code `fork()` in task 0 should not be called as a function. The method implemented in the program is to execute this syscall using the gcc function inline form as shown below (see line 23 of the program):

By declaring an inline function, you can have gcc integrate the code of the function into the code that called it. This will speed up code execution because it saves the overhead of function calls. In addition, if any of the actual arguments is a constant, then these known values at compile time may make the code simpler without including all the code for the inline function. See the relevant instructions in Chapter 3.

---

23 static inline `_syscall0`(int, `fork`)

---

Where `_syscall0()` is the inline macro code in `unistd.h`, which calls the Linux system call interrupt `int 0x80` in the form of embedded assembly. According to the macro definition on line 150 of the `include/unistd.h` file, we expand this macro and replace it with the above line. It can be seen that this statement is actually an `int fork()` creation process system call, as shown below.

---

```
// The definition of _syscall0() in the unistd.h file. That is, the system calls the macro
// function without parameters: type name(void).
150 #define _syscall0(type,name) \
151 type name(void) \
152 { \
153 long __res; \
154 __asm__ volatile ("int $0x80" \           // syscall INT 0x80
155                  : "=a" (__res) \       // return value -> eax(__res)
156                  : "0" (__NR_##name)); \ // input is syscall number __NR_name.
157 if (__res >= 0) \                       // If return value >=0, the value is returned.
158     return (type) __res; \
159 errno = -__res; \                       // Otherwise set error number and return -1.
160 return -1; \
161 }
```

---

According to the above definition, we can expand `_syscall0(int, fork)`. After substituting the 23rd line, we can get the following statement:

---

```
static inline int fork(void)
{
    long __res;
    __asm__ volatile ("int $0x80" : "=a" (__res) : "0" (__NR_fork));
    if (__res >= 0)
        return (int) __res;
    errno = -__res;
    return -1;
}
```

---

It can be seen that this is an inline function definition. gcc will insert the statement in the above "function" body

directly into the code that calls the `fork()` statement, so executing `fork()` will not cause a function call. In addition, the last 0 in the macro name string "syscall0" means no parameter, and 1 means 1 parameter. If the system call has 1 parameter, then the macro `_syscall1()` should be used.

Although the above syscall execution interrupt instruction INT can not avoid using the stack, but the syscall uses the kernel state stack of the task instead of the user stack, and each task has its own independent kernel state stack, so the syscall will not affect the user state stack discussed here.

In addition, during the process of creating new process init (process 1), the system has some special handling. In fact, process 0 and process init use the same code and data memory pages (640KB) in the kernel, but the code that is executed is not in one place, so in fact they also use the same user stack area at the same time. When copying the page directory and page table entries of its parent process (process 0) for the new process init, the 640KB page table entry's attribute of process 0 has not been changed (still readable and writable), but the corresponding attribute of process 1 is set to read-only. Therefore, when process 1 begins execution, its access to the user stack will result in a page write protection exception, which will cause the kernel's memory manager (mm) to allocate a memory page for process 1 in the main memory area, and copy the corresponding pages in task 0's stack to this new page. From this point on, the user mode stack of task 1 begins to have its own independent memory page. That is, after accessing the stack from task 1, the user stacks of task 0 and task 1 become independent of each other. Therefore, in order not to cause conflicts, task 0 must be required to prohibit the use of the user stack area before task 1 performs the stack operation.

In addition, because the order in which the kernel schedules each process is random, it is possible to run task 0 first after task 1 is created for task 1. Therefore, after task 0 executes the `fork()` operation, the subsequent `pause()` function must also be implemented in the form of an inline function to avoid task 0 using the user stack before task 1.

When a process in the system (such as the child process of the init process, process 2) has executed the `execve()` call, the code and data of process 2 will be located in the main memory area, so the system can use the copy-on-write technology to handle the creation and execution of other new processes at any time thereafter.

For Linux, all tasks run in user mode, including many system applications, such as shell programs, network subsystem programs, and so on. The library files in the kernel source code `lib/` directory (except the `string.c` program) are designed to provide function support for the newly created processes. The kernel code itself at the 0 privilege level does not use these library functions.

## 7.1.2 Program Annotations

Program 7-1 linux/init/main.c

---

```

1 /*
2  * linux/init/main.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 // unistd.h is a standard symbol constant and type file. It defines various symbol consts
8 // and types and declares various functions. If the symbol __LIBRARY__ is also defined,
9 // the system call number and the inline assembly code syscall0() are also included.
10 #define __LIBRARY__
11 #include <unistd.h>
12 #include <time.h>          // time type header file. tm structure, time function prototypes.
```

```

11 /*
12  * we need this inline - forking from kernel space will result
13  * in NO COPY ON WRITE (!!!), until an execve is executed. This
14  * is no problem, but for the stack. This is handled by not letting
15  * main() use the stack at all after fork(). Thus, no function
16  * calls - which means inline code for fork too, as otherwise we
17  * would use the stack upon exit from 'fork()'.
18  *
19  * Actually only pause and fork are needed inline, so that there
20  * won't be any messing with the stack from main(), but we define
21  * some others too.
22  */
// Linux does not use copy-on-write when creating processes in kernel space. main() uses
// the inline fork() and pause() after moving to user mode (task 0), so it is guaranteed
// that the user stack of task 0 is not used.
// After executing moveto_user_mode(), the main() program is running as task 0. Task 0 is
// the parent of all the child processes that will be created. When it creates a child
// process (task or process 1, init), since the task 1 code is also in kernel space, no
// copy-on-write functionality is used. At this point, task 0 and task 1 use the same user
// stack space together, so you don't want to have any operations on the stack when running
// in task 0 environment to avoid messing up the stack. After executing fork() again and
// executing the execve() function, the loaded program is no longer in kernel space, so you
// can use the copy-on-write technique. See section 5.3, "How the Linux Kernel Uses Memory".

// The following _syscall0() is the inline macro code defined in unistd.h. It calls Linux's
// system call interrupt 0x80 in embedded assembly form. This interrupt is the entry point
// for all syscalls. The statement is actually a creating process syscall int fork(). You
// can expand it and you will understand it immediately. The last 0 in the syscall0 name
// indicates it contains no parameters, and 1 indicates 1 parameter. See include/unistd.h,
// lines 150-161.
// syscall pause() : Suspend process execution until a signal is received.
// syscall setup (void * BIOS): linux initialization (only called in this program).
// syscall sync(): update or synchronize the file system.
23 static inline _syscall0(int, fork)
24 static inline _syscall0(int, pause)
25 static inline _syscall1(int, setup, void *, BIOS)
26 static inline _syscall0(int, sync)
27

// <linux/tty.h> defines parameters and constants for tty_io, serial communication.
// <linux/sched.h> scheduler header file defines the task structure task_struct, the data
// of the initial task 0, and some embedded assembly function macro statements about
// the descriptor parameter settings and acquisition.
// <linux/head.h> head file. A simple structure for the segment descriptor is defined,
// along with several selector constants.
// <asm/system.h> system header file. An embedded assembly macro that defines or modifies
// descriptors/interrupt gates, etc. is defined.
// <asm/io.h> io header file. Defines the function that operates on the io port in the
// form of a macro's embedded assembler.
// <stddef.h> standard definition header file. NULL, offsetof(TYPE, MEMBER) is defined.
// <stdarg.h> Standard parameter file. Define a list of variable parameters in the form
// of macros. It mainly describes one type (va_list) and three macros (va_start, va_arg
// and va_end) for the vsprintf, vprintf, and vfprintf functions.

```

```

// <unistd.h> Linux standard header file. Various symbol constants and types are defined
// and various functions are declared. If __LIBRARY__ is defined, it also includes the
// system call number and the inline assembly _syscall0().
// <fcntl.h> File control header file. The definition of the operation control constant
// symbol used for the file and its descriptors.
// <sys/types.h> type header file. The basic system data types are defined.
// <linux/fs.h> fs header file. Define file table structure (file, buffer_head, m_inode, etc.)
// <string.h> string header file. mainly defines embedded functions for string operations.
28 #include <linux/tty.h> // teletype terminal.
29 #include <linux/sched.h> // scheduler.
30 #include <linux/head.h> // kernel head.
31 #include <asm/system.h> // system machine.
32 #include <asm/io.h> // port io.
33
34 #include <stddef.h> // standard definition.
35 #include <stdarg.h> // standard arguments.
36 #include <unistd.h>
37 #include <fcntl.h> // file control.
38 #include <sys/types.h> // date types.
39
40 #include <linux/fs.h> // file system.
42 #include <string.h> // string operations.
43
44 static char printbuf[1024]; // cache for kernel to display information.
45
46 extern char *strcpy(); // external functions defined elsewhere.
47 extern int vsprintf(); // Formatted output into a string (vsprintf.c, 92).
48 extern void init(void); // Function prototype, init (168).
49 extern void blk_dev_init(void); // Block dev init (blk_drv/ll_rw_blk.c, 210)
50 extern void chr_dev_init(void); // Char dev init(chr_drv/tty_io.c, 402)
51 extern void hd_init(void); // Hard disk init(blk_drv/hd.c, 378)
52 extern void floppy_init(void); // Floppy init (blk_drv/floppy.c, 469)
53 extern void mem_init(long start, long end); // Memory init (mm/memory.c, 443)
54 extern long rd_init(long mem_start, int length); // Ramdisk init (blk_drv/ramdisk.c, 52)
55 extern long kernel_mktime(struct tm * tm); // Calc boot time (kernel/mktime.c, 41)
56
// kernel-specific sprintf() function. This function is used to generate a formatted
// message and output it to the specified buffer str. The parameter '*fmt' specifies the
// format in which the output will be used, refer to the standard C language book. This
// function uses vsprintf() to put the formatted string into the str buffer, see the
// printf() function on line 179.
57 static int sprintf(char * str, const char *fmt, ...)
58 {
59     va_list args;
60     int i;
61
62     va_start(args, fmt);
63     i = vsprintf(str, fmt, args);
64     va_end(args);
65     return i;
66 }
67
68 /*

```

```

69 * This is set up by the setup-routine at boot-time
70 */
// The following three lines forcibly convert the specified linear address to a pointer of
// the given data type and obtain the contents pointed. Since the kernel code segments are
// mapped to locations starting from the physical address zero, these linear addresses are
// also the corresponding physical addresses. See Table 6-3 in Chapter 6 for the meaning of
// the memory values at these specified addresses. See line 125 below for the drive_info
// structure.
71 #define EXT_MEM_K (*(unsigned short *)0x90002) // Extended Mem size(KB) beyond 1MB
72 #define CON_ROWS ((*(unsigned short *)0x9000e) & 0xff) // Console rows and columns.
73 #define CON_COLS (((*(unsigned short *)0x9000e) & 0xff00) >> 8)
74 #define DRIVE_INFO (*(struct drive_info *)0x90080) // Harddisk parameter table(32 bytes)
75 #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC) // Root fs dev number.
76 #define ORIG_SWAP_DEV (*(unsigned short *)0x901FA) // Swap file dev number.
77
78 /*
79 * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
80 * and this seems to work. I anybody has more info on the real-time
81 * clock I'd be interested. Most of this was trial and error, and some
82 * bios-listing reading. Urghh.
83 */
84
// This macro reads the CMOS real time clock information. Outb_p and inb_p are port input
// and output macros defined in include/asm/io.h. 0x70 is the write address port, and 0x71
// is read data port. 0x80|addr is the CMOS memory address to be read.
85 #define CMOS_READ(addr) ({ \
86 outb_p(0x80|addr, 0x70); \ // Output CMOS addr (0x80|addr) to be read to port 0x70
87 inb_p(0x71); \ // Read 1 byte from port 0x71 and return the byte.
88 })
89
// Define a macro to convert a BCD code to a binary value. The BCD code uses a half byte
// (4 bits) to represent a decimal number, so one byte can represent two decimal numbers.
// '(val)&15' takes the decimal digits represented by BCD, and '((val)>>4)*10' takes the
// decimal tens digit of BCD and multiplies it by 10. The last two are added together,
// which is the actual binary value of a byte BCD code.
90 #define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)
91
// This function takes CMOS real clock information as the boot time and saves it to the
// global variable startup_time (seconds). See the description of CMOS memory list later.
// The function kernel_mktime() is used to calculate the number of seconds elapsed from
// 0:00 on January 1, 1970 to the date of boot, as the boot time.
92 static void time_init(void)
93 {
94     struct tm time; // defined in include/time.h
95
// CMOS access is very slow. In order to reduce the time error, after reading all the
// values in the following loop, if the second value changes in CMOS at this time, then
// all the values are read again. This allows the core to control the CMOS time error to
// within 1 second.
96     do {
97         time.tm_sec = CMOS_READ(0); // current time seconds (BCD code).
98         time.tm_min = CMOS_READ(2); // current minutes.
99         time.tm_hour = CMOS_READ(4); // current hours.

```



```

100         time.tm_mday = CMOS_READ(7);    // day in a month.
101         time.tm_mon = CMOS_READ(8);     // current month (1-12).
102         time.tm_year = CMOS_READ(9);    // current year.
103     } while (time.tm_sec != CMOS_READ(0));
104     BCD_TO_BIN(time.tm_sec);             // BCD to binary value.
105     BCD_TO_BIN(time.tm_min);
106     BCD_TO_BIN(time.tm_hour);
107     BCD_TO_BIN(time.tm_mday);
108     BCD_TO_BIN(time.tm_mon);
109     BCD_TO_BIN(time.tm_year);
110     time.tm_mon--;                       // range of months in tm_mon is 0-11.
111     startup_time = kernel_mktime(&time); // calc boot time (kernel/mktime.c, 41)
112 }
113
114 // Defines some static variables that can only be accessed by main.c source file.
115 static long memory_end = 0;             // machine physical memory size (bytes).
116 static long buffer_memory_end = 0;     // buffer end address.
117 static long main_memory_start = 0;     // the main memory start position.
118 static char term[32];                  // terminal settings (environment parameter).
119
120 // Command line and environment parameters used when manipulating the /etc/rc file.
121 static char * argv_rc[] = { "/bin/sh", NULL }; // parameter string array
122 static char * envp_rc[] = { "HOME=/", NULL, NULL }; // environment string array
123
124 // The command line arguments and environment parameters used to run the login shell.
125 // The character "-" in argv[0] on line 122 is a flag passed to the shell program sh. By
126 // identifying this flag, the sh program is executed as a login shell. Its execution is
127 // not the same as executing sh at the shell prompt.
128 static char * argv[] = { "-/bin/sh", NULL }; // as above.
129 static char * envp[] = { "HOME=/usr/root", NULL, NULL };
130
131 struct drive_info { char dummy[32]; } drive_info; // hard disk parameter table.
132
133 // Main() is the kernel initialization main program. After the initialization is
134 // completed, it will run in task 0 (idle task is idle task).
135 void main(void) /* This really IS void, no error here. */
136 { /* The startup routine assumes (well, ...) this */
137 /*
138  * Interrupts are still disabled. Do necessary setups, then
139  * enable them
140  */
141 // First save the root file system device number and the swap file device number. Then set
142 // the console screen row and column number environment variable TERM according to the
143 // information obtained in the setup.s program. Then use it to set the environment
144 // variables used by the etc/rc file and the shell program in the init process. Then copy
145 // the hard disk parameter table at memory 0x90080.
146 // ROOT_DEV has been declared as extern int on line 206 in linux/fs.h file, and SWAP_DEV
147 // has the same declaration in the linux/mm.h file. The mm.h file here is not explicitly
148 // listed in front of the program, as it is already included in the linux/sched.h file
149 // included earlier.
150 ROOT_DEV = ORIG_ROOT_DEV; // ROOT_DEV defined in fs/super.c, 29
151 SWAP_DEV = ORIG_SWAP_DEV; // SWAP_DEV defined in mm/swap.c, 36
152 sprintf(term, "TERM=con%d%d", CON_COLS, CON_ROWS);

```

```

136     envp[1] = term;
137     envp_rc[1] = term;
138     drive_info = DRIVE_INFO;           // Copy hd parameter table at 0x90080

// The location of cache and main memory area is then set according to the physical memory
// capacity of the machine.
// Cache end address -> buffer_memory_end; Machine memory capacity -> memory_end;
// Main memory start address -> main_memory_start.
139     memory_end = (1<<20) + (EXT_MEM_K<<10); // mem size = 1MB + extended mem (bytes)
140     memory_end &= 0xfffff000;               // Ignore less than 4KB (1 page)
141     if (memory_end > 16*1024*1024)          // If mem size > 16MB, calculated as 16MB
142         memory_end = 16*1024*1024;
143     if (memory_end > 12*1024*1024)          // If mem size > 12MB, set buffer end = 4MB
144         buffer_memory_end = 4*1024*1024;
145     else if (memory_end > 6*1024*1024)      // if size > 6Mb, set buffer end = 2Mb
146         buffer_memory_end = 2*1024*1024;
147     else
148         buffer_memory_end = 1*1024*1024; // Otherwise set buffer end = 1Mb
149     main_memory_start = buffer_memory_end;

// If symbol RAMDISK is defined in the Makefile, the virtual disk is initialized. At this
// point, the main memory area will be reduced. See kernel/blk_drv/ramdisk.c.
150 #ifdef RAMDISK
151     main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
152 #endif
// The following is the initialization of all aspects of the kernel. It is better to
// follow into each init function when reading.
153     mem_init(main_memory_start, memory_end); // Main mem area (mm/memory.c, 443)
154     trap_init();                          // trap gate (hw vector) init (kernel/traps.c, 181)
155     blk_dev_init();                       // block dev init (blk_drv/ll_rw_blk.c, 210)
156     chr_dev_init();                      // char dev init (chr_drv/tty_io.c, 402)
157     tty_init();                          // tty init (chr_drv/tty_io.c, 105)
158     time_init();                         // setting boot time (line 92)
159     sched_init();                       // scheduler init (kernel/sched.c, 385)
160     buffer_init(buffer_memory_end);      // buffer init(fs/buffer.c, 348)
161     hd_init();                          // harddisk init (blk_drv/hd.c, 378)
162     floppy_init();                      // floppy init(blk_drv/floppy.c, 469)
163     sti();                             // All init has completed, so enable interrupt.

// The task 0 is started by manipulating data in the stack and using interrupt return
// instruction. Then immediately derive a new task 1 (known as init process) in task 0
// and execute init() in the new task. For the child process being created, fork() will
// return 0, and for the original process (parent process) it will return the process
// number pid of the child process.
164     move_to_user_mode();                // see head file include/asm/system.h.
165     if (!fork()) {                      /* we count on this going ok */
166         init();                        // run init() in task 1 (init process).
167     }

// The following code runs in task 0.
168 /*
169  * NOTE!! For any other task 'pause()' would mean we have to get a
170  * signal to awaken, but task0 is the sole exception (see 'schedule()')

```

---

```

171  * as task 0 gets activated at every idle moment (when no other tasks
172  * can run). For task0 'pause()' just means we go check if some other
173  * task can run, and if not we return here.
174  */
    // The pause() syscall converts task 0 into an interruptible wait state before executing
    // the scheduler function. However, the scheduler will switch back to task 0 as long as it
    // finds that no other tasks in the system can run, and does not depend on the state of
    // task 0. See (kernel/sched.c, 144).
175      for(;;)
176          __asm__ ("int $0x80::\"a\" (\_\_NR\_pause):\"ax\");          // syscall pause()
177  }
178
    // The printf() function produces formatting information and outputs it to the standard
    // output device stdout(1) for display. The parameter '*fmt' specifies the format used for
    // the output, see the standard C language book. The program uses vsprintf() to put the
    // formatted string into the printbuf buffer and then use write() to output the contents
    // to stdout. See the kernel/vsprintf.c for the implementation of vsprintf() function.
179  static int printf(const char *fmt, ...)
180  {
181      va\_list args;
182      int i;
183
184      va\_start(args, fmt);
185      write(1, printbuf, i=vsprintf(printbuf, fmt, args));
186      va\_end(args);
187      return i;
188  }
189
    // The init() function runs in the newly created process 1 (or init process). It first
    // initializes the environment of the first program to be executed (shell), then loads the
    // program as a login shell and executes it.
190  void init(void)
191  {
192      int pid, i;
193
    // setup() is a syscall that reads the hard disk parameters including the partition
    // table information and loads the virtual disk (if it exists) and installs the root file
    // system device. This function is defined with a macro on line 25, and the corresponding
    // function is sys_setup(). See kernel/blk_drv/hd.c, line 74 for its implementation.
194      setup((void *) &drive\_info);

    // The terminal console device "/dev/tty0" is opened in read/write mode. Since this is the
    // first time the system opened a file, the resulting file handle number (file descriptor)
    // is definitely 0. The file handle (0) corresponds to the default console standard input
    // device stdin of the UNIX-like operating system. It is copied here and opened separately
    // in read and write to produce a standard output handle stdout (1) and a standard error
    // output handle stderr (2). The "(void)" prefix in front of the function is used to force
    // the function to return no value.
195      (void) open("/dev/tty1", O\_RDWR, 0);
196      (void) dup(0);          // duplicate file handle 0 to produce stdout (1).
197      (void) dup(0);          // duplicate file handle 0 to produce stderr (2).

```

---

```

// Print the buffer blocks (1024 bytes per block) and total number of bytes, as well as
// free bytes in the main memory area.
298     printf("%d buffers = %d bytes buffer space\n\r", NR_BUFFERS,
299             NR_BUFFERS*BLOCK_SIZE);
300     printf("Free mem: %d bytes\n\r", memory_end-main_memory_start);

// Next create a child process (task 2) and run the commands in the /etc/rc file in that
// child process. For the child process being created, fork() will return 0, and for the
// original process it will return the process number pid of the child process. So lines
// 202-206 are the code that is executed in the child process. The child process code
// first redirects the standard input (stdin) to the /etc/rc file, then runs the /bin/sh
// program with the execve(). The program reads the commands in the rc file from stdin and
// executes them in an interpreted manner. The parameters and environment variables used
// by the sh are given by the argv_rc and envp_rc arrays, respectively.
// Closing handle 0 and immediately opening the /etc/rc file redirects the standard input
// stdin to the /etc/rc file. This allows you to read the contents of the /etc/rc file
// through a console read operation. Since sh runs non-interactively, it will exit
// immediately after executing the rc file, and process 2 will end. For a description of
// the execve(), see the fs/exec.c program, line 207. Error code when function _exit()
// exits: 1- Operation is not permitted; 2-File or directory does not exist.
301     if (!(pid=fork())) {
302         close(0);
303         if (open("/etc/rc", O_RDONLY, 0))
304             _exit(1); // open failed, (lib/_exit.c, 10)
305         execve("/bin/sh", argv_rc, envp_rc); // execute program /bin/sh
306         _exit(2);
307     }

// Below is the statement executed by process 1. wait() waits for the child process to
// stop or terminate, and the return value should be the process identifier (pid) of the
// child process. The following code acts as the parent process waiting for the end of the
// child process. &i stores the return status information. If the return value of wait()
// is not equal to the child process id, continue to wait.
308     if (pid>0)
309         while (pid != wait(&i))
310             /* nothing */;

// If the code is executed here, the child process just created has finished executing rc
// file (or file does not exist), so the child process automatically stops or terminates.
// In the loop below, a child process is created again to run login and console shell program.
// The new child process will first close all previously leftover handles and create a new
// session. Then reopen /dev/tty0 as stdin and copy it into stdout and stderr, and execute
// the /bin/sh program again. But this time the arguments and environment arrays used by
// the shell are another set (see lines 122--123 above). At this point the parent process
// (process 1) runs wait() again. If the child process stops once again, an error message
// is displayed, and then the code continue to try..., forming a "big" infinite running loop.
311     while (1) {
312         if ((pid=fork())<0) {
313             printf("Fork failed in init\r\n");
314             continue;
315         }
316         if (!pid) { // new process
317             close(0);close(1);close(2);

```

```

218         setsid(); // create a new session.
219         (void) open("/dev/tty1", O_RDWR, 0);
220         (void) dup(0);
221         (void) dup(0);
222         _exit(execve("/bin/sh", argv, envp));
223     }
224     while (1)
225         if (pid == wait(&i))
226             break;
227     printf("\\n\\rchild %d died with code %04x\\n\\r", pid, i);
228     sync(); // flush the buffer.
229 }
230 _exit(0); /* NOTE! _exit, not exit() */
// Both _exit() and exit() are used to terminate a function normally. But _exit() is
// directly a sys_exit syscall, and exit() is usually a function in the normal library.
// It performs some cleanup operations, such as executing each termination code, closing
// all standard IOs, etc., and then calling sys_exit.
231 }
232

```

## 7.1.3 Reference Information

### 7.1.3.1 CMOS

The CMOS memory of a PC is a 64- or 128-byte memory block powered by a battery, usually part of the real-time clock (RTC) chip. Some machines have a larger CMOS memory capacity. The 64-byte CMOS was originally used on IBM PC-XT machines to store clock and date information in a BCD format. Since this information only uses 14 bytes, the remaining bytes can be used to store some system configuration data.

The CMOS address space is outside the base memory address space, so executable code is not included. To access it you need to go through I/O ports. 0x70 is the address port and 0x71 is the data port. In order to read the byte of the specified offset, the OUT instruction must first be used to send the offset value of the byte to port 0x70, and then use IN instruction to read the byte from data port 0x71. Similarly, for a write operation, it is first necessary to send the offset of the byte to port 0x70, and then write the data to port 0x71.

The main.c program line 86 statement does not need to perform an OR operation for the byte address with 0x80. Because the CMOS memory capacity at that time has not exceeded 128 bytes, it does not have any effect on OR operation with 0x80. After the kernel 1.0, the operation is removed (see the code from line 42 of the kernel driver/block/hd.c in v1.0). Table 7-1 is a short list of CMOS memory information.

Table 7-1 CMOS 64-byte information profile

Address	Description	Address	Description
0x00	Current Seconds (real clock)	0x11	Reserved
0x01	Alarm Seconds	0x12	Hard drive type
0x02	Current Minutes (real clock)	0x13	Reserved
0x03	Alarm Seconds	0x14	Device byte
0x04	Current Hours (real clock)	0x15	Basic memory (low byte)
0x05	Alarm Hours	0x16	Basic memory (high byte)
0x06	Current day of the week (real clock)	0x17	Extended memory (low byte)

0x07	Date of the day in a month (real clock)	0x18	Extended memory (high byte)
0x08	Current month (real clock)	0x19-0x2d	Reserved
0x09	Current year (real time clock)	0x2e	Checksum (low byte)
0x0a	RTC Status Register A	0x2f	Checksum (high byte)
0x0b	RTC Status Register B	0x30	Extended memory above 1MB (low byte)
0x0c	RTC Status Register C	0x31	Extended memory above 1MB (high byte)
0x0d	RTC Status Register D	0x32	Current century
0x0e	POST diagnostic status byte	0x33	Information flag
0x0f	Shutdown status byte	0x34-0x3f	Reserved
0x10	Floppy disk drive type		

### 7.1.3.2 Forking a new process

Fork() is a system call function that copies the current process and creates a new entry in the process table that is almost identical to the original process (called the parent process) and executes the same code. But the new process (child process) has its own data space and environment parameters. The main purpose of creating a new process is to run the program concurrently, or use the exec() cluster function to execute other different programs in the new process.

At the return position of the fork, the parent process will resume execution, and the child process will just begin execution. In the parent process, the call to fork() returns the process ID of the child process, and in the child process fork() returns a value of 0. In this way, although it is still executed in the same program at this time, they have started to fork and each executes its own code. If the fork() call fails, a value less than 0 is returned. As shown in Figure 7-3.

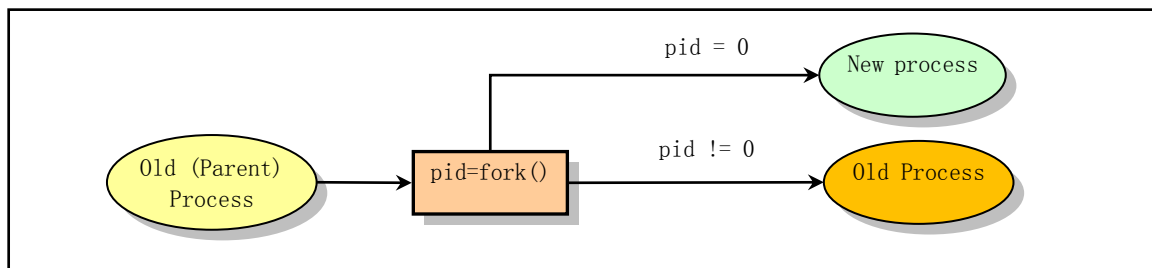


Figure 7-3 Call fork() to create a new process

The init() uses the return value of the fork() to distinguish and execute different code parts. Lines 201 and 216 in the main.c program are the code blocks that the child process starts executing (using the execve() system call to execute other programs, where sh is executed), and lines 208 and 224 are the code blocks executed by the parent process.

exit() can be called to exit the execution of the program when the program has finished or is necessary to terminate. This function terminates the process and releases the kernel resources it occupies. The parent process can use the wait() call to view or wait for the child process to exit, and obtain the exit status information of the terminated process.

### 7.1.3.3 Concept of session

As we said earlier, a program is an executable file, and a process is an instance of a program that is executing. In the kernel, each process is identified by a different positive integer greater than zero, called the process identification number pid (Process ID). A process can create one or more child processes through fork(),

and these processes can form a process group. For example, for a pipe command typed on the shell command line,

---

```
[root]# cat main.c | grep for | more
```

---

Each of these commands: cat, grep, and more belong to a process group.

A process group is a collection of one or more processes. Similar to a process, each process group has a unique process group identification number gid (Group ID), and it is also a positive integer. Each process group has a process called a group leader. The group leader process is a process whose pid is equal to the process group number gid. A process can participate in an existing process group or create a new process group by calling `setpgid()`. The concept of process group has many uses, but the most common one is that we issue a termination signal to the foreground execution program on the terminal (usually by pressing the Ctrl-C key) and terminate all processes in the entire process group. For example, if we issue a termination signal to the above pipe command, the three commands will terminate execution at the same time.

A session is a collection of one or more process groups. Normally, all the programs executed after the user logs in belong to a session, and the login shell is the session leader, and the terminal used by it is the control terminal of the session. Therefore, the first process of a session is also commonly referred to as a Controlling process. When we log out, all processes belonging to our session will be terminated, and this is one of the main uses of the concept of the session. The `setsid()` function is used to create a new session, usually called by the environment initializer. The relationship between processes, process groups, and session periods is shown in Figure 7-4.

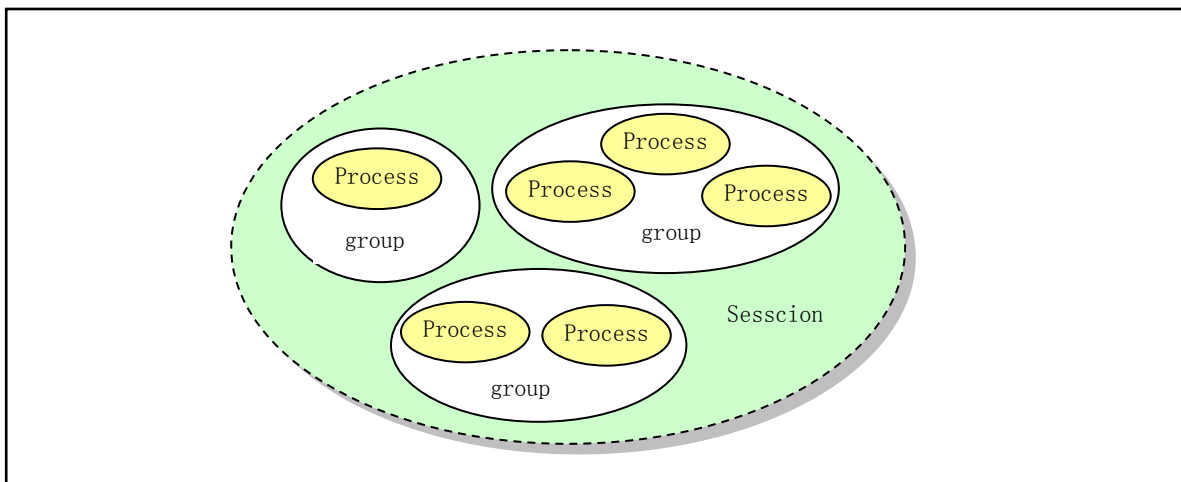


Figure 7-4 Relationship between processes, process groups, and sessions

Several process groups in a session are divided into a foreground process group and one or several background process groups. A terminal can only act as a control terminal for a session. A foreground process group is a process group that has a control terminal in the session, and other process groups in the session become a background process group. The control terminal corresponds to the `/dev/tty` device file, so if a process needs to access the control terminal, you can directly read and write the `/dev/tty` file.

## 7.2 Environment initialization

After the kernel is initialized, the system also needs to perform further environment initialization according to the specific configuration, in order to truly have the working environment of a common system. On lines 205 and 222 above, the `init()` function directly executes the command interpreter (shell) `/bin/sh`, which is not the case in actually available systems. In order to have the login function and the ability of multiple people to use the system at the same time, the usual system is to execute the system environment initialization program `init.c` here or in a similar place, and the program will be based on the setting information of the configuration file in the `/etc/` directory. Each terminal device supported in the system creates a child process, and runs the terminal initialization setting program `agetty` (collectively called `getty`) in the child process. The `getty` will display the user login prompt message "login:" on the terminal. When the user types in the username, `getty` is replaced with the login program. After verifying the correctness of the user's input password, the login program finally calls the shell program and enters the shell interaction interface. The execution relationship between them is shown in Figure 7-5.

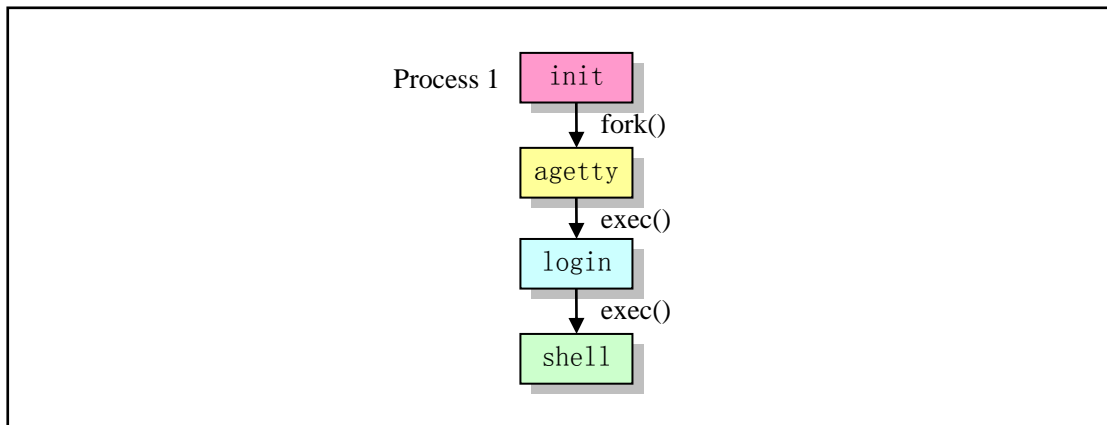


Figure 7-5 Procedures for environment initialization

Although these programs (`init`, `getty`, `login`, `shell`) are not part of the kernel, a basic understanding of their role will facilitate understanding of the many functions that the kernel provides.

The main job of `init` process is to execute the commands set in `/etc/rc` file, and then create a child process for each terminal device that is allowed to log in using `fork()` according to the information in the `/etc/inittab` file, and run the `agetty` (`getty`) program in each newly created child process. At this point, the `init` process calls `wait()` and waits for the end of the child process. Whenever one of its child processes ends and exits, it will know which sub-process of the corresponding terminal is finished according to the `pid` returned by `wait()`, so a new sub-process will be created for the corresponding terminal device, and re-execute the `agetty` program in the child process. In this way, each allowed terminal device always has a corresponding process waiting for processing.

Under normal operation, `init` determines that `agetty` is working to allow the user to log in and to collect orphaned processes. An orphan process is one that has its parent process ended. All processes in Linux must belong to a single process tree, so orphaned processes must be gathered. When the system is shut down, `init` is responsible for killing all other processes, unmounting all filesystems and stopping the processor, as well as any other work that is configured to do.



The main task of the `getty` program is to set the terminal type, properties, speed, and line discipline. It opens and initializes a tty port, displays a prompt, and waits for the user to type in the username. This program can only be executed by the superuser. Usually, if the `/etc/issue` text file exists, `getty` will first display the text message, then display the login prompt information (for example: `plinux login:` ), read the login name entered by the user, and then execute the login program.

The login program is mainly used to require the login user to enter a password. According to the user name entered by the user, it obtains the corresponding user's login item from the password file `passwd`, then calls `getpass()` to display the "password:" prompt message, reads the password typed by the user, and then uses the encryption algorithm to the input password, and compared to the `pw_passwd` field in the user entry in the password file. If the password entered by the user is invalid, the login program will exit with the error code 1, indicating that the login process failed. At this time, the `wait()` of the parent process (process `init`) will return the pid of the exit process, so `init` will create a child process again according to the recorded information, and execute the `agetty` program again for the terminal device in the child process. This repeats the above process.

If the password entered by the user is correct, login will change the current working directory to the user's starting working directory specified in the password file, and modify the access rights to the terminal device to user read/write and group write, and set the group ID of the process. Then use the information obtained to initialize environment variable information, such as the starting directory (`HOME=`), the shell program used (`SHELL=`), the user name (`USER=` and `LOGNAME=`), and the default path sequence of the system executive (`PATH=`). The text message in the `/etc/motd` file (message-of-the-day) is then displayed and checked to see if the user has mail information. Finally, the login program changes to the user ID of the logged in user and executes the shell program specified in the user item in the password file, such as `bash` or `csh`.

If the shell name in the password file `/etc/passwd` does not specify which shell to use, the system will use the default `/bin/sh` program. If the user's home directory is not specified, the default root directory `/` will be used. For a description of some of the execution options and special access restrictions for the login program, see the online manual page (`man 8 login`) on Linux.

A shell program is a complex command-line interpreter that is executed when a user logs into the system for interactive operations. It is where the user interacts with the computer. It takes the information entered by the user and then executes the command. The user can interact directly to the shell on the terminal, or use the shell script file to input to the shell.

When starts executing the shell during login, the first character of the parameter `argv[0]` is '-', indicating that the shell is executed as a login shell. At this point, the shell program will perform some operations corresponding to the login process based on the character. The login shell will first read the command from the `/etc/profile` file and the `.profile` file (if it exists) and execute it. If the `ENV` environment variable is set when entering the shell, or if the variable is set in the `.profile` file when logging in to the shell, the shell will next read the command from the file and execute it. Therefore, the user should put the command to be executed at login time in the `.profile` file, and put the command to be executed every time the shell is executed in the file specified by the `ENV` variable. The way to set the `ENV` environment variable is to put the following statement in the `.profile` file in your home directory:

---

```
ENV=$HOME/.anyfilename; export ENV
```

---

When executing the shell, in addition to some of the specified options, if the command line argument is

also specified, the shell will treat the first argument as a script filename and execute the command. The rest of the parameters are treated as shell parameters (\$1, \$2, etc.). Otherwise the shell program will read the command from its standard input.

There are many options for executing a shell program, see the instructions in the online manual page for `sh` on a Linux system.

## 7.3 Summary

Through the code analysis in this chapter, we know that for the kernel 0.12, as long as the root file system is a MINIX, and it contains files `/etc/rc`, `/bin/sh`, `/dev/*`, and some directories (`/etc/`, `/dev/`, `/bin/`, `/home/`, `/home/root/`), we can form a simple root file system to make the Linux system run.

From here on, for the reading of subsequent chapters, the `main.c` program can be used as a main line, and does not need to be read in chapter order. If the reader does not understand the memory paging mechanism, it is recommended to first read the contents of chapter 10 memory management.

In order to understand the contents of the following chapters smoothly, the author strongly hopes that readers can review the mechanism of 32-bit protection mode operation at this time. Read the relevant contents in the appendix in detail, or refer to the Intel 80x86 books for protection mode.

If you get here in order of chapters, then you should have a general understanding of the initialization process of the Linux kernel. But you might also ask the question: "After generating a series of processes, how does the system run these processes in a time-sharing manner or how to schedule them to run? That is how the 'wheels' turn around?" . The answer is not complicated: the kernel is run by executing the `schedule()` and the timer clock interrupt handler `_timer_interrupt`. The kernel sets a clock interrupt every 10 milliseconds, and during the interrupt processing, the next state of the process is determined by calling the `do_timer()` function to check the current execution of all processes. According to the state of each process, the scheduler schedules each process to execute sequentially.

















When the process needs to wait for a while due to the temporary lack of resources, it will indirectly call the `schedule()` through the `sleep_on()` to voluntarily hand over the CPU rights to the other processes. As for which process the system will run next, it is completely determined by `schedule()` according to the current state and priority of all processes. For a process that is always in a runnable state, when the clock interrupt process determines that the time slice it is running has been used up, the process switch operation is performed in `do_timer()`, and the CPU usage rights of the process are reluctantly deprived, and the kernel will schedule other processes to run.

The scheduling function `schedule()` and the clock interrupt handler procedure are one of the important topics in the next chapter.

## 8 Kernel Code (kernel)

The linux/kernel/ directory contains 10 C files and 2 assembly language files, as well as a Makefile, as shown in List 8-1. Comments on the code in three subdirectories will be made in subsequent chapters. This chapter mainly comments on these 12 code files. First, we give a general introduction to the basic functions of all programs, so that we can get a general understanding of the functions implemented by these kernel code and the calling relationship between them, and then make detailed comments on each code file.

List 8-1 linux/kernel/

	Filename	Size	Last Modified Time (GMT)	Description
	<a href="#">blk_drv/</a>		1992-01-16 14:39:00	
	<a href="#">chr_drv/</a>		1992-01-16 14:37:00	
	<a href="#">math/</a>		1992-01-16 14:37:00	
	<a href="#">Makefile</a>	4034 bytes	1992-01-12 19:49:12	
	<a href="#">asm.s</a>	2422 bytes	1991-12-18 16:40:03	
	<a href="#">exit.c</a>	10554 bytes	1992-01-13 21:28:02	
	<a href="#">fork.c</a>	3951 bytes	1992-01-13 21:52:19	
	<a href="#">mktime.c</a>	1461 bytes	1991-10-02 14:16:29	
	<a href="#">panic.c</a>	448 bytes	1991-10-17 14:22:02	
	<a href="#">printk.c</a>	537 bytes	1992-01-10 23:13:59	
	<a href="#">sched.c</a>	9296 bytes	1992-01-12 15:30:13	
	<a href="#">signal.c</a>	5265 bytes	1992-01-10 00:30:25	
	<a href="#">sys.c</a>	12003 bytes	1992-01-11 00:15:19	
	<a href="#">sys_call.s</a>	5704 bytes	1992-01-06 21:10:59	
	<a href="#">traps.c</a>	5090 bytes	1991-12-18 19:14:43	
	<a href="#">vsprintf.c</a>	4800 bytes	1991-10-02 14:16:29	

### 8.1 Main Functions

The code files in this directory can be divided into three categories, one is the hardware (exception) interrupt handler files; the other is the system call service handler files; the third is the general function files such as process scheduling, see Figure 8-1. We now provide a more detailed explanation of the functionality implemented based on this classification.

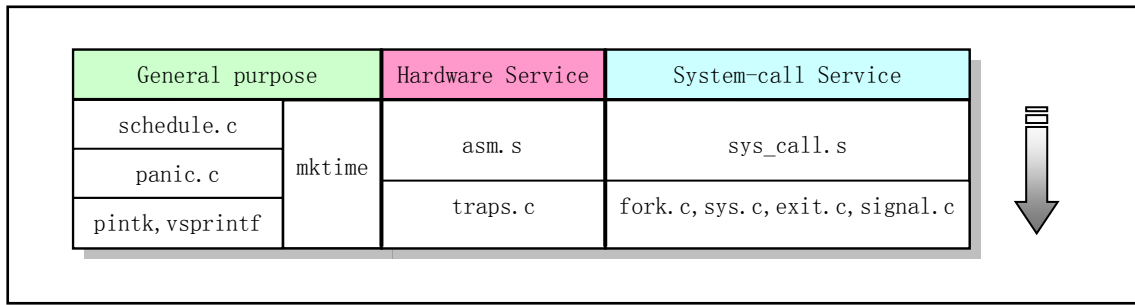


Figure 8-1 Invocation hierarchical relationships between each files

### 8.1.1 Interrupt Processing

There are two files related to interrupt handling: `asm.s` and `traps.c`. `asm.s` is used to implement the assembly processing part of the interrupt service caused by most hardware exceptions, while the `traps.c` implements the C function part called in the `asm.s` interrupt processing. This C function part sometimes called Bottom-halves of the interrupt handling. In addition, several other hardware interrupt handlers are implemented in the files `sys_call.s` and `mm/page.s`. See Figure 5-21 for the connection and function of the 8259A programmable interrupt control chip in the PC.

In Linux systems, the interrupt service function is provided by the kernel, so the interrupt handler uses the kernel state stack of the process. Before the user program (process) passes control to the interrupt handler, the CPU first pushes at least 12 bytes (EFLAGS, CS, and EIP) into the stack of the interrupt handler, that is the kernel state stack of the process. See Figure 8-2(a). This situation is similar to a far call (inter-segment call). The CPU pushes the code segment selector and offset of the return address onto the stack. Another point that is similar to inter-segment calls is that the 80X86 CPU pushes information onto the stack of destination code (interrupt handler code) instead of the interrupted code stack. If the priority level changes, such as from user level to kernel level, the CPU also pushes the stack segment SS and stack pointer ESP of the original code onto the stack of the interrupt handler. But after the kernel is initialized, the kernel code is executed using the kernel state stack of the process, so the stack of the destination code here refers to the kernel state stack of the process, and the stack of interrupted code is of course the user state stack of the process. So when an interrupt occurs, the interrupt handler uses the kernel state stack of the process. In addition, the CPU always pushes the contents of the EFLAGS onto the stack. A schematic diagram of the contents of the stack with priority changes is shown in Figure 8 (c) and (d).

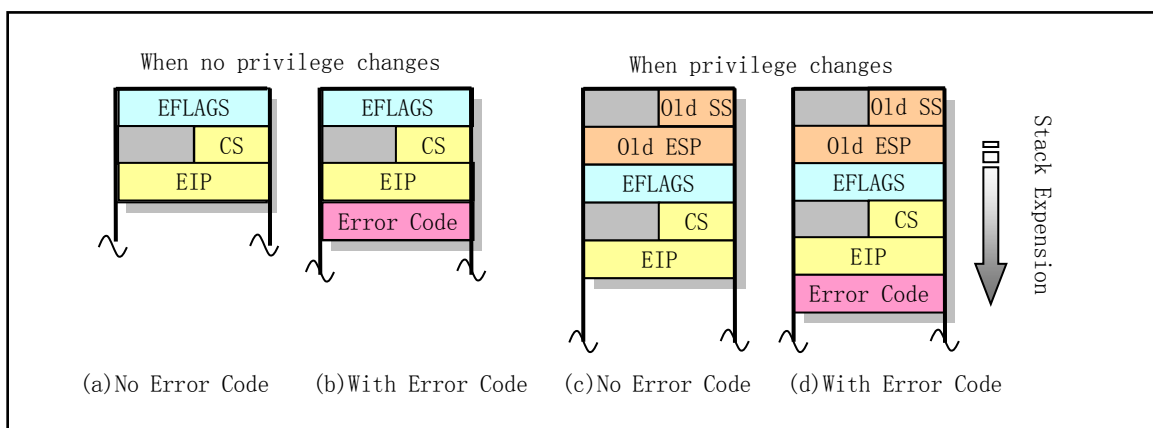


Figure 8-2 The contents of the stack when an interrupt occurs

The `asm.s` code file mainly deals with the processing of Intel's reserved interrupt `INT0--INT16`, and the remaining reserved interrupts (`INT17--INT31`) are reserved for future expansion by Intel Corporation. The processing of 16 interrupts (`INT32--INT47`) corresponding to the IRQ pins of the PIC chip will be set in the initialization routines of various hardware such as clock, keyboard, floppy disk, math coprocessor, hard disk, etc. . The handler for the Linux system call interrupt `INT128 (INT0x80)` will be given in `kernel/sys_call.s`. The specific definition of each interrupt is described in the Reference Information section given after the code file comment.

When some exceptions cause an interrupt, the CPU internally generates an error code that is pushed onto the stack (such as the exception interrupt `INT8` and `INT10 -- INT14`, as shown in Figure 8-2 (b)), while the other interrupts do not carry this error code (for example, a divide-by-zero error and a boundary check error), therefore, the `asm.s` divides the interrupt into two types according to whether or not the error code is carried. But the process is the same as without the error code. The processing of an interrupt caused by a hardware exception is shown in Figure 8-3.

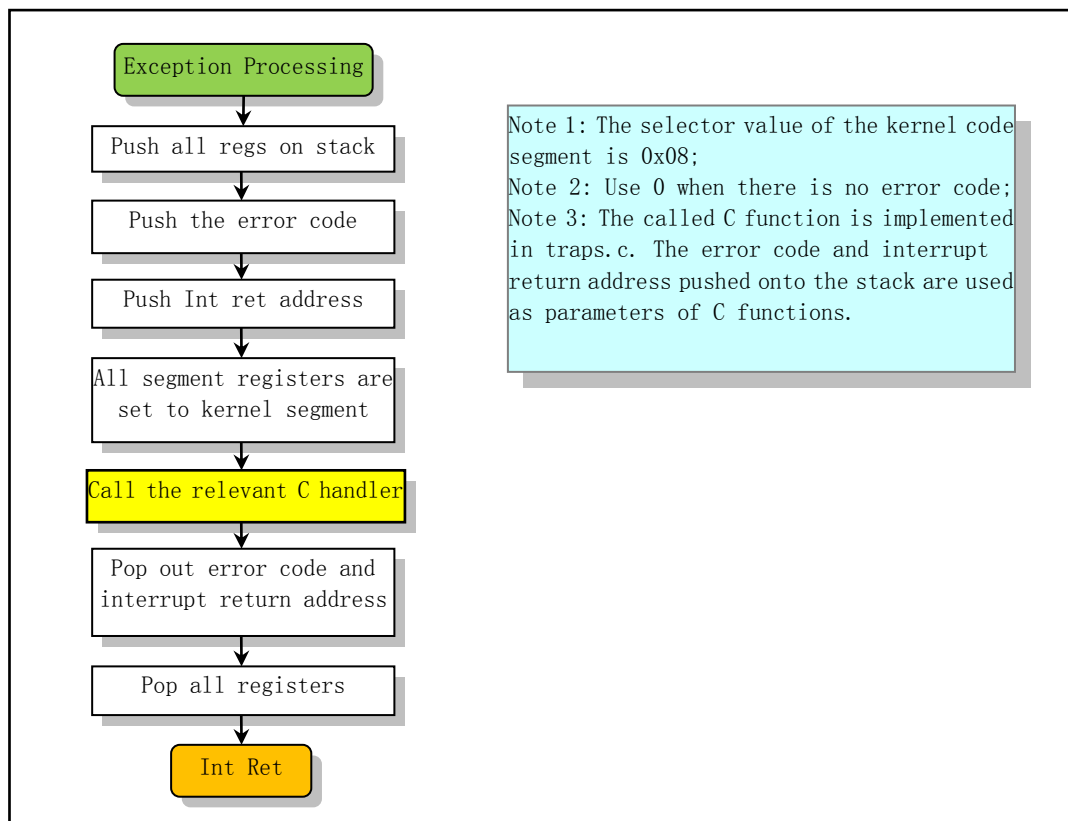


Figure 8-3 Hardware exception (fault, trap) processing flow

### 8.1.2 System-call handling

In Linux, the application needs to use the interrupt `INT 0x80` when using system resources, and the function call number needs to be placed in the register `eax`. If you need to pass parameters to the interrupt handler, you can use the registers `ebx`, `ecx`, and `edx` to store the parameters. Therefore the interrupt call is called a system call (`syscall`). The relevant files for implementing system calls include `sys_call.s`, `fork.c`, `signal.c`, `sys.c`,

and `exit.c` files.

The `sys_call.s` acts like the `asm.s` program in hardware interrupt handling. In addition, the program handles clock interrupts and hard disk and floppy disk interrupts. The interrupt service functions in `fork.c` and `signal.c` are similar to the `traps.c` program, which provide C handlers for system interrupt calls. The `fork.c` provides two C handlers for creating processes: `find_empty_process()` and `copy_process()`. The `signal.c` provides a function `do_signal()` on the processing of the process signal, which is used during system call interrupt processing. It also includes implementations of four system calls.

The `sys.c` and `exit.c` programs implement some other `sys_xxx()` system call functions. These `sys_xxx()` functions are the handlers that are called for the corresponding system call. Some functions are implemented in assembly language, such as `sys_execve()`, while others are implemented in C (for example, four syscalls functions in `signal.c`).

Based on the simple naming conventions for these interrupt-related function's names, we can understand this: Usually the C functions starting with 'do\_' is either a function common to the system calls or a function specific to a system call; , and the functions starting with 'sys\_' is usually a special handler for the specified system call. For example, `do_signal()` is basically a function that all system calls must execute, while `sys_pause()` and `sys_execve()` are C-processor functions specific to a system call.

### 8.1.3 Other general-purpose programs

These programs include `schedule.c`, `mktime.c`, `panic.c`, `printk.c`, and `vsprintf.c`.

`Schedule.c` contains the most frequently used functions `schedule()`, `sleep_on()`, and `wakeup()`. It is the kernel's core scheduler, which is used to switch the execution of a process or change the execution state of a process. It also includes functions for system clock interrupts and floppy drive timing. The `mktime.c` contains only one time function `mktime()` used by the kernel, which is called only once in `init/main.c`. `panic.c` includes a `panic()` function that displays an error message and stops the kernel when an error occurs in kernel. `printk.c` and `vsprintf.c` are kernel supporting programs that implement the kernel-specific display function `printk()` and the string-formatted output function `vsprintf()`.

## 8.2 asm.s

### 8.2.1 Function description

The `asm.s` assembly file contains the underlying code of handler procedures for most of the exceptions detected by the CPU, as well as the exception handling code for the math coprocessor (FPU). This program has a close relationship with `traps.c`. The main processing method of this program is to call the corresponding C function program in `traps.c` in the interrupt handler, display the error location and error code, and then exit the interrupt.

It is helpful to refer to the kernel stack change diagram for the current task in Figure 8-4 when reading this code, where each row represents 4 bytes. For the interrupt process without an error code, refer to Figure 8 4(a) for the changes of the stack pointer position. The stack pointer `esp` refers to the interrupt return address (`esp0` in the figure) before starting the execution of the corresponding interrupt service routine. When the `do_divide_error()` or other C function address is pushed onto the stack, the pointer position is at `esp1`. At this point, the program uses the `swap` instruction to put the address of the function into the `eax` register, and the original `eax` value is saved to the stack. After the program puts some registers onto the stack, the stack pointer position is at `esp2`. Before the formal call to `do_divide_error()`, the program will push the address of the original

eip onto the stack (that is, placed at location esp3) when the interrupt handler begins executing, and by plus 8 to go back to the location esp2 before pop out all registers

For the interrupt that the CPU generates an error code, refer to Figure 8-4(b) for the changes of the stack pointer position. The stack pointer points to esp0 in the figure just before the interrupt service routine is executed. After the do\_double\_fault() or other C function address to be called is pushed onto the stack, the stack pointer location is at esp1. At this time, the program saves the values of the eax and ebx registers at the positions of esp0 and esp1 by using two exchange instructions, and exchanges the error code into the eax registers; the function address is swapped into ebx register. Subsequent processing is the same as in Figure 8-4(a) above.

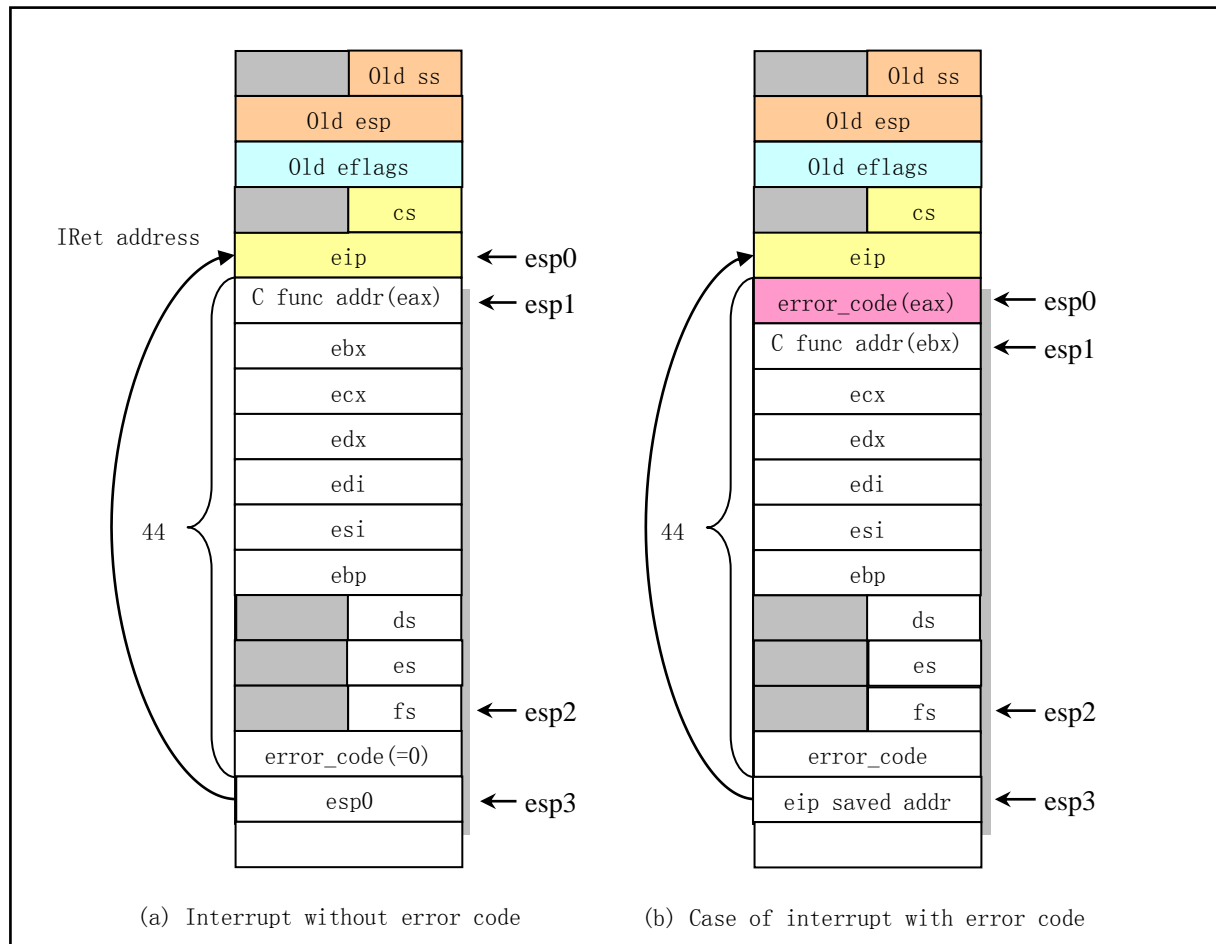


Figure 8-4 Schematic diagram of kernel stack changes during error handling

The reason for putting the error code and esp0 on the stack before the formal invocation to do\_divide\_error() is to use error code and esp0 as parameters do\_divide\_error(). In traps.c, the function signature is:

```
void do_divide_error(long esp, long error_code)
```

Therefore, the position and error code of the error can be printed in this C function. The processing of the remaining exceptions in the file is basically similar to the process described here.

## 8.2.2 Code Annotation

Program 8-1 linux/kernel/asm.s

---

```

1  /*
2  *  linux/kernel/asm.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  asm.s contains the low-level code for most hardware faults.
9  *  page_exception is handled by the mm, so that isn't here. This
10 *  file also handles (hopefully) fpu-exceptions due to TS-bit, as
11 *  the fpu must be properly saved/resored. This hasn't been tested.
12 */
13 # TS - Task Switched, bit-3 in CR0. refer to section 4.1.3
14 # This file mainly deals with the processing of Intel reserved interrupts INT0 -- INT16.
15 # The following are some global function declarations, actually in traps.c.
16 .globl _divide_error, _debug, _nmi, _int3, _overflow, _bounds, _invalid_op
17 .globl _double_fault, _coprocessor_segment_overrun
18 .globl _invalid_TSS, _segment_not_present, _stack_segment
19 .globl _general_protection, _coprocessor_error, _irq13, _reserved
20 .globl _alignment_check
21
22 # The following procedure deals with exceptions without error codes.
23 # Int0 -- Handles errors of divided by zero.      Type: error;   Error code: None.
24 # When executing a DIV or IDIV instruction, if the divisor is 0, the CPU will generate
25 # this exception. This exception is also generated when EAX (or AX, AL) does not hold the
26 # result of a legal divide operation. The label '_do_divide_error' on line 21 is actually
27 # the corresponding name in the object compiled from C function do_divide_error(). This
28 # function is in traps.c, line 101.
29
30 _divide_error:
31     pushl $_do_divide_error      # first push the C function address.
32 no_error_code:                  # no error code processing entry.
33     xchgl %eax, (%esp)           # _do_divide_error in stack exchanged with EAX
34     pushl %ebx
35     pushl %ecx
36     pushl %edx
37     pushl %edi
38     pushl %esi
39     pushl %ebp
40     push %ds                    # occupied 4 bytes in stack.
41     push %es
42     push %fs
43     pushl $0                    # "error code"          # 0 as error code
44     lea 44(%esp), %edx           # get effective addr in esp to edx, points to the
45     pushl %edx                  # ip of original interrupted code (at esp0).
46     movl $0x10, %edx            # Initialize ds, es, fs to kernel data seg selector.
47     mov %dx, %ds
48     mov %dx, %es

```



```

39      mov %dx,%fs
# The '*' in the following statement indicates the register value as an address. This
# statement represents invocation to a routine whos address is specified by the operand.
# This is an indirect call. The meaning of this sentence is to call the C routine
# specified, such as do_divide_error(). After the C function returns, a number 8 is added
# to the stack pointer (line 41), which is equivalent to executing twice pop instructions,
# used to discard the C routine's two parameters (pushed on stack at line 33 and 35), and
# let the stack pointer esp point back to the location of register fs.
40      call *%eax          # like: do_divide_error(long esp, long error_code)
41      addl $8,%esp
42      pop %fs
43      pop %es
44      pop %ds
45      popl %ebp
46      popl %esi
47      popl %edi
48      popl %edx
49      popl %ecx
50      popl %ebx
51      popl %eax          # recover original EAX
52      iret
53
# Int1 -- debug interrupt entry point.   Type: Error/Trap (Fault/Trap); No error code.
# This exception is raised when the TF flag in EFLAGS is set. The CPU generates this
# exception when a hardware breakpoint is found, or an instruction trace trap or task
# swap trap is turned on, or debug register access is invalid (error).
54 _debug:
55      pushl $_do_int3      # _do_debug # push C routine address
56      jmp no_error_code    # line 22.
57
# Int2 - Non Maskable Interrupt entry point.   Type: Trap; No error code.
# This is the only hardware interrupt that is given a fixed interrupt vector. Whenever an
# NMI signal is received, the CPU internally generates an interrupt vector 2 and performs
# a standard interrupt acknowledge cycle, thus saving time. NMI is usually reserved for
# use with extremely important hardware events. When the CPU receives an NMI signal and
# begins executing its interrupt handler, all subsequent hardware interrupts are ignored.
58 _nmi:
59      pushl $_do_nmi
60      jmp no_error_code
61
# Int3 - The breakpoint instruction entry point. Type: Trap; No error code.
# The interrupt caused by the int 3 instruction is independent of the hardware interrupt.
# This instruction is usually inserted into the code by the debugger, and the processing
# method is same as _debug.
62 _int3:
63      pushl $_do_int3
64      jmp no_error_code
65
# Int4 -- Overflow error interrupt entry point.   Type: Trap; no error code.
# This interrupt is raised by the CPU executing the INTO instruction when the OF flag is
# set in EFLAGS. Usually used by compilers to track arithmetic calculation overflows.
66 _overflow:
67      pushl $_do_overflow

```

```

68      jmp no_error_code
69
# Int5 - Bounds check error interrupt entry point.   Type: Error; No error code.
# An interrupt that is raised when the operand is outside the valid range. This interrupt
# is generated when the BOUND instruction fails. The BOUND instruction has 3 operands. If
# the first one is not between the other two, an exception of 5 is generated.
70 _bounds:
71     pushl $_do_bounds
72     jmp no_error_code
73
# Int6 -- Invalid opcode interrupt entry point.   Type: Error; no error code.
# The interrupt caused by the CPU actuator detecting an invalid opcode.
74 _invalid_op:
75     pushl $_do_invalid_op
76     jmp no_error_code
77
# Int9 -- The coprocessor segment overrun entry point.   Type: Abandon; No error code.
# This exception is basically equivalent to coprocessor error protection. Because when
# the floating-point instruction operand is too large, we have the opportunity to load or
# save a floating-point value that exceeds the data segment.
78 _coprocessor_segment_overrun:
79     pushl $_do_coprocessor_segment_overrun
80     jmp no_error_code
81
# Int15 -- The entry point for other interrupts reserved by Intel.
82 _reserved:
83     pushl $_do_reserved
84     jmp no_error_code
85
# Int45 -- (0x20 + 13) Math coprocessor hardware interrupts set by the Linux kernel.
# When the coprocessor performs an operation, it will issue an IRQ13 interrupt signal to
# inform CPU that the operation is complete. When the 80387 is performing calculations,
# CPU waits for its operation to complete. On line 89 below, 0xF0 is the co-processing
# port used to clear the latch. By writing to this port, this interrupt will eliminate the
# CPU's BUSY continuation signal and reactivate the 80387 processor extension request pin
# PEREQ. This operation is mainly to ensure that the CPU responds to this interrupt before
# continuing to execute any instruction of 80387.
86 _irq13:
87     pushl %eax
88     xorb %al,%al
89     outb %al,$0xF0
90     movb $0x20,%al
91     outb %al,$0x20          # sent EOI (End of Interrupt) to 8259's master chip.
92     jmp 1f                 # delay a while.
93 1:     jmp 1f
94 1:     outb %al,$0xA0        # sent EOI to 8259's slave chip.
95     popl %eax
96     jmp _coprocessor_error  # code in system_call.s
97
# When the following interrupt is called, the CPU will push the error code onto the stack
# after interrupting the return address, so the error code will also need to be popped up
# when returning (see Figure 5.3(b)).

```

# Int8 -- double fault. Type: Abandon; There is an error code.  
 # Usually when the CPU invokes a exception handler and detects a new exception, the two  
 # exceptions can be handled serially. However, there are few situations in which the CPU  
 # cannot handle them serially, and the double fault exception is triggered at this time.

```

98 _double_fault:
99     pushl $_do_double_fault    # addr of C routine pushed onto stack.
100 error_code:
101     xchgl %eax, 4(%esp)        # error code <-> %eax, original eax is in stored stack.
102     xchgl %ebx, (%esp)        # &function <-> %ebx, original ebx is stored in stack.
103     pushl %ecx
104     pushl %edx
105     pushl %edi
106     pushl %esi
107     pushl %ebp
108     push %ds
109     push %es
110     push %fs
111     pushl %eax                # error code
112     lea 44(%esp), %eax        # offset
113     pushl %eax
114     movl $0x10, %eax          # set kernel data segment selector.
115     mov %ax, %ds
116     mov %ax, %es
117     mov %ax, %fs
118     call *%ebx                # indirect invocation to C routine
119     addl $8, %esp             # discard used parameters.
120     pop %fs
121     pop %es
122     pop %ds
123     popl %ebp
124     popl %esi
125     popl %edi
126     popl %edx
127     popl %ecx
128     popl %ebx
129     popl %eax
130     iret
131

```

# Int10 -- Invalid task status segment (TSS). Type: Error; there is an error code.  
 # The CPU attempts to switch to a process, and the TSS of the process is invalid.  
 # Depending on which part of the TSS caused an exception, when the TSS length exceeds 104  
 # bytes, this exception is generated in the current task, and the handover is terminated.  
 # Other problems can cause this exception to occur in new tasks after the switch.

```

132 _invalid_TSS:
133     pushl $_do_invalid_TSS
134     jmp error_code
135

```

# Int11 -- The segment does not present. Type: Error; there is an error code.  
 # The referenced segment is not in memory. The flag in the segment descriptor indicates  
 # that the segment is not in memory.

```

136 _segment_not_present:
137     pushl $_do_segment_not_present
138     jmp error_code

```

[139](#)

```
# Int12 -- Stack exception. Type: Error; there is an error code.
# The instruction operation attempted to exceed the stack segment range, or the stack
# segment is not in memory. This is a special case of exceptions 11 and 13. Some operating
# systems can use this exception to determine when more stack space should be allocated
# for the program.
```

[140](#) `_stack_segment:`[141](#) `pushl $_do_stack_segment`[142](#) `jmp error_code`[143](#)

```
# Int13 -- General protection exception. Type: Error; there is an error code.
# Indicates a protection violation that does not belong to any other class. If an exception
# is generated without a corresponding exception vector (0-16), it will usually fall back
# to this class.
```

[144](#) `_general_protection:`[145](#) `pushl $_do_general_protection`[146](#) `jmp error_code`[147](#)

```
# Int17 -- Boundary alignment check error.
# This exception is raised when privilege level 3 (user-level) data is non-boundicallly
# aligned when memory boundary checking is enabled.
```

[148](#) `_alignment_check:`[149](#) `pushl $_do_alignment_check`[150](#) `jmp error_code`[151](#)

```
# int7 -- _device_not_available in file kernel/sys_call.s, line 158.
# int14 -- _page_fault in file mm/page.s, line 14.
# int16 -- _coprocessor_error in file kernel/sys_call.s, line 140.
# int 0x20 -- _timer_interrupt in file kernel/sys_call.s, line 189.
# int 0x80 -- _system_call in file kernel/sys_call.s, line 84.
```

## 8.2.3 Information

### 8.2.3.1 Intel reserved interrupt vector definition

Here is a summary of the Intel reserved interrupt vector, as shown in Table 8–1.

Table 8-1 Exceptions and Interrupts reserved by Intel Co.

Vector No	Name	Type	Error Code	Signal	Source
0	Devide error	Fault (Error)	No	SIGFPE	DIV and IDIV instructions.
1	Debug	Fault/Trap	No	SIGTRAP	Any code or data reference or the INT instruction.
2	nmi	Interrupt	No		Non maskable external interrupt.
3	Breakpoint	Trap	No	SIGTRAP	INT 3 instruction.
4	Overflow	Trap	No	SIGSEGV	INTO instruction.
5	Bounds check	Fault	No	SIGSEGV	BOUND instruction.
6	Invalid Opcode	Fault	No	SIGILL	UD2 instruction or reserved opcode.
7	Device not available	Fault	No	SIGSEGV	Floating-point or WAIT/FWAIT instruction.

8	Double fault	Abort	Yes(0)	SIGSEGV	Any instruction that can generate an exception, NMI, or an INTR.
9	Coprocessor seg overrun	Abort	No	SIGFPE	Floating-point instruction.
10	Invalid TSS	Fault	Yes	SIGSEGV	Task switch or TSS access.
11	Segment not present	Fault	Yes	SIGBUS	Loading segment registers or accessing system segments.
12	Stack segment	Fault	Yes	SIGBUS	Stack operations and SS register loads.
13	General protection	Fault	Yes	SIGSEGV	Any memory reference and other protection checks.
14	Page fault	Fault	Yes	SIGSEGV	Any memory reference.
15	Intel reserved		No		
16	Coprocessor error	Fault	No	SIGFPE	Floating-point or WAIT/FWAIT
17	Alignment check	Fault	Yes(0)		Any data reference in memory.
20-31	Intel reserved.				
32-255	User Defined interrupts	Interrupt			External interrupt or INT n instruction.

## 8.3 traps.c

### 8.3.1 Function Description

The traps.c program mainly includes some C functions used in exception handling, which are used to be called by the exception handling low-level code asm.s. to display debugging message such as error location and error code. The die() generic function is used to display detailed error information in interrupt processing. The final initialization function trap\_init() of the program is called in the previous init/main.c to initialize the hardware exception handling interrupt vector (trap gate) and enable the interrupt request signal to arrive. Please refer to the previous asm.s file when reading this program.

From the beginning of this program, we will encounter many assembly statements embedded in C language programs. See Section 3.3.2 for the basic syntax of embedded assembly statements.

### 8.3.2 Code Annotation

Program 8-2 linux/kernel/traps.c

```

1 /*
2  * linux/kernel/traps.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * 'Traps.c' handles hardware traps and faults after we have saved some
9  * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
10 * to mainly kill the offending process (probably by giving it a signal,
11 * but possibly by killing it outright if necessary).
12 */
13 // <string.h> header file. Mainly defines some embedded functions about string operations.

```

```

// <linux/head.h> Head header file. A simple structure for the segment descriptor is
//     defined, along with several selector constants.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the
//     data of the initial task 0, and some embedded assembly function macro statements
//     about the descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//     commonly used functions of the kernel.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//     descriptors/interrupt gates, etc. is defined.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//     for segment register operations.
// <asm/io.h> Io header file. Defines the function that operates on the io port in the form
//     of a macro's embedded assembler.
13 #include <string.h>
14
15 #include <linux/head.h>
16 #include <linux/sched.h>
17 #include <linux/kernel.h>
18 #include <asm/system.h>
19 #include <asm/segment.h>
20 #include <asm/io.h>
21
// The following statements define three embedded assembly macro functions.
// A compound statement enclosed in parentheses (statements in curly braces) can be used as
// an expression, with the last __res being its output value. Line 23 defines a register
// variable __res. This variable will be saved in a register for quick access and operation.
// If you want to specify a register (such as eax), then we can write the sentence as
// "register char __res asm("ax");". Refer to section 3.3.2 for detailed descriptions.
//
// Function: Take a byte at address addr in segment seg.
// Parameters: seg - the segment selector; addr - the specified address in the segment.
// Output: %0 - eax (__res); Input: %1 - eax (seg); %2 - memory address (*(addr)).
22 #define get_seg_byte(seg, addr) ({ \
23     register char __res; \
24     __asm__( "push %%fs; mov %%ax, %%fs; movb %%fs:%2, %%al; pop %%fs" \
25             : "=a" (__res) : "0" (seg), "m" (*(addr))) ; \
26     __res; })
27
// Function: Take a long word (4 bytes) at the address addr in the segment seg.
// Parameters: seg - the segment selector; addr - the specified address in the segment.
// Output: %0 - eax (__res); Input: %1 - eax (seg); %2 - memory address (*(addr)).
28 #define get_seg_long(seg, addr) ({ \
29     register unsigned long __res; \
30     __asm__( "push %%fs; mov %%ax, %%fs; movl %%fs:%2, %%eax; pop %%fs" \
31             : "=a" (__res) : "0" (seg), "m" (*(addr))) ; \
32     __res; })
33
// Function: Take the value (selector) of the fs segment register .
// Output: %0 - eax (__res).
34 #define fs() ({ \
35     register unsigned short __res; \
36     __asm__( "mov %%fs, %%ax" : "=a" (__res) ); \
37     __res; })

```

```

38 // Some function prototypes are defined below.
39 void page\_exception(void); // page_fault (mm/page.s, 14).
40
41 void divide\_error(void); // Int0 (kernel/asm.s, 20).
42 void debug(void); // int1 (kernel/asm.s, 54).
43 void nmi(void); // int2 (kernel/asm.s, 58).
44 void int3(void); // int3 (kernel/asm.s, 62).
45 void overflow(void); // int4 (kernel/asm.s, 66).
46 void bounds(void); // int5 (kernel/asm.s, 70).
47 void invalid\_op(void); // int6 (kernel/asm.s, 74).
48 void device\_not\_available(void); // int7 (kernel/sys_call.s, 158).
49 void double\_fault(void); // int8 (kernel/asm.s, 98).
50 void coprocessor\_segment\_overrun(void); // int9 (kernel/asm.s, 78).
51 void invalid\_TSS(void); // int10 (kernel/asm.s, 132).
52 void segment\_not\_present(void); // int11 (kernel/asm.s, 136).
53 void stack\_segment(void); // int12 (kernel/asm.s, 140).
54 void general\_protection(void); // int13 (kernel/asm.s, 144).
55 void page\_fault(void); // int14 (mm/page.s, 14).
56 void coprocessor\_error(void); // int16 (kernel/sys_call.s, 140).
57 void reserved(void); // int15 (kernel/asm.s, 82).
58 void parallel\_interrupt(void); // int39 (kernel/sys_call.s, 295).
59 void irq13(void); // int45 (kernel/asm.s, 86) Coprocessor handling.
60 void alignment\_check(void); // int46 (kernel/asm.s, 148).
61
// This subroutine is used to print the error name, error code, program's CS:EIP, EFLAGS,
// ESP, fs segment, segment messages, process pid, task no, and 10-byte instruction code.
// If the stack is in user data segment, then 16 bytes of stack content is also printed
// out. These information can be used for debugging.
// Parameter:
// str - Error name string pointer;
// esp_ptr - Pointer to info of the interrupted program on stack (see esp0 in Figure 8-4);
// nr - Error code. For exception with no error code, this parameter is always 0.
62 static void die(char * str, long esp_ptr, long nr)
63 {
64     long * esp = (long *) esp_ptr;
65     int i;
66
67     printk("s: %04x\n|r", str, nr&0xffff);
// The next statement prints CS:EIP, EFLAGS, and SS:ESP for the currently calling process.
// As can be seen from Figure8-4, here esp[0] is esp0 in the figure. So we take this
// statement apart and look at it as:
// (1) EIP:\t%04x:%p\n -- esp[1] segment selector(CS), esp[0] is EIP;
// (2) EFLAGS:\t%p -- esp[2] is EFLAGS;
// (3) ESP:\t%04x:%p\n -- esp[4] is SS, esp[3] is ESP
68     printk("EIP: \t%04x:%p\nEFLAGS: \t%p\nESP: \t%04x:%p\n",
69         esp[1], esp[0], esp[2], esp[4], esp[3]);
70     printk("fs: %04x\n", fs());
71     printk("base: %p, limit: %p\n", get\_base(current->ldt[1]), get\_limit(0x17));
72     if (esp[4] == 0x17) { // if SS == 0x17, means in user segment,
73         printk("Stack: "); // print 16 bytes data in user stack too.
74         for (i=0; i<4; i++)
75             printk("p ", get\_seg\_long(0x17, i+(long *) esp[3]));

```

```

76         printk("\n");
77     }
78     str(i); // get task no. (include/linux/sched.h, 210)
79     printk("Pid: %d, process nr: %d\n|r", current->pid, 0xffff & i);
80     for(i=0; i<10; i++)
81         printk("%02x ", 0xff & get_seg_byte(esp[1], (i+(char *)esp[0])));
82     printk("\n|r");
83     do_exit(11); /* play segment exception */
84 }
85
86 // The following functions are called by the interrupt handler in asm.s.
87 void do_double_fault(long esp, long error_code)
88 {
89     die("double fault", esp, error_code);
90 }
91 void do_general_protection(long esp, long error_code)
92 {
93     die("general protection", esp, error_code);
94 }
95 void do_alignment_check(long esp, long error_code)
96 {
97     die("alignment check", esp, error_code);
98 }
99 void do_divide_error(long esp, long error_code)
100 {
101     die("divide error", esp, error_code);
102 }
103
104 // These parameters are the register values that are sequentially pushed onto the stack
105 // after entering the interrupt. See lines 24-35 in asm.s file.
106 void do_int3(long * esp, long error_code,
107             long fs, long es, long ds,
108             long ebp, long esi, long edi,
109             long edx, long ecx, long ebx, long eax)
110 {
111     int tr;
112
113     __asm__("str %%ax": "=a" (tr): "" (0)); // get task register TR -> tr
114     printk("eax|t|tebx|t|tecx|t|tedx|n|r%8x|t%8x|t%8x|t%8x|n|r",
115            eax, ebx, ecx, edx);
116     printk("esi|t|tedi|t|tebp|t|tesp|n|r%8x|t%8x|t%8x|t%8x|n|r",
117            esi, edi, ebp, (long) esp);
118     printk("\n|rds|tes|tfs|ttr|n|r%4x|t%4x|t%4x|t%4x|n|r",
119            ds, es, fs, tr);
120     printk("EIP: %8x CS: %4x EFLAGS: %8x|n|r", esp[0], esp[1], esp[2]);
121 }
122 void do_nmi(long esp, long error_code)
123 {
124     die("nmi", esp, error_code);
125 }

```



```
126 }
127
128 void do\_debug(long esp, long error_code)
129 {
130     die("debug", esp, error_code);
131 }
132
133 void do\_overflow(long esp, long error_code)
134 {
135     die("overflow", esp, error_code);
136 }
137
138 void do\_bounds(long esp, long error_code)
139 {
140     die("bounds", esp, error_code);
141 }
142
143 void do\_invalid\_op(long esp, long error_code)
144 {
145     die("invalid operand", esp, error_code);
146 }
147
148 void do\_device\_not\_available(long esp, long error_code)
149 {
150     die("device not available", esp, error_code);
151 }
152
153 void do\_coprocessor\_segment\_overnrun(long esp, long error_code)
154 {
155     die("coprocessor segment overrun", esp, error_code);
156 }
157
158 void do\_invalid\_TSS(long esp, long error_code)
159 {
160     die("invalid TSS", esp, error_code);
161 }
162
163 void do\_segment\_not\_present(long esp, long error_code)
164 {
165     die("segment not present", esp, error_code);
166 }
167
168 void do\_stack\_segment(long esp, long error_code)
169 {
170     die("stack segment", esp, error_code);
171 }
172
173 void do\_coprocessor\_error(long esp, long error_code)
174 {
175     if (last\_task\_used\_math != current)
176         return;
177     die("coprocessor error", esp, error_code);
178 }
```

```

179
180 void do_reserved(long esp, long error_code)
181 {
182     die("reserved (15, 17-47) error", esp, error_code);
183 }
184
// The following are exception (trap) initializers for setting their interrupt call gates
// (vectors) separately. Both set_trap_gate() and set_system_gate() use the Trap Gate in
// the interrupt descriptor table IDT. The main difference between them is that the former
// sets the privilege level to 0 and the latter to 3. Therefore, breakpoint trap int3,
// overflow interrupt, and bounds error interrupt can be called by any program. Both of
// these functions are embedded assembly macros, see include/asm/system.h, lines 36, 39.
185 void trap_init(void)
186 {
187     int i;
188
189     set_trap_gate(0, &divide_error);
190     set_trap_gate(1, &debug);
191     set_trap_gate(2, &nmi);
192     set_system_gate(3, &int3);           /* int3-5 can be called from all */
193     set_system_gate(4, &overflow);
194     set_system_gate(5, &bounds);
195     set_trap_gate(6, &invalid_op);
196     set_trap_gate(7, &device not available);
197     set_trap_gate(8, &double fault);
198     set_trap_gate(9, &coprocessor segment overrun);
199     set_trap_gate(10, &invalid TSS);
200     set_trap_gate(11, &segment not present);
201     set_trap_gate(12, &stack segment);
202     set_trap_gate(13, &general protection);
203     set_trap_gate(14, &page fault);
204     set_trap_gate(15, &reserved);
205     set_trap_gate(16, &coprocessor error);
206     set_trap_gate(17, &alignment check);

// The trap gates of int17--int47 are all set to reserved first, and they will be re-set
// after each hardware initialization.
207     for (i=18; i<48; i++)
208         set_trap_gate(i, &reserved);

// Set the coprocessor int45 (0x20+13) trap gate descriptor below and allow it to generate
// an interrupt request. The coprocessor IRQ13 is connected to the 8259 slave chip's IR5 pin.
// Lines 210-211 are used to allow the coprocessor to send an interrupt request signal. In
// addition, the gate descriptor of the int39 (0x20+7) of the parallel port 1 is also set
// here, and the interrupt request number IRQ7 is connected to the IR7 pin of the 8259
// main chip. Refer to figure 2-6.
//
// The line 210 statement sends an operation command word OCW1 to 8259. This command is
// used to set the 8259 Interrupt Mask Register IMR. 0x21 is the main chip port. The mask
// code is read from it and written immediately after AND 0xfb (0b1111011), indicating
// that the interrupt request mask bit M2 corresponding to the interrupt request IR2 is
// cleared. As shown in figure 2-6, the request pin INT of the slave chip is connected to
// the IR2 pin of the master chip, so the statement indicates that the interrupt request

```

```
// signal from the slave chip is enabled.
// similarly, the line 211 statement performs a similar operation for the slave chip. 0xA1
// is the slave chip port. Read the mask code from it and write it immediately after AND
// 0xdf (0b11011111), indicating that the interrupt mask bit M2 for the IR5 on slave chip
// is cleared. Since the coprocessor is connected to the IR5 pin, this statement enables
// the coprocessor to send interrupt request signal IRQ13.
209     set_trap_gate(45, &irq13);
210     outb_p(inb_p(0x21)&0xfb, 0x21);           // enable IRQ2 of master chip.
211     outb(inb_p(0xA1)&0xdf, 0xA1);           // enable IRQ13 of slave chip.
212     set_trap_gate(39, &parallel_interrupt); // set parallel 1 gate
213 }
214
```

---

## 8.4 sys\_call.s

Linux uses the interrupt invocation method to implement the access interface between the user and the kernel resources. The `sys_call.s` program mainly implements the system call `INT 0x80` entry processing and signal detection processing, and gives the underlying interfaces of the two system functions, namely `sys_execve` and `sys_fork`. Interrupt handlers for coprocessor errors (`INT 16`), device not exist (`INT7`), clock interrupt (`INT32`), hard disk interrupt (`INT46`), floppy disk interrupt (`INT38`) are also listed.

### 8.4.1 Function descriptions

In Linux 0.12, application programs use `INT 0x80` and function number in register `EAX` to use various services provided by the kernel, and these services are called system call (syscall) services. Usually users do not use the system call service directly, but use it through the interface functions provided in general libraries (such as `libc`). For example, the system call `fork` that creates the process can directly use the function `fork()` in the library. The `INT 0x80` invocation is executed in this function and the result is returned to the user program.

In the kernel, the C function implementation code for all system call services is distributed throughout the kernel. The kernel sequentially arranges them into a function pointer (address) table according to the system call function number, and then calls the corresponding system service function during `INT 0x80` processing.

In addition, this source file also includes the entry processing code of several other interrupt calls. The implementation process and steps of these interrupt entry codes are basically the same. For soft interrupts (`system_call`, `coprocessor_error`, `device_not_available`), the processing can be basically divided into the following two steps: First prepare for calling the corresponding C function handler, pushing some parameters onto the stack. The system call can take up to 3 parameters, which are passed in via the registers `EBX`, `ECX` and `EDX`. Then call the C function to process the corresponding function. After the processing returns, the signal bitmap of the current task is detected, and a signal with the smallest value (highest priority) is processed and the signal in the signal bitmap is reset.

For the interrupt sent by the hardware `IRQ`, the processing first sends an end of interrupt instruction `EOI` to the interrupt control chip `8259A`, then calls the corresponding C function program. For the clock interrupt, the signal bitmap of the current task is also detected.

#### 8.4.1.1 Interrupt Service Entry Processing

For the interrupt handling of the system call (`int 0x80`), the program can be thought of as an "interface". In fact, the processing of each system call is basically done by calling the corresponding C function. The entire

process of the system call is shown in Figure 8-5.

This program will first check if the syscall function number in EAX is valid (within a given range), and then save some of the registers that will be used onto the stack. By default, the Linux kernel uses the segment registers DS, ES for kernel data segments and FS for user data segments. Then call the C function of the corresponding syscall through the above address jump table (sys\_call\_table). After the C function returns, the program pushes the return value onto the stack and saves it.

Next, the program looks at the status of the process that performed this invocation. If the process state changes from the running state to another state due to the operation of the above C function or other conditions, or because the time slice has run out (counter==0), then the process scheduling function schedule() ('jmp \_schedule') is called. . Since the return address 'ret\_from\_sys\_call' has been pushed onto the stack before executing "jmp \_schedule", it will eventually return to 'ret\_from\_sys\_call' and continue execution after end of schedule().

The code starting at the 'ret\_from\_sys\_call' label performs some post-processing. The main operation is to determine whether the current process is the initial process 0, and if so, directly exit the system call, and the interrupt returns. Otherwise, according to the code segment descriptor and the stack used, it is judged whether the process is a normal process, if not, then it is a kernel process (for example, initial process 1) or the like. The stack content is also immediately popped out and the syscall interrupt is exited. A piece of code at the end is used to handle the signal of the process. If the signal bitmap of the process structure indicates that the process has received a signal, then the signal handler do\_signal() is called.

Finally, the program restores the contents of the saved registers, exits the interrupt processing and returns to the interrupted program. If there is a signal, the program will first "return" to the corresponding signal processing function to execute, and then return to the program that calls system\_call.

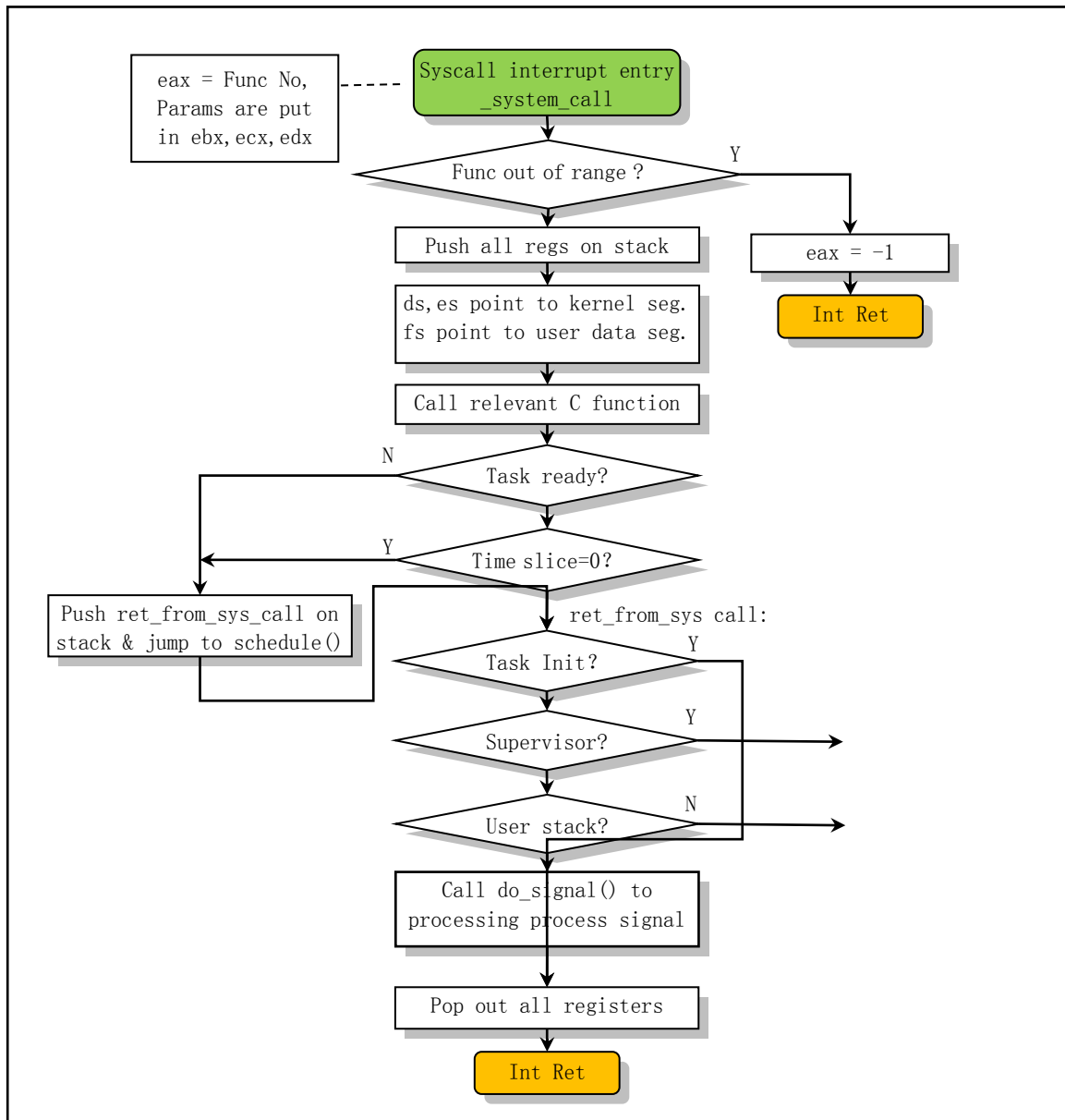


Figure 8-5 System call processing flow diagram

### 8.4.1.2 Syscall parameter passing method

Regarding the parameter transfer issue in the system call INT 0x80, the Linux kernel uses several general-purpose registers as a channel for parameter passing. In the Linux 0.12 system, the program uses the registers EBX, ECX, and EDX to pass parameters, and can pass three parameters directly to the system call service procedure (not including syscall function number in the EAX register). If a pointer to a user-space data block is used, the user program can pass more data information to the system call procedure.

As mentioned above, during the system call processing, the segment registers DS and ES point to the kernel data space and the FS is set to user data space. Therefore, in the actual data block transfer procedure, the Linux kernel can use the FS register to perform data copying between the kernel data space and the user data space, and the kernel program does not need to perform any check operation on the data boundary range during the copying process. The boundary check is done automatically by CPU. The actual data transfer work in the kernel can be done using functions such as `get_fs_byte()` and `put_fs_byte()`, see the implementations for these functions in the `include/asm/segment.h` file.

This method of using registers to pass parameters has a distinct advantage. That is, when the system interrupt service routine is entered, the registers that pass the parameters are also automatically placed on the kernel state stack, and when the process exits from the interrupt call, the kernel state stack is also popped, so the kernel does not have to specialize them. This method is the simplest and fastest method of parameter transfer that Mr. Linus knew at the time.

## 8.4.2 Code Annotation

Program 8-3 linux/kernel/sys\_call.s

---

```

1  /*
2  *  linux/kernel/system_call.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  system_call.s  contains the system-call low-level handling routines.
9  *  This also contains the timer-interrupt handler, as some of the code is
10 *  the same. The hd- and floppy-interrupts are also here.
11 *
12 *  NOTE: This code handles signal-recognition, which happens every time
13 *  after a timer-interrupt and after each system call. Ordinary interrupts
14 *  don't handle signal-recognition, as that would clutter them up totally
15 *  unnecessarily.
16 *
17 *  Stack layout in 'ret_from_system_call':
18 *
19 *      0(%esp) - %eax
20 *      4(%esp) - %ebx
21 *      8(%esp) - %ecx
22 *      C(%esp) - %edx
23 *      10(%esp) - original %eax      (-1 if not system call)
24 *      14(%esp) - %fs
25 *      18(%esp) - %es
26 *      1C(%esp) - %ds
27 *      20(%esp) - %eip
28 *      24(%esp) - %cs
29 *      28(%esp) - %eflags
30 *      2C(%esp) - %oldesp
31 *      30(%esp) - %oldss
32 */
# The general interrupt procedure in the original Linus comment above refers to interrupts
# other than the system-call (int 0x80) and clock interrupt (int 0x20). These interrupts
# occur randomly in the kernel state or user state. If the signal recognition is also
# handled during these interrupts, it may conflict with the process of identifying the
# signal during the system-call and clock interrupt. This violates the non-preemptive
# principle of kernel code. It is therefore not necessary for the system to process the
# signals in these "other" interrupts, nor to do so.
33
34 SIG_CHLD      = 17                # signal SIG_CHLD (child stop or end).
35
36 EAX           = 0x00              # offset of each register in the stack.

```

```

37 EBX          = 0x04
38 ECX          = 0x08
39 EDX          = 0x0C
40 ORIG_EAX     = 0x10          # If not a syscall (other interrupts), the value is -1
41 FS           = 0x14
42 ES           = 0x18
43 DS           = 0x1C
44 EIP          = 0x20          # Line 44-48 is automatically pushed onto stack by CPU.
45 CS           = 0x24
46 EFLAGS       = 0x28
47 OLDESP       = 0x2C          # old SS:ESP is also pushed when privilege level changed
48 OLDSS        = 0x30
49
# To facilitate access to the data structure in assembly, the offsets of the fields in
# structure of task and signal are given here.
# These are the field offsets in the task_struct, see include/linux/sched.h, line 105.
50 state = 0          # these are offsets into the task-struct.
51 counter = 4        # task runtime counts (ticks), time slice.
52 priority = 8       # counter=priority when task starts running, the longer it runs.
53 signal = 12        # signal bitmap, signal = bit offset + 1
54 sigaction = 16     # MUST be 16 (=len of sigaction)
55 blocked = (33*16)  # blocked signal offset
56
57 # offsets within sigaction          # see include/signal.h, line 55.
58 sa_handler = 0
59 sa_mask = 4
60 sa_flags = 8
61 sa_restorer = 12      # refer to the description of kernel/signal.c
62
63 nr_system_calls = 82    # total number of system calls in Linux 0.12.
64
65 ENOSYS = 38            # system-call number error code
66
67 /*
68  * Ok, I get parallel printer interrupts while using the floppy for some
69  * strange reason. Urgel. Now I just ignore them.
70  */
71 .globl _system_call, _sys_fork, _timer_interrupt, _sys_execve
72 .globl _hd_interrupt, _floppy_interrupt, _parallel_interrupt
73 .globl _device_not_available, _coprocessor_error
74
# The error code -ENOSYS will be returned if the system-call number is incorrect.
75 .align 2                # Memory is 4 bytes aligned.
76 bad_sys_call:
77     pushl $-ENOSYS      # set -ENOSYS in eax
78     jmp ret_from_sys_call

# scheduler re-execute entry. The scheduler starts at (kernel/sched.c, line 119).
# When scheduler schedule() returns, it continues running from ret_from_sys_call.
79 .align 2
80 reschedule:
81     pushl $ret_from_sys_call
82     jmp _schedule

```

```

##### Int 0x80 -- Linux system call entry point (int 0x80, call number in eax).
83 .align 2
84 _system_call:
85     push %ds                # save original seg registers.
86     push %es
87     push %fs
88     pushl %eax              # save the orig_eax

# A system-call can take up to 3 parameters or no parameters. The EBX, ECX, and EDX pushed
# onto the stack are loaded with parameters of the corresponding C function (see line 99).
# The order in which these registers are pushed is specified by GNU gcc. The first parameter
# can be stored in EBX, the second is in ECX, and the third is in EDX.
# The system-calls can be found in macros on lines 150--200 in file include/unistd.h.
89     pushl %edx
90     pushl %ecx              # push %ebx,%ecx,%edx as parameters
91     pushl %ebx              # to the system call

# After saving segment registers above, here we set DS, ES to point to the kernel data
# segment, and FS to the current local data segment of the user program that performs
# this syscall. Note that in Linux 0.12, the code and data memory segments allocated to
# the task is overlapped, their segment base address and limits are the same.
92     movl $0x10,%edx        # set up ds,es to kernel space
93     mov %dx,%ds
94     mov %dx,%es
95     movl $0x17,%edx        # fs points to local data space
96     mov %dx,%fs
97     cmpl _NR_syscalls,%eax  # syscall nr is valid ?
98     jae bad_sys_call

# The meaning of the operand in the following sentence is: [_sys_call_table + %eax * 4].
# See the explanation after the program and section 3.2.3. Sys_call_table[] is an array
# of pointers defined in include/linux/sys.h. This array contains the addresses of all 82
# syscall C handlers.
99     call _sys_call_table(,%eax,4)  # call C function indirectly.
100    pushl %eax               # the return value.

# Lines 101-106 below check the status of the current task. If it is not in the running
# state (state is not equal to 0) then execute the scheduler. If the task is in the
# running state, but its time slice has been used up (counter=0), then the scheduler is
# also executed. For example, when a process in the background process group performs
# control terminal read/write operations, all processes in the background group will
# receive a SIGTTIN or SIGTTOU signal by default, causing all processes in the process
# group to be in a stopped state, and the current process will return immediately.
101 2:
102    movl _current,%eax        # structure pointer -> eax
103    cmpl $0,state(%eax)      # state
104    jne reschedule
105    cmpl $0,counter(%eax)     # counter
106    je reschedule

# The following code executes the recognition of the signal after returning from the C
# function. When other interrupt service routines exit, they will also jump to here for

```



```

# processing before exiting the interrupt process. For example, the processor error
# interrupt int 16 on the following 131 lines.
# Here, it is first determined whether the current task is the initial task0, and if so,
# it is not necessary to perform signal processing on it and return directly. Note that
# _task on line 109 corresponds to task[] array in the C program, and direct reference
# to it is equivalent to referencing task[0].
107 ret_from_sys_call:
108     movl _current,%eax
109     cmpl _task,%eax           # task[0] cannot have signals
110     je 3f                    # forward jump to label 3 (line 129), exit

# Check whether the program is a user task by checking the code segment selector of the
# original calling program, and if not, exit the interrupt directly (task cannot be
# preempted while in kernel mode). Otherwise, the signal of the task is checked.
# Here, the selector is compared with 0x000f to determine whether it is a user task.
# Value 0x000f represents the selector of the user code segment (RPL=3, local table,
# code segment). If not, it means that an interrupt handler (such as INT 16) jumps to
# line 107 and runs here. For this case, jump and exit the interrupt. In addition, if
# the original stack segment selector is not 0x17 (not in the user segment), it also
# indicates that the caller of the system-call is not a user task, and also exits.
111     cmpw $0x0f,CS(%esp)      # was old code segment supervisor ?
112     jne 3f
113     cmpw $0x17,OLDSS(%esp)    # was stack segment = 0x17 ?
114     jne 3f

# The following code (lines 115-128) is used to process the signal of the current task.
# Here, the signal bitmap (32 bits, each bit represents 1 kind of signal) in the current
# task structure is first obtained, and then the signal block code is used to block the
# impermissible signals. Then take the signal with the smallest value and reset the
# corresponding bit of the signal in the original bitmap. Finally, using this signal as
# one of the parameters, call do_signal() (in kernel/signal.c, 128). After the do_signal()
# or signal handler returns, if the return value is not 0, then see if you need to switch
# processes or continue processing other signals.
115     movl signal(%eax),%ebx    # signal bitmap -> ebx.
116     movl blocked(%eax),%ecx   # signals blocked -> ecx.
117     notl %ecx
118     andl %ebx,%ecx           # get a bitmap of permissible signals
119     bsfl %ecx,%ecx           # scan the bitmap from bit0, located none zero bit.
120     je 3f                    # exit if none.
121     btrl %ecx,%ebx           # reset the signal.
122     movl %ebx,signal(%eax)    # store the new bitmap -> current->signal
123     incl %ecx                # adjust signal starting from 1 (1--32).
124     pushl %ecx               # as parameter.
125     call _do_signal          # do_signal() (kernel/signal.c, 128)
126     popl %ecx               # discard the parameter.
127     testl %eax, %eax         # check return value.
128     jne 2b                  # see if we need to switch tasks, or do more signals

129 3:    popl %eax              # contains ret code pushed at line 100.
130     popl %ebx
131     popl %ecx
132     popl %edx
133     addl $4, %esp           # skip orig_eax

```

```

134     pop %fs
135     pop %es
136     pop %ds
137     iret
138
139     ##### Int16 -- Processor error interrupt.   Type: Error; no error code.
140     # This is an external hardware exception. When the coprocessor detects that it has an
141     # error, it notifies the CPU via the ERROR pin. The following code is used to process
142     # the error signal issued by the coprocessor and jump to execute the C function
143     # math_error(). After returning, it will jump to the label 'ret_from_sys_call' to
144     # continue execution.
145     .align 2
146     _coprocessor_error:
147     push %ds
148     push %es
149     push %fs
150     pushl $-1                # fill in -1 for orig_eax    # not an syscall
151     pushl %edx
152     pushl %ecx
153     pushl %ebx
154     pushl %eax
155     movl $0x10,%eax          # ds,es point to kernel data seg.
156     mov %ax,%ds
157     mov %ax,%es
158     movl $0x17,%eax          # fs point to (user data seg)
159     mov %ax,%fs
160     pushl $ret_from_sys_call
161     jmp _math_error          # math_error() (kernel/math/error.c, 11).
162
163     ##### Int7 -- The device or coprocessor does not exist.   Type: Error; no error code.
164     # If the EM (analog) flag in control register CR0 is set, the interrupt is raised when
165     # CPU executes a coprocessor instruction, so the CPU has a chance to have the interrupt
166     # handler emulate the coprocessor instruction (line 181).
167     # The flag TS of CR0 is set when the CPU performs task switch. TS can be used to determine
168     # when the content in the coprocessor does not match the task being executed by the CPU.
169     # This interrupt is raised when the CPU is running a coprocessor escape instruction and
170     # finds that TS is set. At this point you can save the coprocessor content of the previous
171     # task and restore coprocessor execution status of the new task (line 176). See kernel/sched.c,
172     # line 92. The interrupt will eventually be transferred to the label 'ret_from_sys_call'
173     # for execution (detection and processing of the signal).
174     .align 2
175     _device_not_available:
176     push %ds
177     push %es
178     push %fs
179     pushl $-1                # fill in -1 for orig_eax
180     pushl %edx
181     pushl %ecx
182     pushl %ebx
183     pushl %eax
184     movl $0x10,%eax          # ds,es to kernel data seg.
185     mov %ax,%ds
186     mov %ax,%es

```

```

170      movl $0x17,%eax          # fs to user data seg.
171      mov %ax,%fs

# The following code clears the flag TS and get CRO. If the coprocessor emulation flag EM
# is not set, indicating that it is not an interrupt caused by EM, the task coprocessor
# state is restored, the C function math_state_restore() is executed, and the code at
# ret_from_sys_call is executed upon return.
172      pushl $ret_from_sys_call
173      clts                    # clear TS so that we can use math
174      movl %cr0,%eax
175      testl $0x4,%eax         # EM (math emulation bit)
176      je _math_state_restore

# If the EM flag is set, execute the math simulation function math_emulate().
177      pushl %ebp
178      pushl %esi
179      pushl %edi
180      pushl $0                # temporary storage for ORIG_EIP
181      call _math_emulate      # (math/math_emulate.c, line 476)
182      addl $4,%esp            # discard temporary data.
183      popl %edi
184      popl %esi
185      popl %ebp
186      ret                    # ret to ret_from_sys_call
187

##### Int32 -- (int 0x20) Clock interrupt handler.
# The clock interrupt frequency is set to 100Hz (include/linux/sched.h, 4). The timing
# chip 8253/8254 is initialized at (kernel/sched.c, 438). Here jiffies add 1 every 10
# milliseconds. This code increments jiffies by 1, sends EOI to 8259 controller, and then
# calls the C function do_timer(long CPL) with current privilege level as a parameter.
188 .align 2
189 _timer_interrupt:
190     push %ds                 # save ds,es and put kernel data space
191     push %es                 # into them. %fs is used by _system_call
192     push %fs
193     pushl $-1               # fill in -1 for orig_eax
194     pushl %edx               # we save %eax,%ecx,%edx as gcc doesn't
195     pushl %ecx               # save those across function calls. %ebx
196     pushl %ebx               # is saved as we use that in ret_sys_call
197     pushl %eax
198     movl $0x10,%eax         # ds,es to kernel
199     mov %ax,%ds
200     mov %ax,%es
201     movl $0x17,%eax         # fs to user
202     mov %ax,%fs
203     incl _jiffies
204     movb $0x20,%al          # EOI to interrupt controller #1
205     outb %al,$0x20

# The current privilege level (0 or 3) in the selector (CS segment) that executes the
# system-call is fetched and pushed onto the stack as a parameter to do_timer. The
# do_timer() function performs task switching, timing, etc., and is implemented in
# kernel/sched.c, line 324.

```

```

206      movl CS(%esp),%eax
207      andl $3,%eax          # %eax is CPL (0 or 3, 0=supervisor)
208      pushl %eax
209      call _do_timer        # 'do_timer(long CPL)' does everything from
210      addl $4,%esp          # task switching to accounting ...
211      jmp ret_from_sys_call
212
##### This is sys_execve() syscall. The C function do_execve() is called with the caller's
# code pointer as a parameter.  function do_execve() is in fs/exec.c, line 207.
213 .align 2
214 _sys_execve:
215      lea EIP(%esp),%eax    # eax points to old eip on the stack.
216      pushl %eax
217      call _do_execve
218      addl $4,%esp          # discard pushed eip.
219      ret
220
##### The sys_fork() call, used to create a child process, is syscall function 2.
# First call the C function find_empty_process() to get a process last_pid. If a negative
# number is returned, the current task array is full. Otherwise call copy_process() to
# copy the process.
221 .align 2
222 _sys_fork:
223      call _find_empty_process # get last_pid (kernel/fork.c, 143)
224      testl %eax,%eax          # pid in eax, if negative then ret.
225      js 1f
226      push %gs
227      pushl %esi
228      pushl %edi
229      pushl %ebp
230      pushl %eax
231      call _copy_process      # copy_process() (kernel/fork.c, 68).
232      addl $20,%esp           # discard.
233 1:      ret
234
##### Int 46 -- (int 0x2E) Harddisk interrupt handler, which responds to IRQ14.
# This interrupt occurs when the requested hard disk operation is completed or an error
# occurs. (See kernel/blk_drv/hd.c).
# The code first sends an EOI instruction to 8259A slave chip, then takes function pointer
# in variable do_hd into EDI, and sets the do_hd to NULL. Then check if the edi function
# pointer is null. If it is null, let edi point to unexpected_hd_interrupt() to display
# the error message. The EOI instruction is then sent to the 8259A master chip, and the
# function pointed to by EDI is called: read_intr(), write_intr(), or unexpected_hd_interrupt().
235 _hd_interrupt:
236      pushl %eax
237      pushl %ecx
238      pushl %edx
239      push %ds
240      push %es
241      push %fs
242      movl $0x10,%eax        # ds,es poing to kernel data seg.
243      mov %ax,%ds
244      mov %ax,%es

```

```

245      movl $0x17,%eax          # fs point to user data seg.
246      mov %ax,%fs
247      movb $0x20,%al
248      outb %al,$0xA0          # EOI to interrupt controller #1
249      jmp 1f                  # give port chance to breathe
250 1:      jmp 1f
      # do_hd is defined as a function pointer that will be assigned the address of read_intr()
      # or write_intr() function. The do_hd pointer variable is set to NULL after being placed
      # in the edx register. Then the resulting function pointer is tested. If the pointer is
      # NULL, the pointer is assigned to C function unexpected_hd_interrupt() to handle the
      # unknown hard disk interrupt.
251 1:      xorl %edx,%edx
252      movl %edx,_hd_timeout    # hd_timeout set to 0,controller produces INT in time.
253      xchgl _do_hd,%edx
254      testl %edx,%edx
255      jne 1f                  # if null, point to unexpected_hd_interrupt().
256      movl $_unexpected_hd_interrupt,%edx
257 1:      outb %al,$0x20        # send EOI to 8259A master chip.
258      call *%edx              # "interesting" way of handling intr.
259      pop %fs
260      pop %es
261      pop %ds
262      popl %edx
263      popl %ecx
264      popl %eax
265      iret
266
##### Int38 -- (int 0x26) floppy drive interrupt handler, handles the interrupt request IRQ6.
# The processing is basically the same as the above of the hard disk. (kernel/blk_drv/floppy.c).
# The following code first sends an EOI instruction to the 8259A interrupt controller
# master chip. Then take the function pointer in the variable do_floppy into the eax register,
# and set do_floppy to NULL. Then check if the eax function pointer is NULL. If it is NULL,
# then let eax point to unexpected_floppy_interrupt () to display the error message. Then
# call the function pointed to by eax: rw_interrupt, seek_interrupt, recal_interrupt,
# reset_interrupt or unexpected_floppy_interrupt.
267 _floppy_interrupt:
268      pushl %eax
269      pushl %ecx
270      pushl %edx
271      push %ds
272      push %es
273      push %fs
274      movl $0x10,%eax          # ds,es point kernel data seg.
275      mov %ax,%ds
276      mov %ax,%es
277      movl $0x17,%eax          # fs point to user data seg.
278      mov %ax,%fs
279      movb $0x20,%al          # send EOI to 8259A master chip.
280      outb %al,$0x20          # EOI to interrupt controller #1
281      xorl %eax,%eax
282      xchgl _do_floppy,%eax
283      testl %eax,%eax          # function pointer NULL ?
284      jne 1f                  # yes, point to unexpected_floppy_interrupt()

```

```

285     movl $_unexpected_floppy_interrupt,%eax
286 1:    call *%eax           # "interesting" way of handling intr.
287     pop %fs              # function pointed by do_floppy
288     pop %es
289     pop %ds
290     popl %edx
291     popl %ecx
292     popl %eax
293     iret
294
#### Int 39 -- (int 0x27) Parallel port interrupt handler, corresponding to IRQ7.
# The kernel has not implemented this handler, only the EOI instruction is sent here.
295 _parallel_interrupt:
296     pushl %eax
297     movb $0x20,%al
298     outb %al,$0x20
299     popl %eax
300     iret

```

### 8.4.3 Reference Information

#### 8.4.3.1 32-bit addressing for GNU assembly language

The GNU assembly language uses AT&T's assembly syntax. See Section 3.2.3 for a detailed introduction and examples of this. Here is just an introduction to the addressing method and some examples. The AT&T and Intel assembly language addressing operand formats are as follows:

---

```

AT&T:  disp (base, index, scale)
Intel: [base + index * scale + disp]

```

---

Where disp is a optional offset, base is 32-bit base address, index is 32-bit index register, and scale is scale factor (1, 2, 4, 8, default is 1). Although the two assembly language addressing formats are slightly different, the specific addressing locations are actually identical. The addressing position calculation method in the above format is:  $\text{disp} + \text{base} + \text{index} * \text{scale}$

You don't need to write all of these fields when you apply, but there must be one in disp and base. Here are some examples.

Table 8-2 Memory addressing examples

Addressing requirements	AT&T format	Intel format
Addressing a specified C variable 'booga'	_booga	[_booga]
Addressing the location pointed to by register	(%eax)	[eax]
Addressing a variable by using the contents of the register as the base address	_variable(%eax)	[eax + _variable]
Address a value in an int array (scale value is 4)	_array(, %eax, 4)	[eax*4 + _array]
Use direct addressing offset *(p+1), where p is the char's pointer, placed in %eax.	1(%eax)	[eax+1]
Addresses the specified character in an 8-byte array of records. Where eax is the index, and ebx is the	_array(%ebx, %eax, 8)	[ebx + eax * 8 + _array]

offset of the specified character in the record.		
--	--	--

### 8.4.3.2 Adding a System Call

To add a new system-call to your kernel, we should first decide what its exact purpose is. Linux systems do not promote a system call for multiple purposes (except for `ioctl()` system calls). In addition, we need to determine the parameters, return values, and error code for the new system-call. The interface of the system call should be as simple as possible, so the parameters should be as few as possible. Also, the versatility and portability of system-calls should be considered in design. If we want to add a new system call to Linux 0.12, then we need to do the following things.

First, write the handler for the new system call in the relevant program, such as the function named `sys_sethostname()`. This function is used to modify the computer name of the system. Usually this C function can be placed in the `kernel/sys.c` program. In addition, since the `thisname` structure is used, it is also necessary to move the `thisname` structure (lines 218-220) in `sys_uname()` outside of the function.

---

```
#define MAXHOSTNAMELEN 8
int sys_sethostname(char *name, int len)
{
    int    i;

    if (!suser())
        return -EPERM;
    if (len > MAXHOSTNAMELEN)
        return -EINVAL;
    for (i=0; i < len; i++) {
        if ((thisname.nodename[i] = get_fs_byte(name+i)) == 0)
            break;
    }
    if (thisname.nodename[i]) {
        thisname.nodename[i>MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
    }
    return 0;
}
```

---

Then add the new system call number and prototype definition in the `include/unistd.h` file. For example, you can add a function number after line 149 and add a prototype definition after line 279:

---

```
// The new system call number.
#define __NR_sethostname    87
// The new system calls function prototype.
int sethostname(char *name, int len);
```

---

Then add the external function declaration in the `include/linux/sys.h` file and insert the name of the new system call handler at the end of the function pointer table `sys_call_table`, as shown below. Note that the function names must be arranged in strict order of syscall function numbers.

---

```
extern int sys_sethostname();
// Function pointer array table.
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
```

---

```
...,
sys_lstat, sys_readlink, sys_uselib, sys_sethostname };
```

---

Then modify line 63 of the sys\_call.s file to increase the total number of nr\_system\_calls by one. At this point you can recompile the kernel. Finally, refer to the implementation of the library function in the lib/ directory to add a new system call library function sethostname() to the libc library.

---

```
#define __LIBRARY__
#include <unistd.h>

_syscall2(int, sethostname, char *, name, int, len);
```

---

### 8.4.3.3 Using System-Calls Directly in Assembly File

Below is a simple assembly example asm.s given by Mr. Linus in explaining the relationship and difference between as86 and GNU as. This example shows how to program a stand-alone program in assembly language on a Linux system. That is, it is not necessary to use a start code module (such as crt0.o) and a function in the library. The procedure is as follows:

---

```
.text
_entry:
    movl $4,%eax           # syscall nr, write op.
    movl $1,%ebx           # paras: fhandle, stdout.
    movl $message,%ecx     # paras: buff pointer.
    movl $12,%edx          # paras: size.
    int $0x80
    movl $1,%eax           # syscall no, exit.
    int $0x80

message:
    .ascii "Hello World\n"
```

---

There are two system-calls used: 4 - write file operation sys\_write() and 1 - exit program sys\_exit(). The C function executed by the write system-call is declared as sys\_write(int fd, char \*buf, int len), see the program fs/read\_write.c, starting at line 83. It comes with 3 parameters. These three parameters are stored in registers EBX, ECX, and EDX before calling the system call. The steps to compile and execute the program are as follows:

---

```
[/usr/root]# as -o asm.o asm.s
[/usr/root]# ld -o asm asm.o
[/usr/root]# ./asm
Hello World
[/usr/root]#
```

---



## 8.5 mktime.c

The mktime.c program is used to calculate the boot time of the kernel-specific UNIX calendar time.

### 8.5.1 Functions

The program has only one function `kernel_mktime()`, which is only used by the kernel and is used to calculate the number of seconds (calendar time) from 0:00 on January 1, 1970 to the date of booting, as the boot time. This function is exactly the same as the `mktime()` function provided in the standard C library, which converts the time represented by the `tm` structure into UNIX calendar time. However, since the kernel is not a normal program, you cannot call functions in the development environment library, so you must write one yourself.

### 8.5.2 Code Annotation

Program 8-4 linux/kernel/mktime.c

---

```

1  /*
2   * linux/kernel/mktime.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  // Time type header file. The most important of these is the definition of the tm
8  // structure and some function prototypes related to time.
9  #include <time.h>
10
11 /*
12  * This isn't the library routine, it is only used in the kernel.
13  * as such, we don't care about years<1970 etc, but assume everything
14  * is ok. Similarly, TZ etc is happily ignored. We just do everything
15  * as easily as possible. Let's find something public for the library
16  * routines (although I think minix times is public).
17  */
18 /*
19  * PS. I hate whoever thought up the year 1970 - couldn't they have gotten
20  * a leap-year instead? I also hate Gregorius, pope or no. I'm grumpy.
21  */
22 #define MINUTE 60                // 1 minute in seconds.
23 #define HOUR (60*MINUTE)        // 1 hour in seconds.
24 #define DAY (24*HOUR)           // 1 day in seconds.
25 #define YEAR (365*DAY)          // 1 year in seconds.
26
27 /* interestingly, we assume leap-years */
28 // The start time seconds at the beginning of each month is defined in a year limit.
29 static int month[12] = {
30     0,
31     DAY*(31),
32     DAY*(31+29),
33     DAY*(31+29+31),
34     DAY*(31+29+31+30),

```

```

32     DAY*(31+29+31+30+31),
33     DAY*(31+29+31+30+31+30),
34     DAY*(31+29+31+30+31+30+31),
35     DAY*(31+29+31+30+31+30+31+31),
36     DAY*(31+29+31+30+31+30+31+31+30),
37     DAY*(31+29+31+30+31+30+31+31+30+31),
38     DAY*(31+29+31+30+31+30+31+31+30+31+30)
39 };
40
41 // This function calculates the number of seconds elapsed from 0:00 on January 1, 1970 to
42 // the date of machine boot, as the boot time. The fields in the tm have been assigned in
43 // init/main.c and the information is taken from CMOS.
44 long kernel_mktime(struct tm * tm)
45 {
46     long res;
47     int year;
48
49     // First calculate the number of years that have passed since 1970. Because there is a
50     // 2-digit representation here, there will be a year 2000 problem. We can simply solve
51     // this problem by adding a statement to the front: if (tm->tm_year<70) tm->tm_year += 100;
52     // Since the year y of UNIX is calculated from 1970. It is a leap year until 1972, so the
53     // third year (71, 72, 73) is the first leap year, so the calculation method of the leap
54     // year from 1970 should be 1 + (y - 3) / 4, that is (y + 1)/4.
55     // res = the nr of secs in these years + the nr of secs in each leap year + the nr of secs
56     // from current year to current month. In addition, the number of days in February in the
57     // month[] array contains the number of days in leap year, that is, the number of days in
58     // February is one day more. Therefore, if the year is not a leap year and the current
59     // month is greater than February, we will subtract this day. Since we counted from 1970,
60     // the detection method for the leap year is: (y + 2) can be divided by 4. If not, it's
61     // not a leap year.
62     // if (tm->tm_year<70) tm->tm_year += 100;
63     year = tm->tm_year - 70;
64     /* magic offsets (y+1) needed to get leapyears right. */
65     res = YEAR*year + DAY*((year+1)/4);
66     res += month[tm->tm_mon];
67     /* and (y+2) here. If it wasn't a leap-year, we have to adjust */
68     if (tm->tm_mon>1 && ((year+2)%4))
69         res -= DAY;
70     res += DAY*(tm->tm_mday-1);           // nr of days in the past month in secs.
71     res += HOUR*tm->tm_hour;             // the past hours of the day in secs.
72     res += MINUTE*tm->tm_min;             // the past minutes of the hour in secs.
73     res += tm->tm_sec;                   // nr of secs that have passed in 1 minute.
74     return res;                          // the nr of secs elapsed since 1970.
75 }
76
77

```

## 8.5.3 Information

### 8.5.3.1 Calculation method for leap year

The basic calculation method for leap years is:

If y can be divisible by 4 and cannot be divisible by 100, or can be divisible by 400, then y is a leap year.

## 8.6 sched.c

### 8.6.1 Function description

The sched.c source file contains codes for scheduling tasks in the kernel. It includes several basic functions for scheduling (sleep\_on(), wakeup(), schedule(), etc.), as well as some simple system-call functions (such as getpid()). The timer function do\_timer() of the system clock interrupt service routine is also in this program. In addition, in order to facilitate the floppy disk drive timing processing programming, Mr. Linus also put several functions related to floppy disk timing into this program.

The code for these basic functions is not long, but it is somewhat abstract and difficult to understand. Fortunately, there are already many textbooks that have a more in-depth introduction and discussion. Therefore, you can refer to other books to describe these functions when you study. Before we start to annotate and analyze the code, let's take a look at the principle of the scheduler, sleep, and wake-up functions.

#### 8.6.1.1 Schedule function

The schedule function schedule() is responsible for selecting the next task (process) to run in the system. It first checks all tasks and wakes up any task that has received signals. The specific method is to check the alarm timing value 'alarm' for each task in the task array. If the task's alarm time has expired ( jiffies > alarm ), set the SIGALRM signal in its signal bitmap and clear the alarm value. Jiffies is the number of ticks from machine boot time (10ms/tick, defined in sched.h). If there are other signals besides the blocked signal in the signal bitmap of the task and the task is in an interruptible sleep state (TASK\_INTERRUPTIBLE), then the task is set to ready state (TASK\_RUNNING).

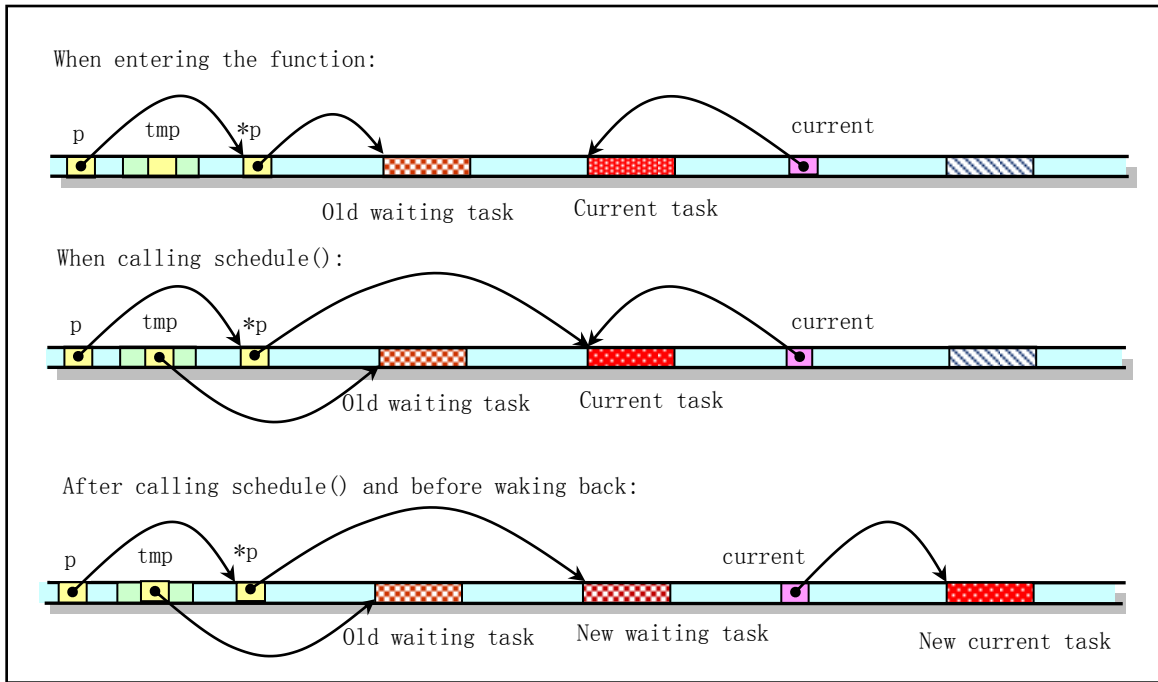
This is followed by the core processing part of the scheduling function. This part of the code selects the task to be executed later based on the time slice and priority mechanism of the task. It first loops through all the tasks in the task array, selects the task with the largest counter value of the remaining execution time, and switches to the task with the switch\_to() function.

If the counter value of all ready-to-run tasks is equal to zero, it means that the time slice of all tasks has been run out at the moment. Then, according to the task priority value 'priority', reset the running time slice value 'counter' of each task, and then cycle through the execution time slice values of all tasks again.

#### 8.6.1.2 Sleep and wake-up functions

The other two functions worth mentioning are the sleep function sleep\_on() and the wake-up function wake\_up(). Although these two functions are very short, they are harder to understand than the schedule() function. Before we look at the code, let's make some explanations by means of the diagram. Simply put, the main function of the sleep\_on() is to temporarily switch the process out onto the waiting queue for a period of time when the resource requested by the process (or task) is being used or not in memory. When the process is switched back, it will continue to run. The way to put in the wait queue takes advantage of the tmp pointer in the function as the link for each waiting task.

The function involves the operation of three task pointers: \*p, tmp, and current. \*p is the wait queue head pointer, such as the i\_wait pointer of the file system memory i node, the buffer\_wait pointer in the memory buffer operation, etc.; tmp is a temporary pointer established on the function stack, stored on the current task kernel state stack; 'current' is a pointer to the current task. For the changes of these pointers in memory, we can use the schematic diagram of Figure 8-6 to illustrate. The long bars in the figure represent a sequence of memory bytes.

Figure 8-6 Schematic diagram of pointer changes in the `sleep_on()`.

When entering the function, the queue head pointer `*p` points to the task structure (process descriptor) that has been waiting in the wait queue. Of course, there is no waiting task on the wait queue when the system first starts executing. Therefore, the original waiting task in the above figure does not exist at the beginning, and `*p` points to NULL.

Through the pointer operations, before the scheduler function is called, the queue head pointer points to the current task structure, and the temporary pointer 'tmp' in the function points to the original waiting task. Before executing the scheduler and before the task is woken up and returned to execution, the current task pointer is directed to the new current task, and the CPU switches to execute in the new task. In this way, the execution of the `sleep_on()` function causes the tmp pointer to point to the original waiting task pointed to by the queue head pointer in the queue, and the queue head pointer points to the newly added waiting task, that is, the task of calling this function. Thus, by the link function of the temporary pointer tmp on the stack, when several processes call the function for waiting for the same resource, the kernel program implicitly constructs a waiting queue. See the waiting queue diagram in Figure 8-7. The figure shows the situation when a third task is inserted into the head of the queue. From the figure we can more easily understand the wait queue formation process of the `sleep_on()` function.

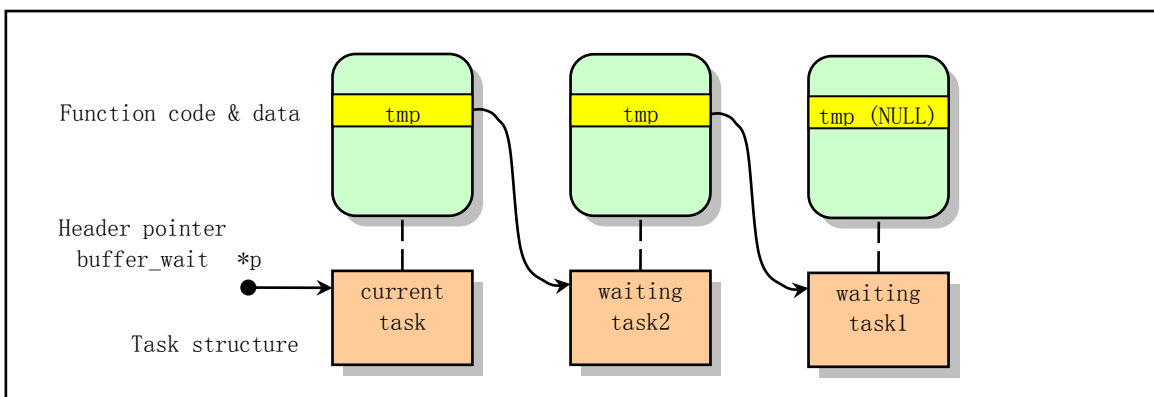


Figure 8-7 The implicit task wait queue for the sleep\_on()

After inserting the process into the wait queue, the sleep\_on() function calls the schedule() function to execute another process. When the process is awakened and re-executed, the subsequent statements are executed, and a process that enters the waiting queue earlier than it wakes up. Note that the so-called wake-up here does not mean that the process is in the execution state, but in the ready state that can be scheduled to execute.

The wakeup function wake\_up() is used to put the specified task waiting for available resources into a ready state (TASK\_RUNNING). This function is a generic wakeup function. In some cases, such as reading a block of data on a disk, since any task in the wait queue may be awakened first, it is also necessary to empty the pointer of the wake-up task structure. In this way, when the process that goes to sleep is awakened and the sleep\_on() is re-executed, there is no need to wake up the process.

There is also a function interruptible\_sleep\_on(), whose structure is basically similar to sleep\_on(), except that the current task is set to an interruptible wait state before scheduling, and after the task is awakened, it is necessary to determine whether there is any task entered later. If so, schedule them to run first. Starting at kernel 0.12, the two functions are combined into one and only use the state of the task as a parameter to distinguish between the two cases.

When reading the code in this file, it is best to refer to the comments in the include/linux/sched.h file for a more complete understanding of the kernel's scheduling mechanism.

## 8.6.2 Code Annotation

Program 8-5 linux/kernel/sched.c

---

```

1  /*
2   *  linux/kernel/sched.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  'sched.c' is the main kernel file. It contains scheduling primitives
9   *  (sleep_on, wakeup, schedule etc) as well as a number of simple system
10  *  call functions (type getpid(), which just extracts a field from
11  *  current-task
12  */
13  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
14  //      data of the initial task 0, and some embedded assembly function macro statements
15  //      about the descriptor parameter settings and acquisition.
16  // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
17  //      commonly used functions of the kernel.
18  // <linux/sys.h> The system calls the header file. Contains 72 system call C function
19  //      handlers, starting with 'sys_'.
20  // <linux/fdreg.h> Floppy disk file. Contains some definitions of floppy disk controller
21  //      parameters.
22  // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
23  //      descriptors/interrupt gates, etc. is defined.
24  // <asm/io.h> Io header file. Defines the function that operates on the io port in the form
25  //      of a macro's embedded assembler.

```

```

// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//     for segment register operations.
// <signal.h> Signal header file. Define signal symbol constants, signal structures, and
//     signal manipulation function prototypes.
13 #include <linux/sched.h>
14 #include <linux/kernel.h>
15 #include <linux/sys.h>
16 #include <linux/fdreg.h>
17 #include <asm/system.h>
18 #include <asm/io.h>
19 #include <asm/segment.h>
20
21 #include <signal.h>
22
// The macro takes the binary value of the corresponding bit of the signal nr in the
// signal bitmap. The signal number range is 1-32. For example, the bitmap value of
// signal 5 is 1<<(5-1) = 16 = 00010000b.
// All signals except the SIGKILL and SIGSTOP signals are blockable:
// BLOCKABLE = (11111111, 11111011, 11111110, 11111111b).
23 #define S(nr) (1<<((nr)-1))
24 #define BLOCKABLE (~(S(SIGKILL) | S(SIGSTOP)))
25
// Kernel debugging function. prints the pid, process status, kernel stack free bytes
// (approximately) and task's younger & older siblings for the specified task nr.
// Because task's data and its kernel state stack are on the same memory page (4096 bytes
// per page), and the kernel state stack begins from the end of the page downward, so
// variable j on line 28 represents the maximum kernel stack capacity, or the lowest top
// position of the task's kernel stack.
// Parameters:
// nr - task no;    p - task structure pointer.
26 void show_task(int nr, struct task_struct * p)
27 {
28     int i, j = 4096 - sizeof(struct task_struct);
29
30     printk("d: pid=%d, state=%d, father=%d, child=%d, ", nr, p->pid,
31           p->state, p->p_pptr->pid, p->p_cptr ? p->p_cptr->pid : -1);
32     i=0;
33     while (i<j && !((char *) (p+1))[i]) // Detects nr of zero bytes after task struct.
34         i++;
35     printk("d/%d chars free in kstack\n\r", i, j);
36     printk("    PC=%08X ", *(1019 + (unsigned long *) p));
37     if (p->p_ysptr || p->p_osptr)
38         printk("    Younger sib=%d, older sib=%d\n\r",
39               p->p_ysptr ? p->p_ysptr->pid : -1,
40               p->p_osptr ? p->p_osptr->pid : -1);
41     else
42         printk("    \n\r");
43 }
44
// Displays status information for all tasks in the system.
// NR_TASKS is the max nr of tasks in the system (64), defined in line 6 of linux/sched.h.
45 void show_state(void)
46 {

```

```

47     int i;
48
49     printk("\rTask-info: \n\r");
50     for (i=0; i<NR_TASKS; i++)
51         if (task[i])
52             show_task(i, task[i]);
53 }
54
55 // The input clock frequency of 8253 counter/timer chip is about 1.193180 MHz. The Linux
56 // kernel expects the timer interrupt frequency to be 100 Hz, that is, a clock interrupt
57 // is issued every 10 ms. So here LATCH is the initial value of setting the 8253 chip, see
58 // line 438.
59 #define LATCH (1193180/HZ)
60
61 extern void mem_use(void);          // [??] not defined anywhere.
62
63 extern int timer_interrupt(void);   // kernel/system_call.s, 189
64 extern int system_call(void);      // kernel/system_call.s, 84
65
66 // Each task (process) has its own kernel-state stack. This defines the task union,
67 // consisting of the task structure and the stack array. Because the data structure of a
68 // task and its kernel-state stack are placed in the same memory page, its data segment
69 // selector can be obtained from the stack segment register SS.
70 // Line 67 below sets data for the initial task (initial data is in linux/sched.h, 156).
71 union task_union {
72     struct task_struct task;
73     char stack[PAGE_SIZE];
74 };
75
76 static union task_union init_task = {INIT_TASK,};
77
78 // The nr of ticks from system start (10ms/tick). A tick is made every time the timer
79 // chip's interrupt occurs.
80 // The qualifier 'volatile', its English meaning is easy to change, unstable. The meaning
81 // of this qualifier is to indicate to the compiler that the contents of the variable may
82 // change as a result of modifications by other programs. Usually when a variable is
83 // declared in a program, the compiler will try to put it in a general-purpose register,
84 // such as EBX, to improve access efficiency. After that, it generally does not care about
85 // the original value of the variable in memory. If other programs or devices modify the
86 // value of this variable in memory at this time, the value in EBX will not be updated. To
87 // solve this issue, a volatile qualifier is created, so that the code must take its value
88 // from the specified memory location when referring to the variable. Here, gcc is required
89 // not to optimize the jiffies, nor to move the location, and to take its value from memory.
90 // Because the counter/timer chip interrupt processing process and other programs will modify
91 // its value.
92 unsigned long volatile jiffies=0;    // kernel pulse (ticks)
93 unsigned long startup_time=0;        // total seconds from 1970:0:0:0
94 int jiffies_offset = 0;              /* # clock ticks to add to get "true
95                                     time". Should always be less than
96                                     1 second's worth. For time fanatics
97                                     who like to synchronize their machines
98                                     to WWW :->) */
99

```

```

// current task pointer and points to task 0 during initialization.
77 struct task\_struct *current = &(init_task.task);
78 struct task\_struct *last_task_used_math = NULL;
79
// Define an array of task pointers. The first item is initialized to the task data
// structure of the initial task (task 0).
80 struct task\_struct * task[NR_TASKS] = {&(init_task.task), };
81
// Define the user stack (array), a total of 1K items, size 4K bytes. Used as a kernel
// stack during kernel initialization. After initialization is complete, it will be used
// as the user mode stack for task 0. It is the kernel stack before running task 0 and is
// later used as the user state stack for tasks 0 and 1.
// The following structure is used to set the stack SS: ESP, see head.s, line 23. SS is
// set to the kernel data segment selector (0x10), and ESP is set to point to the end of
// the last item in the user_stack array. This is because Intel CPU performs the stack
// operation by first decrementing the stack pointer SP and then saving the contents of
// the stack at the SP pointer.
82 long user\_stack [ PAGE\_SIZE>>2 ] ;
83
84 struct {
85     long * a;
86     short b;
87     } stack_start = { & user\_stack [PAGE\_SIZE>>2] , 0x10 };
88 /*
89  * 'math_state_restore()' saves the current math information in the
90  * old math state array, and gets the new ones from the current task
91  */
// After the task is scheduled to be exchanged, this function is used to save the math
// coprocessor state (context) of the original task and restore the coprocessor context
// of the new task scheduled.
92 void math\_state\_restore()
93 {
// Return if the task has not changed (the previous task is the current task). Here
// "previous task" refers to the task that has just been exchanged out.
// In addition, the WAIT instruction must be executed before the coprocessor instructions.
// If the previous task used a coprocessor, its state is saved to task's field of TSS.
94     if (last\_task\_used\_math == current)
95         return;
96     __asm__("fwait");
97     if (last\_task\_used\_math) {
98         __asm__("fnsave %0"::"m" (last\_task\_used\_math->tss.i387));
99     }
// Now, 'last_task_used_math' points to the current task, in case the current task is
// swapped out. At this point, if the current task has used the coprocessor, its state is
// restored. Otherwise, it is the first time to use, so the initialization command is sent
// to the coprocessor, and the coprocessor flag is set.
100     last\_task\_used\_math=current;
101     if (current->used_math) {
102         __asm__("frstor %0"::"m" (current->tss.i387));
103     } else {
104         __asm__("fninit"::); // send initial cmd to the math.
105         current->used_math=1; // set used math flag.
106     }

```



```

107 }
108
109 /*
110  * 'schedule()' is the scheduler function. This is GOOD CODE! There
111  * probably won't be any reason to change this, as it should work well
112  * in all circumstances (ie gives IO-bound processes good response etc).
113  * The one thing you might take a look at is the signal-handler code here.
114  *
115  * NOTE!! Task 0 is the 'idle' task, which gets called when no other
116  * tasks can run. It can not be killed, and it cannot sleep. The 'state'
117  * information in task[0] is never used.
118  */
119 void schedule(void)
120 {
121     int i,next,c;
122     struct task_struct ** p;          // pointer's pointer of task struct.
123
124     /* check alarm, wake up any interruptible tasks that have got a signal */
125
126     // Start checking the alarm from the last task in the task array. Skip empty pointer items
127     // when looping.
128     for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
129         if (*p) {
130             // If the task timeout is set and has expired (jiffies>timeout), then timeout is reset to
131             // 0, and if the task is in TASK_INTERRUPTIBLE sleep state, then it is put into the ready
132             // state (TASK_RUNNING).
133             if ((*p)->timeout && (*p)->timeout < jiffies) {
134                 (*p)->timeout = 0;
135                 if ((*p)->state == TASK_INTERRUPTIBLE)
136                     (*p)->state = TASK_RUNNING;
137             }
138             // If the timeout alarm value of the SIGALRM signal of the task is set and has expired
139             // (alarm<jiffies), the SIGALRM signal is set in the signal bitmap, that is, the SIGALRM
140             // signal is sent to the task, and then the alarm is cleared. The default action for this
141             // signal is to terminate the task.
142             if ((*p)->alarm && (*p)->alarm < jiffies) {
143                 (*p)->signal |= (1<<(SIGALRM-1));
144                 (*p)->alarm = 0;
145             }
146             // If there are other signals in the signal bitmap in addition to the blocked signal and
147             // the task is in an interruptible state, then set the task to ready state (TASK_RUNNING).
148             // Where '~(_BLOCKABLE & (*p)->blocked)' is used to ignore blocked signals, but SIGKILL
149             // and SIGSTOP signals cannot be blocked.
150             if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
151                 (*p)->state==TASK_INTERRUPTIBLE)
152                 (*p)->state=TASK_RUNNING;      // set ready.
153         }
154
155     /* this is the scheduler proper: */
156
157     while (1) {
158         c = -1;
159         next = 0;

```

```

147         i = NR_TASKS;
148         p = &task[NR_TASKS];
// This code is also looped from the last task of the task array and skips the empty
// slots. It compares the counter (the countdown number of task run time) of each ready
// state task. Which value is large, it means that there are still many running times of
// the task, and next points to the task number of that task.
149         while (--i) {
150             if (!*--p)
151                 continue;
152             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
153                 c = (*p)->counter, next = i;
154         }
// If the comparison results in a result with a counter value not equal to 0, or if there
// is no executable task in the system (c is still -1, next=0), then exit the outer while
// loop (144 lines), execute the latter task switching macro(line 161). Otherwise, the
// counter value of each task is updated according to the priority of each task, and then
// back to 144 lines for re-comparison. The counter value is calculated as
// counter = counter /2 + priority
// Note that the calculation process here does not consider the state of the process.
155         if (c) break;
156         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
157             if (*p)
158                 (*p)->counter = ((*p)->counter >> 1) +
159                     (*p)->priority;
160     }
// The following macro (sched.h) takes the selected task next as the current task and
// switches to the task to run. Because next is initialized to 0 on line 146, next is
// always 0 if there are no other tasks in the system to run. Therefore, the scheduler
// will run task 0 when the system is idle. At this time, task 0 only executes the pause()
// syscall, and causes this function to be called again.
161     switch to(next);
162 }
163
// This is the pause() system-call, which is used to convert the current task's state to
// an interruptible wait state (TASK_INTERRUPTIBLE) and reschedule.
// This system-call will cause the process to go to sleep until a signal is received. This
// signal is used to terminate the process or cause the process to call a signal capture
// function. Pause() returns only if a signal is caught and the signal capture handler
// returns. At this point the pause() return value should be -1 and errno is set to EINTR.
// It has not been fully implemented yet (until kernel 0.95).
164 int sys_pause(void)
165 {
166     current->state = TASK_INTERRUPTIBLE;
167     schedule();
168     return 0;
169 }
170
// The following function sets the current task to an interruptible or uninterruptible
// sleep state and has the sleep queue head pointer point to the current task.
// The function parameter p is the wait pointer of the task queue; the parameter state is
// the state used by the task sleep: TASK_UNINTERRUPTIBLE or TASK_INTERRUPTIBLE. Tasks in
// the uninterruptible sleep state require the kernel to explicitly wake up using the
// wake_up() function; tasks in the interruptible sleep state can be woken up by signal,

```

```

// task timeout, etc. (set to ready state TASK_RUNNING).
// *** Note that because this code is not very mature, there are some issues with it.
171 static inline void sleep_on(struct task_struct **p, int state)
172 {
173     struct task_struct *tmp;
174
175     // First, if the pointer is invalid, it will exit. (The object pointed to by the pointer
176     // can be NULL, but the pointer itself will not be 0). If the current task is task 0, the
177     // kernel panic.
178     if (!p)
179         return;
180     if (current == &(init_task.task))
181         panic("task[0] trying to sleep");
182     // Then let tmp point to the task already on waiting queue (if any), such as inode->i_wait,
183     // and point the sleep queue head to the current task. This inserts the current task into
184     // the wait queue of *p. The current task is then placed in the specified wait state and
185     // rescheduling is performed.
186     tmp = *p;
187     *p = current;
188     current->state = state;
189     repeat: schedule();
190
191     // Only when this waiting task is awakened will the program continue to execute from
192     // here. Indicates that the process has been explicitly woken up and executed.
193     // If there are still waiting tasks in the queue, and the task pointed to by the queue
194     // header *p is not the current task, then there is still tasks entering the queue after
195     // the task is inserted. Therefore, we should also awaken these subsequent tasks entered
196     // laterly. So the task indicated by the queue header is set to the ready state first, and
197     // the current task itself is set to the uninterruptible wait state. That is, after waiting
198     // for these subsequent queued tasks to be awakened, the current task itself can then be
199     // woken up by using the wake_up() function. Then jump to label repeat and re-execute the
200     // schedule() function.
201     if (*p && *p != current) {
202         (*p).state = 0;
203         current->state = TASK_UNINTERRUPTIBLE;
204         goto repeat;
205     }
206
207     // Execution here, indicating that this task is really awakened to execute. At this point
208     // the queue head should point to this task. If it is empty, it indicates that there is a
209     // problem with the schedule, and a warning message is displayed. Finally, we let the head
210     // point to the task that entered the queue in front of us (*p = tmp). If there is such a
211     // task, that is, there are tasks in the queue (tmp is not empty), it will be woken up.
212     // Therefore, the task that first enters the queue will eventually set the wait queue
213     // header to NULL when it is run after wakeup.
214     if (!*p)
215         printk("Warning: *P = NULL\n|r");
216     if (*p = tmp)
217         tmp->state=0;
218 }
219
220 // Set the current task to the interruptible wait state (TASK_INTERRUPTIBLE) and put it
221 // into the wait queue specified by the head pointer *p. This wait state task can be

```

```

// awakened by means of signals, task timeouts, and the like.
194 void interruptible_sleep_on(struct task_struct **p)
195 {
196     __sleep_on(p, TASK_INTERRUPTIBLE);
197 }
198
// Set the current task to the interruptible wait state (TASK_UNINTERRUPTIBLE) and put
// it into the wait queue specified by the head pointer *p. This wait state task can only
// be awakened by wait_up() function.
199 void sleep_on(struct task_struct **p)
200 {
201     __sleep_on(p, TASK_UNINTERRUPTIBLE);
202 }
203

// Wake up the uninterruptible wait task. *p is the task wait queue head pointer. Since
// the new wait task is inserted at the wait queue head, the wake up is the last task to
// enter the wait queue. If the task is already in a stopped or zombie state, a warning
// message is displayed.
204 void wake_up(struct task_struct **p)
205 {
206     if (p && *p) {
207         if ((*p).state == TASK_STOPPED)
208             printk("wake_up: TASK_STOPPED");
209         if ((*p).state == TASK_ZOMBIE)
210             printk("wake_up: TASK_ZOMBIE");
211         (*p).state=0; // TASK_RUNNING
212     }
213 }
214
215 /*
216  * OK, here are some floppy things that shouldn't be in the kernel
217  * proper. They are here because the floppy needs a timer, and this
218  * was the easiest way of doing it.
219  */
// The following 220--281 lines of code are used to handle the floppy drive timing.
// Before reading this code, please take a look at the instructions in the chapter on
// block devices for floppy drivers (floppy.c), or look at this code when reading the
// floppy block device driver.
//
// The array wait_motor[] is used to store the process pointer waiting for drive motor
// to start up to normal speed. The array index 0-3 corresponds to the floppy drive A-D.
// The array mon_timer[] stores the number of ticks required for each floppy drive motor
// to start. The default startup time in the program is 50 ticks (0.5 seconds).
// The array moff_timer[] stores the time each floppy drive needs to maintain before the
// motor stalls. The program is set to 10,000 ticks (100 seconds).
220 static struct task_struct * wait_motor[4] = {NULL, NULL, NULL, NULL};
221 static int mon_timer[4]={0, 0, 0, 0};
222 static int moff_timer[4]={0, 0, 0, 0};

// The following variables correspond to the current digital output register (DOR) in the
// floppy drive controller. The definition of each bit of this register is as follows:
// Bits 7-4: control the activation of the drive D-A motor separately. 1-Start; 0-Close.

```

```

// Bit 3:1 - enable DMA and interrupt requests; 0 - disable DMA and interrupt requests.
// Bit 2:1 - start floppy drive controller(FDC); 0 - reset FDC.
// Bits 1-0: Used to select the floppy drive A-D.
// The initial value set here is: Allow DMA and interrupt request, start FDC.
223 unsigned char current_DOR = 0x0C;
224
// Specifies the wait time for the floppy drive from begin to normal operation.
// Parameter nr is floppy drive number (0--3), the return value is the nr of ticks.
// The variable selected is the selected floppy drive flag (blk_drv/floppy.c, line 123).
// The mask is the start motor bits in the selected floppy drive DOR, and the upper 4 bits
// are the floppy drive start motor flags.
225 int ticks to floppy on(unsigned int nr)
226 {
227     extern unsigned char selected;
228     unsigned char mask = 0x10 << nr;
229
// The system has up to 4 floppy drives. First, set the time (100 seconds) that the
// specified floppy drive nr needs to stop before it stops. Then take the current DOR
// value into variable mask, and set the motor start flag of the specified floppy drive
// in it.
230     if (nr>3)
231         panic("floppy_on: nr>3");
232     moff_timer[nr]=10000;           /* 100 s = very big :-) */
233     cli();                         /* use floppy_off to turn it off */
234     mask |= current_DOR;
// If the floppy drive is not currently selected, first reset the selection bits of other
// floppy drive, then set the floppy drive selection bit.
235     if (!selected) {
236         mask &= 0xFC;
237         mask |= nr;
238     }
// If the current value of DOR is different from the required value, a new value (mask) is
// output to the FDC digital output port, and if the motor that is required to start is
// not yet started, the motor start timer value of the corresponding floppy drive is set
// (HZ/2 = 0.5) Seconds or 50 ticks). If it has been started, set the startup timing to 2
// ticks, which can meet the requirements of the following depreciation in do_floppy_timer().
// The current digital output register current_DOR is updated thereafter.
239     if (mask != current_DOR) {
240         outb(mask, FD_DOR);
241         if ((mask ^ current_DOR) & 0xf0)
242             mon_timer[nr] = HZ/2;
243         else if (mon_timer[nr] < 2)
244             mon_timer[nr] = 2;
245         current_DOR = mask;
246     }
247     sti();                         // enable int.
248     return mon_timer[nr];          // return time value required to start motor.
249 }
250
// wait for a period of time required to start the floppy drive motor.
// Sets the delay time required for the motor of the specified floppy drive from start to
// normal speed, then sleeps. During the timer interrupt process, the delay value set here
// is decremented. When the delay expires, it will wake up the waiting process here.

```

```

251 void floppy_on(unsigned int nr)
252 {
    // Disable interrupt. If the motor start timer has not expired, the current process is
    // always placed in an uninterruptible sleep state and placed in a queue waiting for the
    // motor to run. Then open the interrupt.
253     cli();
254     while (ticks_to_floppy_on(nr))
255         sleep_on(nr+wait_motor);
256     sti();
257 }
258
    // Set to turn off the motor stall timer (3 seconds).
    // If you do not use this function to explicitly turn off the specified floppy drive
    // motor, it will be turned off after the motor is turned on for 100 seconds.
259 void floppy_off(unsigned int nr)
260 {
261     moff_timer[nr]=3*HZ;
262 }
263
    // The floppy disk timer subroutine. Update the motor start timing value and the motor off
    // stall count value. This subroutine is called during the system timer interrupt, so the
    // system is called once every time a tick (10ms) is passed, and the value of the motor on
    // or off timer is updated at each time. If a motor stall timing expires, the DOR motor
    // start bit is reset.
264 void do_floppy_timer(void)
265 {
266     int i;
267     unsigned char mask = 0x10;
268
    // For the four floppy drives that the system have, check the floppy drives in use one by
    // one. Skip if it is not the motor specified by DOR. If the motor start timer expires,
    // the process is woken up. If the motor off timer expires, reset motor's start bit.
269     for (i=0 ; i<4 ; i++,mask <=> 1) {
270         if (!(mask & current_DOR))
271             continue;
272         if (mon_timer[i]) {
273             if (!--mon_timer[i]) // if motor on timer expires
274                 wake_up(i+wait_motor); // wake up the process.
275         } else if (!moff_timer[i]) {
276             current_DOR &= ~mask; // reset motor start bit
277             outb(current_DOR, FD_DOR); // update DOR.
278         } else
279             moff_timer[i]--;
280     }
281 }
282
    // Below is the code for the kernel timer. There can be up to 64 timers.
    // Lines 285-289 defines linked timer list structure and timer array. The linked timer
    // list is dedicated to the floppy drive to turn on and off motor for timing operation.
    // This type of timer is similar to the dynamic timer in modern Linux systems and is
    // intended for use only by the kernel.
283 #define TIME_REQUESTS 64
284

```

```

285 static struct timer_list {
286     long jiffies;                // Timer ticks.
287     void (*fn)();                // Timer handler.
288     struct timer_list * next;    // points to the next timer.
289 } timer_list[TIME REQUESTS], * next_timer = NULL; // next_timer is timer queue head.
290
    // Add timer subroutine. The input parameters are the specified timing values (ticks) and
    // the associated handler. The floppy disk driver uses this function to perform a delay
    // operation to start or shut down the motor.
    // jiffies - number of timed ticks; *fn() - function to be executed when the time is up.
291 void add_timer(long jiffies, void (*fn)(void))
292 {
293     struct timer_list * p;
294
    // If the timer handler pointer to be added is null, the function is exited.
295     if (!fn)
296         return;
297     cli();
    // If the timer time value <=0, its handler is called immediately and the timer is not
    // added to the linked list.
298     if (jiffies <= 0)
299         (fn)();
300     else {
    // Otherwise, find a free entry from the timer array.
301         for (p = timer_list ; p < timer_list + TIME REQUESTS ; p++)
302             if (!p->fn)
303                 break;
    // If the timer array has been used up, the system crashes :-). Otherwise, the timer data
    // structure is filled with information and linked into the list header.
304         if (p >= timer_list + TIME REQUESTS)
305             panic("No more time requests free");
306         p->fn = fn;
307         p->jiffies = jiffies;
308         p->next = next_timer;
309         next_timer = p;
    // The linked list items are sorted from early to late according to the time value.
    // Subtract the number of ticks needed before sorting. In this way, when processing the
    // timer, it is only necessary to check whether the timing of the first item expires.
310         while (p->next && p->next->jiffies < p->jiffies) {
311             p->jiffies -= p->next->jiffies;
312             fn = p->fn;
313             p->fn = p->next->fn;
314             p->next->fn = fn;
315             jiffies = p->jiffies;
316             p->jiffies = p->next->jiffies;
317             p->next->jiffies = jiffies;
318             p = p->next;
319         }
320     }
321     sti();
322 }
323
    /// The C function called in the timer interrupt handler. Called in _timer_interrupt

```

```

// (line 189, 209) in the sys_call.s file. The parameter cpl is the current privilege
// level 0 or 3, which is the privilege level in the code selector being executed when
// the interrupt occurs. Cpl=0 means that the kernel code is being executed when the
// interrupt occurs; cpl=3 means that the user code is being executed when the interrupt
// occurs. For a task, if its execution time slice is used up, the task is switched. At
// this point the function will perform a timing update.
324 void do\_timer(long cpl)
325 {
326     static int blanked = 0;
327
// First determine if you need to perform a screen blankout operation. If the blankcount
// is not zero, or the black screen delay interval blankinterval is 0, then if the screen
// is already in a black screen (black screen flag blanked = 1), the screen is restored.
// If the blankcount is not zero, it is decremented and the black screen flag is reset.
328     if (blankcount || !blankinterval) {
329         if (blanked)
330             unblank\_screen();
331         if (blankcount)
332             blankcount--;
333         blanked = 0;
// Otherwise, if the black screen flag is not set, the screen will be blank and the flag
// will be set.
334     } else if (!blanked) {
335         blank\_screen();
336         blanked = 1;
337     }

// Next, we handle the hard disk operation timeout issue. If the hard disk timeout count
// is decremented to 0, the hard disk access timeout processing is performed.
338     if (hd\_timeout)
339         if (!--hd\_timeout)
340             hd\_times\_out();           // blk_drv/hdc, line 318
341
// If the beep counts is reached, the beep is turned off. (cmd is sent to port 0x61, reset
// bits 0 and 1. Bit 0 controls the counter 2 of 8253 chip, bit 1 controls the speaker ).
342     if (beepcount)                     // beep ticks (chr_drv/console.c, 950)
343         if (!--beepcount)
344             sysbeepstop();           // chr_drv/console.c, 944.
345
// If the current privilege level (cpl) is 0 (the highest, indicating that the kernel
// program is working), then the kernel code runtime stime is incremented; if cpl > 0,
// it means that the general user program is working, adding utime.
346     if (cpl)
347         current->utime++;
348     else
349         current->stime++;
350
// If a timer exists, the value of the first timer in the linked list is decremented by
// one. If it is equal to 0, the corresponding handler is called, and the handler pointer
// is set to null, and then the timer is removed.
351     if (next\_timer) {                 // timer list header.
352         next\_timer->jiffies--;
353         while (next\_timer && next\_timer->jiffies <= 0) {

```



```

354         void (*fn)(void);           // a function pointer definition.
355
356         fn = next_timer->fn;
357         next_timer->fn = NULL;
358         next_timer = next_timer->next;
359         (fn)();                      // call the timer handler.
360     }
361 }
// If the motor enable bit in the DOR of the current floppy disk controller FDC is set,
// the floppy disk timer routine is executed.
362     if (current_DOR & 0xf0)
363         do floppy timer();
// If the task still has run time, exit here to continue running the task. Otherwise, the
// current task running count is set to 0. And if it is running in the kernel code when
// the interrupt occurs, it returns, otherwise it means that the user program is being
// executed, so the scheduler is called to try to perform task switching operation.
364     if ((--current->counter)>0) return;
365     current->counter=0;
366     if (!cpl) return;                // kernel code
367     schedule();
368 }
369
// System-call function - Sets the alarm timer value (in seconds).
// If the parameter seconds > 0, the new timing is set, and the remaining interval is
// returned to the original timing, otherwise it returns 0.
// The unit of the alarm field in the process data structure is tick, which is the sum of
// the system tick value jiffies and the timing value, ie 'jiffies + HZ* seconds', where
// the constant HZ = 100. The main operation of this function is to set the alarm field
// and convert between two time units.
370 int sys_alarm(long seconds)
371 {
372     int old = current->alarm;
373
374     if (old)
375         old = (old - jiffies) / HZ;
376     current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
377     return (old);
378 }
379
// Get current process pid.
380 int sys_getpid(void)
381 {
382     return current->pid;
383 }
384
// Get parent pid - ppid.
385 int sys_getppid(void)
386 {
387     return current->p_pptr->pid;
388 }
389
// Get current user id.
390 int sys_getuid(void)

```

---

```

391 {
392     return current->uid;
393 }
394
395 // Get effective user id - euid.
396 int sys_geteuid(void)
397 {
398     return current->euid;
399 }
400 // Get group id - gid
401 int sys_getgid(void)
402 {
403     return current->gid;
404 }
405 // Get effective group id - egid.
406 int sys_getegid(void)
407 {
408     return current->egid;
409 }
410 // System call function -- Reduce the priority of using the CPU (someone will use it?).
411 // parameter increment should be limited to a value greater than 0.
412 int sys_nice(long increment)
413 {
414     if (current->priority-increment>0)
415         current->priority -= increment;
416     return 0;
417 }
418 // The initialization subroutine of the kernel scheduler.
419 void sched_init(void)
420 {
421     int i;
422     struct desc_struct * p;          // descriptor structure pointer
423
424     // At the beginning of Linux development, the kernel was not mature. The kernel code is
425     // often modified. Mr. Linus feared that he had inadvertently modified these critical
426     // data structures, causing incompatibility with the POSIX standard, so add the
427     // following statement here. This is not necessary, it is purely to remind himself and
428     // others who modify the kernel code.
429     if (sizeof(struct sigaction) != 16)        // signal struct
430         panic("Struct sigaction MUST be 16 bytes");
431
432     // The task state segment (TSS) descriptor and the local data table (LDT) descriptor of
433     // the initial task (task 0) are set in the global descriptor table (GDT).
434     // The value of FIRST_TSS_ENTRY and FIRST_LDT_ENTRY is 4 and 5 respectively, defined in
435     // file linux/sched.h. gdt is a descriptor array (linux/head.h), it associated with the
436     // base address _gdt in file head.s, line 234. Therefore, gdt + FIRST_TSS_ENTRY is
437     // gdt[FIRST_TSS_ENTRY] (ie gdt[4]), which is the address of item 4 of the gdt array.
438     // See asm/system.h, line 65.
439     set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));

```

```

425     set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task,ldt));
    // Clear the task array and descriptor table entries (note that starting with i=1, so the
    // descriptor for the initial task is still there).
426     p = gdt+2+FIRST_TSS_ENTRY;
427     for(i=1;i<NR_TASKS;i++) {
428         task[i] = NULL;
429         p->a=p->b=0;
430         p++;
431         p->a=p->b=0;
432         p++;
433     }
434     /* Clear NT, so that we won't have troubles with that later on */
    // The NT flag in EFLAGS is used to control nested calls to tasks. When NT is set, the
    // current interrupt task will cause a task switch when the IRET instruction is executed.
    // NT indicates whether the back link field in the TSS is valid. Invalid when NT=0.
435     __asm__ ("pushfl ; andl $0xffffbfff, (%esp) ; popfl");    // reset NT

    // The TSS segment selector of task 0 is loaded into the task register (TR). The LDT
    // segment selector is loaded into the local descriptor table register (LDTR). Note! The
    // selector of the corresponding LDT descriptor in the GDT is loaded into the LDTR. It
    // only explicitly loads this time. Later, the loading of the new task LDT is
    // automatically loaded by the CPU according to the LDT entry in the TSS.
436     ltr(0);    // include/linux/sched.h, 157-158
437     lldt(0);    // 0 is task no.

    // The following code is used to initialize the 8253 timer. Channel 0, select working
    // mode 3, binary counting mode. The output pin of channel 0 is connected to the IRQ0 of
    // the interrupt control master chip, which issues an IRQ0 request every 10 milliseconds.
    // LATCH is the initial timing count value.
438     outb_p(0x36,0x43);    /* binary, mode 3, LSB/MSB, ch 0 */
439     outb_p(LATCH & 0xff , 0x40);    /* LSB */
440     outb(LATCH >> 8 , 0x40);    /* MSB */

    // Set the timer interrupt handler. Modify the interrupt controller mask code to enable
    // timer interrupt occurs. Then set system-call interrupt gate. The macro definitions of
    // the descriptors are at lines 33 and 39 of file asm/system.h.
441     set_intr_gate(0x20,&timer_interrupt);
442     outb(inb_p(0x21)&~0x01,0x21);    // change int mask, enable timer.
443     set_system_gate(0x80,&system_call);
444 }
445

```

## 8.6.3 Information

### 8.6.3.1 Floppy Drive Controller

For the application in the above program, only the I/O port used by the floppy disk controller (FDC) is briefly introduced here. For a detailed description of FDC programming, see the explanations in Chapter 9 after floppy.c. Four ports need to be accessed when programming the FDC. These ports correspond to one or more registers on the controller. For a normal floppy disk controller there are some ports shown in Table 8-3.

Table 8-3 Floppy drive controller ports

I/O port	Port name	Read/Write	Register name
----------	-----------	------------	---------------

0x3f2	FD_DOR	Write only	Digital output register (digital controller register)
0x3f4	FD_STATUS	Read only	FDC main status register
0x3f5	FD_DATA	Read/Write	FDC data register
0x3f7	FD_DIR	Read only	Digital input register
0x3f7	FD_DCR	Write only	Drive control register (transfer rate control)

The digital output register DOR (or digital control) is an 8-bit register that controls driver motor turn-on, driver select, start/reset FDC, and enable/disable DMA and interrupt requests.

The FDC's main status register is also an 8-bit register that reflects the basic state of the FDC and floppy disk drive FDD. Typically, the status bits of the main status register are read before the CPU sends a command to the FDC or before the FDC obtains the result of the operation to determine if the current FDC data register is ready and to determine the direction of data transfer.

The data port of the FDC corresponds to multiple registers (write-only command register and parameter register, read-only result register), but only one register can appear on data port 0x3f5 at any one time. When accessing a write-only register, the DIO direction bit of the main state control must be 0 (CPU → FDC), and vice versa when accessing the read-only register. When reading the result, the result is only read after the FDC is not busy. Usually, the result data has a maximum of 7 bytes.

The floppy disk controller can accept a total of 15 commands. Each command goes through three phases: the command phase, the execution phase, and the results phase.

The command phase is that the CPU sends command bytes and parameter bytes to the FDC. The first byte is always the command byte (command code) followed by a parameter of 0-8 bytes.

The execution phase is the operation specified by the FDC execution command. In the execution phase, the CPU does not intervene. Generally, the FDC issues an interrupt request to know the end of the command execution. If the FDC command sent by the CPU is to transfer data, the FDC can be performed in an interrupt mode or in a DMA manner. The interrupt mode transfers 1 byte at a time. The DMA mode is under the management of the DMA controller, and the FDC and the memory transfer data until all the data is transmitted. At this time, the DMA controller notifies the FDC of the transmission byte count termination signal, and finally the FDC issues an interrupt request signal to inform the CPU that the execution phase is over.

The result phase is that the CPU reads the FDC data register return value to obtain the result of the FDC command execution. The result data returned is 0-7 bytes in length. For commands that do not return result data, the FDC should be sent a detect interrupt status command to get the status of the operation.

### 8.6.3.2 Programmable Timer/Counter Controller

#### 1. Intel 8253 (8254) chip

The Intel 8253 (or 8254) is a programmable Timer/Counter chip that solves the time control problems typically encountered in computers, ie, produces precise time delays under software control. The chip provides three independent 16-bit counter channels. Each channel can work in different operating modes, and these can be set using software. The 8254 is an updated product of the 8253 chip. The main functions are basically the same, except that the 8254 chip adds a readback command. In the following description, we use 8253 to refer to the 8253 and 8254 chips, and only point out where they differ in their functions.

The programming of the 8253 chip is relatively simple and can produce the desired delays of various lengths of time. A block diagram of the 8253 (8254) chip is shown in Figure 8-8.

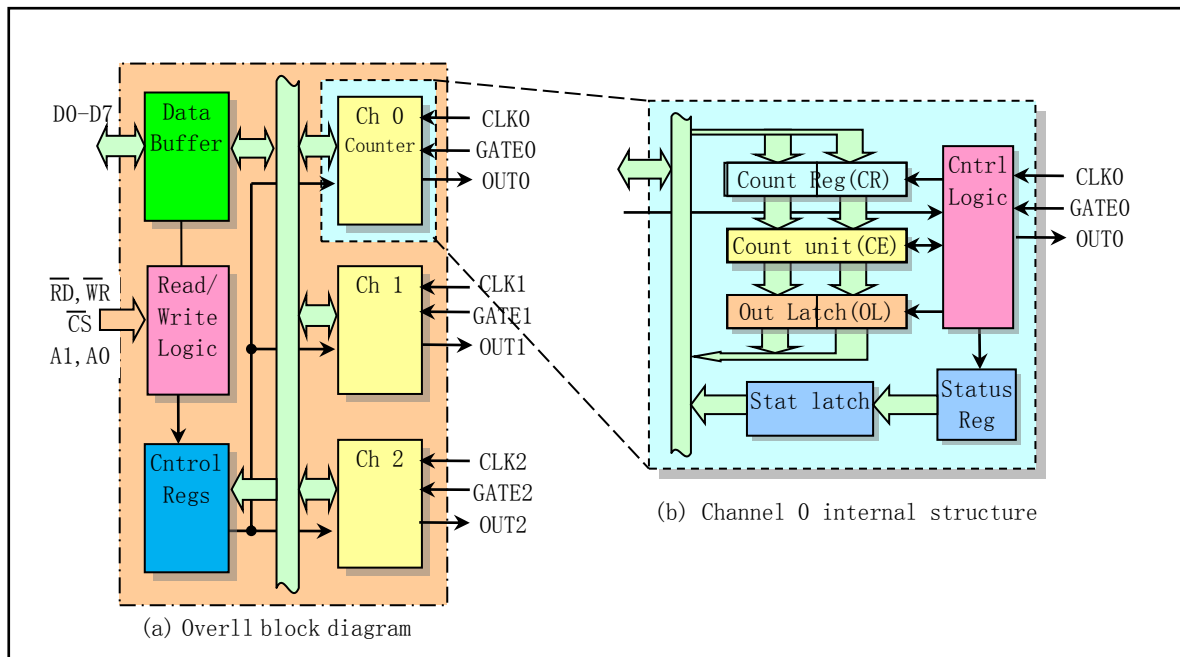


Figure 8-8 8253 (8254) Timer/Counter chip internal structure

The 3-state, bidirectional 8-bit Data Bus Buffer is used to interface with the system data bus. Read/Write Logic is used to receive input signals from the system bus and generate control signals that are input to other parts. Address lines A1, A0 are used to select one of the three counter channels or Control Word Registers that need to be read/written. Usually they are connected to the A0, A1 address line of the system. The read and write pins RD, WR and chip select pin CS are used by the CPU to control the read and write operations of the 8253 chip. The Control Word Register is used by the CPU to set how the specified counter works. It is a write-only register. But for the 8254 chip, you can use the Read-Back Command to read the status information. The three independent counter channels function exactly the same, each of which can work in different ways. The Control Word register will determine how each counter works. The CLK pin of each counter is connected to the clock frequency generator (crystal oscillator). The 8253 has a clock input frequency of up to 2.6MHz, while the 8254 can be up to 10MHz. Pin GATE is the gate control input of the counter, which is used to control the start and stop of the counter and the output state of the counter. Pin OUT is the output signal terminal of the counter.

Figure 8-8(b) is an internal logic block diagram of one of the counter channels. The status register will contain the current contents of the control word register and the status of the output and the Null Count Flag when locked. The actual counter is the CE (counting unit) in the figure. It is a 16-bit pre-settable synchronous down counter. The output latch OL (Output Latch) is composed of two 8-bit latches, OLm and OL1, which represent the high byte and low byte of the latch, respectively. Usually the contents of the two output latches change following the content change of the counting unit CE, but if the chip receives a counter latch command, then their contents will be locked. Until the CPU reads their contents, they will continue to follow the CE content changes. Note that the value of CE is unreadable. Whenever you need to read the count value, the contents of the latch OL are always output. The other two in Figure 8-8(b) are 8-bit registers called the Count Register (CR). When the CPU writes a new count value to the counter channel, the initial count value is stored in these two registers and then copied to the count unit CE. These two registers will be cleared when the counter is programmed. Therefore, after the initial count value is saved in the count register CR, it is sent to the counting unit CE. When GATE is enabled, the counting unit performs a countdown operation under the action of the clock pulse CLK. Each time it is decremented by one, until the count value is decremented to zero, a

signal is sent to the OUT pin.

## 2. 8253 (8254) chip programming

When the system is just powered up, the state of the 8253 is unknown. By writing a control word and an initial count value to the 8253, we can program a counter we want to use. For counters that are not used we don't have to program them. Table 8–4 shows the format of the contents of the control registers.

Table 8-4 8253 (8254) chip control word format

Bit	Name	Description
7	SC1	SC1, SC0 are used to select counter channel 0-2, or to read back the command. 00 - Channel 0; 01 - Channel 1; 02 - Channel 2; 11 - Readback command (only in 8254).
6	SC0	
5	RW1	RW1 and RW0 are used for counter read/write operation selection. 00 - indicates a register latch command; 01 - Reads/writes the low byte (LSB); 10 - Read/write high byte (MSB); 11 - Read/write low byte first, then high byte.
4	RW0	
3	M2	M2-M0 is used to select the working mode of the specified channel. 000 - mode 0; 001 - mode 1; 010 - mode 2; 011 - mode 3; 100 - mode 4; 101 - mode 5.
2	M1	
1	M0	
0	BCD	Count value format selection. 0 - 16 bit binary count; 1 - 4 BCD code counts.

When the CPU performs a write operation, if the A1 and A0 lines are 11 (in this case, the corresponding port 0x43 on the PC microcomputer), the control word is written into the control word register. The contents of the control word specify the counter channel being programmed. The initial count value is written to the specified counter. When A1 and A0 are 00, 01, and 10 (corresponding to PC ports 0x40, 0x41, and 0x42, respectively), one of the three counters is selected. In a write operation, the control word must be written first and then the initial count value. The initial count value must be written in the format set in the control word (binary or BCD code format). When the counter starts working, we can still rewrite the new initial value to the specified counter at any time. This does not affect how the counters that have been set work.

During the read operation, there are three ways to read the counter's current count value for the 8254 chip: (1) simple read operation; (2) use counter latch command; (3) use readback command. The first method must temporarily stop the clock input of the counter using the GATE pin or the corresponding logic circuit during reading. Otherwise the counting operation may be in progress, resulting in an incorrect result of the reading. The second method is to use the counter latch command. The command is first sent to the control word register before the read operation, and the two bits (00) of D5, D4 indicate that the counter latch command is sent instead of the control word command. When the counter receives the command, it latches the count value in the counting unit CE into the output latch register OL. At this point, if the CPU does not read the contents of the OL, the value in the OL will remain the same, even if you send another counter latch command. Only after the CPU performs the reading of the counter operation, the contents of the OL will automatically follow the counting unit CE to change. The third method is to use the readback command. But only the 8254 has this feature. This command allows the program to detect the current count value, how the counter is run, and the current output status and NULL count flag. Similar to the second method, after the count value is locked, the contents of the OL will automatically follow the counting unit CE again after the CPU performs the reading of the counter operation.

## 3. Counter working mode

The 3 counter channels of 8253/8254 can work independently. There are 6 ways to choose from.

(1) Mode 0 - Interrupt on terminal count

After this mode is set, the output pin OUT is low and remains low until the count is decremented to zero. At this time, OUT goes high and stays high until a new count value is written or the control word is reset to mode 0. This method is usually used for event counting. This mode is characterized by the use of the GATE pin to control the count pause; the output goes high at the end of the count as an interrupt signal; the initial count value can be reloaded during the count and re-executed after the count high byte is received.

(2) Mode 1 - Hardware Retriggerable One-shot

When working in this mode, OUT is initially at a high level. The counter is ready after the CPU has written the control word and the initial count value. At this point, the GATE pin rising edge triggers the counter to start operation and OUT turns low. Until the end of the count (0), OUT goes high. During the counting period or after the counting is completed, GATE will go high again and trigger the counter to load the initial count value and restart the counting operation. For this mode of operation, the GATE signal does not work.

(3) Mode 2 - Rate Generator

The function of this mode is similar to an N divider. Usually used to generate real-time clock interrupts. OUT is high in the initial state. When the count value is decremented to 1, OUT goes low and then goes high. The interval is one CLK pulse width. At this point the counter will reload the initial value and repeat the above process. Therefore, for the case where the initial count value is N, a low-level pulse signal is output every N clock pulses. In this way GATE can control the pause and continuation of the count. When GATE goes high, the counter is reloaded with the initial value and begins to recount.

(4) Mode 3 - Square Wave Mode

This method is usually used for the baud rate generator. This mode is similar to mode 2, but the OUT output is a square wave. If the initial count value is N, the frequency of the square wave is one-N of the input clock CLK. The characteristic of this mode is that the square wave duty cycle is about 1 to 1 (slightly different when N is odd), and if the new initial value is reset during the counter decrement, the new initial value takes effect only after the previous count is completed.

(5) Mode 4 - Software Triggered Strobe

OUT is high in the initial state. When the count ends, OUT will output a low level of the clock pulse width and then go high (low level gate). The counting operation is "triggered" by writing the initial count value. In this mode of operation, the GATE pin can control the count pause (1 allow count), but does not affect the state of OUT. If a new initial value is written during the counting process, the counter will use the new value to recount after one clock pulse.

(6) Mode 5 - Hardware Triggered Strobe

OUT is high in the initial state. The counting operation will be triggered by the rising edge of the GATE pin. When the count is over, OUT will output a low level of the clock CLK pulse width and then go high. After writing the control word and the initial value, the counter does not immediately load the initial count value and starts working. It will only be triggered to start operation after a CLK clock pulse after the GATE pin goes high.

For PC/AT and its compatible microcomputer system, the 8254 chip is used. Three timer/counter channels are used for the clock timing interrupt signal, the dynamic memory DRAM refresh timing circuit, and the host speaker tone synthesis. The input clock frequencies of the three counters are all 1.193180MHz. The connection diagram of the 8254 chip in the PC/AT microcomputer is shown in Figure 8-9. The A1 and A0 pins are connected to the system address lines A1 and A0, and the 8254 chip is selected when the system address line A9--A2 signal is 0b0010000. Therefore, the I/O port range of 8254 chip is 0x40--0x43. Among them,

0x40--0x42 corresponds to select counter channel 0--2, and 0x43 corresponds to the control word register write port.

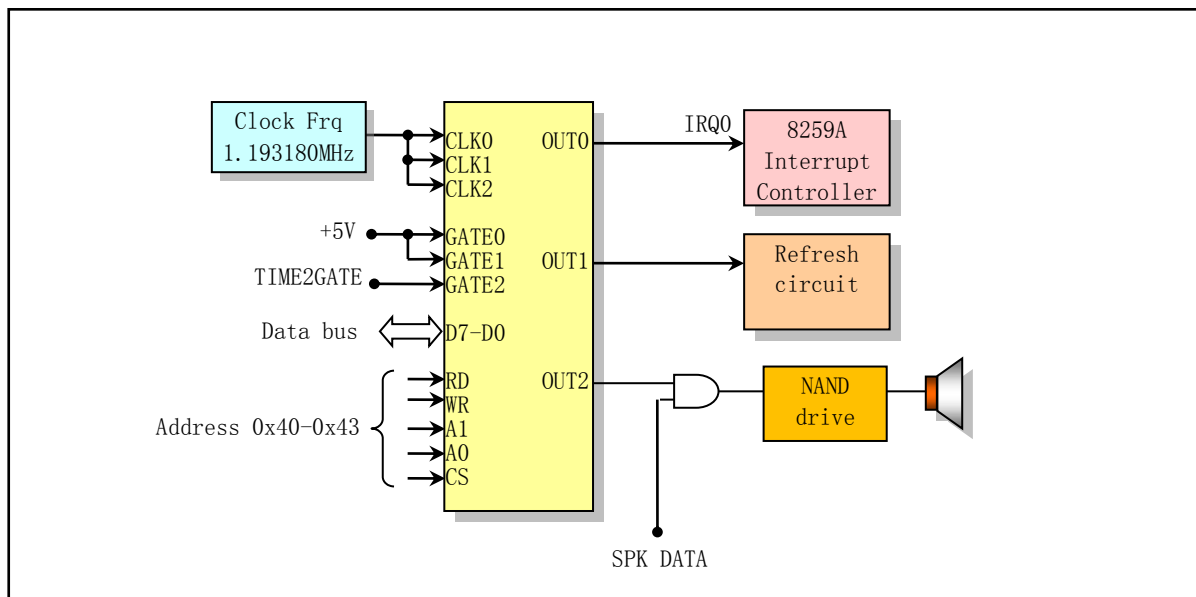


Figure 8-9 Timer/counter chip connection diagram in PC

For counter channel 0, its GATE pin is fixed high. When the system is powered on, it is set to work in mode 3 (square wave generator mode), and the initial count value is set to 0 by default, which means that the count value is 65536 (0--65535). Therefore, the OUT0 pin emits a square wave signal with a frequency of 18.2HZ (1.193180MHz/65536) every second. OUT0 is connected to the level 0 interrupt requester of the programmable interrupt controller 8259 chip. Therefore, using the rising edge of the square wave can trigger an interrupt request, causing the system to issue an interrupt request every 54.9ms (1000ms/18.2).

The GATE pin of counter channel 1 is also directly connected to the high level, so it is in the allowable count state. It works in mode 2 (frequency generator mode) and the initial value is usually set to 18. This counter is used to send a RAM refresh signal to the refresh circuit of the DMA controller channel 2 of the PC/XT system or the PC/AT system. A signal is output approximately every 15 microseconds with an output frequency of  $1.19318/18 = 66.288$  KHz.

The GATE pin (TIME2GATE) of counter channel 2 is connected to the D0 pin of the 8255A chip port B or equivalent logic. The SPK DATA in Figure 8-9 is connected to the D1 pin or equivalent logic of the 8255A chip port B (0x61). This counter channel is used to make the main speaker sound, but it can also be used as a normal timer with the 8255A chip (or its equivalent).

In Linux 0.12, the kernel only reinitializes counter channel 0 of the 8254 so that the counter operates in mode 3, the initial count value is binary, and the initial count value is set to LATCH (1193180/100). That is, the counter 0 sends a square wave rising edge signal every 10 milliseconds to generate an interrupt request signal (IRQ0). Therefore, the control word written to the 8254 is 0x36 (0b00110110), and then the low byte and the high byte of the initial count value are written. The initial count value low byte and high byte value are (LATCH & 0xff) and (LATCH >> 8), respectively. The interrupt request generated by this interval timing is the pulse of the Linux 0.12 kernel working. It is used to periodically switch the currently executing tasks, count the amount of system resources (time) used by each task, and implement kernel timing operations.



## 8.7 signal.c

### 8.7.1 Function Description

The `signal.c` program involves all the functions related to signal processing in the kernel. In UNIX-like systems, signals are a "software interrupt" processing mechanism. There are many more complicated programs that use signals. The signaling mechanism provides a way to handle asynchronous events, which can be used as a simple message mechanism for communication between processes, allowing one process to send signals to another. The signal is usually a positive integer, which does not carry any other information except to indicate its own signal class. For example, when a child process terminates or ends, a `SIGCHLD` signal is sent to the parent process to inform it about the current state of the child process; use the system function `kill()` to send termination execution signal to all child processes in the same group; typing `ctrl-C` will generate a `SIGINT` signal that is sent to the foreground process to terminate it. In addition, when an alarm timer set by the process expires, the system sends a `SIGALRM` signal to the process; when a hardware exception occurs, the system also sends a corresponding signal to the executing process.

Signal processing mechanisms existed in very early UNIX systems, but the methods of signal processing in earlier UNIX kernels were not as reliable. Signals may be lost, and it is sometimes difficult for a process to close a specified signal while processing a critical area code. Later POSIX provided a way to reliably process signals. In order to maintain compatibility, the Linux kernel provides both methods for processing signals.

#### 8.7.1.1 Signals in Linux

In Linux kernel code, bits in an unsigned long integer (32 bits) are typically used to represent a variety of different signals, so there can be up to 32 different signals in the system. In this version of the Linux kernel, 22 different signals are defined. 20 of these signals are those specified in the POSIX.1 standard, and the other 2 are Linux-specific signals: `SIGUNUSED` (undefined) and `SIGSTKFLT` (stack error). The former can represent all other signal types that the system does not currently support. For the specific names and definitions of these 22 signals, please refer to Table 8-4 of the signal list after the program. Also refer to the contents of the `include/signal.h` header file.

When a process receives a signal, there are three different ways of processing or operation: one is to ignore the signal, but two signals cannot be ignored (`SIGKILL` and `SIGSTOP`). The second approach is that the process defines its own signal handler to process the signal. The third is to perform the system's default signal processing operations.

1. Ignore this signal. Most signals can be ignored by the process. But there are two signals that can't be ignored: `SIGKILL` and `SIGSTOP`. The reason is to give the superuser a definite way to terminate or stop any process specified. In addition, if the signal generated by some hardware exceptions is ignored (for example, divided by 0), the behavior or state of the process may become agnostic.
2. Capture the signal. In order to perform the capture operation, we must first tell the kernel to call our custom signal handler when the specified signal occurs. In this handler we can do anything. Of course, you can do nothing and play the same role of ignoring the signal. An example of a custom signal processing function is: If we create some temporary files during program execution, then we can define a function to capture the `SIGTERM` (terminate execution) signal and do some cleanup in the function. The `SIGTERM` signal is the default signal sent by the `kill` command.
3. Perform the default action. The process does not process the signal, and the signal is processed by the system's corresponding default signal handler. The kernel provides a default action for each type of signal. Usually these default actions are to terminate the execution of the process. See the description in the

post-program signal list (Table 8-4).

### 8.7.1.2 Signal Processing Implementation

The signal.c program mainly includes: 1) the access signal blocking code system-calls `sys_ssetmask()` and `sys_sgetmask()`, 2) signal handling syscall `sys_signal()` (the traditional signal handling function), 3) handler for modification signal actions syscall `sys_sigaction()` (the reliable signal handler), 4) and the function `do_signal()` that processes the signal in the system call interrupt handler. The send signal function `send_sig()` and the notification parent process function `tell_father()` are included in another source file (`exit.c`). In addition, the name prefix 'sig' in the code is short for signal.

The functions of `signal()` and `sigaction()` are similar, and can be used to change the signal handler. However, `signal()` is the traditional way for the kernel to process signals, which can cause signal loss at certain special times. When the user wants to use his own signal handler (signal handle), the user needs to use the `signal()` or `sigaction()` syscall to first set the `sigaction[]` structure array item in the task's data structure, and put the signal handler and some attributes in the structure for itself. When the kernel returns from a system-call or some interrupt procedures, it will detect if the current process receives a signal. If a specific signal specified by the user is received, the kernel executes the user-defined signal processing service program according to the structure item in `sigaction[]` in the process task data structure.

#### 1. `signal()` fuction

On line 62 of the `include/signal.h` header file, the `signal()` function signature is declared as follows:

---

```
void (*signal(int signr, void (*handler)(int)))(int);
```

---

This `signal()` function takes two arguments. One is to specify the signal `signr` to be captured; the other is the new signal handler pointer `void (*handler)(int)`. This new signal handler is a function pointer with no return value and an integer parameter that is passed to the handler when the specified signal occurs.

The prototype declaration of the `signal()` function seems complicated, but if we define a type like this:

---

```
typedef void sigfunc(int);
```

---

Then we can rewrite the prototype of the `signal()` function to the following simple form:

---

```
sigfunc *signal(int signr, sigfunc *handler);
```

---

The `signal()` function installs a new signal handler for the signal `signr`. The signal handler can be a signal processing function specified by the user, or it can be a specific function pointer `SIG_IGN` or `SIG_DFL` provided by the kernel. When the specified signal arrives, the signal is ignored if the associated signal processing handler is set to `SIG_IGN`. If the signal handler is `SIG_DFL`, then the default operation of the signal is performed. Otherwise, if the signal handler is set to a user's signal handler, the kernel first resets the signal handle to its default handle, performs an implementation-related signal blocking operation, and then invokes the specified signal handler.

The `signal()` function returns the original signal handler, and the returned handler is also a function pointer with no return value and with an integer argument, and after the new handler is called once, it is restored to the default processing handler value `SIG_DFL`.

In the include/signal.h file (starting on line 46), the default handle SIG\_DFL and the ignore handle SIG\_IGN are defined as:

---

```
#define SIG_DFL      ((void (*)(int))0)
#define SIG_IGN      ((void (*)(int))1)
```

---

They all represent function pointers with no return value, respectively, as required by the second parameter in the signal() function. The pointer values are 0 and 1, respectively. These two pointer values are logically the function address values that are not possible in the actual program. Therefore, in the signal() function, it is possible to judge whether to use the default signal processing handle or ignore the processing of the signal based on the two special pointer values. Of course, SIGKILL and SIGSTOP cannot be ignored. See the processing statements on lines 155-162 in the program listing below for related code.

When a program starts executing, the system sets its way to process all signals as SIG\_DFL or SIG\_IGN. In addition, when the program fork() a child process, the child process inherits the signal processing mode (signal mask) of the parent process. Therefore, the way the parent process sets and processes the signal is equally valid in the child process.

In order to continuously capture a specified signal, an example of the usual use of the signal() function in the user program is as follows.

---

```
void sig_handler(int signr)
{
    signal(SIGINT, sig_handler);    // re-install the handler for the next capture.
    ...                            // do something.
}

main ()
{
    signal(SIGINT, sig_handler);    // set signal handler in main.
    ...
}
```

---

The reason that the signal() function is unreliable is that when the signal has already occurred and enters its own set of signal processing functions, it is possible that another signal will occur during this time before re-setting its own handler. But at this point the system has set the handle to the default value. Therefore, it is possible to cause signal loss.

## 2. sigaction() function

The sigaction() function uses the sigaction data structure to hold the information of the specified signal. It is a reliable mechanism for the kernel to process signals. It allows us to easily view or modify the processing handle of a given signal. This function is a superset of the signal(). The declaration of this function in the include/signal.h header file (line 73) is:

---

```
int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
```

---

The parameter sig is the signal we need to view or modify the signal processing handler, and the last two parameters are pointers to the sigaction structure. When the parameter act pointer is not NULL, the behavior of

the specified signal can be modified according to the information in the act structure. When oldact is not empty, the kernel will return the original settings of the signal in the structure. The sigaction structure is as follows:

---

```

48 struct sigaction {
49     void (*sa_handler)(int);           // signal handler.
50     sigset_t sa_mask;                  // signal mask.
51     int sa_flags;
52     void (*sa_restorer)(void);         // internal restorer pointer.
53 };

```

---

When modifying a signal processing method, if the processing handler `sa_handler` is not the default `SIG_DFL` or the ignore handler `SIG_IGN`, then before the `sa_handler` can be called, the `sa_mask` field specifies one signal set that needs to be added to the process signal mask bitmap. If the signal handler returns, the system will restore the original signal mask bitmap of the process. This way we can block some of the specified signals when a signal handler is called. When the signal handler is called, the new signal mask bitmap automatically includes the currently transmitted signal, blocking the continued transmission of the signal. Thus, we can ensure that the same signal is blocked without being lost during the processing of a specified signal until the processing is completed. In addition, when a signal is blocked and occurs multiple times, usually only one sample is saved, that is, when the blocking is released, the signal processing handler is called again only once for the same multiple signals that are blocked. After we modify the processing handler of a signal, the handler is used until it is changed again. This is not the same as the traditional `signal()` function. The `signal()` function will restore it to the default handler of the signal after the handler processing has finished.

The `sa_flags` in the `sigaction` structure are used to specify other options for processing signals. These options are used to change some of the default processes in the signal handling. For their definitions, see the description in the `include/signal.h` file (lines 37-40).

---

```

// The symbol constant value that the sigaction structure sa_flags field can take.
37 #define SA_NOCLDSTOP    1           // ignore SIGCHLD if child stopped.
38 #define SA_INTERRUPT    0x20000000 // not restart syscall after interrupt by sig.
39 #define SA_NOMASK        0x40000000 // do not mask the current signal.
40 #define SA_ONESHOT       0x80000000 // restore to default handler after finished.

```

---

The last field in the `sigaction` structure and the parameter `restorer` of the `sys_signal()` function are both function pointers. It is provided by the `libc` library when compiling and linking programs, to clean up the user-mode stack after the signal handler ends, and to restore the return value of the system-call stored in `eax`, as detailed below.

### 3. `do_signal()` function

The `do_signal()` function is a preprocessor for the signal in the kernel system-call (`int 0x80`) interrupt handler. Each time a process calls a system-call or a timer interrupt occurs, if the process has received a signal, the function inserts the signal's processing handle (ie, the signal handler) into the user program stack. In this way, the signal handler is executed immediately after the current system-call returns, and then the user's program is executed, as shown in Figure 8-10.

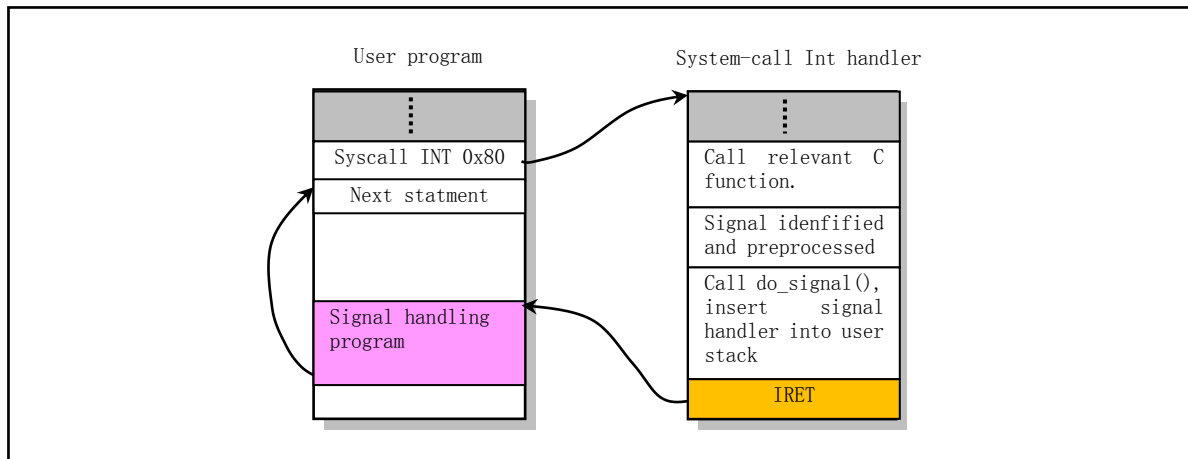
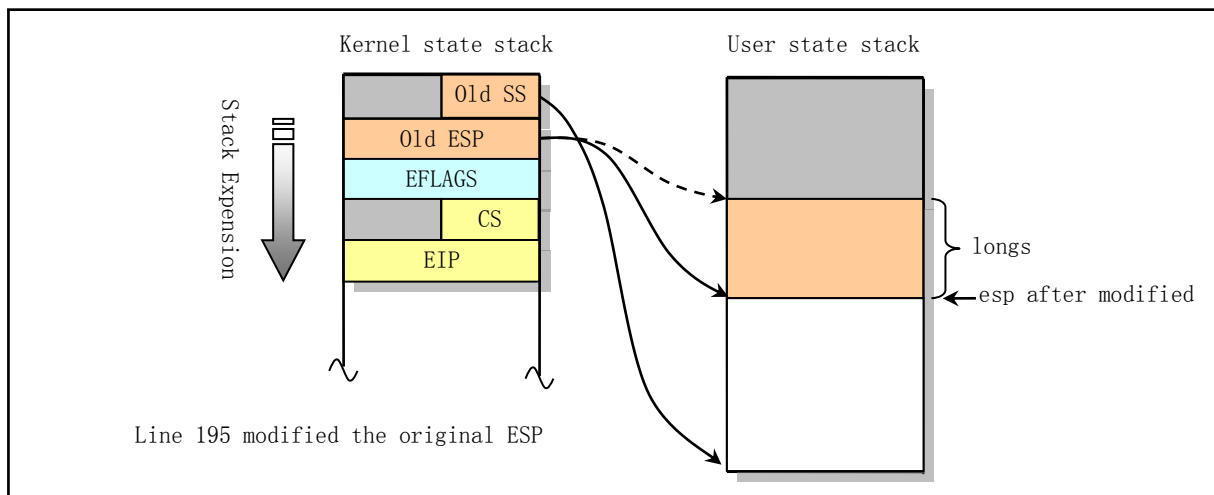


Figure 8-10 How the signal handler is called.

Before inserting the parameters of the signal handler into the user stack, the `do_signal()` function first expands the user program stack pointer down longs (see line 195 in the program below) and then adds the relevant parameters to it. See Figure 8-11. Since the code starting with line 193 of the `do_signal()` function is hard to understand, we will describe it in detail below.

When the user program calls a system-call just entering the kernel, the kernel state stack of the process is automatically pushed into the content by the CPU, as shown in Figure 8-11. That is: the SS, ESP and the CS and EIP of the next instruction in the user program. After processing the specified system-call and preparing to call `do_signal()` (that is, after the 124 lines in file `sys_call.s`), the contents of the kernel state stack are shown in the left side of Figure 8-12. So the arguments to `do_signal()` is what are on the kernel-state stack.

Figure 8-11 Modification of the user stack by the `do_signal()`

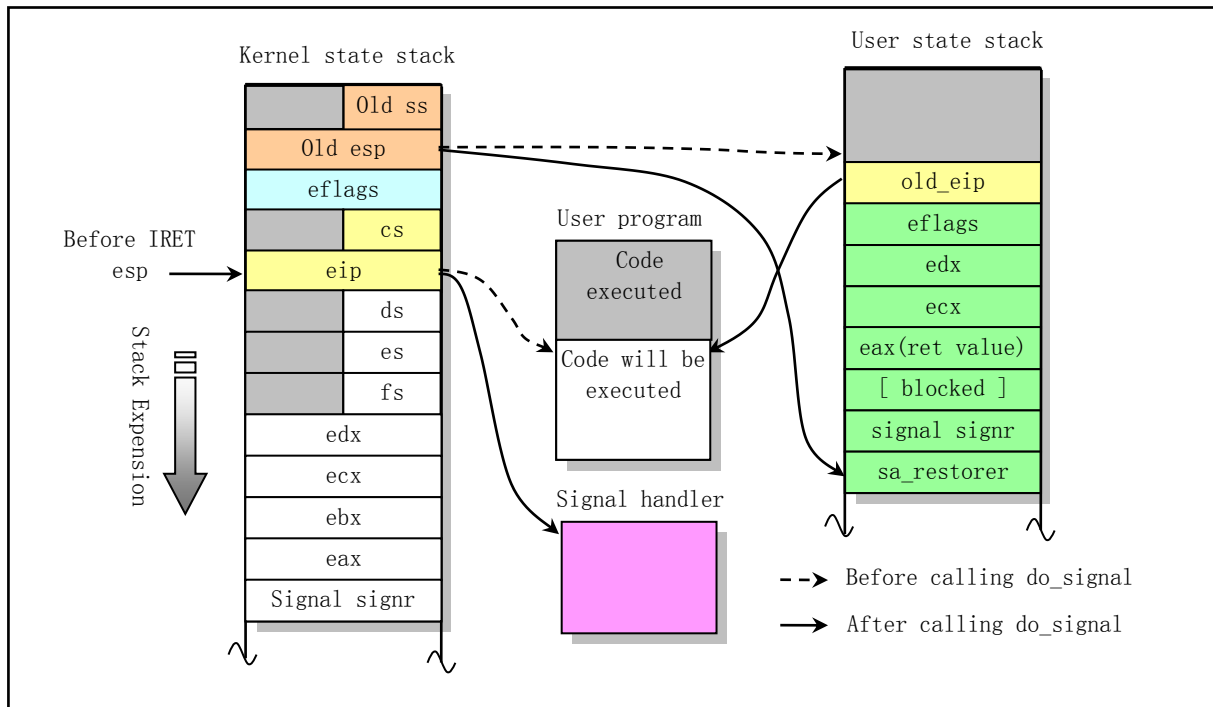


Figure 8-12 The specific process of modifying the user state stack

After `do_signal()` determines and processes the two default signal handlers (`SIG_IGN` and `SIG_DFL`), if the user customizes the signal handler, then from the 193th line, `do_signal()` starts to prepare inserting the user-defined handler into the user state stack. It first saves the return execution point pointer `eip` of the original user program in the kernel state stack as `old_eip` in user stack, and then replaces the `eip` with the custom handler `sa_handler`, that is, the `eip` in the kernel state stack in the figure points to `sa_handler`. Next, the user state stack is extended downward by 7 or 8 long words by subtracting the `longs` value from the "original `esp`" saved in the kernel state. Finally, some of the register contents on the kernel stack are copied into this space, as shown in the right side of Figure 8-12.

The kernel code puts a total of 7 to 8 values onto the user state stack. Let us now explain what these values mean and why they are placed. `old_eip` is the return address of the original user program, which is reserved before the `eip` is replaced with the signal handler address on the kernel stack. `eflags`, `edx`, and `ecx` are the values of the original user program before the system-call is invoked. They are basically the parameters used by the system-call. After the system-call returns, these register values of the user programs still need to be restored. The return value of the system-call is stored in `eax`. If the processed signal also allows itself to be received, the blocked code blocked for that process is also stored on the stack. The next one is the signal `signr`.

The last one is the pointer `sa_restorer` of the signal activity recovery function. This recovery function is not set by the user because only one signal value `signr` and one signal handler are provided when the user defines the `signal()` function.

The following is a simple example of setting a custom signal processing handler for the `SIGINT`. By default, pressing the Ctrl-C key combination will generate a `SIGINT` signal.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
```

```
void handler(int sig)
```

```
// user defined signal handler.
```

```
{
    printf("The signal is %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);           // restore default handler of SIGINT
}                                              // some system will restored automatically

int main()
{
    (void) signal(SIGINT, handler);           // set user defined handler for SIGINT.
    while (1) {
        printf("Signal test.\n");
        sleep(1);                             // wait a second.
    }
}
```

---

Among them, the signal handler function () will be called when the signal SIGINT appears, and then return to the main program to continue execution. The function first outputs a message and then sets the processing of the SIGINT signal to the default signal handler. So when you press the Ctrl-C key combination a second time, SIG\_DFL will let the program finish running.

#### 4. sa\_restorer function

So, where does the sa\_restorer function come from? In fact, it is provided by the function library. This function is available in the Linux Libc-2.2.2 library file (misc/subdir) and is defined as follows:

---

```
.globl __sig_restore
.globl __mask_sig_restore
# Use this restorer function if there is no blocked
__sig_restore:
    addl $4,%esp          # discard the signal signr
    popl %eax             # restore system-call ret value in eax
    popl %ecx             # restore user original registers
    popl %edx
    popfl                 # restore user eflags.
    ret
# If there is blocked, use the following restorer function. Blocked for use by ssetmask.
__mask_sig_restore:
    addl $4,%esp          # discard signal signr
    call __ssetmask       # set signal mask old blocking
    addl $4,%esp          # discard blocked.
    popl %eax
    popl %ecx
    popl %edx
    popfl
    ret
```

---

The main purpose of this function is to restore the return value and some register contents after the user program executes the syscall after the signal handler ends, and clear the signal value signr. When compiling a user-defined signal handler, the compiler invokes the signal syscall in the libc library to insert the sa\_restorer() function into the user program. The function implementation of the signal syscall in the library file is shown below.

---

```
01 #define __LIBRARY__
02 #include <unistd.h>
03
04 extern void __sig_restore();
05 extern void __mask_sig_restore();
06
07 // The signal() wrapper function called by the user in the library.
08 void (*signal(int sig, __sig_handler_t func))(int)
09 {
10     void (*res)();
11     register int __foebx __asm__ ("bx") = sig;
12     __asm__ ("int $0x80":"=a" (res):
13             "0" (__NR_signal), "r" (__foebx), "c" (func), "d" ((long) __sig_restore));
14     return res;
15 }
16
17 // The sigaction() function called by the user.
18 int sigaction(int sig, struct sigaction * sa, struct sigaction * old)
19 {
20     register int __foebx __asm__ ("bx") = sig;
21     if (sa->sa_flags & SA_NOMASK)
22         sa->sa_restorer=__sig_restore;
23     else
24         sa->sa_restorer=__mask_sig_restore;
25     __asm__ ("int $0x80":"=a" (sig)
26             : "0" (__NR_sigaction), "r" (__foebx), "c" (sa), "d" (old));
27     if (sig>=0)
28         return 0;
29     errno = -sig;
30     return -1;
31 }
```

---

The `sa_restorer()` is responsible for cleaning up the register value of the user program and the return value of the system-call after the signal handler is executed, as if it had not run the signal handler, but returned directly from the system-call.

Finally, let us explain the flow of the signal being processed. After `do_signal()` is executed, `sys_call.s` will pop the stack all values below the `eip` on the process kernel state stack. After executing the `IRET` instruction, the CPU will pop the `cs:eip`, `eflags`, and `ss:esp` on the kernel state stack, and return to the user state to execute the program. But since the `eip` has now been replaced with a pointer to the signal handler, the user-defined signal handler is executed immediately. After the signal handler is executed, the CPU will transfer control to the recovery program pointed to by `sa_restorer` through the `RET` instruction. The `sa_restorer` program will do some user-level stack cleanup, which will skip the signal value `signr` on the stack and pop the return value `eax` and the registers `ecx`, `edx` and the flag register `eflags` after the system-call. The state of each register and CPU after the system call is completely restored. Finally, the `eip` of the original user program (that is, `old_eip` on the stack) is popped up by the `RET` instruction of `sa_restorer`, and the user program is returned to execute.

### 8.7.1.3 Process suspension

The `signal.c` program also includes a `sys_sigsuspend()` syscall implementation that temporarily replaces the process signal mask with the given set in the argument and then suspends the process until a signal is received. The syscall is declared as a signature with three parameters:



---

```
int sys_sigsuspend(int restart, unsigned long old_mask, unsigned long set)
```

---

Where restart is a restart indicator. If it's 0, then we save the current mask in the oldmask, and then block the process until we receive a signal; if it's non-zero, then we restore the original mask from the saved oldmask, and return normally.

Although the syscall has three parameters, the general user program will call through the library when using it. This function in the library only uses a form with a set parameter:

---

```
int sigsuspend(unsigned long set)
```

---

This is the form in which the syscall is used in the C library. The first two parameters will be processed by the sigsuspend() library function. The general implementation of this library function is similar to the following code:

---

```
#define __LIBRARY__
#include <unistd.h>

int sigsuspend(sigset_t *sigmask)
{
    int res;

    register int __foebx __asm__ ("bx") = 0;
    __asm__ ("int $0x80"
            : "=a" (res)
            : "0" (__NR_sigsuspend), "r" (__foebx), "c" (0), "d" (*sigmask)
            : "bx", "cx");
    if (res >= 0)
        return res;
    errno = -res;
    return -1;
}
```

---

Here, the register variable \_\_foebx is the above 'restart'. When the syscall is called for the first time, it is 0, and the original blocking code is saved (old\_mask), and 'restart' is set to a non-zero value. So when the process calls the syscall for the second time, it will restore the blocking code that the process originally saved in old\_mask.

#### 8.7.1.4 Restart of a system-call interrupted by signal

If a process receives a signal while it is blocked while executing a slow system-call, the system-call is interrupted by the signal and no longer continues. At this point, the system-call will return an error message, and the corresponding global error code variable errno is set to EINTR, indicating that the system call is interrupted by a signal. For example, when reading or writing pipes, terminal devices or network devices, if the read data does not exist or the device cannot accept the data immediately, the calling program of the system-call will be blocked all the time. Therefore, for some slow system-calls, signals can be used to interrupt them and return to the user program when necessary. This also includes system-calls such as pause() and wait().

However, in some cases it is not necessary for the user program to personally handle the system-call that was interrupted by the signal. Because sometimes the user does not know if the device is a low speed device. If the program can run interactively, it may read and write low-speed devices. If the signal is captured in such a program and the system does not provide an automatic restart function for the system-call, then the program needs to detect the error return code each time the system-call is read or written. If it is interrupted by a signal, it needs to be read and written again. For example, when performing a read operation, if it is interrupted by a signal, then in order for it to continue the read operation, the user will be asked to write the following code:

---

```
again:
    if (( n = read(fd, buff, BUFFSIZE)) < 0 ) {
        if (errno == EINTR)
            goto again;          /* an interrupted syscall */
    }
```

---

In order to prevent the user program from having to deal with certain interrupted system-call situations, a restart (re-execute) function for some interrupted system-calls is introduced while processing the signal. System-calls that are automatically restarted include: `ioctl`, `read`, `write`, `wait`, and `waitpid`. The first three system-calls are interrupted by the signal only when operating on low-speed devices, while `wait` and `waitpid` are always interrupted when the signal is captured.

When handling a signal, depending on the flag set in the `sigaction` structure, it is possible to select whether to restart the interrupted system-call. In the Linux 0.12 kernel, if the `SA_INTERRUPT` flag is set in the `sigaction` structure (the system-call can be interrupted) and the relevant signals are not `SIGCONT`, `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, and `SIGTTOU`, the system-call will be interrupted when a signal is received. Otherwise the kernel will automatically re-execute the interrupted system-call. The method of execution is to first restore the value of the original register `eax` when the system-call is invoked, and then subtract two bytes from the `eip` of user program, that is, let `eip` redirect to the system-call `int 0x80` instruction.

For the current Linux system, the flag `SA_INTERRUPT` has been discarded. Instead, the flag `SA_RESTART`, which has the opposite meaning, is required to restart the interrupted system-call after the signal handler is executed.

## 8.7.2 Code Comments

Program 8-6 linux/kernel/signal.c

---

```
1 /*
2  * linux/kernel/signal.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
8 //     data of the initial task 0, and some embedded assembly function macro statements
9 //     about the descriptor parameter settings and acquisition.
10 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
11 //     commonly used functions of the kernel.
12 // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
13 //     descriptors/interrupt gates, etc. is defined.
14 // <signal.h> Signal header file. Define signal symbol constants, signal structures, and
```

```

//      signal manipulation function prototypes.
// <errno.h> Error number header file. Contains various error numbers in the system.
//      (Linus was introduced from minix).
7 #include <linux/sched.h>
8 #include <linux/kernel.h>
9 #include <asm/segment.h>
10
11 #include <signal.h>
12 #include <errno.h>
13
// Get the current task signal mask bitmap (mask, block code). The abbreviation 'sgetmask'
// can be broken down into 'signal-get-mask'.
14 int sys_sgetmask()
15 {
16     return current->blocked;
17 }
18
// Set a new signal mask bitmap. The signals SIGKILL and SIGSTOP cannot be masked.
// The return value is the original signal mask bitmap.
19 int sys_ssetmask(int newmask)
20 {
21     int old=current->blocked;
22
23     current->blocked = newmask & ~(1<<(SIGKILL-1)) & ~(1<<(SIGSTOP-1));
24     return old;
25 }
26
// Detects and acquires signals received but blocked. Bitmaps that have not yet processed
// the signal will be placed in the set.
27 int sys_sigpending(sigset_t *set)
28 {
29     /* fill in "set" with signals pending but blocked. */
// First verify that the user storage space provided should have 4 bytes. The bitmap of
// the signal that has not been handled and blocked is then filled in at the position
// indicated by the set pointer.
30     verify_area(set,4);
31     put_fs_long(current->blocked & current->signal, (unsigned long *)set);
32     return 0;
33 }
34
35 /* atomically swap in the new signal mask, and wait for a signal.
36 *
37 * we need to play some games with syscall restarting. We get help
38 * from the syscall library interface. Note that we need to coordinate
39 * the calling convention with the libc routine.
40 *
41 * "set" is just the sigmask as described in 1003.1-1988, 3.3.7.
42 * It is assumed that sigset_t can be passed as a 32 bit quantity.
43 *
44 * "restart" holds a restart indication. If it's non-zero, then we
45 * install the old mask, and return normally. If it's zero, we store
46 * the current mask in old_mask and block until a signal comes in.
47 */

```

```

// The syscall temporarily replaces the signal mask with the given set in the parameter
// and then suspends the process until a signal is received.
// restart is an interrupted system-call restart indicator. When the syscall is invoked
// for the first time, it is 0, and the original blocking code is saved (old_mask), and
// restart is set to a non-zero value. Therefore, when the process invokes the syscall
// for the second time, it will restore the blocking code that the process originally
// saved in old_mask.
// The pause() syscall will cause the process that called it to go to sleep until it
// receives a signal. This signal may either terminate the execution of the process or
// cause the process to execute the corresponding signal capture function.
48 int sys\_sigsuspend(int restart, unsigned long old_mask, unsigned long set)
49 {
50     extern int sys\_pause(void);
51
// If the restart flag is not 0, it means that the program is to be re-run. The original
// process blocking code previously saved in the old_mask is restored, and the code -EINTR
// is returned (the system call is interrupted by the signal).
52     if (restart) {
53         /* we're restarting */
54         current->blocked = old_mask;
55         return -EINTR;
56     }
// Otherwise, the restart flag is 0. Indicates that it is the first call. So first set the
// restart flag (set to 1), save the current blocking code to old_mask, and replace the
// process's blocking code with set. Then call sys_pause() to let the process sleep and
// wait for the signal to arrive. When the process receives a signal, pause() will return,
// and the process will execute the signal handler, then the call returns the -ERESTARTNOINTR
// code and exits. This return code indicates that after the signal is handled, it is
// required to return to the system-call to continue running.
57     /* we're not restarting. do the work */
58     *(&restart) = 1;
59     *(&old_mask) = current->blocked;
60     current->blocked = set;
61     (void) sys\_pause(); /* return after a signal arrives */
62     return -ERESTARTNOINTR; /* handle the signal, and come back */
63 }
64
// Copy the sigaction data to the fs data segment to. That is, copy from the kernel space
// to the user (task) data segment.
// First verify that the mem space at variable to is large enough. Then copy the sigaction
// structure data into the fs segment (user) space. The macro put_fs_byte() is implemented
// in include/asm/segment.h.
65 static inline void save\_old(char * from, char * to)
66 {
67     int i;
68
69     verify\_area(to, sizeof(struct sigaction));
70     for (i=0 ; i< sizeof(struct sigaction) ; i++) {
71         put\_fs\_byte(*from, to);
72         from++;
73         to++;
74     }
75 }

```

```

76 // Copy the sigaction data from the fs data segment 'from' to 'to'. That is, copy from the
77 // user data space to the kernel data segment.
78 static inline void get\_new(char * from, char * to)
79 {
80     int i;
81     for (i=0 ; i< sizeof(struct sigaction) ; i++)
82         *(to++) = get fs byte(from++);
83 }
84
85 // The signal() syscall, similar to sigaction(). Install a new signal handler for the
86 // specified signal. The signal handler can be a user-specified function, or it can be
87 // SIG_DFL (the default handler) or SIG_IGN (ignored).
88 // Parameters: signum - the specified signal; handler - the specified signal handler;
89 // restorer - the recovery function pointer. This function is provided by libc library.
90 // It is used to restore the original value of several registers and the return value of
91 // the syscall when the syscall returns after the end of signal handler, just as if the
92 // syscall does not execute the signal handler and returns directly to the user program.
93 // The function returns the original signal handler.
94 int sys\_signal(int signum, long handler, long restorer)
95 {
96     struct sigaction tmp;
97
98     // First verify that the signal is within the valid range (1--32) and must not be signal
99     // SIGKILL (and SIGSTOP). Because these two signals cannot be captured by the process.
100    // Then build the sigaction structure content according to the provided parameters.
101    // sa_handler is the specified signal handler. sa_mask is the signal mask. sa_flags is
102    // some combination of flags when executed. Here, the signal handler is set to the default
103    // after only one use, and the signal is allowed to be received in its own handler.
104    if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
105        return -EINVAL;
106    tmp.sa_handler = (void (*)(int)) handler;
107    tmp.sa_mask = 0;
108    tmp.sa_flags = SA\_ONESHOT | SA\_NOMASK;
109    tmp.sa_restorer = (void (*)(void)) restorer;
110    // Then take the original signal handler and set the sigaction structure. Finally return
111    // the original signal handler.
112    handler = (long) current->sigaction[signum-1].sa_handler;
113    current->sigaction[signum-1] = tmp;
114    return handler;
115 }
116
117 // Sigaction() system-call. Change the operation of the process when it receives a signal.
118 // Signum is any signal other than SIGKILL. [If the new action is not empty] the new
119 // operation is installed. If the oldaction pointer is not empty, the original operation
120 // is retained to the oldaction. Returns 0 if successful, otherwise -EINVAL.
121 int sys\_sigaction(int signum, const struct sigaction * action,
122 struct sigaction * oldaction)
123 {
124     struct sigaction tmp;
125
126     // First verify that the signal is within the valid range (1--32) and must not be signal

```

```

// SIGKILL (and SIGSTOP). Because these two signals cannot be captured by the process.
// Then set a new action in the sigaction structure of the signal. If the oldaction
// pointer is not empty, the original operation pointer is saved to the location pointed
// to by the oldaction.
105     if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
106         return -EINVAL;
107     tmp = current->sigaction[signum-1];
108     get_new((char *) action,
109            (char *) (signum-1+current->sigaction));
110     if (oldaction)
111         save_old((char *) &tmp, (char *) oldaction);
// If we allow the signal to be received in its own signal handler, then the mask is 0,
// otherwise the mask is set to mask this signal.
112     if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
113         current->sigaction[signum-1].sa_mask = 0;
114     else
115         current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
116     return 0;
117 }
118
119 /*
120  * Routine writes a core dump image in the current directory.
121  * Currently not implemented.
122  */
123 int core_dump(long signr)
124 {
125     return(0);    /* We didn't do a dump */
126 }
127
// The real signal pre-processing code in system-call interrupt handler.
// The main purpose of this code is to insert the signal processing handler into the user
// program stack, and immediately execute the signal handler after system-call returns,
// and then continue to execute the user program. The parameters of this function include
// all values pushed step by step onto the stack from entering the system-call handler to
// the location before calling this function (sys_call.s, line 125). These values include
// (the line number in sys_call.s):
// (1)The user stack address ss and esp, the eflags, and the return addresses cs and eip
//     that are pushed onto kernel-state stack by the interrupt instruction;
// (2)The values of the segment registers ds, es, fs and registers eax (orig_eax), edx,
//     ecx, and ebx pushed onto the stack just after entering system_call on lines 85-91;
// (3)After the sys_call_table is called on line 100, the return value (eax) of the
//     system-call is pushed.
// (4)The currently processed signal (signer) is pushed onto the stack on line 124.
128 int do_signal(long signr, long eax, long ebx, long ecx, long edx, long orig_eax,
129             long fs, long es, long ds,
130             long eip, long cs, long eflags,
131             unsigned long * esp, long ss)
132 {
133     unsigned long sa_handler;
134     long old_eip=eip;
135     struct sigaction * sa = current->sigaction + signr - 1;
136     int longs;    // current->sigaction[signr-1]
137

```

```

138     unsigned long * tmp_esp;
139
140     // The debug statement. The relevant information is printed when notdef is defined.
141 #ifdef notdef
142     printf("pid: %d, signr: %x, eax=%d, oeax = %d, int=%d\n",
143         current->pid, signr, eax, orig_eax,
144         sa->sa_flags & SA\_INTERRUPT);
145 #endif
146
147     // The orig_eax value will be -1 if it's not a system-call interrupt but is called during
148     // other interrupt service (see line 144 of sys_call.s). So when orig_eax is not equal to -1,
149     // it means that this function was called in the handling of a system-call. In the waitpid()
150     // function in kernel/exit.c, if SIGCHLD signal is received, or if reading data from a
151     // pipeline but no data is read, and if the process receives any non-blocking signal, it
152     // will be returned with a return value of -ERESTARTSYS. It indicates that the process can
153     // be interrupted, but the system-call is restarted after execution continues. Return code
154     // -ERESTARTNOINTR indicates that after the signal is handled, it is required to return to
155     // the system-call to continue running, that is, the system-call will not be interrupted.
156     // Therefore, the following statement shows that if this function is called in the system-call,
157     // and the return code eax of the system-call is equal to -ERESTARTSYS or -ERESTARTNOINTR,
158     // the following processing is performed (actually, it has not returned to user program).
159     if ((orig_eax != -1) &&
160         ((eax == -ERESTARTSYS) || (eax == -ERESTARTNOINTR))) {
161
162         // If the system-call return code is -ERESTARTSYS, and the sigaction contains flag
163         // SA_INTERRUPT or the signal is less than SIGCONT or greater than SIGTTOU (ie the signal
164         // is not SIGCONT, SIGSTOP, SIGSTP, SIGTTIN, or SIGTTOU), the modified return value of the
165         // system-call is eax = -EINTR, which is the system-call interrupted by the signal.
166         if ((eax == -ERESTARTSYS) && ((sa->sa_flags & SA\_INTERRUPT) ||
167             signr < SIGCONT || signr > SIGTTOU))
168             *(&eax) = -EINTR;
169         else {
170             // Otherwise, the eax is restored to the value before the system-call is called, and the
171             // original program instruction pointer is subtracted by 2 bytes. That is, when returning
172             // to the user program, let the program restart to execute the system-call that was
173             // interrupted by the signal.
174             *(&eax) = orig_eax;
175             *(&eip) = old_eip - 2;
176         }
177     }
178
179     // If the signal handle is SIG_IGN (1, the handle is ignored by default), the signal is
180     // not processed and returned directly.
181     sa_handler = (unsigned long) sa->sa_handler;
182     if (sa_handler==1)
183         return(1);    /* Ignore, see if there are more signals... */
184
185     // If the handle is SIG_DFL (0, default processing), the default method is used for
186     // processing according to the specific signal.
187     if (!sa_handler) {
188         switch (signr) {
189             // If the signal is the following two, it is also ignored and returned.
190             case SIGCONT:
191             case SIGCHLD:
192                 return(1);    /* Ignore, ... */

```

```

// If the signal is one of the following four signals, the current process state is set to
// the stop state TASK_STOPPED. If the parent process of the current process does not set
// the sigaction flag SA_NOCLDSTOP of the SIGCHLD signal, then the SIGCHLD signal is sent
// to the parent process. The SA_NOCLDSTOP flag indicates that the SIGCHLD signal is not
// generated when the child process stops executing or continues execution.
164         case SIGSTOP:
165         case SIGTSTP:
166         case SIGTTIN:
167         case SIGTTOU:
168             current->state = TASK_STOPPED;
169             current->exit_code = signr;
170             if (!(current->p_pptr->sigaction[SIGCHLD-1].sa_flags &
171                 SA_NOCLDSTOP))
172                 current->p_pptr->signal |= (1<<(SIGCHLD-1));
173             return(1); /* Reschedule another event */
174
// If the signal is one of the following 6 signals, if the signal generates a core dump,
// call do_exit() to exit with the exit code as signr|0x80. Otherwise the exit code is the
// signal value. The parameters of do_exit() are the return code and the exit status
// information provided by the program. It can be used as status information for the
// wait() or waitpid() functions. See lines 13-19 of the sys/wait.h file. wait() or
// waitpid() can use these macros to get the exit status code of the child process or the
// reason (signal) of the child process termination.
175         case SIGQUIT:
176         case SIGILL:
177         case SIGTRAP:
178         case SIGIOT:
179         case SIGFPE:
180         case SIGSEGV:
181             if (core_dump(signr))
182                 do_exit(signr|0x80);
183             /* fall through */
184         default:
185             do_exit(signr);
186     }
187 }
188 /*
189  * OK, we're invoking a handler
190  */
// If the signal handler only needs to be called once, the handler is left blank. Note
// that the signal handler has been previously saved in the sa_handler pointer.
// When the system-call enters the kernel code, the user program return address (eip, cs)
// is saved in the kernel state stack. The following code modifies the code pointer eip on
// the kernel state stack to point to the signal handler, and also pushed sa_restorer,
// signr, process mask (if SA_NOMASK is not set), eax, ecx, edx as parameters onto the
// user stack. The original program return pointer and eflag register are also pushed onto
// the user stack. Therefore, when the syscall returns to the user program, the user's
// signal handler is executed first, and then the user program is continued.
191     if (sa->sa_flags & SA_ONESHOT)
192         sa->sa_handler = NULL;

// Let the user's next code instruction pointer eip on the kernel state stack point to the
// signal handler. Since the C function is passed by value, you need to use the form

```



---

```

// "*(&eip)" when assigning values to eip.
// The sa_mask field of the sigaction structure gives the set of signals that should be
// masked during the execution of the current signal handler. At the same time, the current
// signal will also be blocked. However, if the SA_NOMASK flag is used in sa_flags, the
// current signal will not be masked. If the signal handler is allowed to receive its own
// signal, the signal blocking code of the process also needs to be pushed onto the stack.
193     *(&eip) = sa_handler;
194     longs = (sa->sa_flags & SA_NOMASK)?7:8;
// Extend the user stack pointer of the original user program by 7 (or 8) long words (used
// to store the parameters of the signal handler, etc.) and check the memory usage (if the
// memory is out of bounds, allocate a new page, etc.).
195     *(&esp) -= longs;
196     verify_area(esp, longs*4);
// Store sa_restorer, signal signer, mask code blocked (if SA_NOMASK is set), eax, ecx,
// edx, eflags, and user program code pointer from bottom to top in the user stack.
197     tmp_esp=esp;
198     put_fs_long((long) sa->sa_restorer, tmp_esp++);
199     put_fs_long(signr, tmp_esp++);
200     if (!(sa->sa_flags & SA_NOMASK))
201         put_fs_long(current->blocked, tmp_esp++);
202     put_fs_long(eax, tmp_esp++);
203     put_fs_long(ecx, tmp_esp++);
204     put_fs_long(edx, tmp_esp++);
205     put_fs_long(eflags, tmp_esp++);
206     put_fs_long(old_eip, tmp_esp++);
207     current->blocked |= sa->sa_mask; // Fill in with sa_mask bitmap
208     return(0); // Continue, execute handler */
209 }
210

```

---

## 8.7.3 Information

### 8.7.3.1 Signal Description

The signal in the process is a simple message used for communication between processes, usually a label value between 1 to 31, and does not carry any other information. The signals supported by the Linux 0.12 kernel are shown in Table 8-5.

Table 8-5 Process signals

Sig	Name	Description	Default operation
1	SIGHUP	(Hangup) The kernel generates this signal when you no longer have a control terminal, or when you turn off Xterm or disconnect the modem. Since the background programs do not have control terminals, they often use SIGHUP to signal that they need to re-read their configuration files.	(Abort) Hang up control terminal or process.
2	SIGINT	(Interrupt) An interrupt from the keyboard. Usually the terminal driver will bind it to ^C.	(Abort) Terminate program.
3	SIGQUIT	(Quit) The exit interrupt from keyboard. Usually the terminal driver will bind it to ^\.	(Dump) Terminated and a dump core file is generated.
4	SIGILL	(Illegal Instruction) The program has an error or an illegal operation	(Dump)

		command has been executed.	
5	SIGTRAP	(Breakpoint/Trace Trap) Used for debugging, tracking breakpoints.	
6	SIGABRT	(Abort) Abandon execution and end abnormally.	(Dump)
6	SIGIOT	(IO Trap) Same as SIGABRT	(Dump)
7	SIGUNUSED	(Unused) Not used.	
8	SIGFPE	(Floating Point Exception) Floating exception.	(Dump)
9	SIGKILL	(Kill) The program was terminated. This signal cannot be captured or ignored. If you want to terminate a process immediately, you will send a signal 9. Note that the program will have no chance to do the cleanup.	(Abort)
10	SIGUSR1	(User defined Signal 1) User defined signal 1.	(Abort)
11	SIGSEGV	(Segmentation Violation) This signal is generated when the program references invalid memory. For example: addressing unmapped memory; addressing unlicensed memory.	(Dump)
12	SIGUSR2	(User defined Signal 2) Reserved for user programs for IPC or other purposes.	(Abort)
13	SIGPIPE	(Pipe) This signal is generated when a program writes to a socket or pipe, and there is no reader.	(Abort)
14	SIGALRM	(Alarm) This signal is generated after the delay time set by the user using alarm syscall. This signal is often used to determine syscall timeouts.	(Abort)
15	SIGTERM	(Terminate) Used to kindly require a program to terminate. It is the default signal for kill. Unlike SIGKILL, this signal can be captured so that it can be cleaned up before exiting.	(Abort)
16	SIGSTKFLT	(Stack fault on coprocessor) Coprocessor stack error.	(Abort)
17	SIGCHLD	(Child) The child process issued. The child process has been stopped or terminated. You can change its meaning and use it for other usage.	(Ignore) The child process stops or ends.
18	SIGCONT	(Continue) This signal causes the process stopped by SIGSTOP to resume operation. Can be captured.	(Continue) Resume process running.
19	SIGSTOP	(Stop) Stop process running. This signal cannot be captured or ignored.	(Stop) Stop process running.
20	SIGTSTP	(Terminal Stop) Send a stop key sequence to the terminal. This signal can be captured or ignored.	(Stop)
21	SIGTTIN	(TTY Input on Background) The background process attempts to read data from a terminal that is no longer under control, at which point the process will be stopped until the SIGCONT signal is received. This signal can be captured or ignored.	(Stop)
22	SIGTTOU	(TTY Output on Background) The background process attempts to output data to a terminal that is no longer under control, at which point the process will be stopped until the SIGCONT signal is received. This signal can be captured or ignored.	(Stop)

## 8.8 exit.c

### 8.8.1 Function Description

The `exit.c` program mainly implements the processing related to the termination and exit of process. These include process release, session (process group) termination, and program exit handlers, as well as system-calls such as killing processes, terminating processes, and suspending processes. It also includes signal sending function `send_sig()`, and function `tell_father()` that notifies parent the termination of child process.

The release process function `release()` deletes the specified task pointer in the task array, releases the relevant memory page according to the specified task pointer, and immediately causes the kernel to reschedule the running task. The process group termination function `kill_session()` is used to send a signal to the process whose session number is the same as the current process ID. The system-call `sys_kill()` is used to send any specified signal to a process. Depending on the parameter `pid`, the system-call sends a signal to a different process or group of processes. The handling of the various situations is listed in the program comments.

The program exit handling function `do_exit()` is called in the interrupt handler of the exit system-call. It first releases the memory page occupied by the code and data segments of the current process. If the current process has child process, set the father field of the child process to 1, that is, change the parent of the child process to process 1 (init process). If the child process is already in a zombie state, the child termination signal `SIGCHLD` is sent to process 1. Then close all files opened by the current process, release the used terminal device, and coprocessor device. If the current process is the first process of the process group, then all related processes need to be terminated. The current process is then set to a zombie state, the exit code is set, and the child termination signal `SIGCHLD` is sent to its parent process. Finally let the kernel reschedule the running task.

The system-call `waitpid()` is used to suspend current process until the child process specified by `pid` exits (terminates) or receives a signal requesting termination of the process, or a signal handler needs to be called. If the child process pointed to by `pid` has already exited (which has become a so-called zombie process), this call will return immediately. All resources used by the child process will be released. The specific operation of this function is also handled differently according to its parameters. See the relevant comments in the code for details.

### 8.8.2 Code Comments

Program 8-7 linux/kernel/exit.c

---

```
1 /*  
2  * linux/kernel/exit.c  
3  *  
4  * (C) 1991 Linus Torvalds  
5  */  
6  
7 #define DEBUG\_PROC\_TREE  
8  
9 // <errno.h> Error number header file. Contains various error numbers in the system.  
10 // <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal  
11 // manipulation function prototypes.
```

```

// <sys/wait.h> Wait header file. Define the system call wait() core waitpid() and related
//      constant symbols.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//      used functions of the kernel.
// <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
//      communication.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//      for segment register operations.
9 #include <errno.h>
10 #include <signal.h>
11 #include <sys/wait.h>
12
13 #include <linux/sched.h>
14 #include <linux/kernel.h>
15 #include <linux/tty.h>
16 #include <asm/segment.h>
17
18 int sys_pause(void);      // put in sleep until receive a signal (kernel/sched.c, 164).
19 int sys_close(int fd);    // close a file (fs/open.c, 219).
20
// Frees task slot occupied by the process and mem page occupied by its task structure. The
// parameter p is a pointer to the task data structure. This function is called in the sys_kill()
// and sys_waitpid() that follow.
// The program scans the task pointer array task[] to find the specified task. If found, the
// task slot is first emptied and then the memory page occupied by the task data structure is
// released. Finally execute the scheduler and exit immediately upon return. If the item
// corresponding to the specified task is not found in the task array, the kernel panic.
21 void release(struct task_struct * p)
22 {
23     int i;
24
// Exit if the given task structure pointer is NULL. If the pointer points to the current process,
// a warning message is displayed to exit. Suicide is not allowed here even you are not a creature!
25     if (!p)
26         return;
27     if (p == current) {
28         printk("task releasing itself\n\r");
29         return;
30     }
// The following loop statement scans the array of task structure pointers to find the specified
// task p. If found, the corresponding item in the task pointer array is set to NULL, and the
// associated pointer between the task is updated, and the memory page occupied by the task
// p data structure is released. Finally, exit after scheduler returns. If task p is not found,
// the kernel code is wrong, and an error message is displayed and the kernel crashes. In addition,
// the code that updates the links removes the task p from the doubly linked list.
31     for (i=1 ; i<NR_TASKS ; i++)
32         if (task[i]==p) {
33             task[i]=NULL;
34             /* Update links */
// The following code operates on the linked list. If p is not the last (oldest) child process,

```

```

// let the old sibling (neighbor) points to the young sibling. If p is not the latest child
// process, let the newer sibling points to the older sibling. If task p is the latest child
// process, you need to update its parent's latest child pointer cptr to point to its old sibling.
// Refer to figure 5-20.
// osptr (old sibling pointer) points to the sibling process created earlier than p.
// ysptr (younger sibling pointer) points to the sibling process created after p.
// pptr (parent pointer) points to the parent process of p.
// cptr (child pointer) parent process points to the last created child.
35         if (p->p_osptr)
36             p->p_osptr->p_ysptr = p->p_ysptr;
37         if (p->p_ysptr)
38             p->p_ysptr->p_osptr = p->p_osptr;
39         else
40             p->p_pptr->p_cptr = p->p_osptr;
41         free\_page((long)p);
42         schedule();
43         return;
44     }
45     panic("trying to release non-existent task");
46 }
47
48 #ifdef DEBUG\_PROC\_TREE
// If symbol DEBUG_PROC_TREE is defined, the following code is included at compile time.
49 /*
50  * Check to see if a task_struct pointer is present in the task[] array
51  * Return 0 if found, and 1 if not found.
52  */
53 int bad\_task\_ptr(struct task\_struct *p)
54 {
55     int    i;
56
57     if (!p)
58         return 0;
59     for (i=0 ; i<NR\_TASKS ; i++)
60         if (task[i] == p)
61             return 0;
62     return 1;
63 }
64
65 /*
66  * This routine scans the pid tree and make sure the rep invariant still
67  * holds. Used for debugging only, since it's very slow....
68  *
69  * It looks a lot scarier than it really is.... we're doing nothing more
70  * than verifying the doubly-linked list found in p_ysptr and p_osptr,
71  * and checking it corresponds with the process tree defined by p_cptr and
72  * p_pptr;
73  */
74 void audit\_ptree()
75 {
76     int    i;
77
// The loop scans all tasks except task 0 in the system and checks the correctness of the

```

```

// four pointers (pptr, cptr, ysptr, and osptr). Skip if the task array slot is empty.
78     for (i=1 ; i<NR_TASKS ; i++) {
79         if (!task[i])
80             continue;
// If the task's parent pointer p_pptr does not point to any process (that is, it does not
// exist in the task array), a warning message is displayed. The following statements perform
// similar operations on cptr, ysptr, and osptr.
81         if (bad_task_ptr(task[i]->p_pptr))
82             printk("Warning, pid %d's parent link is bad\n",
83                 task[i]->pid);
84         if (bad_task_ptr(task[i]->p_cptr))
85             printk("Warning, pid %d's child link is bad\n",
86                 task[i]->pid);
87         if (bad_task_ptr(task[i]->p_ysptr))
88             printk("Warning, pid %d's ys link is bad\n",
89                 task[i]->pid);
90         if (bad_task_ptr(task[i]->p_osptr))
91             printk("Warning, pid %d's os link is bad\n",
92                 task[i]->pid);
// If the task's parent pointer p_pptr points to itself, a warning message is displayed.
// The following statements perform similar operations on cptr, ysptr, and osptr.
93         if (task[i]->p_pptr == task[i])
94             printk("Warning, pid %d parent link points to self\n");
95         if (task[i]->p_cptr == task[i])
96             printk("Warning, pid %d child link points to self\n");
97         if (task[i]->p_ysptr == task[i])
98             printk("Warning, pid %d ys link points to self\n");
99         if (task[i]->p_osptr == task[i])
100            printk("Warning, pid %d os link points to self\n");
// If the task has a old sibling process (that was created earlier than itself), then check
// if they have a common parent and check if the ysptr pointer of the buddy points to this process
// correctly, otherwise a warning message is displayed.
101        if (task[i]->p_osptr) {
102            if (task[i]->p_pptr != task[i]->p_osptr->p_pptr)
103                printk(
104                    "Warning, pid %d older sibling %d parent is %d\n",
105                    task[i]->pid, task[i]->p_osptr->pid,
106                    task[i]->p_osptr->p_pptr->pid);
107            if (task[i]->p_osptr->p_ysptr != task[i])
108                printk(
109                    "Warning, pid %d older sibling %d has mismatched ys link\n",
110                    task[i]->pid, task[i]->p_osptr->pid);
111        }
// If the task has a young sibling process (that was created later than itself), then check
// if they have a common parent and check if the osptr pointer of the younger correctly points
// to this process, otherwise a warning message is displayed.
112        if (task[i]->p_ysptr) {
113            if (task[i]->p_pptr != task[i]->p_ysptr->p_pptr)
114                printk(
115                    "Warning, pid %d younger sibling %d parent is %d\n",
116                    task[i]->pid, task[i]->p_osptr->pid,
117                    task[i]->p_osptr->p_pptr->pid);
118            if (task[i]->p_ysptr->p_osptr != task[i])

```

```

119             printk(
120                 "Warning, pid %d younger sibling %d has mismatched os link\n",
121                 task[i]->pid, task[i]->p_ysptr->pid);
122         }
// If the task's latest child pointer cptr is not empty, then check if the child's parent is
// the process, and check if the child's younger pointer ysptr is empty. If not, a warning
// message is displayed.
123         if (task[i]->p_cptr) {
124             if (task[i]->p_cptr->p_pptr != task[i])
125                 printk(
126                     "Warning, pid %d youngest child %d has mismatched parent link\n",
127                     task[i]->pid, task[i]->p_cptr->pid);
128             if (task[i]->p_cptr->p_ysptr)
129                 printk(
130                     "Warning, pid %d youngest child %d has non-NULL ys link\n",
131                     task[i]->pid, task[i]->p_cptr->pid);
132         }
133     }
134 }
135 #endif /* DEBUG_PROC_TREE */
136
// Send a signal sig to task p with privilege priv.
// sig - the signal; p - a pointer to the task; priv - a flag that forces the signal to be sent,
// ie the right to send signal without regard to process user attribute or level.
// The function first checks the correctness of the parameters and then determines if the
// condition is met. If it is satisfied, it sends a signal sig to the process and exits,
// otherwise it returns an unlicensed error code.
137 static inline int send\_sig(long sig, struct task\_struct * p, int priv)
138 {
// If there is no permission, and the effective user ID (euid) of the current process is
// different from the process p, and is not a superuser, then there is no right to send a
// signal to p. suser() is defined as (current->euid==0), used to check if it's a superuser.
139     if (!p)
140         return -EINVAL;
141     if (!priv && (current->euid!=p->euid) && !suser())
142         return -EPERM;
// If the signal to be send is SIGKILL or SIGCONT, then if the process p receiving signal
// is stopped at this time, it is set to the ready state (TASK_RUNNING). Then modify the
// signal bitmap process p to remove (reset) the signals SIGSTOP, SIGTSTP, SIGTTIN, and
// SIGTTOU that will cause the process to stop.
143     if ((sig == SIGKILL) || (sig == SIGCONT)) {
144         if (p->state == TASK\_STOPPED)
145             p->state = TASK\_RUNNING;
146         p->exit_code = 0;
147         p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1)) |
148                       (1<<(SIGTTIN-1)) | (1<<(SIGTTOU-1)) );
149     }
150     /* If the signal will be ignored, don't even post it */
151     if ((int) p->sigaction[sig-1].sa_handler == 1)
152         return 0;
153     /* Depends on order SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU */
// If the signal is one of SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU, then it is necessary to
// stop process p from running. Therefore (if SIGCONT is set in the signal bitmap of p), it

```

```

// is necessary to reset the SIGCONT bit in the bitmap.
154     if ((sig >= SIGSTOP) && (sig <= SIGTTOU))
155         p->signal &= ~(1<<(SIGCONT-1));
156     /* Actually deliver the signal */
157     p->signal |= (1<<(sig-1));
158     return 0;
159 }
160
// Get the session id based on the process group id pgrp.
// The code scans task array, looks for process with the group id pgrp, and returns its session
// id. If no process is found for the specified group pgrp, -1 is returned.
161 int session_of_pgrp(int pgrp)
162 {
163     struct task_struct **p;
164
165     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
166         if ((*p)->pgrp == pgrp)
167             return((*p)->session);
168     return -1;
169 }
170
// Kill the process group (send a signal to the group).
// Parameters: grp - process group id; sig - signal; priv - privilege.
// That is, the signal sig is sent to each process in the process group pgrp. As long as it
// is successfully sent to a process, it will return 0. Otherwise, if no process is found for
// the group pgrp, the error code -ESRCH is returned. If the process whose group is pgrp is
// found, but the signal transmission fails, the error code is returned.
171 int kill_pg(int pgrp, int sig, int priv)
172 {
173     struct task_struct **p;
174     int err, retval = -ESRCH;           // ESRCH - error search.
175     int found = 0;
176
177     // First check if the given signal and process group are valid, then scan all tasks in the
178     // system. If the process with the group id pgrp is scanned, the signal sig is sent to it.
179     // As long as the signal is sent successfully, the function will return 0 at the end.
180     if (sig<1 || sig>32 || pgrp<=0)
181         return -EINVAL;
182     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
183         if ((*p)->pgrp == pgrp) {
184             if (sig && (err = send_sig(sig, *p, priv)))
185                 retval = err;
186             else
187                 found++;
188         }
189     return(found ? 0 : retval);
190 }
191
// Kill the process (send a signal to the process).
// Parameters: pid - the process id; sig - the signal; priv - the privilege.
// That is, the signal sig is sent to the process with the pid. If the process of the pid is
// found, then if the signal is sent successfully, it returns 0, otherwise return error code.
// If the process with the pid is not found, the error code -ESRCH is returned

```



```

189 int kill_proc(int pid, int sig, int priv)
190 {
191     struct task_struct **p;
192
193     if (sig<1 || sig>32)
194         return -EINVAL;
195     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
196         if ((*p)->pid == pid)
197             return(sig ? send_sig(sig,*p,priv) : 0);
198     return(-ESRCH);
199 }
200
201 /*
202  * POSIX specifies that kill(-1,sig) is unspecified, but what we have
203  * is probably wrong. Should make it like BSD or SYSV.
204  */
205 // The system-call kill() can be used to send any signal to process or process group.
206 // Parameters: pid - the process id; sig - the signal that needs to be sent.
207 // If pid > 0, the signal is sent to the process whose process id is pid.
208 // If pid = 0, the signal is sent to all processes in the group of the current process.
209 // If pid = -1, the signal is sent to all processes except the first (initial) process.
210 // If pid < -1, the signal will be sent to all processes in the group -pid.
211 // If sig is 0, no signal is sent, but error check is still performed. Ret 0 if succeed.
212 //
213 // This function scans the task array and sends the signal sig to the process that satisfies
214 // the condition according to pid. If pid is equal to 0, it indicates that the current process
215 // is the group leader, so it is necessary to force the signal sig to be sent to all processes
216 // in the group.
217 int sys_kill(int pid,int sig)
218 {
219     struct task_struct **p = NR_TASKS + task;    // points to the last item.
220     int err, retval = 0;
221
222     if (!pid)
223         return(kill_pg(current->pid,sig,0));
224     if (pid == -1) {
225         while (--p > &FIRST_TASK)
226             if (err = send_sig(sig,*p,0))
227                 retval = err;
228         return(retval);
229     }
230     if (pid < 0)
231         return(kill_pg(-pid,sig,0));
232     /* Normal kill */
233     return(kill_proc(pid,sig,0));
234 }
235
236 /*
237  * Determine if a process group is "orphaned", according to the POSIX
238  * definition in 2.2.2.52. Orphaned process groups are not to be affected
239  * by terminal-generated stop signals. Newly orphaned process groups are
240  * to receive a SIGHUP and a SIGCONT.
241  */

```

```

230 * "I ask you, have you ever known what it is to be an orphan?"
231 */
// The POSIX P1003.1 section 2.2.2.52 mentioned above is a description of the orphan process
// group. In both cases, when a process terminates, it may cause the process group to become
// "orphan." The connection between a process group and a parent outside its group depends
// on both the parent and its child processes. Therefore, if the last process outside the group
// that is connected to the parent process, or the immediate descendant of the last parent
// process, terminates, then the process group becomes an orphan process group. In either case,
// if the termination of the process causes the process group to become an orphaned group, all
// processes in the group are disconnected from their job control shell.
// The job control shell will no longer have any information about the existence of this process
// group. The process in the group that is in a stopped state will disappear forever. To solve
// this problem, a newly generated orphan process group containing a stop state process needs
// to receive a SIGHUP signal and a SIGCONT signal to indicate that they have disconnected from
// their session.
// The SIGHUP signal will cause members of the process group to be terminated unless they capture
// or ignore the SIGHUP signal. The SIGCONT signal will continue to run those processes that
// are not terminated by the SIGHUP signal. However, in most cases, if one of the processes
// in the group is in a stopped state, all processes in the group may be in a stopped state.
//
// Check if a process group is an orphan. Returns 0 if not; returns 1 if it is. The code loop
// scans the task array. If the task item is empty, or the process group id is different from
// the specified one, or the process is already in a zombie state, or the process's parent is
// an init process, then the scanned process is not a member of the group, or the request is
// not met, so skip over. Otherwise, the process is a member of the group and its parent is
// not the init process. At this time, if the group id of the parent is not equal to the group
// id pgrp, but the session of the parent is equal to the session of the process, it means that
// they belong to the same session. Therefore the specified pgrp group is certainly not an orphan
// group, otherwise...sighing.
232 int is_orphaned_pgrp(int pgrp)
233 {
234     struct task_struct **p;
235
236     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
237         if (!(*p) ||
238             ((*p)->pgrp != pgrp) ||
239             ((*p)->state == TASK_ZOMBIE) ||
240             ((*p)->p_pptr->pid == 1))
241             continue;
242         if (((*p)->p_pptr->pgrp != pgrp) &&
243             ((*p)->p_pptr->session == (*p)->session))
244             return 0;
245     }
246     return(1);    /* (sighing) "Often!" */
247 }
248
// Check if the process group contains a job (process group) that is in a stopped state.
// Returns 1 if there is none; returns 0 if none. The search method is to scan the entire task
// array and check if any processes belonging to the group pgrp are in a stopped state.
249 static int has_stopped_jobs(int pgrp)
250 {
251     struct task_struct ** p;
252

```

```

253     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
254         if ((*p)->pgrp != pgrp)
255             continue;
256         if ((*p)->state == TASK_STOPPED)
257             return(1);
258     }
259     return(0);
260 }
261
262 // Program exit handling function. Invoked by the syscall sys_exit() at line 365 below.
263 // The function will process it according to the characteristics of the current process itself,
264 // and set the current process state to TASK_ZOMBIE, and finally call the schedule() function
265 // to execute other processes, and will not return.
266 volatile void do_exit(long code)
267 {
268     struct task_struct *p;
269     int i;
270
271     // First free the memory page occupied by the current process code and data segments.
272     // The first argument of the function free_page_tables() (the get_base() return value) indicates
273     // the starting base address in the CPU linear address space, and the second (get_limit() return
274     // value) indicates the byte length to be released. The current->ldt[1] in the get_base() macro
275     // gives the location of the process code segment descriptor, and current->ldt[2] gives the
276     // location of the data segment descriptor. 0x0f in get_limit() is the selector of the code
277     // segment, and 0x17 is the selector of the data segment. That is, when the segment base address
278     // is taken, the address of the segment descriptor is used as a parameter, and when the segment
279     // limit (length) is taken, the segment selector is used as a parameter (since there is a dedecated
280     // instruction LSL which can be used to get segment limit through a selector).
281     // The free_page_tables() is located at the beginning of line 69 in mm/memory.c file;
282     // The get_base() and get_limit() macros are located at line 265 in include/linux/sched.h.
283     free_page_tables(get_base(current->ldt[1]), get_limit(0x0f));
284     free_page_tables(get_base(current->ldt[2]), get_limit(0x17));
285
286     // Then, all the files opened by the current process are closed, and the working directory pwd,
287     // the root directory, the i-node of the executable file, and the library file are synchronized,
288     // and the i-nodes are put back and are respectively blanked (released). Then set state of the
289     // process to zombie state (TASK_ZOMBIE) and set process exit code.
290     for (i=0 ; i<NR_OPEN ; i++)
291         if (current->filp[i])
292             sys_close(i);
293     iput(current->pwd);
294     current->pwd = NULL;
295     iput(current->root);
296     current->root = NULL;
297     iput(current->executable);
298     current->executable = NULL;
299     iput(current->library);
300     current->library = NULL;
301     current->state = TASK_ZOMBIE;
302     current->exit_code = code;
303     /*
304      * Check to see if any process groups have become orphaned
305      * as a result of our exiting, and if they have any stopped

```

```

285      * jobs, send them a SIGUP and then a SIGCONT.  (POSIX 3.2.2.2)
286      *
287      * Case i: Our father is in a different pgrp than we are
288      * and we were the only connection outside, so our pgrp
289      * is about to become orphaned.
290      */
// POSIX 3.2.2.2 (1991 version) is a description of the exit() function. If the process group
// in which the parent is located is different from the current process, but all are in the
// same session, and the process group in which the current process is located is going to be
// an orphan, and the current process group contains job in stopped state, then two signals
// should be sent to the group: SIGHUP and SIGCONT.
291      if ((current->p_pptr->pgrp != current->pgrp) &&
292          (current->p_pptr->session == current->session) &&
293          is_orphaned_pgrp(current->pgrp) &&
294          has_stopped_jobs(current->pgrp)) {
295          kill_pg(current->pgrp, SIGHUP, 1);
296          kill_pg(current->pgrp, SIGCONT, 1);
297      }
298      /* Let father know we died */
299      current->p_pptr->signal |= (1<<(SIGCHLD-1));
300
301      /*
302      * This loop does two things:
303      *
304      * A. Make init inherit all the child processes
305      * B. Check to see if any process groups have become orphaned
306      *    as a result of our exiting, and if they have any stopped
307      *    jons, send them a SIGUP and then a SIGCONT.  (POSIX 3.2.2.2)
308      */
// If the current process has child processes (whose p_cptr points to the most recently created
// child), then process 1 (init process) becomes the parent of all its child processes. If the
// child process is already in zombie state, the child terminating signal SIGCHLD is sent to
// the init process (parent).
309      if (p = current->p_cptr) {
310          while (1) {
311              p->p_pptr = task[1];
312              if (p->state == TASK_ZOMBIE)
313                  task[1]->signal |= (1<<(SIGCHLD-1));
314              /*
315              * process group orphan check
316              * Case ii: Our child is in a different pgrp
317              * than we are, and it was the only connection
318              * outside, so the child pgrp is now orphaned.
319              */
// If the child is not in the same group as the current process but belongs to the same session,
// and the process group in which the current process is located is going to be an orphan, and
// this group has a job (process) in stopped state, then It is necessary to send two signals
// to this group: SIGHUP and SIGCONT. If the child process has sibling processes, continue to
// loop through these sibling processes.
320              if ((p->pgrp != current->pgrp) &&
321                  (p->session == current->session) &&
322                  is_orphaned_pgrp(p->pgrp) &&
323                  has_stopped_jobs(p->pgrp)) {

```

```

324         kill_pg(p->pgrp, SIGHUP, 1);
325         kill_pg(p->pgrp, SIGCONT, 1);
326     }
327     if (p->p_osptr) {
328         p = p->p_osptr;
329         continue;
330     }
331     /*
332      * This is it; link everything into init's children
333      * and leave
334      */
    // Through the above processing, all the sibling processes of the child process have been
    // processed. At this point p points to the oldest sibling of the child process. So all these
    // siblings are added to the doubly linked list header of the child process of the init process.
    // After joining, the p_cptr of the init process points to the youngest child of the current
    // process, and the oldest sibling child process p_osptr points to the youngest child process,
    // while the youngest process's p_ysptr points to the oldest sibling subprocess. Finally, the
    // p_cptr pointer of the current process is set to null and the loop is exited.
335     p->p_osptr = task[1]->p_cptr;
336     task[1]->p_cptr->p_ysptr = p;
337     task[1]->p_cptr = current->p_cptr;
338     current->p_cptr = 0;
339     break;
340 }
341 }
    // If the current process is a session leader, and if it has a control terminal, it first sends
    // a signal SIGHUP to the group using the control terminal, and then releases the terminal.
    // Then scan the task array and empty (cancel) the terminal of the process in the session.
342     if (current->leader) {
343         struct task_struct **p;
344         struct tty_struct *tty;
345
346         if (current->tty >= 0) {
347             tty = TTY_TABLE(current->tty);
348             if (tty->pgrp > 0)
349                 kill_pg(tty->pgrp, SIGHUP, 1);
350             tty->pgrp = 0;
351             tty->session = 0;
352         }
353         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
354             if ((*p)->session == current->session)
355                 (*p)->tty = -1;
356     }
    // If the current process used the coprocessor last time, then set it to NULL to discard the
    // message. In addition, if debug symbol is defined, the audit process tree function is called.
    // Finally, the scheduler is called to re-schedule the process to run, so that the parent process
    // can handle other aftermath of the zombie process.
357     if (last_task_used_math == current)
358         last_task_used_math = NULL;
359 #ifdef DEBUG_PROC_TREE
360     audit_ptree();
361 #endif
362     schedule();

```

```

363 }
364
// The system-call exit(), terminate the process. The parameter error_code is the exit status
// information provided by the user program, only the low byte is valid. Shifting error_code
// to the left by 8 bits is a requirement of the wait() or waitpid() function. The low byte
// will be used to save the status of wait(). For example, if a process is in a suspended state
// (TASK_STOPPED), its low byte is equal to 0x7f. See lines 13-19 of the sys/wait.h file. wait()
// or waitpid() can use these macros to get the exit status code of the child process or the
// reason (signal) of the child termination.
365 int sys_exit(int error_code)
366 {
367     do_exit((error_code&0xff)<<8);
368 }
369
// The system-call waitpid(). Suspend the current process until the child specified by pid exits
// (terminates) or receives a signal requesting termination of the process, or needs to call
// a signal handler. If the child process pointed to by pid has already exited (which has become
// a so-called zombie process), this syscall will return immediately. All resources used by
// the child will be released.
// If pid > 0, waiting for a child whose process id is equal to pid.
// If pid = 0, waiting for a child whose group is equal to current process group.
// If pid < -1, waiting for any child, its process group is equal to absolute value of pid.
// If pid = -1, Indicates waiting for any child process.
// If options = WUNTRACED, means that if child is stopped, it return immediately.
// If options = WNOHANG, means that if no child exits or terminates, it returns immediately.
// If the return state pointer stat_addr is not empty, the state info is saved there.
// parameters:
// pid - the process id; *stat_addr - pointer to the state info; options - waitpid option.
370 int sys_waitpid(pid_t pid, unsigned long * stat_addr, int options)
371 {
372     int flag; // selected child in ready or sleep state.
373     struct task_struct *p;
374     unsigned long oldblocked;
375
// First verify that there is enough memory space to store the status information. Then reset
// the flag. The child process sibling list is then scanned starting from the youngest child
// of the current process.
376     verify_area(stat_addr, 4);
377 repeat:
378     flag=0;
379     for (p = current->p_cptra ; p ; p = p->p_osptr) {
// If the waiting child process number pid>0 and it is not equal to the pid of the scanned child
// process p, it indicates that it is another child process of the current process, then skips
// the process and then scans the next process. Otherwise, it means that the waiting child process
// pid is found, so it jumps to line 390 to continue execution.
380         if (pid>0) {
381             if (p->pid != pid)
382                 continue;
// Otherwise, if you specify a pid=0 for the waiting process, it means that you are waiting
// for any child processes whose process group id is equal to the current process group id.
// If the process group id of the scanned process p is not equal to the group id of the current
// process, it is skipped. Otherwise, it means that a child process whose process group id is

```

```

// equal to the current process group id is found, and then jumps to line 390 to continue
// execution.
383         } else if (!pid) {
384             if (p->pgrp != current->pgrp)
385                 continue;
// Otherwise, if the specified pid < -1, it means that any child whose process group id is equal
// to the absolute value of pid are waiting. If the group id of the scanned process p is not
// equal to the absolute value of the pid, it is skipped. Otherwise, it means that a child whose
// process group id is equal to the absolute value of pid is found, and then jumps to line 390
// to continue execution.
386         } else if (pid != -1) {
387             if (p->pgrp != -pid)
388                 continue;
389         }
// If the first three do not match the pid, it means that the current process is waiting for
// any of its child processes (this time pid = -1).
//
// At this point, the selected process p is either its process id equal to the specified pid,
// or any child process in the current process group, or a child whose process id is equal to
// the absolute value of the pid, or any child process (pid is equal to -1). Next, it is processed
// according to the state of this selected process p.
//
// When the process p is in a stopped state, if the WUNTRACED option is not set at this time,
// it means that the program does not need to return immediately, or the exit code of the child
// process is equal to 0, so the scanning continues to process other child processes. If WUNTRACED
// is set and the child exit code is not 0, then the exit code is moved to the high byte, OR
// the status message 0x7f, then placed in *stat_addr, and the child pid is returned immediately
// after resetting child exit code. Here the return status of 0x7f makes the WIFSTOPPED() macro
// true, see file include/sys/wait.h, line 14.
390         switch (p->state) {
391             case TASK\_STOPPED:
392                 if (!(options & WUNTRACED) ||
393                     !p->exit_code)
394                     continue;
395                 put\_fs\_long((p->exit_code << 8) | 0x7f,
396                     stat_addr);
397                 p->exit_code = 0;
398                 return p->pid;
// If the child process p is in a dead state, it first accumulates the time it runs in the user
// mode and the kernel state into the current process (parent process). Then take out the pid
// and exit code of the child process, put the exit code into the return status position stat_addr
// and release the child process. Finally, return the exit code and pid of the child process.
// If the debug process tree symbol is defined, the process tree audit function is called.
399             case TASK\_ZOMBIE:
400                 current->cutime += p->utime;
401                 current->cstime += p->stime;
402                 flag = p->pid;
403                 put\_fs\_long(p->exit_code, stat_addr);
404                 release(p);
405 #ifdef DEBUG\_PROC\_TREE
406                 audit\_ptree();
407 #endif
408                 return flag;

```

```
// If the state of this child p is neither stopped nor in zombie state, then set flag = 1.
// This Indicates that a child process that meets the requirements has been found, but it is
// in a running state or sleep state.
409         default:
410             flag=1;
411             continue;
412     }
413 }
// After the scan of the task array is finished, if the flag is set, it indicates that the child
// process that meets the wait condition is not in the exit or zombie state. At this time, if
// the WNOHANG option has been set, it will immediately return 0 and exit. Otherwise, the current
// process is placed in an interruptible wait state, and current process signal blocking bitmap
// is saved and modified to allow it to receive the SIGCHLD signal, then execute the scheduler.
414     if (flag) {
415         if (options & WNOHANG)
416             return 0;
417         current->state=TASK_INTERRUPTIBLE;
418         oldblocked = current->blocked;
419         current->blocked &= ~(1<<(SIGCHLD-1));
420         schedule();

// When the system starts executing this process again, if the process receives an unmasked
// signal other than SIGCHLD, it will return with the exit code "Restart System Call". Otherwise
// jump to the repeat label at the beginning of the function and repeat the processing steps.
421         current->blocked = oldblocked;
422         if (current->signal & ~(current->blocked | (1<<(SIGCHLD-1))))
423             return -ERESTARTSYS;
424         else
425             goto repeat;
426     }
// If flag = 0, it means that no subprocess that meets the requirements is found, then an error
// code is returned (the child process does not exist).
427     return -ECHILD;
428 }
429
```

---

## 8.9 fork.c

### 8.9.1 Function Description

The `fork()` system-call is used to create child process. All processes in Linux are child processes of process 0 (task 0). The `fork.c` program includes a set of auxiliary processing functions for `sys_fork()` (starting at line 222 in `kernel/sys_calls.s`). It gives two C functions used in the `sys_fork()` system-call: `find_empty_process()` and `copy_process()`. It also includes the process memory area validation and memory allocation functions `verify_area()` and `copy_mem()`.

`copy_process()` is used to create and copy the code segment and data segment of the process and the environment. In the procedure of process replication, the work mainly involves the setting of information in the process data structure. The system first requests a page for the new process in the main memory area to store its



task structure information, and copies all the contents of the current process task structure as a template for the new process task structure.

The code then modifies the contents of the copied task structure. First, the code sets the current process as the parent of the new process, clears the signal bitmap and resets the statistics for the new process. Then set registers in the new process task status segment (TSS) according to the current process environment. Since the new process return value should be 0, you need to set `tss.eax = 0`. The new process kernel state stack pointer `tss.esp0` is set to the top of the memory page where the new process task structure is located, and the stack segment `tss.ss0` is set to the kernel data segment selector. `Tss.ltd` is set to the index value of the LDT descriptor in the GDT. If the current process uses a coprocessor, you also need to save the full state of the coprocessor to the `tss.i387` structure of the new process.

After that, the system sets the base address and limit of the new task code segment and data segment, and copies the page directory entry and page table entry of the current process paging management. If there is a file in the parent process that is open, the corresponding file in the child process is also open, so you need to increase the number of times the corresponding file is opened by one. The TSS and LDT descriptor entries for the new task are then set in the GDT, where the base address information points to `tss` and `ldt` in the new process task structure. Finally, set the new task to a runnable state and return a new process id to the current process.

Figure 8-13 is a position adjustment diagram for verifying the starting position and range in the memory verification function `verify_area()`. Because the memory write verification function `write_verify()` needs to operate in units of memory pages (4096 bytes), before calling `write_verify()`, it is necessary to adjust the starting position of the verification to the start position of the page, and correspondingly to the verification range.

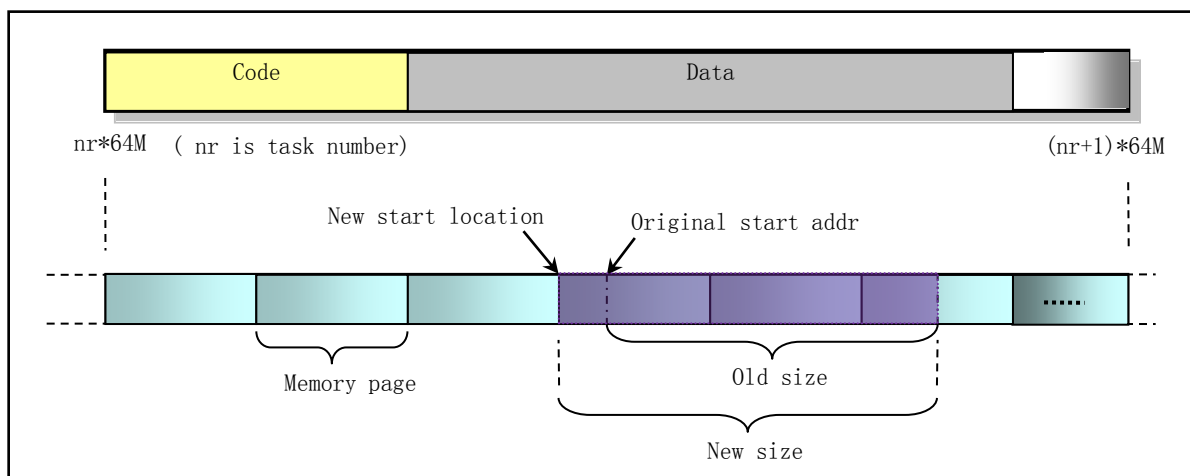


Figure 8-13 Memory verification range and starting position adjustment

The role of `fork()` is briefly described above based on the purpose of each function in the `fork.c` program. Here we will give a little more explanation on it. In general, `fork()` will first apply for a page of memory for the new process to copy the task data structure (also known as process control block, PCB) information of the parent process, and then modify the copied task data structure for the new process. These fields include resetting the registers of the TSS structure in the task structure by using the registers that is gradually pushed into stack when the system-call interrupt occurs (ie, the parameter of `copy_process()`), so that the state of the new process keeps the parent process state before entering the interrupt. The program then determines the starting position ( $nr \cdot 64MB$ ) in the linear address space for the new process. For the segmentation mechanism of the CPU, the

code segment and data segment of Linux 0.12 are exactly the same in the linear address space. The page directory entry and page table entry for the parent process are then copied for the new process. For the Linux 0.12 kernel, all programs share a page directory table at the beginning of the physical memory, and the page table of the new process needs to apply for another memory page.

During the execution of `fork()`, the kernel does not immediately allocate code and data memory pages for new processes. The new process will work with the parent process to use the code and data memory pages already in the parent process. Only the memory pages that are accessed when one of the processes accesses the memory in write mode will be copied to the newly requested memory page before the write operation.

## 8.9.2 Code Comments

Program 8-8 linux/kernel/fork.c

---

```

1  /*
2  *  linux/kernel/fork.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  'fork.c' contains the help-routines for the 'fork' system call
9  *  (see also system_call.s), and some misc functions ('verify_area').
10 *  Fork is rather simple, once you get the hang of it, but the memory
11 *  management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'
12 */
13 // <errno.h> Error number header file. Contains various error numbers in the system.
14 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
15 //   of the initial task 0, and some embedded assembly function macro statements about the
16 //   descriptor parameter settings and acquisition.
17 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
18 //   used functions of the kernel.
19 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
20 //   segment register operations.
21 // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
22 //   descriptors/interrupt gates, etc. is defined.
23 #include <errno.h>
24
25 #include <linux/sched.h>
26 #include <linux/kernel.h>
27 #include <asm/segment.h>
28 #include <asm/system.h>
29
30 // Write page verification. If page is not writable, copy the page. See mm/memory.c, L274.
31 extern void write_verify(unsigned long address);
32
33 long last_pid=0;          // The latest process id, generated by get_empty_process().
34
35
36 // The pre-write verification function for the process space.
37 // For the 80386 CPU, the privilege level 0 code is executed without regard to whether the page
38 // in user space is page protected. Therefore, the data page protection flag in the user space

```

```

// does not work when the kernel code is executed, and the copy-on-write mechanism loses its
// effect. The verify_area() function is used to solve this problem. However, for the 80486
// or later CPU, there is a write protection flag WP (bit 16) in the control register CR0. The
// kernel can disable the privilege level 0 code to write data to the user space read-only page
// by setting the flag. Therefore, the 80486 and above CPUs can achieve the same purpose of
// this function by setting this flag.
//
// This function performs a pre-write detection operation for the range of logical addresses
// from addr to (addr + size). Since the detection is performed on a page-by-page basis, the
// program first needs to find the 'start' address of the page where 'addr' is located, and
// then 'start' plus the base address of the process data segment, so that this 'start' is changed
// into an address in the linear space of the CPU 4G. Finally, write_verify() is called cyclically
// to perform pre-write verification on the memory space of the specified size. If the page
// is read-only, a share check and copy page operation (copy-on-write) is performed.
24 void verify\_area(void * addr, int size)
25 {
26     unsigned long start;
27
// First, the start address 'start' is adjusted to the start position of the page, and the size
// of the verification area is adjusted accordingly. The start & 0xfff in the next sentence
// is used to get the offset in the page. The original verification range 'size' plus this offset
// is expanded to the range value starting from the beginning of the page. Therefore, it is
// also necessary to adjust the verification start position 'start' to the page boundary. See
// Figure 8-13 above.
28     start = (unsigned long) addr;
29     size += start & 0xfff;
30     start &= 0xfffff000;           // now start is logical address.

// Next, add the base address of the process data segment and turn 'start' into the address
// in the linear space of the system. Then loop to write page verification. If the page is not
// writable, copy the page (mm/memory.c, line 274).
31     start += get\_base(current->ldt[2]);    // include/linux/sched.h, line 277
32     while (size>0) {
33         size -= 4096;
34         write\_verify(start);
35         start += 4096;
36     }
37 }
38
// Copy memory page table.
// The parameter nr is the new task number; p is the new task data structure pointer. This function
// sets the code segment and data segment base address, limit, and copy the page table for the
// new task in the linear address space. Since the Linux system uses copy-on-write technology,
// only the new page directory entries and page table entries are set up for the new process,
// and the actual physical memory pages are not allocated for the new process. At this point,
// the new process shares all memory pages with its parent process. Returns 0 if successful,
// otherwise it returns the error code.
39 int copy\_mem(int nr, struct task\_struct * p)
40 {
41     unsigned long old_data_base, new_data_base, data_limit;
42     unsigned long old_code_base, new_code_base, code_limit;
43
// First, the limits in code and data segment descriptors in current process LDT is taken.

```

```

// 0x0f is the code segment selector; 0x17 is the data segment selector. Then take the base
// address of the code & data segments of the current process in the linear address space. Since
// the Linux 0.12 kernel does not support the separation of code and data segments, it is necessary
// to check whether the code segment and the data segment base address are the same, and the
// length of the data segment is required to be at least not less than the length of the code
// segment (see Figure 5-12), otherwise the kernel displays an error message and stops running.
// get_limit() and get_base() are defined on lines 277, 279 in file include/linux/sched.h.
44     code_limit=get_limit(0x0f);
45     data_limit=get_limit(0x17);
46     old_code_base = get_base(current->ldt[1]);
47     old_data_base = get_base(current->ldt[2]);
48     if (old_data_base != old_code_base)
49         panic("We don't support separate I&D");
50     if (data_limit < code_limit)
51         panic("Bad data_limit");
// Then set the base address of the new process in the linear address space equal to
// (64MB * task no), and use this value to set the base address in the segment descriptor in
// the new process LDT. Then set the page directory entry and page table entry of the new process,
// that is, copy the page directory entry and page table entry of the current process (parent
// process). At this point, the child shares the memory page of the parent process. Normally,
// copy_page_tables() returns 0. Otherwise, it indicates an error, and the page entry that was
// just applied is released.
52     new_data_base = new_code_base = nr * TASK_SIZE;
53     p->start_code = new_code_base;
54     set_base(p->ldt[1], new_code_base);
55     set_base(p->ldt[2], new_data_base);
56     if (copy_page_tables(old_data_base, new_data_base, data_limit)) {
57         free_page_tables(new_data_base, data_limit);
58         return -ENOMEM;
59     }
60     return 0;
61 }
62
63 /*
64  * Ok, this is the main fork-routine. It copies the system process
65  * information (task[nr]) and sets up the necessary registers. It
66  * also copies the data segment in it's entirety.
67  */
// Copy process information.
// The parameters of this function start from the handler that enters the system-call interrupt
// INT 0x80, and until this function is called (sys_call.s line 231), These registers are
// gradually pushed into the kernel state stack of the process. The values (parameters) that
// are pushed onto the stack in sys_call.s file include:
// 1) The user stack ss, esp, eflags, and the return addresses cs, eip pushed when executing
// the INT instruction;
// 2) ds, es, fs, and edx, ecx, ebx pushed on stack on lines 85-91 just after entering;
// 3) The return address pushed when the sys_fork() in sys_call_table is called on line 97
// (represented by none);
// 4) On lines 226-230, gs, esi, edi, ebp, eax(nr) are pushed before calling copy_process().
// Among them, nr is the task array item index assigned by calling find_empty_process().
68 int copy_process(int nr, long ebp, long edi, long esi, long gs, long none,
69                 long ebx, long ecx, long edx, long orig_eax,
70                 long fs, long es, long ds,

```

```

71         long eip, long cs, long eflags, long esp, long ss)
72 {
73     struct task\_struct *p;
74     int i;
75     struct file *f;
76
77     // The code first allocates memory for the new task structure (if the allocation fails, it returns
78     // an error code and exits). Then put the new task structure pointer into the nr item of the
79     // task array. Where nr is task no, which is returned by the previous find_empty_process().
80     // Then copy the contents of the task structure of the current process to the beginning of the
81     // memory page p just applied.
82     p = (struct task\_struct *) get\_free\_page();
83     if (!p)
84         return -EAGAIN;
85     task[nr] = p;
86     *p = *current; /* NOTE! this doesn't copy the supervisor stack */
87
88     // Then some modifications are made to the content of the copied process structure as the task
89     // structure of the new process. The state of the new process is first set to an uninterruptible
90     // wait state to prevent the kernel from scheduling its execution. Then set the process ID pid
91     // of the new process and initialize the process run time slice value equal to its priority
92     // value (typically 15 ticks). Then reset the signal bitmap, the alarm timer, the session leader
93     // flag, the running time statistics in the kernel and user mode, and the system time start_time
94     // at which the process starts running.
95     p->state = TASK\_UNINTERRUPTIBLE;
96     p->pid = last\_pid; // new pid obtained from find_empty_process()
97     p->counter = p->priority; // run time slice value (number of ticks).
98     p->signal = 0; // signal bitmap.
99     p->alarm = 0; // alarm timer.
100    p->leader = 0; /* process leadership doesn't inherit */
101    p->utime = p->stime = 0; // user state and core state running time.
102    p->cutime = p->cstime = 0; // child's user state and core state running time.
103    p->start_time = jiffies; // the start time of the process (current time ticks).
104
105    // Now modify the task status section TSS content (see the description after the program list).
106    // Since the system allocates 1 page of memory to the task structure p, esp0 = (PAGE_SIZE +
107    // (long) p) causes esp0 to point exactly to the top of the page. ss0:esp0 is used as a stack
108    // for the program to execute in kernel mode.
109    // In addition, as we already know in Chapter 5, each task has two segment descriptors in the
110    // GDT table: one is the task's TSS segment descriptor and the other is the task's LDT table
111    // segment descriptor. The statement on line 110 is to store the selector of the LDT segment
112    // descriptor of this task into the TSS segment. When performing task switch, the CPU
113    // automatically loads the LDT segment selector from the TSS into the LDTR register.
114    p->tss.back_link = 0;
115    p->tss.esp0 = PAGE\_SIZE + (long) p; // Task kernel state stack pointer.
116    p->tss.ss0 = 0x10; // selector for the kernel state stack.
117    p->tss.eip = eip;
118    p->tss.eflags = eflags;
119    p->tss.eax = 0; // This is why the new process will return 0.
120    p->tss.ecx = ecx;
121    p->tss.edx = edx;
122    p->tss.ebx = ebx;
123    p->tss.esp = esp;

```

```

101     p->tss.ebp = ebp;
102     p->tss.esi = esi;
103     p->tss.edi = edi;
104     p->tss.es = es & 0xffff;           // The segment register has only 16 bits.
105     p->tss.cs = cs & 0xffff;
106     p->tss.ss = ss & 0xffff;
107     p->tss.ds = ds & 0xffff;
108     p->tss.fs = fs & 0xffff;
109     p->tss.gs = gs & 0xffff;
110     p->tss.ldt = LDT(nr);           // The selector for the task LDT descriptor (in GDT).
111     p->tss.trace_bitmap = 0x80000000; // (High 16 bits are valid).

// If the current task uses a coprocessor, its context is saved. The instruction CLTS is used
// to clear the task exchange flag TS in the control register CR0. The CPU sets this flag whenever
// a task switch occurs. This flag is used to manage the math coprocessor: if this flag is set,
// then each ESC instruction will be caught (Exception 7). If the coprocessor presence flag
// MP is also set, the WAIT instruction will also be captured. Therefore, if a task switch occurs
// after an ESC instruction begins execution, the contents of the coprocessor may need to be
// saved before executing the new ESC instruction. The capture handler will store the contents
// of the coprocessor and resets the TS flag. The instruction FNSAVE is used to save all states
// of the coprocessor to the memory area specified by the destination operand (tss.i387).
112     if (last task used math == current)
113         __asm__ ("clts ; fnsave %0 ; frstor %0"::"m" (p->tss.i387));

// Next, the process page table is copied, that is, the base address and the limit in the new
// task code and data segment descriptors are set, and the page table is copied. If an error
// occurs (the return value is not 0), the corresponding entry in the task array is reset and
// the memory page allocated for the new task structure is released.
114     if (copy mem(nr,p)) {           // The return is not 0, indicating an error.
115         task[nr] = NULL;
116         free page((long) p);
117         return -EAGAIN;
118     }

// Because the newly created child will share the open file with the parent process, if the
// file is opened in the parent, the number of times the corresponding file is opened needs
// to be increased by one. For the same reason, you need to increase the number of reference
// of the i nodes of the current process (parent process) by 1 for pwd, root, and executable.
119     for (i=0; i<NR_OPEN;i++)
120         if (f=p->filp[i])
121             f->f_count++;
122     if (current->pwd)
123         current->pwd->i_count++;
124     if (current->root)
125         current->root->i_count++;
126     if (current->executable)
127         current->executable->i_count++;
128     if (current->library)
129         current->library->i_count++;

// Subsequently, the new task TSS and LDT segment descriptors entries are set in the GDT. The
// limit of both segments is set to 104 bytes (see include/asm/system.h, lines 52-66). Then
// set the relationship list pointers between the processes, that is, insert the new process
// into the child process linked list of the current process. That is, the parent process of
// the new process is set to the current process, and the latest child process pointer p_cpnr

```

---

```

// and the young sibling process pointer p_ysptr of the new process are set to be empty. Then
// let the new process's buddy process pointer p_osptr be set equal to the parent's latest child
// pointer. If the current process has other child processes, let the young sibling pointer
// p_ysptr of the neighboring process point to the new process, and let child pointer of the
// current process point to this new process. Then set the new process to the ready state and
// finally return the new process id. set_tss_desc() and set_ldt_desc() are defined in file
// include/asm/system.h, 52-66. "gdt+(nr<<1)+FIRST_TSS_ENTRY" is the address of the TSS
// descriptor of the task nr in the global table. Since each task occupies 2 items in the GDT
// table, 'nr<<1' should be included in the above formula. Note that the task register TR
// is automatically loaded by the CPU during task switching.
130     set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY, &(p->tss));
131     set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY, &(p->ldt));
132     p->p_pptr = current;           // parent pointer.
133     p->p_cptr = 0;
134     p->p_ysptr = 0;
135     p->p_osptr = current->p_cptr;   // old sibling.
136     if (p->p_osptr)                // if old sibling exist, its young
137         p->p_osptr->p_ysptr = p;    // sibling points to this new process.
138     current->p_cptr = p;           // I am the current new child.
139     p->state = TASK_RUNNING;       /* do this last, just in case */
140     return last_pid;
141 }
142
// Obtain a unique process number last_pid for the new process. The function returns the task
// number (array item) in the task array.
// First get the new process id. If the global variable last_pid incremented by one is outside
// the representation range, re-use the pid number from 1. Then search for the pid just set
// in the task array to see if it is already in use. If it is, then jump to the beginning of
// the function to regain a pid number. Otherwise it means we have found a unique pid and it
// is last_pid. Then look for a free entry for the new task in the task array and return the
// item number. Note that last_pid is a global variable and does not need to be returned. In
// addition, if 64 items in the task array have been fully occupied at this time, an error code
// is returned.
143 int find_empty_process(void)
144 {
145     int i;
146
147     repeat:
148         if ((++last_pid)<0) last_pid=1;
149         for(i=0 ; i<NR_TASKS ; i++)
150             if (task[i] && ((task[i]->pid == last_pid) ||
151                             (task[i]->pgrp == last_pid)))
152                 goto repeat;
153         for(i=1 ; i<NR_TASKS ; i++)           // Task 0 is excluded.
154             if (!task[i])
155                 return i;
156         return -EAGAIN;
157 }
158

```

---

## 8.9.3 Information

### 8.9.3.1 Task Status Segment (TSS)

Figure 8-14 below shows the contents of the task state segment (TSS). The TSS for each task is saved in the task data structure `task_struct`. Please refer to Chapter 4 for a detailed description of it.

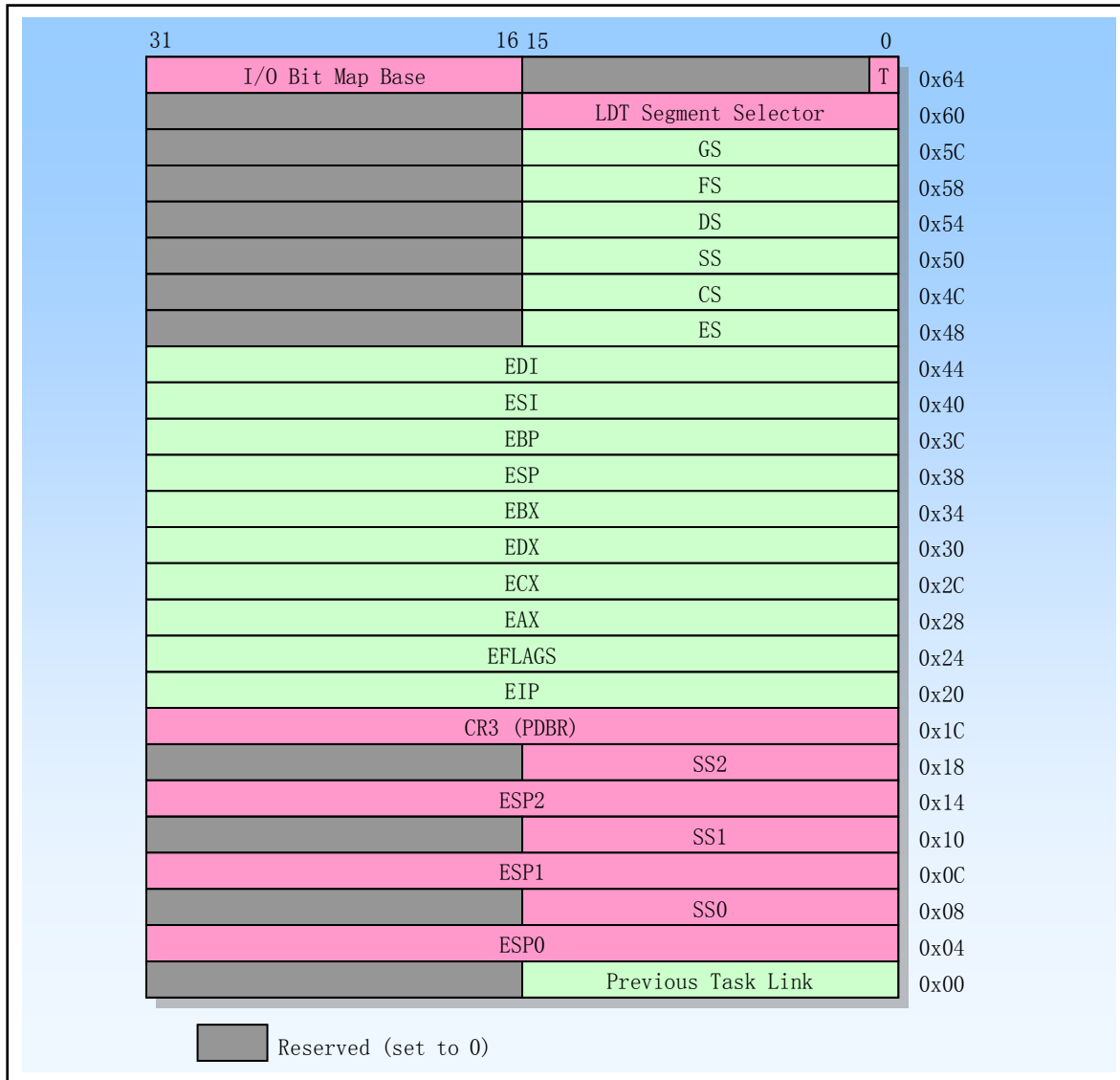


Figure 8-14 Information in the task status segment TSS.

All information required by the CPU management task is stored in a special type of segment, task state segment (TSS). The figure shows the TSS format for performing the 80386 task.

The fields in TSS can be divided into two categories: The first part is a set of information that is dynamically updated when the CPU performs a task switch. These fields are: general purpose registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI), segment registers (ES, CS, SS, DS, FS, GS), flag registers (EFLAGS), instruction pointers (EIP), the selector of the previous TSS that performed the task (updated only when returning). The second type of field is a static set of information that the CPU will read but will not change. These fields are: the LDT selector for the task, the register containing the base address of the task page directory



(PDBR), the stack pointer for privilege level 0-2, the T bit that causes the CPU to generate a debug exception when the task is switched, I/O-bit bitmap base address (the upper limit of the length is the upper limit of the length of the TSS, as described in the TSS descriptor).

The task status segment can be stored anywhere in the linear space. Similar to other types of segments, the TSS is also defined by descriptors. The TSS of the currently executing task is indicated by the task register (TR). The instructions LTR and STR are used to modify and read the selector in the task register (visible portion of the task register).

Each bit in the I/O bit map corresponds to one I/O port. For example, the bit of port 41 is the base address of the I/O bitmap +5, and the bit offset is 1. In protected mode, when an I/O instruction is encountered (IN, INS, OUT, and OUTS), the CPU first checks if the current privilege level is less than the IOPL of the flag register. If this condition is met, the I/O operation is executed. If not, the CPU will check the I/O bit map in the TSS. If the corresponding bit is set, a general protection exception will occur, otherwise the I/O operation will be performed.

If the I/O bitmap base address is set to be greater than or equal to the TSS segment limit length, it means that the TSS segment does not have an I/O permission bitmap, then all I/O instructions of the current privilege layer  $CPL > IOPL$  will result in exception protection. By default, the Linux 0.12 kernel sets the I/O bitmap base address to 0x8000, which is obviously larger than the TSS segment limit of 104 bytes, so there is no I/O permission bitmap in the Linux 0.12 kernel.

In Linux 0.12, SS0:ESP0 in the figure is used to store the stack pointer of the task running in kernel mode. SS1: ESP1 and SS2: ESP2 correspond to the stack pointers used when running privilege levels 1 and 2, respectively. These two privilege levels are not used in Linux. The stack pointer is stored in the SS:ESP register when the task is working in user mode. As can be seen from the above, each time the task enters the kernel state, the initial position of the kernel state stack pointer is unchanged, which is at the top position of the page where the task data structure is located.

## 8.10 sys.c

### 8.10.1 Function Description

The sys.c program contains many implementation functions for system-calls. Among them, if the function only has the return value -ENOSYS, it means that this version of the Linux kernel has not yet implemented this function, you can refer to the current kernel code to understand their implementation. For a description of all system-call functions, see the header file include/linux/sys.h.

The program contains a lot of functions related to process ID (pid), process group ID (pgrp or pgid), user ID (uid), user group ID (gid), actual user ID (ruid), valid user ID (euid), and session ID. (session) and other operation functions. The following is a brief description of these IDs.

A user has a user ID (uid) and a user group ID (gid). These two IDs are the IDs set for the user in the passwd file, and are often referred to as real user IDs (ruids) and real group IDs (rgids). In the i-node information of each file, the host user ID and group ID are saved, which indicate the file owner and the user group to which it belongs, and are mainly used for the authority discriminating operation when accessing or executing the file. In addition, in the task data structure of a process, three user IDs and group IDs are saved for different functions, as shown in Table 8-6.

Table 8-6 User ID and group ID associated with the process

Type	User ID	Group ID
Process	gid - User ID, indicating the user who owns the process.	gid - the group ID that indicates the user group that owns the process.
Efficient	euid - A efficient user ID indicating the access rights to the file.	egid - the efficient group ID. Indicate the permission to access the file.
Saved	suid - the saved user ID. When the set-user-ID flag of the executable file is set, the suid of the execution file is saved in the suid. Otherwise suid is equal to the euid of the process.	sgid - the saved group ID. When the set-group-ID flag of the execution file is set, the gid of the execution file is stored in the sgid. Otherwise sgid is equal to the process's egid.

The uid and gid of the process are the user ID and group ID of the process owner, that is, the real user ID (ruid) and the real group ID (rgid) of the process. Superusers can modify them using the functions `set_uid()` and `set_gid()`. The effective user ID and effective group ID are used for permission judgment when the process accesses the file.

The saved user ID (suid) and the saved group ID (sgid) are used by the process to access a file whose set-user-ID or set-group-ID flag is set. When executing a program, the euid of the process is usually the real user ID, and the egid is usually the real group ID. Therefore, the process can only access the effective user of the process, the files specified by the effective user group, or other files that are allowed to access. However, if the set-user-ID flag of a file is set, then the effective user ID of the process is set to the user ID of the file owner, so the process can access the restricted file with this flag set, and The user ID of the file owner is saved in the suid. Similarly, the set-group-ID flag of the file has a similar effect and is treated the same.

For example, if the owner of a program file is a superuser, but the program sets the set-user-ID flag, then when the program is run by a process, the effective user ID (euid) of the process will be set to the super user's ID (0). So this process has the privileges of the superuser. A practical example is the `passwd` command for Linux. This command is a program that sets the set-user-ID, thus allowing the user to modify their own password. Because the program needs to write the user's new password to the `/etc/passwd` file, and only the superuser has write access to the file, the `passwd` program needs to use the set-user-ID flag.

In addition, the process also has a process ID (pid) that identifies its own attribute, a process group ID (pgrp or pgid) of the owning process group, and a session ID (session) of the owning session. These three IDs are used to indicate the relationship between the process and the process, regardless of the user ID and the group ID.

## 8.10.2 Code comments

Program 8-9 linux/kernel/sys.c

```

1 /*
2  * linux/kernel/sys.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
// <errno.h> Error number header file. Contains various error numbers in the system.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
// of the initial task 0, and some embedded assembly function macro statements about the
// descriptor parameter settings and acquisition.

```

```

// <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
//     communication.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//     commonly used functions of the kernel.
// <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk
//     type (HD_TYPE) options.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
// <sys/times.h> Defines the running time structure tms and the times() function prototype in
//     the process.
// <sys/utsname.h> System name structure header file.
// <sys/param.h> Parameter file. Some hardware-related parameter values are given.
// <sys/resource.h> Resource file. Contains information on the limits and utilization of system
//     resources used by processes.
// <string.h> String header file. Defines some embedded functions about string operations.
7 #include <errno.h>
8
9 #include <linux/sched.h>
10 #include <linux/tty.h>
11 #include <linux/kernel.h>
12 #include <linux/config.h>
13 #include <asm/segment.h>
14 #include <sys/times.h>
15 #include <sys/utsname.h>
16 #include <sys/param.h>
17 #include <sys/resource.h>
18 #include <string.h>
19
20 /*
21  * The timezone where the local system is located. Used as a default by some
22  * programs who obtain this value by using gettimeofday.
23  */
// The time zone structure 1st field (tz_minuteswest) represents the number of minutes west
// of GMT; the second field (tz_dsttime) is daylight saving time (DST) adjustment type. This
// structure is defined in include/sys/time.h.
24 struct timezone sys\_tz = { 0, 0};
25
// Obtain the session ID of the process group according to the process group ID pgrp.
// This function is implemented in file kernel/exit.c, line 161.
26 extern int session\_of\_pgrp(int pgrp);
27
// Fetch the date and time (ftime - fetch time).
// System-calls with return value -ENOSYS indicates it has not been implemented.
28 int sys\_ftime()
29 {
30     return -ENOSYS;
31 }
32
33 int sys\_break()
34 {
35     return -ENOSYS;
36 }
37

```

```

    // Used by the current process to debug the child process.
38 int sys_ptrace()
39 {
40     return -ENOSYS;
41 }
42
    // Change and print the terminal line settings.
43 int sys_stty()
44 {
45     return -ENOSYS;
46 }
47
    // Get the terminal line setting information.
48 int sys_gtty()
49 {
50     return -ENOSYS;
51 }
52
    // Rename the filename.
53 int sys_rename()
54 {
55     return -ENOSYS;
56 }
57
58 int sys_prof()
59 {
60     return -ENOSYS;
61 }
62
63 /*
64  * This is done BSD-style, with no consideration of the saved gid, except
65  * that if you set the effective gid, it sets the saved gid too. This
66  * makes it possible for a setgid program to completely drop its privileges,
67  * which is often a useful assertion to make when you are doing a security
68  * audit over a program.
69  *
70  * The general idea is that a program which uses just setregid() will be
71  * 100% compatible with BSD. A program which uses just setgid() will be
72  * 100% compatible with POSIX w/ Saved ID's.
73  */
    // Set the real and/or effective group ID (gid) of the current task. If the task does not have
    // superuser privileges, then only its real group ID and effective group ID can be swapped.
    // If the task has superuser privileges, you can set the effective and real group IDs arbitrarily,
    // and the saved gid (sgid) is set to a effective gid (egid). The real group ID (rgid) refers
    // to the current gid of the process.
74 int sys_setregid(int rgid, int egid)
75 {
76     if (rgid > 0) {
77         if ((current->gid == rgid) ||
78             suser())
79             current->gid = rgid;
80         else
81             return(-EPERM);

```

```

82     }
83     if (egid>0) {
84         if ((current->gid == egid) ||
85             (current->egid == egid) ||
86             suser()) {
87             current->egid = egid;
88             current->sgid = egid;
89         } else
90             return(-EPERM);
91     }
92     return 0;
93 }
94
95 /*
96  * setgid() is implemented like SysV w/ SAVED_IDS
97  */
98 // Set the process group id (gid). If the task does not have superuser privileges, it can use
99 // setgid() to set its effective gid to its saved gid (sgid) or its real gid (rgid). If the
100 // task has superuser privileges, the rgid, egid, and sgid are all set to the gid specified
101 // by the parameter.
102 int sys\_setgid(int gid)
103 {
104     if (suser())
105         current->gid = current->egid = current->sgid = gid;
106     else if ((gid == current->gid) || (gid == current->sgid))
107         current->egid = gid;
108     else
109         return -EPERM;
110     return 0;
111 }
112
113 // Turn process billing on or off.
114 int sys\_acct()
115 {
116     return -ENOSYS;
117 }
118
119 // Map any physical memory to the virtual address space of the process.
120 int sys\_phys()
121 {
122     return -ENOSYS;
123 }
124
125 int sys\_lock()
126 {
127     return -ENOSYS;
128 }
129
130 int sys\_mpx()
131 {
132     return -ENOSYS;
133 }
134
135 int sys\_rlimit(int resource, int rlim_cur, int rlim_max)
136 {
137     return -ENOSYS;
138 }
139
140 int sys\_setrlimit(int resource, int rlim_cur, int rlim_max)
141 {
142     return -ENOSYS;
143 }
144
145 int sys\_swapoff(int fd)
146 {
147     return -ENOSYS;
148 }
149
150 int sys\_sysfs(int fd, int cmd, void *arg)
151 {
152     return -ENOSYS;
153 }
154
155 int sys\_timerfd\_create(int clock_id, int flags)
156 {
157     return -ENOSYS;
158 }
159
160 int sys\_timerfd\_settime(int fd, int flags, struct itimerspec *value)
161 {
162     return -ENOSYS;
163 }
164
165 int sys\_tkill(pid_t pid, int sig)
166 {
167     return -ENOSYS;
168 }
169
170 int sys\_uio(int fd, int cmd, struct uio *uio)
171 {
172     return -ENOSYS;
173 }
174
175 int sys\_uselib(char *path)
176 {
177     return -ENOSYS;
178 }
179
180 int sys\_waitid(int idtype, int id, struct kevent *ident, int options,
181               struct kevent *identdata, struct kevent *data)
182 {
183     return -ENOSYS;
184 }
185
186 int sys\_waitpid(pid_t pid, int *status, int options)
187 {
188     return -ENOSYS;
189 }
190
191 int sys\_writev(int fd, struct iovec *iov, int iovcnt)
192 {
193     return -ENOSYS;
194 }
195
196 int sys\_zswap(int fd, int cmd, void *arg)
197 {
198     return -ENOSYS;
199 }
200
201 int sys\_zswapoff(int fd)
202 {
203     return -ENOSYS;
204 }
205
206 int sys\_zswapstat(int fd)
207 {
208     return -ENOSYS;
209 }
210
211 int sys\_zswapctl(int fd, int cmd, void *arg)
212 {
213     return -ENOSYS;
214 }
215
216 int sys\_zswapinfo(int fd)
217 {
218     return -ENOSYS;
219 }
220
221 int sys\_zswapset(int fd, int cmd, void *arg)
222 {
223     return -ENOSYS;
224 }
225
226 int sys\_zswapstatfs(int fd)
227 {
228     return -ENOSYS;
229 }
230
231 int sys\_zswapstatfs2(int fd)
232 {
233     return -ENOSYS;
234 }
235
236 int sys\_zswapstatfs3(int fd)
237 {
238     return -ENOSYS;
239 }
240
241 int sys\_zswapstatfs4(int fd)
242 {
243     return -ENOSYS;
244 }
245
246 int sys\_zswapstatfs5(int fd)
247 {
248     return -ENOSYS;
249 }
250
251 int sys\_zswapstatfs6(int fd)
252 {
253     return -ENOSYS;
254 }
255
256 int sys\_zswapstatfs7(int fd)
257 {
258     return -ENOSYS;
259 }
260
261 int sys\_zswapstatfs8(int fd)
262 {
263     return -ENOSYS;
264 }
265
266 int sys\_zswapstatfs9(int fd)
267 {
268     return -ENOSYS;
269 }
270
271 int sys\_zswapstatfs10(int fd)
272 {
273     return -ENOSYS;
274 }
275
276 int sys\_zswapstatfs11(int fd)
277 {
278     return -ENOSYS;
279 }
280
281 int sys\_zswapstatfs12(int fd)
282 {
283     return -ENOSYS;
284 }
285
286 int sys\_zswapstatfs13(int fd)
287 {
288     return -ENOSYS;
289 }
290
291 int sys\_zswapstatfs14(int fd)
292 {
293     return -ENOSYS;
294 }
295
296 int sys\_zswapstatfs15(int fd)
297 {
298     return -ENOSYS;
299 }
300
301 int sys\_zswapstatfs16(int fd)
302 {
303     return -ENOSYS;
304 }
305
306 int sys\_zswapstatfs17(int fd)
307 {
308     return -ENOSYS;
309 }
310
311 int sys\_zswapstatfs18(int fd)
312 {
313     return -ENOSYS;
314 }
315
316 int sys\_zswapstatfs19(int fd)
317 {
318     return -ENOSYS;
319 }
320
321 int sys\_zswapstatfs20(int fd)
322 {
323     return -ENOSYS;
324 }
325
326 int sys\_zswapstatfs21(int fd)
327 {
328     return -ENOSYS;
329 }
330
331 int sys\_zswapstatfs22(int fd)
332 {
333     return -ENOSYS;
334 }
335
336 int sys\_zswapstatfs23(int fd)
337 {
338     return -ENOSYS;
339 }
340
341 int sys\_zswapstatfs24(int fd)
342 {
343     return -ENOSYS;
344 }
345
346 int sys\_zswapstatfs25(int fd)
347 {
348     return -ENOSYS;
349 }
350
351 int sys\_zswapstatfs26(int fd)
352 {
353     return -ENOSYS;
354 }
355
356 int sys\_zswapstatfs27(int fd)
357 {
358     return -ENOSYS;
359 }
360
361 int sys\_zswapstatfs28(int fd)
362 {
363     return -ENOSYS;
364 }
365
366 int sys\_zswapstatfs29(int fd)
367 {
368     return -ENOSYS;
369 }
370
371 int sys\_zswapstatfs30(int fd)
372 {
373     return -ENOSYS;
374 }
375
376 int sys\_zswapstatfs31(int fd)
377 {
378     return -ENOSYS;
379 }
380
381 int sys\_zswapstatfs32(int fd)
382 {
383     return -ENOSYS;
384 }
385
386 int sys\_zswapstatfs33(int fd)
387 {
388     return -ENOSYS;
389 }
390
391 int sys\_zswapstatfs34(int fd)
392 {
393     return -ENOSYS;
394 }
395
396 int sys\_zswapstatfs35(int fd)
397 {
398     return -ENOSYS;
399 }
400
401 int sys\_zswapstatfs36(int fd)
402 {
403     return -ENOSYS;
404 }
405
406 int sys\_zswapstatfs37(int fd)
407 {
408     return -ENOSYS;
409 }
410
411 int sys\_zswapstatfs38(int fd)
412 {
413     return -ENOSYS;
414 }
415
416 int sys\_zswapstatfs39(int fd)
417 {
418     return -ENOSYS;
419 }
420
421 int sys\_zswapstatfs40(int fd)
422 {
423     return -ENOSYS;
424 }
425
426 int sys\_zswapstatfs41(int fd)
427 {
428     return -ENOSYS;
429 }
430
431 int sys\_zswapstatfs42(int fd)
432 {
433     return -ENOSYS;
434 }
435
436 int sys\_zswapstatfs43(int fd)
437 {
438     return -ENOSYS;
439 }
440
441 int sys\_zswapstatfs44(int fd)
442 {
443     return -ENOSYS;
444 }
445
446 int sys\_zswapstatfs45(int fd)
447 {
448     return -ENOSYS;
449 }
450
451 int sys\_zswapstatfs46(int fd)
452 {
453     return -ENOSYS;
454 }
455
456 int sys\_zswapstatfs47(int fd)
457 {
458     return -ENOSYS;
459 }
460
461 int sys\_zswapstatfs48(int fd)
462 {
463     return -ENOSYS;
464 }
465
466 int sys\_zswapstatfs49(int fd)
467 {
468     return -ENOSYS;
469 }
470
471 int sys\_zswapstatfs50(int fd)
472 {
473     return -ENOSYS;
474 }
475
476 int sys\_zswapstatfs51(int fd)
477 {
478     return -ENOSYS;
479 }
480
481 int sys\_zswapstatfs52(int fd)
482 {
483     return -ENOSYS;
484 }
485
486 int sys\_zswapstatfs53(int fd)
487 {
488     return -ENOSYS;
489 }
490
491 int sys\_zswapstatfs54(int fd)
492 {
493     return -ENOSYS;
494 }
495
496 int sys\_zswapstatfs55(int fd)
497 {
498     return -ENOSYS;
499 }
500
501 int sys\_zswapstatfs56(int fd)
502 {
503     return -ENOSYS;
504 }
505
506 int sys\_zswapstatfs57(int fd)
507 {
508     return -ENOSYS;
509 }
510
511 int sys\_zswapstatfs58(int fd)
512 {
513     return -ENOSYS;
514 }
515
516 int sys\_zswapstatfs59(int fd)
517 {
518     return -ENOSYS;
519 }
520
521 int sys\_zswapstatfs60(int fd)
522 {
523     return -ENOSYS;
524 }
525
526 int sys\_zswapstatfs61(int fd)
527 {
528     return -ENOSYS;
529 }
530
531 int sys\_zswapstatfs62(int fd)
532 {
533     return -ENOSYS;
534 }
535
536 int sys\_zswapstatfs63(int fd)
537 {
538     return -ENOSYS;
539 }
540
541 int sys\_zswapstatfs64(int fd)
542 {
543     return -ENOSYS;
544 }
545
546 int sys\_zswapstatfs65(int fd)
547 {
548     return -ENOSYS;
549 }
550
551 int sys\_zswapstatfs66(int fd)
552 {
553     return -ENOSYS;
554 }
555
556 int sys\_zswapstatfs67(int fd)
557 {
558     return -ENOSYS;
559 }
560
561 int sys\_zswapstatfs68(int fd)
562 {
563     return -ENOSYS;
564 }
565
566 int sys\_zswapstatfs69(int fd)
567 {
568     return -ENOSYS;
569 }
570
571 int sys\_zswapstatfs70(int fd)
572 {
573     return -ENOSYS;
574 }
575
576 int sys\_zswapstatfs71(int fd)
577 {
578     return -ENOSYS;
579 }
580
581 int sys\_zswapstatfs72(int fd)
582 {
583     return -ENOSYS;
584 }
585
586 int sys\_zswapstatfs73(int fd)
587 {
588     return -ENOSYS;
589 }
590
591 int sys\_zswapstatfs74(int fd)
592 {
593     return -ENOSYS;
594 }
595
596 int sys\_zswapstatfs75(int fd)
597 {
598     return -ENOSYS;
599 }
600
601 int sys\_zswapstatfs76(int fd)
602 {
603     return -ENOSYS;
604 }
605
606 int sys\_zswapstatfs77(int fd)
607 {
608     return -ENOSYS;
609 }
610
611 int sys\_zswapstatfs78(int fd)
612 {
613     return -ENOSYS;
614 }
615
616 int sys\_zswapstatfs79(int fd)
617 {
618     return -ENOSYS;
619 }
620
621 int sys\_zswapstatfs80(int fd)
622 {
623     return -ENOSYS;
624 }
625
626 int sys\_zswapstatfs81(int fd)
627 {
628     return -ENOSYS;
629 }
630
631 int sys\_zswapstatfs82(int fd)
632 {
633     return -ENOSYS;
634 }
635
636 int sys\_zswapstatfs83(int fd)
637 {
638     return -ENOSYS;
639 }
640
641 int sys\_zswapstatfs84(int fd)
642 {
643     return -ENOSYS;
644 }
645
646 int sys\_zswapstatfs85(int fd)
647 {
648     return -ENOSYS;
649 }
650
651 int sys\_zswapstatfs86(int fd)
652 {
653     return -ENOSYS;
654 }
655
656 int sys\_zswapstatfs87(int fd)
657 {
658     return -ENOSYS;
659 }
660
661 int sys\_zswapstatfs88(int fd)
662 {
663     return -ENOSYS;
664 }
665
666 int sys\_zswapstatfs89(int fd)
667 {
668     return -ENOSYS;
669 }
670
671 int sys\_zswapstatfs90(int fd)
672 {
673     return -ENOSYS;
674 }
675
676 int sys\_zswapstatfs91(int fd)
677 {
678     return -ENOSYS;
679 }
680
681 int sys\_zswapstatfs92(int fd)
682 {
683     return -ENOSYS;
684 }
685
686 int sys\_zswapstatfs93(int fd)
687 {
688     return -ENOSYS;
689 }
690
691 int sys\_zswapstatfs94(int fd)
692 {
693     return -ENOSYS;
694 }
695
696 int sys\_zswapstatfs95(int fd)
697 {
698     return -ENOSYS;
699 }
700
701 int sys\_zswapstatfs96(int fd)
702 {
703     return -ENOSYS;
704 }
705
706 int sys\_zswapstatfs97(int fd)
707 {
708     return -ENOSYS;
709 }
710
711 int sys\_zswapstatfs98(int fd)
712 {
713     return -ENOSYS;
714 }
715
716 int sys\_zswapstatfs99(int fd)
717 {
718     return -ENOSYS;
719 }
720
721 int sys\_zswapstatfs100(int fd)
722 {
723     return -ENOSYS;
724 }
725
726 int sys\_zswapstatfs101(int fd)
727 {
728     return -ENOSYS;
729 }
730
731 int sys\_zswapstatfs102(int fd)
732 {
733     return -ENOSYS;
734 }
735
736 int sys\_zswapstatfs103(int fd)
737 {
738     return -ENOSYS;
739 }
740
741 int sys\_zswapstatfs104(int fd)
742 {
743     return -ENOSYS;
744 }
745
746 int sys\_zswapstatfs105(int fd)
747 {
748     return -ENOSYS;
749 }
750
751 int sys\_zswapstatfs106(int fd)
752 {
753     return -ENOSYS;
754 }
755
756 int sys\_zswapstatfs107(int fd)
757 {
758     return -ENOSYS;
759 }
760
761 int sys\_zswapstatfs108(int fd)
762 {
763     return -ENOSYS;
764 }
765
766 int sys\_zswapstatfs109(int fd)
767 {
768     return -ENOSYS;
769 }
770
771 int sys\_zswapstatfs110(int fd)
772 {
773     return -ENOSYS;
774 }
775
776 int sys\_zswapstatfs111(int fd)
777 {
778     return -ENOSYS;
779 }
780
781 int sys\_zswapstatfs112(int fd)
782 {
783     return -ENOSYS;
784 }
785
786 int sys\_zswapstatfs113(int fd)
787 {
788     return -ENOSYS;
789 }
790
791 int sys\_zswapstatfs114(int fd)
792 {
793     return -ENOSYS;
794 }
795
796 int sys\_zswapstatfs115(int fd)
797 {
798     return -ENOSYS;
799 }
800
801 int sys\_zswapstatfs116(int fd)
802 {
803     return -ENOSYS;
804 }
805
806 int sys\_zswapstatfs117(int fd)
807 {
808     return -ENOSYS;
809 }
810
811 int sys\_zswapstatfs118(int fd)
812 {
813     return -ENOSYS;
814 }
815
816 int sys\_zswapstatfs119(int fd)
817 {
818     return -ENOSYS;
819 }
820
821 int sys\_zswapstatfs120(int fd)
822 {
823     return -ENOSYS;
824 }
825
826 int sys\_zswapstatfs121(int fd)
827 {
828     return -ENOSYS;
829 }
830
831 int sys\_zswapstatfs122(int fd)
832 {
833     return -ENOSYS;
834 }
835
836 int sys\_zswapstatfs123(int fd)
837 {
838     return -ENOSYS;
839 }
840
841 int sys\_zswapstatfs124(int fd)
842 {
843     return -ENOSYS;
844 }
845
846 int sys\_zswapstatfs125(int fd)
847 {
848     return -ENOSYS;
849 }
850
851 int sys\_zswapstatfs126(int fd)
852 {
853     return -ENOSYS;
854 }
855
856 int sys\_zswapstatfs127(int fd)
857 {
858     return -ENOSYS;
859 }
860
861 int sys\_zswapstatfs128(int fd)
862 {
863     return -ENOSYS;
864 }
865
866 int sys\_zswapstatfs129(int fd)
867 {
868     return -ENOSYS;
869 }
870
871 int sys\_zswapstatfs130(int fd)
872 {
873     return -ENOSYS;
874 }
875
876 int sys\_zswapstatfs131(int fd)
877 {
878     return -ENOSYS;
879 }
880
881 int sys\_zswapstatfs132(int fd)
882 {
883     return -ENOSYS;
884 }
885
886 int sys\_zswapstatfs133(int fd)
887 {
888     return -ENOSYS;
889 }
890
891 int sys\_zswapstatfs134(int fd)
892 {
893     return -ENOSYS;
894 }
895
896 int sys\_zswapstatfs135(int fd)
897 {
898     return -ENOSYS;
899 }
900
901 int sys\_zswapstatfs136(int fd)
902 {
903     return -ENOSYS;
904 }
905
906 int sys\_zswapstatfs137(int fd)
907 {
908     return -ENOSYS;
909 }
910
911 int sys\_zswapstatfs138(int fd)
912 {
913     return -ENOSYS;
914 }
915
916 int sys\_zswapstatfs139(int fd)
917 {
918     return -ENOSYS;
919 }
920
921 int sys\_zswapstatfs140(int fd)
922 {
923     return -ENOSYS;
924 }
925
926 int sys\_zswapstatfs141(int fd)
927 {
928     return -ENOSYS;
929 }
930
931 int sys\_zswapstatfs142(int fd)
932 {
933     return -ENOSYS;
934 }
935
936 int sys\_zswapstatfs143(int fd)
937 {
938     return -ENOSYS;
939 }
940
941 int sys\_zswapstatfs144(int fd)
942 {
943     return -ENOSYS;
944 }
945
946 int sys\_zswapstatfs145(int fd)
947 {
948     return -ENOSYS;
949 }
950
951 int sys\_zswapstatfs146(int fd)
952 {
953     return -ENOSYS;
954 }
955
956 int sys\_zswapstatfs147(int fd)
957 {
958     return -ENOSYS;
959 }
960
961 int sys\_zswapstatfs148(int fd)
962 {
963     return -ENOSYS;
964 }
965
966 int sys\_zswapstatfs149(int fd)
967 {
968     return -ENOSYS;
969 }
970
971 int sys\_zswapstatfs150(int fd)
972 {
973     return -ENOSYS;
974 }
975
976 int sys\_zswapstatfs151(int fd)
977 {
978     return -ENOSYS;
979 }
980
981 int sys\_zswapstatfs152(int fd)
982 {
983     return -ENOSYS;
984 }
985
986 int sys\_zswapstatfs153(int fd)
987 {
988     return -ENOSYS;
989 }
990
991 int sys\_zswapstatfs154(int fd)
992 {
993     return -ENOSYS;
994 }
995
996 int sys\_zswapstatfs155(int fd)
997 {
998     return -ENOSYS;
999 }
1000
1001 int sys\_zswapstatfs156(int fd)
1002 {
1003     return -ENOSYS;
1004 }
1005
1006 int sys\_zswapstatfs157(int fd)
1007 {
1008     return -ENOSYS;
1009 }
1010
1011 int sys\_zswapstatfs158(int fd)
1012 {
1013     return -ENOSYS;
1014 }
1015
1016 int sys\_zswapstatfs159(int fd)
1017 {
1018     return -ENOSYS;
1019 }
1020
1021 int sys\_zswapstatfs160(int fd)
1022 {
1023     return -ENOSYS;
1024 }
1025
1026 int sys\_zswapstatfs161(int fd)
1027 {
1028     return -ENOSYS;
1029 }
1030
1031 int sys\_zswapstatfs162(int fd)
1032 {
1033     return -ENOSYS;
1034 }
1035
1036 int sys\_zswapstatfs163(int fd)
1037 {
1038     return -ENOSYS;
1039 }
1040
1041 int sys\_zswapstatfs164(int fd)
1042 {
1043     return -ENOSYS;
1044 }
1045
1046 int sys\_zswapstatfs165(int fd)
1047 {
1048     return -ENOSYS;
1049 }
1050
1051 int sys\_zswapstatfs166(int fd)
1052 {
1053     return -ENOSYS;
1054 }
1055
1056 int sys\_zswapstatfs167(int fd)
1057 {
1058     return -ENOSYS;
1059 }
1060
1061 int sys\_zswapstatfs168(int fd)
1062 {
1063     return -ENOSYS;
1064 }
1065
1066 int sys\_zswapstatfs169(int fd)
1067 {
1068     return -ENOSYS;
1069 }
1070
1071 int sys\_zswapstatfs170(int fd)
1072 {
1073     return -ENOSYS;
1074 }
1075
1076 int sys\_zswapstatfs171(int fd)
1077 {
1078     return -ENOSYS;
1079 }
1080
1081 int sys\_zswapstatfs172(int fd)
1082 {
1083     return -ENOSYS;
1084 }
1085
1086 int sys\_zswapstatfs173(int fd)
1087 {
1088     return -ENOSYS;
1089 }
1090
1091 int sys\_zswapstatfs174(int fd)
1092 {
1093     return -ENOSYS;
1094 }
1095
1096 int sys\_zswapstatfs175(int fd)
1097 {
1098     return -ENOSYS;
1099 }
1100
1101 int sys\_zswapstatfs176(int fd)
1102 {
1103     return -ENOSYS;
1104 }
1105
1106 int sys\_zswapstatfs177(int fd)
1107 {
1108     return -ENOSYS;
1109 }
1110
1111 int sys\_zswapstatfs178(int fd)
1112 {
1113     return -ENOSYS;
1114 }
1115
1116 int sys\_zswapstatfs179(int fd)
1117 {
1118     return -ENOSYS;
1119 }
1120
1121 int sys\_zswapstatfs180(int fd)
1122 {
1123     return -ENOSYS;
1124 }
1125
1126 int sys\_zswapstatfs181(int fd)
1127 {
1128     return -ENOSYS;
1129 }
1130
1131 int sys\_zswapstatfs182(int fd)
1132 {
1133     return -ENOSYS;
1134 }
1135
1136 int sys\_zswapstatfs183(int fd)
1137 {
1138     return -ENOSYS;
1139 }
1140
1141 int sys\_zswapstatfs184(int fd)
1142 {
1143     return -ENOSYS;
1144 }
1145
1146 int sys\_zswapstatfs185(int fd)
1147 {
1148     return -ENOSYS;
1149 }
1150
1151 int sys\_zswapstatfs186(int fd)
1152 {
1153     return -ENOSYS;
1154 }
1155
1156 int sys\_zswapstatfs187(int fd)
1157 {
1158     return -ENOSYS;
1159 }
1160
1161 int sys\_zswapstatfs188(int fd)
1162 {
1163     return -ENOSYS;
1164 }
1165
1166 int sys\_zswapstatfs189(int fd)
1167 {
1168     return -ENOSYS;
1169 }
1170
1171 int sys\_zswapstatfs190(int fd)
1172 {
1173     return -ENOSYS;
1174 }
1175
1176 int sys\_zswapstatfs191(int fd)
1177 {
1178     return -ENOSYS;
1179 }
1180
1181 int sys\_zswapstatfs192(int fd)
1182 {
1183     return -ENOSYS;
1184 }
1185
1186 int sys\_zswapstatfs193(int fd)
1187 {
1188     return -ENOSYS;
1189 }
1190
1191 int sys\_zswapstatfs194(int fd)
1192 {
1193     return -ENOSYS;
1194 }
1195
1196 int sys\_zswapstatfs195(int fd)
1197 {
1198     return -ENOSYS;
1199 }
```

```

129 int sys_ulimit()
130 {
131     return -ENOSYS;
132 }
133
134 // Returns the time (in seconds) from January 1, 1970, 00:00:00 GMT.
135 // If parameter tloc is not null, then the time value is also stored there. Since the location
136 // pointed to by the parameter is in user space, you need to use the function put_fs_long()
137 // to store the time value in user space. When running in the kernel, the segment register fs
138 // is pointed to the current user data space by default. So the function can use the fs segment
139 // register to access values in user space.
140 int sys_time(long * tloc)
141 {
142     int i;
143
144     i = CURRENT_TIME;
145     if (tloc) {
146         verify_area(tloc, 4); // Verify mem capacity is sufficient (4 bytes).
147         put_fs_long(i, (unsigned long *)tloc);
148     }
149     return i;
150 }
151
152 /*
153  * Unprivileged users may change the real user id to the effective uid
154  * or vice versa. (BSD-style)
155  *
156  * When you set the effective uid, it sets the saved uid too. This
157  * makes it possible for a setuid program to completely drop its privileges,
158  * which is often a useful assertion to make when you are doing a security
159  * audit over a program.
160  *
161  * The general idea is that a program which uses just setreuid() will be
162  * 100% compatible with BSD. A program which uses just setuid() will be
163  * 100% compatible with POSIX w/ Saved ID's.
164  */
165 // Set the real and/or effective user id (uid) of the task. If the task does not have superuser
166 // privileges, then only its real uid (ruid) and effective uid (euid) can be swapped. If the
167 // task has superuser privileges, you can arbitrarily set effective and real user IDs. The saved
168 // uid (suid) is set to the same value as the euid.
169 int sys_setreuid(int ruid, int euid)
170 {
171     int old_ruid = current->uid;
172
173     if (ruid > 0) {
174         if ((current->euid == ruid) ||
175             (old_ruid == ruid) ||
176             suser())
177             current->uid = ruid;
178         else
179             return(-EPERM);
180     }
181     if (euid > 0) {

```

```

172         if ((old_ruid == euid) ||
173             (current->euid == euid) ||
174             suser()) {
175             current->euid = euid;
176             current->suid = euid;
177         } else {
178             current->uid = old_ruid;
179             return(-EPERM);
180         }
181     }
182     return 0;
183 }
184
185 /*
186  * setuid() is implemented like SysV w/ SAVED_IDS
187  *
188  * Note that SAVED_ID's is deficient in that a setuid root program
189  * like sendmail, for example, cannot set its uid to be a normal
190  * user and then switch back, because if you're root, setuid() sets
191  * the saved uid too. If you don't like this, blame the bright people
192  * in the POSIX committee and/or USG. Note that the BSD-style setreuid()
193  * will allow a root program to temporarily drop privileges and be able to
194  * regain them by swapping the real and effective uid.
195  */
196 // Set the task user ID (uid). If the task does not have superuser privileges, it can use setuid()
197 // to set its effective uid (euid) to its saved uid (suid) or its real uid (ruid). If the task
198 // has superuser privileges, the ruid, euid, and suid will be set to the uid specified by the
199 // parameter.
200 int sys_setuid(int uid)
201 {
202     if (suser())
203         current->uid = current->euid = current->suid = uid;
204     else if ((uid == current->uid) || (uid == current->suid))
205         current->euid = uid;
206     else
207         return -EPERM;
208     return(0);
209 }
210
211 // Set the system boot time. The parameter tptr is the time value (in seconds) that is counted
212 // from January 1, 1970, at 00:00:00 GMT.
213 // The calling process must have superuser privileges. Where HZ=100 is the operating frequency
214 // of the kernel system. Since the location of the parameter pointer is in user space, you need
215 // to use the function get_fs_long() to access the value. When running in the kernel, the segment
216 // register fs is pointed to the current user data space by default. So the function can use
217 // fs to access values in user space. The current time value provided by the function parameter
218 // minus the time second value (jiffies/HZ) that the system has been running is the boot time
219 // seconds.
220 int sys_stime(long * tptr)
221 {
222     if (!suser())
223         return -EPERM;
224     startup_time = get_fs_long((unsigned long *) tptr) - jiffies/HZ;

```

```

212     jiffies_offset = 0;
213     return 0;
214 }
215
216 // Get the current task runtime statistics.
217 // Returns the task runtime statistics of the tms structure at the user data space pointed to
218 // by tbuf. The tms structure includes the process user runtime, kernel runtime, child user
219 // runtime, and child kernel runtime. The return value of the function is the ticks that the
220 // system runs to the current time.
221 int sys_times(struct tms * tbuf)
222 {
223     if (tbuf) {
224         verify_area(tbuf, sizeof *tbuf);
225         put_fs_long(current->utime, (unsigned long *)&tbuf->tms_utime);
226         put_fs_long(current->stime, (unsigned long *)&tbuf->tms_stime);
227         put_fs_long(current->cutime, (unsigned long *)&tbuf->tms_cutime);
228         put_fs_long(current->cstime, (unsigned long *)&tbuf->tms_cstime);
229     }
230     return jiffies;
231 }
232
233 // Set the program's end position in memory.
234 // When the value of the parameter end_data_seg is reasonable and the system does have enough
235 // memory and the process does not exceed its maximum data segment size, the function sets the
236 // value specified by end_data_seg at the end of the data segment. This value must be greater
237 // than the end of the code and less than 16KB of the end of the stack. The return value is
238 // the new end value of the data segment (if the return value is different from the required
239 // value, an error has occurred). This function is not directly called by the user, but is wrapped
240 // by the libc library function, and the return value is not the same.
241 int sys_brk(unsigned long end_data_seg)
242 {
243     // If the parameter is greater than the end of the code and is less than (stack - 16KB), set
244     // the new data segment end value.
245     if (end_data_seg >= current->end_code &&
246         end_data_seg < current->start_stack - 16384)
247         current->brk = end_data_seg;
248     return current->brk; // Returns the current data segment end value.
249 }
250
251 /*
252  * This needs some heave checking ...
253  * I just haven't get the stomach for it. I also don't fully
254  * understand sessions/pgrp etc. Let somebody who does explain it.
255  *
256  * OK, I think I have the protection semantics right.... this is really
257  * only important on a multi-user system anyway, to make sure one user
258  * can't send a signal to a process owned by another. -TYT, 12/12/91
259  */
260
261 // Set the process group id of the specified process pid to pgid.
262 // The parameter pid is the process id. If the parameter pid is 0, then let this pid be equal
263 // to pid of the current process. The parameter pgid specifies the process group id. If it is
264 // 0, let it be equal to the process group id of the process pid. If the function is used to
265 // move a process from one process group to another, the two process groups must belong to the

```



```

// same session. In this case, the parameter pgid specifies the existing process group ID to
// join, and the session ID of the group must be the same as the process to be joined (L263).
245 int sys_setpgid(int pid, int pgid)
246 {
247     int i;
248
// If the parameter pid is 0, the pid is set to the pid of the current process. If the parameter
// pgid is 0, then pgid is also the pid of the current process. If pgid is less than 0, an invalid
// error code is returned.
249     if (!pid)
250         pid = current->pid;
251     if (!pgid)
252         pgid = current->pid;
253     if (pgid < 0)
254         return -EINVAL;
// Scan the task array for the task with the specified process pid. If the process with the
// process ID is pid is found, and the parent process of the process is the current process
// or the process is the current process, then if the task is already the session leader, an
// error is returned. If the session id of the task is different from the current process, or
// the specified process group id pgid is different from the pid, and the session id of the
// pgid process group is different from the session id of the current process, an error is
// returned. Otherwise, set the pgrp field of the found process to pgid and return 0. If the
// process with the specified pid is not found, the return process doesn't have an error code.
255     for (i=0 ; i<NR\_TASKS ; i++)
256         if (task[i] && (task[i]->pid == pid) &&
257             ((task[i]->p_pptr == current) ||
258              (task[i] == current))) {
259             if (task[i]->leader)
260                 return -EPERM;
261             if ((task[i]->session != current->session) ||
262                 ((pgid != pid) &&
263                  (session of pgrp(pgid) != current->session)))
264                 return -EPERM;
265             task[i]->pgrp = pgid;
266             return 0;
267         }
268     return -ESRCH;
269 }
270
// Returns the process group id of the current process. Equivalent to getpgid(0).
271 int sys_getpgrp(void)
272 {
273     return current->pgrp;
274 }
275
// Create a session (ie set its leader=1) and set its session id = its group id = its process
// id. If the current process is already the session leader and is not a superuser, an error
// is returned. Otherwise, set the current process to be new session leader (leader = 1). The
// session and group id pgrp are set to equal to the process pid, and the current process has
// no control terminal. The last system-call returns the session id.
276 int sys_setsid(void)
277 {
278     if (current->leader && !suser())

```

```

279         return -EPERM;
280     current->leader = 1;
281     current->session = current->pgrp = current->pid;
282     current->tty = -1;           // the current process has no control terminal.
283     return current->pgrp;
284 }
285
286 /*
287  * Supplementary group ID's
288  */
289 // Get the other auxiliary user group id of the current process.
290 // The groups[] array in the task structure holds multiple user group ids to which the process
291 // belongs. The array has a total of NGROUPS items. If the value of an item is NOGROUP (that
292 // is, -1), it means that all items are idle after the start of the item. Otherwise the user
293 // group id is saved in the array item.
294 // The parameter gidsetsize is the maximum number of user group ids that can be stored in the
295 // user cache, that is, the maximum number of items in the group list; the grouplist is the
296 // user space cache that stores these user group numbers.
297 int sys_getgroups(int gidsetsize, gid_t *grouplist)
298 {
299     int i;
300
301     // First verify that the user cache space pointed to by the grouplist is sufficient, and then
302     // obtain the user group id one by one from the groups[] array of the current process structure
303     // and copy it into the user cache. During the copying process, if the number of items in groups[]
304     // is greater than the number specified by the given parameter gidsetsize, it means that the
305     // cache given is too small to accommodate all the groups of the current process. The operation
306     // will return with an error code. If the copy operation is ok, the function will eventually
307     // return the number of copied user group ids.
308     if (gidsetsize)
309         verify_area(grouplist, sizeof(gid_t) * gidsetsize);
310
311     for (i = 0; (i < NGROUPS) && (current->groups[i] != NOGROUP);
312          i++, grouplist++) {
313         if (gidsetsize) {
314             if (i >= gidsetsize)
315                 return -EINVAL;
316             put_fs_word(current->groups[i], (short *) grouplist);
317         }
318     }
319     return(i);           // Returns the number of user group ids.
320 }
321
322 // Set the other secondary user group ids to which the current process belongs.
323 // The parameter gidsetsize is the number of user group ids to be set; the grouplist is the
324 // user space cache containing the user group ids.
325 int sys_setgroups(int gidsetsize, gid_t *grouplist)
326 {
327     int i;
328
329     // First check the validity of the permissions and parameters. Only the superuser can modify
330     // or set the secondary user group ids of the current process, and the number of items cannot
331     // exceed the capacity of the groups [NGROUPS] array. Then copy the user group id one by one

```

```

// from the user buffer to the array. A total of gidsetsize is copied. If the number of copies
// does not fill in groups[], fill in the next item with a value of -1 (NOGROUP). Finally, the
// function returns 0.
311     if (!suser())
312         return -EPERM;
313     if (gidsetsize > NGROUPS)
314         return -EINVAL;
315     for (i = 0; i < gidsetsize; i++, grouplist++) {
316         current->groups[i] = get_fs_word((unsigned short *) grouplist);
317     }
318     if (i < NGROUPS)
319         current->groups[i] = NOGROUP;
320     return 0;
321 }
322
// Check if the current process belongs to user group grp. Yes ret 1, otherwise returns 0.
323 int in_group_p(gid_t grp)
324 {
325     int i;
326
// If the effective group id (egid) of the current process is grp, the process belongs to the
// grp group, the function returns 1. Otherwise, it scans the process's secondary user group
// array for the grp group id. If so, the function also returns 1. If the item with the value
// NOGROUP is scanned, it means that the full valid item has been scanned and no matching group
// id is found, so the function returns 0.
327     if (grp == current->egid)
328         return 1;
329
330     for (i = 0; i < NGROUPS; i++) {
331         if (current->groups[i] == NOGROUP)
332             break;
333         if (current->groups[i] == grp)
334             return 1;
335     }
336     return 0;
337 }
338
// The utsname structure contains some string fields that hold the name of the system. It contains
// 5 fields, which are: the name of the current operating system, the network node name (host
// name), the current operating system release level, the operating system version number, and
// the hardware type name that the system is running. This structure is defined in the
// include/sys/utsname.h file. Here they are set to their default values using the constant
// symbols in include/linux/config.h file. They are: "Linux", "(none)", "0", "0.12", "i386".
339 static struct utsname thisname = {
340     UTS_SYSNAME, UTS_NODENAME, UTS_RELEASE, UTS_VERSION, UTS_MACHINE
341 };
342
// Get system name information.
343 int sys_uname(struct utsname * name)
344 {
345     int i;
346
347     if (!name) return -ERROR;

```

```

348     verify\_area(name, sizeof *name);
349     for(i=0; i<sizeof *name; i++)
350         put\_fs\_byte((char *) &thisname)[i], i+(char *) name);
351     return 0;
352 }
353
354 /*
355  * Only sethostname; gethostname can be implemented by calling uname\(\)
356  */
357 // Set the system host name (network node name).
358 // The parameter name points to the buffer containing the host name string in the user data
359 // area; len is the host name string length.
360 int sys\_sethostname(char *name, int len)
361 {
362     int i;
363
364     // The system hostname can only be set or modified by the superuser, and the hostname length
365     // cannot exceed the maximum length MAXHOSTNAMELEN.
366     if (!suser())
367         return -EPERM;
368     if (len > MAXHOSTNAMELEN)
369         return -EINVAL;
370     for (i=0; i < len; i++) {
371         if ((thisname.nodename[i] = get\_fs\_byte(name+i)) == 0)
372             break;
373     }
374     // After the copy is completed, if the string provided by the user does not contain NULL
375     // characters, if the length of the copied hostname does not exceed MAXHOSTNAMELEN, a NULL
376     // is added after the host name string. If MAXHOSTNAMELEN characters have been filled, change
377     // the last character to NULL.
378     if (thisname.nodename[i]) {
379         thisname.nodename[i > MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
380     }
381     return 0;
382 }
383
384 // Get the resource limits of the current process.
385 // An array rlim[RLIM_NLIMITS] is defined in task's structure to control the boundaries of the
386 // system's use of system resources. Each item in the array is an rlimit structure with two
387 // fields. One specifies the current limit (soft limit) of the specified resource, and the other
388 // indicates the system's maximum limit (hard limit) for the specified resource. Each item of
389 // the rlim[] array corresponds to the limit information of a resource for the current process.
390 // The Linux 0.12 system has a limit on six resources, namely RLIM_NLIMITS=6. Please refer to
391 // lines 41-46 in file include/sys/resource.h. The 'resource' specifies the name of the resource
392 // we are consulting. It is actually the index of the rlim[] array in the task structure. rlim
393 // is a user buffer pointer to the rlimit structure, which is used to store the obtained resource
394 // limit information.
395 int sys\_getrlimit(int resource, struct rlimit *rlim)
396 {
397     // The resource being queried is actually the index value of the rlim[] array in the process
398     // task structure. The index value can of course not be greater than the maximum number of items
399     // in the array RLIM_NLIMITS. After verifying that the user buffer is sufficient, the resource
400     // structure information is copied into it, and returns 0.

```

```

377     if (resource >= RLIM_NLIMITS)
378         return -EINVAL;
379     verify_area(rlim, sizeof *rlim);
380     put_fs_long(current->rlim[resource].rlim_cur,      // Current (soft) limits.
381                (unsigned long *) rlim);
382     put_fs_long(current->rlim[resource].rlim_max,      // System (hard) limits.
383                ((unsigned long *) rlim)+1);
384     return 0;
385 }
386
// Sets resource limits for the current process.
// The parameter resource specifies the resource name for which we set the limit. It is actually
// the index of the rlim[] array in the task structure. The parameter rlim is a user buffer
// pointer to the rlimit structure for kernel to read new resource limits.
387 int sys_setrlimit(int resource, struct rlimit *rlim)
388 {
389     struct rlimit new, *old;
390
// First determine the validity of the parameter resource (the task structure rlim[] index).
// Then let the rlimit structure pointer 'old' point to the current rlimit structure of the
// specified resource. The resource limit information provided by the user is then copied to
// the temporary rlimit structure 'new'. At this time, if the soft limits value or the hard
// limits value in the 'new' structure is greater than the original limits, and the current
// is not a superuser, the permission error is returned. Otherwise, it means that the information
// in 'new' is reasonable or the process is a super user, then the information specified in
// the original process is changed to the information in the 'new' structure, and 0 is returned.
391     if (resource >= RLIM_NLIMITS)
392         return -EINVAL;
393     old = current->rlim + resource;      // old = current->rlim[resource]
394     new.rlim_cur = get_fs_long((unsigned long *) rlim);
395     new.rlim_max = get_fs_long(((unsigned long *) rlim)+1);
396     if (((new.rlim_cur > old->rlim_max) ||
397         (new.rlim_max > old->rlim_max)) &&
398         !suser())
399         return -EPERM;
400     *old = new;
401     return 0;
402 }
403
404 /*
405  * It would make sense to put struct rusage in the task_struct,
406  * except that would make the task_struct be *really big*. After
407  * task_struct gets moved into malloc'ed memory, it would
408  * make sense to do this. It will make moving the rest of the information
409  * a lot simpler! (Which we're not doing right now because we're not
410  * measuring them yet).
411  */
// Get resource usage information of the specified process.
// This syscall provides the current process or its terminated or waiting child resource usage.
// If the parameter 'who' is equal to RUSAGE_SELF, the resource usage information of the current
// process is returned. If 'who' is RUSAGE_CHILDREN, returns the terminated or waiting child
// resource usage information of the current process. Symbolic constants RUSAGE_SELF and
// RUSAGE_CHILDREN and the rusage structure are all defined in file include/sys/resource.h.

```

```

412 int sys\_getrusage(int who, struct rusage *ru)
413 {
414     struct rusage r;
415     unsigned long *lp, *lpend, *dest;
416
417     // First check the validity of the process specified by the parameter 'who'. If 'who' is neither
418     // RUSAGE_SELF (specifying the current process) nor RUSAGE_CHILDREN (specifying the child),
419     // it is returned with an invalid parameter code. Otherwise, after verifying the user buffer
420     // area specified by the pointer ru, the temporary rusage structure area 'r' is cleared.
421     if (who != RUSAGE\_SELF && who != RUSAGE\_CHILDREN)
422         return -EINVAL;
423     verify\_area(ru, sizeof *ru);
424     memset((char *) &r, 0, sizeof(r)); // at the end of include/strings.h
425
426     // If the parameter who is RUSAGE_SELF, the current process resource usage information is copied
427     // into the r structure. If the specified process who is RUSAGE_CHILDREN, the terminated or
428     // waiting child resource usage of the current process is copied to the temporary rusage
429     // structure r. The macros CT_TO_SECS and CT_TO_USECS are used to convert the current system
430     // ticks into seconds and microseconds. They are defined in file include/linux/sched.h.
431     // jiffies_offset is the system parameter error adjustment.
432     if (who == RUSAGE\_SELF) {
433         r.ru_utime.tv_sec = CT\_TO\_SECS(current->utime);
434         r.ru_utime.tv_usec = CT\_TO\_USECS(current->utime);
435         r.ru_stime.tv_sec = CT\_TO\_SECS(current->stime);
436         r.ru_stime.tv_usec = CT\_TO\_USECS(current->stime);
437     } else {
438         r.ru_utime.tv_sec = CT\_TO\_SECS(current->cutime);
439         r.ru_utime.tv_usec = CT\_TO\_USECS(current->cutime);
440         r.ru_stime.tv_sec = CT\_TO\_SECS(current->cstime);
441         r.ru_stime.tv_usec = CT\_TO\_USECS(current->cstime);
442     }
443
444     // Then let the lp pointer point to the r structure, lpend to the end of the r structure, and
445     // the dest pointer to the ru structure in user space. Finally, copy the information in r into
446     // the user space ru and return 0.
447     lp = (unsigned long *) &r;
448     lpend = (unsigned long *) (&r+1);
449     dest = (unsigned long *) ru;
450     for (; lp < lpend; lp++, dest++)
451         put\_fs\_long(*lp, dest);
452     return(0);
453 }
454
455 // Get the current time of the system and return it in the specified format.
456 // The timeval structure contains two fields, seconds and microseconds (tv_sec and tv_usec).
457 // The timezone structure contains two fields, the number of minutes west of Greenwich Mean
458 // Time (tz_minuteswest) and the daylight saving time (dst) adjustment type (tz_dsttime). Both
459 // structures are defined in the include/sys/time.h file.
460 int sys\_gettimeofday(struct timeval *tv, struct timezone *tz)
461 {
462     // If the timeval structure pointer is not empty, the current time (seconds and microseconds)
463     // is returned in the structure; if the pointer of the timezone structure in the given user
464     // data space is not empty, the structure is also returned. The startup_time in the code is
465     // the system boot time (seconds). The macros CT_TO_SECS and CT_TO_USECS are used to convert

```

```

// the current systems ticks to be expressed in seconds and microseconds. They are defined in
// the include/linux/sched.h file. Jiffies_offset is the system ticks error adjustment.
442     if (tv) {
443         verify\_area(tv, sizeof *tv);
444         put\_fs\_long(startup\_time + CT\_TO\_SECS(jiffies+jiffies\_offset),
445                 (unsigned long *) tv);
446         put\_fs\_long(CT\_TO\_USECS(jiffies+jiffies\_offset),
447                 ((unsigned long *) tv)+1);
448     }
449     if (tz) {
450         verify\_area(tz, sizeof *tz);
451         put\_fs\_long(sys\_tz.tz_minuteswest, (unsigned long *) tz);
452         put\_fs\_long(sys\_tz.tz_dsttime, ((unsigned long *) tz)+1);
453     }
454     return 0;
455 }
456
457 /*
458  * The first time we set the timezone, we will warp the clock so that
459  * it is ticking GMT time instead of local time. Presumably,
460  * if someone is setting the timezone then we are running in an
461  * environment where the programs understand about timezones.
462  * This should be done at boot time in the /etc/rc script, as
463  * soon as possible, so that the clock can be set right. Otherwise,
464  * various programs will get confused when the clock gets warped.
465  */
// Set the current time of the system.
// The parameter tv is a pointer to the timeval structure in the user data area. tz is a pointer
// to the timezone structure in the user data area. This operation requires superuser privileges.
// If both are empty, nothing is done and the function returns 0.
466 int sys\_settimeofday(struct timeval *tv, struct timezone *tz)
467 {
468     static int      firsttime = 1;
469     void            adjust\_clock();
470
// Superuser privileges are required to set the current time of the system. If the tz pointer
// is not empty, set the system time zone information, that is, copy the user timezone structure
// information to the sys_tz structure in the system (see line 24). If the system-call is called
// for the first time and the parameter tv pointer is not empty, adjust the system clock value.
471     if (!suser())
472         return -EPERM;
473     if (tz) {
474         sys\_tz.tz_minuteswest = get\_fs\_long((unsigned long *) tz);
475         sys\_tz.tz_dsttime = get\_fs\_long((unsigned long *) tz)+1);
476         if (firsttime) {
477             firsttime = 0;
478             if (!tv)
479                 adjust\_clock();
480         }
481     }
// If the timeval structure pointer tv of the parameter is not empty, the system clock is set
// with the structure information. First, the system time expressed by the second value (sec)
// plus the microsecond value (usec) is obtained from the position indicated by tv, and then

```

---

```

// the system startup time global variable startup_time is modified by the second value, and
// the system error value jiffies_offset is set by the microsecond value.
482     if (tv) {
483         int sec, usec;
484
485         sec = get\_fs\_long((unsigned long *)tv);
486         usec = get\_fs\_long((unsigned long *)tv)+1);
487
488         startup\_time = sec - jiffies/HZ;
489         jiffies\_offset = usec * HZ / 1000000 - jiffies%HZ;
490     }
491     return 0;
492 }
493
494 /*
495  * Adjust the time obtained from the CMOS to be GMT time instead of
496  * local time.
497  *
498  * This is ugly, but preferable to the alternatives. Otherwise we
499  * would either need to write a program to do it in /etc/rc (and risk
500  * confusion if the program gets run more than once; it would also be
501  * hard to make the program warp the clock precisely n hours) or
502  * compile in the timezone information into the kernel. Bad, bad....
503  *
504  * XXX Currently does not adjust for daylight savings time. May not
505  * need to do anything, depending on how smart (dumb?) the BIOS
506  * is. Blast it all.... the best thing to do not depend on the CMOS
507  * clock at all, but get the time via NTP or timed if you're on a
508  * network....
509  *                                     - TYT, 1/1/92
510  */
511 // Adjust the system startup time to the time based on GMT.
512 // The startup_time unit is seconds, so you need to multiply the time zone minute by 60.
513 void adjust\_clock()
514 {
515     startup\_time += sys\_tz.tz_minuteswest*60;
516 }
517
518 // Set the creation file attribute mask of the current process to mask & 0777 and return the
519 // original mask.
520 int sys\_umask(int mask)
521 {
522     int old = current->umask;
523
524     current->umask = mask & 0777;
525     return (old);
526 }

```

---



## 8.11 vsprintf.c

### 8.11.1 Function Description

The program mainly includes the `vsprintf()` function, which formats the parameters and outputs them to the buffer. Since this function is a standard function in the C library, there is basically no content related to the working principle of the kernel, so it can be skipped. In order to understand its application in the kernel, you can directly read the instructions of the function after the code. Please also refer to the C library function manual for how to use the `vsprintf()` function.

### 8.11.2 Code Comments

Program 8-10 linux/kernel/vsprintf.c

---

```
1  /*
2  *  linux/kernel/vsprintf.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /* vsprintf.c -- Lars Wirzenius & Linus Torvalds. */
8  /*
9  * Wirzenius wrote this portably, Torvalds fucked it up :-)
10 */
11 // Lars Wirzenius is a friend of Linus and worked in an office at Helsinki University. When
12 // developing Linux in the summer of 1991, Linus was not very familiar with the C language at
13 // the time, and would not be familiar with the variable parameter list function. So Lars
14 // Wirzenius wrote this code for the kernel to display messages. He later (1998) admitted that
15 // there was a bug in this code that was not discovered until 1994 and was corrected. This bug
16 // is when the * is used as the output field width, the code forgets to increment the pointer
17 // to skip the asterisk. This bug still exists in this code (line 130). His personal homepage
18 // is http://liw.iki.fi/liw/
19
20 // #include <stdarg.h> Standard parameter file. Define a list of variable parameters in the
21 // form of macros. It mainly describes a type (va_list) and three macros (va_start, va_arg
22 // and va_end) for the vsprintf, vprintf, vfprintf functions.
23 // #include <string.h> A string header file. Mainly defines some embedded functions about
24 // string operations.
25 #include <stdarg.h>
26 #include <string.h>
27
28 /* we use this so that we can do without the ctype library */
29 #define is\_digit(c) ((c) >= '0' && (c) <= '9') // check if it's a digital char.
30
31 // Convert a string of characters to an integer. The input is a pointer to a numeric string
32 // pointer and returns the result value. In addition, the pointer will move forward.
33 static int skip\_atoi(const char **s)
34 {
35     int i=0;
36
37     while (is\_digit(**s))
```

```

23         i = i*10 + *((*s)++) - '0';
24     return i;
25 }
26
27 // Here defines symbol constants for various conversion types.
28 #define ZEROPAD 1          /* pad with zero */
29 #define SIGN 2            /* unsigned/signed long */
30 #define PLUS 4            /* show plus */
31 #define SPACE 8           /* space if plus */
32 #define LEFT 16           /* left justified */
33 #define SPECIAL 32        /* 0x */
34 #define SMALL 64          /* use 'abcdef' instead of 'ABCDEF' */
35
36 // Division operation. Input: n is the dividend, base is the divisor; result: n is the quotient,
37 // and the function returns the remainder. See 3.3.2 for embedded assembly.
38 #define do_div(n,base) ({ \
39     int __res; \
40     __asm__ ("divl %4": "=a" (n), "=d" (__res): "0" (n), "1" (0), "r" (base)); \
41     __res; })
42
43 // Converts an integer to a string of the specified radix.
44 // Input: num - integer; base - radix; size - the length of the string;
45 // precision - the length of the number (precision); type - type options.
46 // Output: The converted string is stored in the buffer at the str pointer. The return value
47 // is a pointer to the end of the string after the number is converted to a string.
48 static char * number(char * str, int num, int base, int size, int precision
49 , int type)
50 {
51     char c, sign, tmp[36];
52     const char *digits="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
53     int i;
54
55     // If 'type' indicates a lowercase letter, a lowercase set of letters is defined. If 'type'
56     // indicates that you want to adjust left, the zero-fill flag in 'type' is masked. If the base
57     // is less than 2 or greater than 36, exit the program. That is, the program can only process
58     // numbers with a base between 2-32.
59     if (type&SMALL) digits="0123456789abcdefghijklmnopqrstuvwxyz";
60     if (type&LEFT) type &= ~ZEROPAD;
61     if (base<2 || base>36)
62         return 0;
63
64     // If 'type' indicates zero-filled, set variable c='0', otherwise set c equal space char.
65     // If 'type' indicates a signed number and the value num is less than 0, then negative sign
66     // is set and num is taken as an absolute value. Otherwise, if the 'type' indicates that it
67     // is a plus sign, then set sign=plus, otherwise if the 'type' has a space sign then sign =space,
68     // otherwise set sign to 0.
69     c = (type & ZEROPAD) ? '0' : ' ';
70     if (type&SIGN && num<0) {
71         sign='-';
72         num = -num;
73     } else
74         sign=(type&PLUS) ? '+' : ((type&SPACE) ? ' ' : 0);
75
76     // If signed, the width value is decremented by 1. If the type indicates a special conversion,
77     // then reduce the hexadecimal width by another 2 (for 0x) and for the octal width minus 1 (for

```

```

    // placing a zero before result).
57     if (sign) size--;
58     if (type&SPECIAL)
59         if (base==16) size -= 2;
60         else if (base==8) size--;
    // If num is 0, the temporary string = '0'; otherwise the num is converted to a character form
    // according to the given radix. If the number of digits is greater than the precision, the
    // precision is expanded to the number of digits. The width size minus the number of numeric
    // characters stored.
61     i=0;
62     if (num==0)
63         tmp[i++]='0';
64     else while (num!=0)
65         tmp[i++]=digits[do_div(num, base)];
66     if (i>precision) precision=i;
67     size -= precision;

    // From here on, the resulting conversion result is gradually formed and temporarily placed
    // in the string str. If there is no zero-fill and left-aligned flags in the 'type', Spaces
    // indicated by the remaining width is first filled in str. If the sign is needed, store the
    // sign symbol in it.
68     if (!(type&(ZEROPAD+LEFT)))
69         while(size-->0)
70             *str++ = ' ';
71     if (sign)
72         *str++ = sign;
    // If the 'type' indicates a special conversion, a '0' is placed for the first location the
    // octal result; '0x' is stored for the hexadecimal value.
73     if (type&SPECIAL)
74         if (base==8)
75             *str++ = '0';
76         else if (base==16) {
77             *str++ = '0';
78             *str++ = digits[33];    // 'X' 或 'x'
79         }
    // If there is no left adjust flag in the 'type', the c ('0' or space) is stored in the remaining
    // width, see line 51.
80     if (!(type&LEFT))
81         while(size-->0)
82             *str++ = c;
    // At this time, i holds the number of digits of num. If the number of digits is less than the
    // precision, put (precision - i) '0' in str. Then the converted numeric characters are also
    // filled in str. A total of i.
83     while(i<precision--)
84         *str++ = '0';
85     while(i-->0)
86         *str++ = tmp[i];
    // If the width value is still greater than zero, it means that there is a left adjustment in
    // the 'type' flag. Then put spaces in the remaining width.
87     while(size-->0)
88         *str++ = ' ';
89     return str;    // Returns the pointer to the end of the converted string.
90 }

```

```

91 // The following function sends the formatted output to the string buffer. The parameter fmt
92 // is the format string; args is a pointer to a parameter list; buf is the output string buffer.
93 int vsprintf(char *buf, const char *fmt, va_list args)
94 {
95     int len;
96     int i;
97     char *str;           // Used to hold strings during the conversion.
98     char *s;
99     int *ip;
100
101     int flags;           /* flags to number() */
102
103     int field_width;     /* width of output field */
104     int precision;       /* min. # of digits for integers; max
105                          number of chars for from string */
106     int qualifier;       /* 'h', 'l', or 'L' for integer fields */
107
108     // First, the character pointer is pointed to the buf, and then the format string is scanned,
109     // and each format conversion instruction is processed accordingly.
110     // The format conversion string starts with '%'. Here we scan '%' from the fmt format string
111     // to find the beginning of the format conversion string. Normal characters that are not format
112     // directives are stored in str in order.
113     for (str=buf ; *fmt ; ++fmt) {
114         if (*fmt != '%') {
115             *str++ = *fmt;
116             continue;
117         }
118     }
119
120     // Obtains the flag field in the format string and puts into the flags variable.
121     /* process flags */
122     flags = 0;
123     repeat:
124         ++fmt;           /* this also skips first '%' */
125         switch (*fmt) {
126             case '-': flags |= LEFT; goto repeat;
127             case '+': flags |= PLUS; goto repeat;
128             case ' ': flags |= SPACE; goto repeat;
129             case '#': flags |= SPECIAL; goto repeat;
130             case '0': flags |= ZEROPAD; goto repeat;
131         }
132
133     // Take the current parameter width field value into the field_width variable. If the width
134     // field is a numeric, it is directly taken as the width value. If the width field is the character
135     // '*', it means that the next specifies the width, so va_arg is called to take the width value.
136     // If the width value is less than 0, the negative number indicates that it has the flag field
137     // '-' (left-aligned), so it is necessary to add it to the flag variable and take the field
138     // width value as absolute value.
139     /* get field width */
140     field_width = -1;
141     if (is_digit(*fmt))
142         field_width = skip_atoi(&fmt);
143     else if (*fmt == '*') {

```

```

130      /* it's the next argument */    // bug here, should add "++fmt;"
131      field_width = va\_arg(args, int);
132      if (field_width < 0) {
133          field_width = -field_width;
134          flags |= LEFT;
135      }
136  }
137
138  // The following takes the precision field of the format and puts it into the precision variable.
139  // The flag for the start of the precision field is '.'. The processing is similar to the width
140  // field above. If the precision field is a number, it is taken directly as the precision value.
141  // If the precision field is the character '*', it means that the next parameter specifies the
142  // precision. So call va\_arg to take the precision value. If the width value is less than 0,
143  // the field precision value is taken as 0.
144
145  /* get the precision */
146  precision = -1;
147  if (*fmt == '.') {
148      ++fmt;
149      if (is\_digit(*fmt))
150          precision = skip\_atoi(&fmt);
151      else if (*fmt == '*') {
152          /* it's the next argument */ // should add ++fmt;
153          precision = va\_arg(args, int);
154      }
155      if (precision < 0)
156          precision = 0;
157  }
158
159  // This code analyzes the length modifier and stores it in the qualifer variable. For the meaning
160  // of h, l, L, see the description after the list.
161
162  /* get the conversion qualifier */
163  qualifier = -1;
164  if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
165      qualifier = *fmt;
166      ++fmt;
167  }
168
169  // The conversion format indicator is analyzed below.
170  // If the indicator is 'c', it means that the corresponding parameter should be a character.
171  // At this time, if the flag field indicates that it is not left-aligned, the field is preceded
172  // by a 'width - 1' spaces, and then the parameter character is placed. If the width field is
173  // still greater than 0, it means that it is left-aligned, then add a 'width - 1' spaces after
174  // the parameter character.
175
176  switch (*fmt) {
177      case 'c':
178          if (!(flags & LEFT))
179              while (--field_width > 0)
180                  *str++ = ' ';
181          *str++ = (unsigned char) va\_arg(args, int);
182          while (--field_width > 0)
183              *str++ = ' ';
184          break;

```

```

// If the indicator is 's', it means that the corresponding parameter is a string. Then the
// length of the parameter string is taken first, and if it exceeds the precision field value,
// set the extended precision field equal to string length. If the flag indicates that it's
// not left-aligned, the field is preceded by a 'width-string length' spaces, and then the string
// is placed. If the width is still greater than 0, it means that it is left-aligned, then add
// 'width - string length' spaces after the string.
169         case 's':
170             s = va\_arg(args, char *);
171             len = strlen(s);
172             if (precision < 0)
173                 precision = len;
174             else if (len > precision)
175                 len = precision;
176
177             if (!(flags & LEFT))
178                 while (len < field_width--)
179                     *str++ = ' ';
180             for (i = 0; i < len; ++i)
181                 *str++ = *s++;
182             while (len < field_width--)
183                 *str++ = ' ';
184             break;
185
// If the format character is 'o', it means that the corresponding parameter needs to be
// converted into a string of octal numbers. Call the number() function to handle it.
186         case 'o':
187             str = number(str, va\_arg(args, unsigned long), 8,
188                 field_width, precision, flags);
189             break;
190
// If the format converter is 'p', it means that the corresponding parameter is a pointer type.
// At this time, if the parameter does not set the width field, the default width is 8, and
// you need to add zero. Then invoke the number() function for processing.
191         case 'p':
192             if (field_width == -1) {
193                 field_width = 8;
194                 flags |= ZEROPAD;
195             }
196             str = number(str,
197                 (unsigned long) va\_arg(args, void *), 16,
198                 field_width, precision, flags);
199             break;
200
// If the format converter is 'x' or 'X', it means that the corresponding parameter needs to
// be printed as a hexadecimal output. 'x' means lowercase letters.
201         case 'x':
202             flags |= SMALL;
203         case 'X':
204             str = number(str, va\_arg(args, unsigned long), 16,
205                 field_width, precision, flags);
206             break;
207
// If the format character is 'd', 'i' or 'u', then the parameter is an integer. 'd', 'i' stands

```

---

```

// for symbolic integers, so you need to add a signed flag. 'u' stands for an unsigned integer.
208         case 'd':
209         case 'i':
210             flags |= SIGN;
211         case 'u':
212             str = number(str, va\_arg(args, unsigned long), 10,
213                     field_width, precision, flags);
214             break;
215
// If the format indicator is 'n', it means that the number of characters converted so far is
// saved to the position specified by the corresponding parameter pointer. First use va_arg()
// to get the pointer to the parameter, and then store the number of characters already converted
// into the position pointed to by the pointer.
216         case 'n':
217             ip = va\_arg(args, int *);
218             *ip = (str - buf);
219             break;
220
// If the format converter is not '%', a '%' is written directly into the output string. If
// there is still a character of the format converter, the character is also written directly
// into the output string, and the loop continues to process the format string. Otherwise, it
// means that it has been processed to the end of the format string, then exit the loop.
221         default:
222             if (*fmt != '%')
223                 *str++ = '%';
224             if (*fmt)
225                 *str++ = *fmt;
226             else
227                 --fmt;
228             break;
229     }
230 }
231 *str = '\0'; // add null to the end of the output string.
232 return str-buf; // return string length.
233 }
234

```

---

### 8.11.3 Information

#### 8.11.3.1 Format string of vsprintf()

The vsprintf() function is one of the printf() series. These functions all produce formatted output: A format string fmt that determines the output format is received, and the parameters are formatted with the format string to produce a formatted output. The function declaration form is as follows:

---

```
int vsprintf(char *buf, const char *fmt, va\_list args)
```

---

The other functions in the printf() series declare a form similar to this. Printf will send the output directly to the standard output device stdout (the display / console), so there is no first parameter (buffer pointer) in the above declaration. cprintf also sends the output to the console. fprintf sends the output to a file, so the first argument will be a file handle. The printf with a 'v' character (for example, vfprintf) indicates that the arguments

comes from the `va_list` args of the `va_arg` array. `printf` with a prefix 's' means that the format result is output to the string buffer `buf` (the `buf` should have enough space) and end with a null. The following describes in detail how to use the format string.

### 1. The format string

The format string in the `printf` family is used to control how functions are converted, formatted, and output their parameters. For each format, there must be a corresponding parameter, and too many parameters will be ignored. The format string contains two types of components, one is a simple character that will be copied directly into the output; the other is a conversion indicator string used to format the corresponding parameter.

### 2. The format indicator string

The format of the format indicator string is as follows:

---

```
%[flags][width][.prec][h|l|L][type]
```

---

Each conversion indication string needs to start with a percent sign (%). The meaning of each part is as follows:

- `[flags]` is an optional sequence of flag characters;
- `[width]` is an optional width indicator;
- `[.prec]` is an optional precision indicator;
- `[h|l|L]` Is an optional input length modifier;
- `[type]` is a conversion type character (or a conversion indicator).

● flags control output alignment, numeric symbols, decimal points, trailing zeros, binary, octal, or hexadecimal, see the notes in lines 27-33 above. The flag characters and their meanings are as follows:

'#' Indicates that the corresponding parameter needs to be converted to a "special form". For octal (o), the first character of the converted string must be a zero. For hexadecimal (x or X), the converted string must start with '0x' or '0X'. For e, E, f, F, g, and G, even if there are no decimal places, the conversion result will always have a decimal point. For g or G, the trailing zero will not be deleted.

'0' The conversion result should be zero attached. For d, i, o, u, x, X, e, E, f, g, and G, the left side of the conversion result will be filled with zeros instead of spaces. If the 0 and - flags are both present, the 0 flag will be ignored. For numerical conversion, if the precision field is given, the 0 flag is also ignored.

'-' The converted result will be left-adjusted (left) within the boundaries of the corresponding field. (The default is to make a right adjustment - right). The n conversion is an exception, and the conversion result will be filled with spaces on the right.

' ' A space should be reserved before a positive result resulting from a signed conversion.

'+' Indicates that a symbol (+ or -) is always required before a symbol conversion result. For the default case, only negative numbers use a negative sign.

● width specifies the output string width, which specifies the minimum width value of the field. If the result of the conversion is smaller than the specified width, then the left side (or the right side, if the right adjustment flag is given) needs to be filled with spaces or zeros (determined by the flags) and so on. In addition to using numbers to specify the width field, you can also use '\*' to indicate that the width of the field is given by the next integer parameter. When the width of the conversion value is greater than the width specified by width,



the small width value will not truncate the result under any circumstances. The field width is expanded to include the full result.

- precision is the number that describes the minimum number of output. For d, I, o, u, x, and X conversions, the precision value indicates the number of digits at least. For e, E, f, and F, this value indicates the number of digits that appear after the decimal point. For g or G, indicate the maximum number of significant digits. For s or S conversions, the precision value specifies the maximum number of characters in the output string.

- The length modifier indicator describes the output type form after the integer conversion. In the following description, the 'integer number conversion' represents d, i, o, u, x or X conversion.

- 'hh' Indicates that the subsequent integer conversion corresponds to a signed or unsigned character parameter.

- 'h' Indicates that the subsequent integer conversion corresponds to a signed integer or unsigned short integer parameter.

- 'l' Indicates that the subsequent integer conversion corresponds to a long integer or unsigned long integer argument.

- 'll' Indicates that the subsequent integer conversion corresponds to a long long integer or unsigned long long integer argument.

- 'L' Indicates that the e, E, f, F, g or G conversion result corresponds to a long double precision parameter.

- type is the format of the input parameter type and output that are accepted. The meaning of each conversion indicator is as follows:

- 'd,I' Integer parameters will be converted to signed integers. If there is precision, the minimum number of digits that need to be output is given. If the number of values being converted is small, it will be zeroed to the left. The default precision value is 1.

- 'o,u,x,X' Unsigned integers are converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x or X) representations. x means that lowercase letters (abcdef) are used to represent hexadecimal numbers, and X means uppercase letters (ABCDEF) for hexadecimal numbers. If there is a precision field, it means the minimum number of digits that need to be output. If the number of values being converted is small, it will be zeroed to the left. The default precision value is 1.

- 'e,E' These two conversion characters are used to round the arguments into a form of [-]d.ddde+dd. The number of digits after the decimal point is equal to the precision. If there is no precision field, take the default value of 6. If the precision is 0, no decimals appear. E means that the index is represented by a capital letter E. The index part is always represented by 2 digits. If the value is 0, then the index is 00.

- 'f,F' These two characters are used to round the arguments into a form of [-]ddd.ddd. The number of digits after the decimal point is equal to the precision. If there is no precision field, take the default value of 6. If the precision is 0, no decimals appear. If there is a decimal point, there will be at least one digit at the back.

- 'g,G' These two characters convert the argument to the format of f or e (in the case of G, the F or E format). The precision value specifies the number of integers. If there is no precision field, its default value is 6. If the precision is 0, it is treated as 1. If the index is less than -4 or greater than or equal to the precision when converting, the e format is used. The zeros after the decimal part will be deleted. The decimal point appears only if there is at least one decimal.

- 'c' Indicates that the argument will be converted to an unsigned character and the result of the conversion

will be output.

's' Indicates that the input is required to point to a string, and the string is to end in null. If there is a precision field, only the number of characters required for precision is output, and the string does not have to end in null.

'p' Indicates that a hexadecimal number is output as a pointer.

'n' Used to save the number of characters converted so far to the position specified by the corresponding input pointer. The parameter is not converted.

'%' Indicates that a percent sign is output and no conversion is performed. That is, the entire conversion indication is now '%%'.

## 8.12 printk.c

### 8.12.1 Function Description

printk() is the print (display) function used by the kernel and has the same functionality as printf() in the C standard library. The reason for rewriting such a function is that you can't directly use the fs segment register dedicated to user mode in the kernel code, you need to save it first.

The reason why fs cannot be used directly is because in the actual screen display function tty\_write(), the message to be displayed is taken from the data segment pointed to by the fs segment, that is, in the user program data segment. The message that needs to be displayed in the printk() function is in the kernel data segment, that is, in the kernel data segment pointed to by ds register when executed in the kernel code. Therefore, you need to temporarily use the fs segment register in the printk() function.

The printk() function first uses vsprintf() to format the parameters, and then calls tty\_write() to print the information when the fs segment register is saved.

### 8.12.2 Code Comments

Program 8-11 linux/kernel/printk.c

---

```
1 /*
2  * linux/kernel/printk.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * When in kernel-mode, we cannot use printf, as fs is liable to
9  * point to 'interesting' things. Make a printf with fs-saving, and
10 * all is well.
11 */
12 // <stdarg.h> Standard parameter header file. Define a list of variable parameters in the form
13 // of macros. It mainly describes one type (va_list) and three macros (va_start, va_arg and
14 // va_end) for the vsprintf, vprintf, and vfprintf functions.
15 // <stddef.h> The standard definition header file. NULL, offsetof(TYPE, MEMBER) is defined.
16 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
17 // used functions of the kernel.
18 #include <stdarg.h>
19 #include <stddef.h>
20
```

```
15 #include <linux/kernel.h>
16
17 static char buf[1024];           // Temporary buffer for display.
18
19 // vsprintf() is defined at line 92 in linux/kernel/vsprintf.c.
20 extern int vsprintf(char * buf, const char * fmt, va_list args);
21
22 // The display function used by the kernel.
23 int printk(const char *fmt, ...)
24 {
25     va_list args;                // is actually a character pointer type.
26     int i;
27
28     // First run the parameter processing start function, then use the format string fmt to convert
29     // the parameter list args and output it to buf. The return value i is equal to the length of
30     // the output string. Then run the parameter processing end function. Finally, the console is
31     // called to display the function and return the number of characters displayed.
32     va_start(args, fmt);
33     i=vsprintf(buf, fmt, args);
34     va_end(args);
35     console_print(buf);          // chr_drv/console.c, line 995.
36     return i;
37 }
```

---

## 8.13 panic.c

### 8.13.1 Function Description

The panic() function is used to display kernel error messages and put the system into an infinite dead loop. In many places in the kernel, this function is called if the kernel code has a serious error during execution. Calling the panic() function in many cases is a straightforward approach. This approach follows the UNIX "as concise as possible" principle.

### 8.13.2 Code Comments

Program 8-12 linux/kernel/panic.c

---

```
1 /*
2  * linux/kernel/panic.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * This function is used through-out the kernel (includeinh mm and fs)
9  * to indicate a major problem.
10 */
11 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
12 // used functions of the kernel.
```

```
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
// of the initial task 0, and some embedded assembly function macro statements about the
// descriptor parameter settings and acquisition.
11 #include <linux/kernel.h>
12 #include <linux/sched.h>
13
14 void sys_sync(void);    /* it's really int */ // fs/buffer.c, line 44.
15
// This function is used to display the major error messages that appear in the kernel, and
// run the file system synchronization function, then enter the endless loop - the system
// crashes. If the current process is task 0, it also indicates that the swapper task is in
// error and the file system synchronization function has not been run yet. The volatile keyword
// before the function name is used to tell the compiler gcc that the function will not return.
// This allows gcc to produce better code, and more importantly, use this keyword to avoid false
// warnings (uninitialized variables). This is equivalent to the current gcc attributes:
// 'void panic(const char *s) __attribute__((noreturn));'
16 volatile void panic(const char * s)
17 {
18     printk("Kernel panic: %s\n\r", s);
19     if (current == task[0])
20         printk("In swapper task - not syncing\n\r");
21     else
22         sys_sync();
23     for(;;);
24 }
25
```

---

## 8.14 Summary

This chapter mainly studies the 12 source files in the linux/kernel directory, and gives the implementation of some of the most important mechanisms in the kernel, including system calls, process scheduling, process replication, and process termination processing.

Beginning with the next chapter, we began to learn how the Linux kernel supports three block devices of hard disks, floppy disks, and memory virtual disks. First, the important data structures such as device request items and request queues used by the block device are described, and then the specific operation mode of the block device is described in detail. After that, the five source files in the block device directory are hacked one by one.

## 9 Block Device Driver







One of the main functions of the operating system is to communicate with peripheral I/O devices and to control these peripheral devices with a unified interface. All devices of the operating system can be roughly divided into two types: a block device and a character device. A block device is a device that can be addressed and accessed in units of fixed-size data blocks, such as hard disk devices and floppy disk devices. A character device is a device that operates on a character stream and cannot be addressed. For example, printer devices, network interface devices, and terminal devices. For ease of management and access, the operating system distinguishes these devices uniformly by device number. In the Linux 0.12 kernel, devices are divided into 7 classes, which have a total of 7 major device numbers (0 to 6). The devices in each type can be further differentiated based on the sub (sub, secondary) device numbers. The device types and associated devices for each device number are listed in Table 9-1. It can be seen from the table that some devices (memory devices) can be accessed as either block devices or as character devices. This chapter mainly discusses and describes the implementation principles and methods of block device drivers. A discussion of character devices is presented in the next chapter.

Table 9-1 The major device number in the Linux 0.12 kernel

Major No.	Name	Device type	Description
0	None	None	None
1	ram	Block/Char	Ram devices (virtual disk)
2	fd	Block	floppy device
3	hd	Block	harddisk device
4	ttyx	Char	device (virtual or serial terminal)
5	tty	Char	tty device
6	lp	Char	lp printer

The Linux 0.12 kernel mainly supports three types of block devices: hard disk, floppy disk and memory virtual disk. Since block devices are primarily related to file systems and caches, you can quickly take a look at the contents of the file system chapter before proceeding with this chapter. The source code files covered in this chapter are shown in List 9-1.

List 9-1 linux/kernel/blk\_drv

Filename	Size	Last modified time (GMT)	Desc.
 <a href="#">Makefile</a>	2759 bytes	1992-01-12 19:49:21	
 <a href="#">blk.h</a>	3963 bytes	1991-12-26 20:02:50	
 <a href="#">floppy.c</a>	11660 bytes	1992-01-10 03:45:33	
 <a href="#">hd.c</a>	8331 bytes	1992-01-16 06:39:10	
 <a href="#">ll_rw_blk.c</a>	4734 bytes	1991-12-19 21:26:20	
 <a href="#">ramdisk.c</a>	2740 bytes	1991-12-06 03:08:06	

The purpose of the program in this chapter can be divided into two categories, one is the driver corresponding to each device, such programs are: hard disk driver `hd.c`, floppy disk driver `floppy.c`, and Memory virtual disk driver `ramdisk.c`.

The other class includes only one program, which is used by the other program in the kernel to access the block device interface program `ll_rw_blk.c`. Another file is the block device-specific header file `blk.h`, which provides a uniform setup and the same device request start procedure for these three block devices to interact with the `ll_rw_blk.c` program.

## 9.1 Main Functions

Reading and writing data on hard disk and floppy block devices is performed by an interrupt handler. The amount of data read and written by the kernel each time is in units of one logical block (1024 bytes), while the block device controller accesses the block device in units of sectors (512 bytes). During processing, the kernel uses a read and write request entry wait queue to sequentially buffer the operation of reading and writing multiple logical blocks.

When a program needs to read a logical block on the hard disk, it will apply to the buffer management program, and the program process enters a sleep wait state. The buffer manager first looks in the buffer for whether it has been read before. If there is already in the buffer, the corresponding buffer block header pointer is directly returned to the program and the waiting process is woken up. If the required data block does not exist in the buffer, the buffer manager calls the low-level block read/write function `ll_rw_block()` in this chapter to issue a read data block operation request to the corresponding block device driver. The function creates a request structure item for this and inserts it into the request queue. In order to improve the efficiency of reading and writing the disk and reduce the distance the head moves, the kernel code uses the elevator algorithm to insert the request item into the request queue position where the head movement distance is the smallest.

If the request queue of the block device is empty at this time, it indicates that the block device is not busy at the moment. The kernel then immediately issues a read data command to the controller of the block device. When the controller reads the data into the specified buffer block, it will issue an interrupt request signal and call the corresponding read command post-processing function to process continuing reading of the sectors or end the request. For example, closing operation of the corresponding block device, setting flags about the buffer block data has been updated, and finally woken up the process waiting for the block data.

### 9.1.1 Block Device Requests and Request Queues

According to the above description, we know that the low-level read/write function `ll_rw_block()` is to establish a connection with various block devices through a request item and issue a read/write request operation. For various block devices, the kernel uses a block device table (array) `blk_dev[]` for management. Each block device occupies one item in the block device table. The structure of each block device item in the block device table is (see the file `blk.h` below):

---

```
struct blk\_dev\_struct {
    void (*request_fn)(void);           // A function pointer of requests.
    struct request * current_request;    // current request structure pointer.
};
extern struct blk\_dev\_struct blk\_dev[NR_BLK_DEV]; // Block device table (NR_BLK_DEV = 7).
```

---

The first field is a function pointer that is used to manipulate the request item of the corresponding block device. For example, for a hard disk drive, it is `do_hd_request()`, and for a floppy device it is `do_floppy_request()`. The second field is the current request item structure pointer, which is used to indicate the request item currently being processed by the block device. All the request items in the table are set to `NULL` at the time of initialization.

The block device table will be set during the initialization function of each device in the `init/main.c` program. For ease of extension, Mr. Linus built the block device table into an array indexed by the major device number. In Linux 0.12, there are 7 major device numbers, as shown in Table 9-2. Among them, the main device numbers 1, 2, and 3 correspond to the block devices: a virtual disk, a floppy disk, and a hard disk. All other items in the block device array are set to `NULL` by default.

Table 9-2 The major device no and related operation functions

Main Dev No	Type	Description	Request function
0	None	None	NULL
1	Block/Char	ram, memory dev (ramdisk etc)	<code>do_rd_request()</code>
2	Block	fd, floppy disk device	<code>do_fd_request()</code>
3	Block	hd, hardware disk device	<code>do_hd_request()</code>
4	Char	ttyx device (virtual or serial terminal etc)	NULL
5	Char	tty device	NULL
6	Char	lp printing device	NULL

When the kernel issues a block device read or write or other operation request, the `ll_rw_block()` function will create a device request item using the operation function `do_XX_request()` according to the command specified in its parameter and the device no in the data buffer block header, and inserts it into the request queue using elevator algorithm. The 'XX' in the function name can be one of 'rd', 'fd' or 'hd', representing memory, floppy disk and hard disk block devices. The request item queue consists of items in the request item table (array). There are a total of 32 items. The data structure of each request item is as follows:

---

```

struct request {
    int dev;                // The device no used (-1 means empty).
    int cmd;                // Command (READ or WRITE).
    int errors;             // The nr of errors during operation.
    unsigned long sector;   // Starting sector. (1 block = 2 sectors)
    unsigned long nr_sectors; // Read/write sector number.
    char * buffer;          // Data buffer.
    struct task_struct * waiting; // The place where the task waits for operation.
    struct buffer_head * bh; // Buffer header (include/linux/fs.h, 68).
    struct request * next;   // next request item.
};
extern struct request request[NR_REQUEST]; // Request item array (NR_REQUEST = 32).
```

---

The current request item pointer for each block device, together with the request link list for the device in the request array, constitutes the request queue for the device. Between the items, the next pointer field is used to form a linked list. Therefore, the block device item and the associated request queue form the structure shown in Figure 9-1. The main reason why the request item adopts the array and linked list structure is to satisfy two purposes: First, the array structure of the request item can be used to perform loop operations when searching

for idle request blocks. The search access time complexity is constant, so the program can be written very concisely; secondly, in order to satisfy the elevator algorithm insertion request item operation, it is also necessary to adopt Linked list structure. As shown in Figure9-1, the hard disk device currently has four request items, and the floppy disk device has only one request item, and the virtual disk device does not currently have a read/write request item.

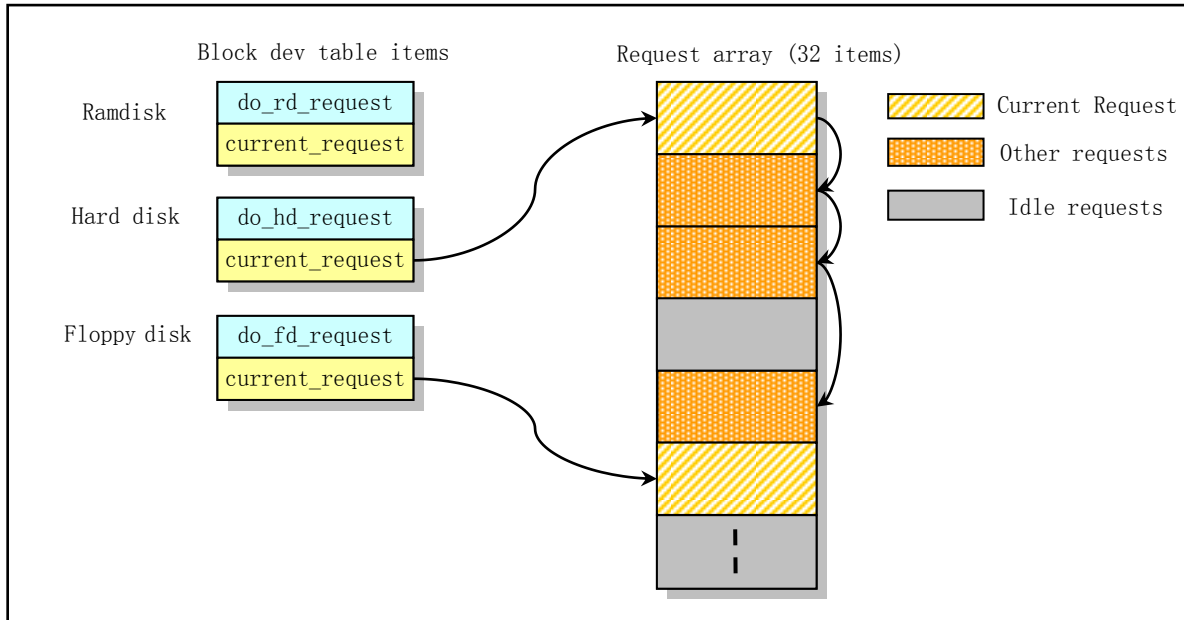


Figure 9-1 Device table entry and request item

For a currently idle block device, when the `ll_rw_block()` function establishes the first request item for it, the current request item pointer `current_request` directly points to the newly created request item, and immediately invokes the request function to start the block device read and write operations. When a block device already has a linked list of several request items, `ll_rw_block()` will use the elevator algorithm to insert the newly created request into the appropriate position of the linked list according to the principle of minimum head movement distance.

In addition, in order to satisfy the priority of the read operation, when searching for the array of request items for establishing a new request item, the search range of the free item for write operation is limited to the first 2/3 of the entire request array, and the remain 1/3 request items is specifically used for the read operation establishment request.

### 9.1.2 Block device access scheduling

Accessing data in blocks such as hard disks and floppy disks is a time consuming operation and affects system performance compared to accessing memory operations. Since the hard disk head seek operation (that is, moving the read/write head from one track to another designated track) takes a long time, it is necessary to sort the order of accessing the disk sectors before sending the operation command to the hard disk controller. That is, the order of each request item in the request linked list is sorted, so that all the disk sector blocks accessed by the request items are operated in order. In the Linux 0.12 kernel, the request items are sorted using the elevator algorithm. The principle of operation is similar to the movement of the elevator -- moving in one direction until the last "request" stop layer in that direction. Then perform the opposite direction of movement. For the disk, the head moves all the way to the center of the disc, or vice versa. See Figure 2-11 for the structure of the hard



disk.

Therefore, the request items are not directly sent to the block device for processing in the received order, but the order of the request items needs to be optimized first. We usually refer to the relevant handler as the I/O scheduler. The I/O scheduler in Linux 0.12 only sorts the request items, and the current popular Linux kernel (such as 2.6.x) I/O scheduler also contains two access to adjacent disk sectors or Multiple request items are merged.

### 9.1.3 Block device operation method

When performing IO access operations between the system (kernel) and the hard disk, you need to consider the interaction between three objects. They are system, controller, and drive (such as hard drive or floppy drive), as shown in Figure 9-2. The system can send commands directly to the controller or wait for the controller to issue an interrupt request; after receiving the command, the controller will control the drive to perform related operations, read/write data or perform other operations. Therefore, we can regard the interrupt signal sent by the controller here as the synchronous operation signal between the three, and the operation steps are as follows:

- First, the system indicates the C function that the controller should invoke during the interrupt caused by the execution of the command, and then sends a read, write, reset or other operation command to the block device controller;
- When the controller completes the specified command, it will issue an interrupt request signal, which will cause the system to execute the interrupt processing of the block device, and call the specified C function to post-process the read/write or other commands after these commands are finished.

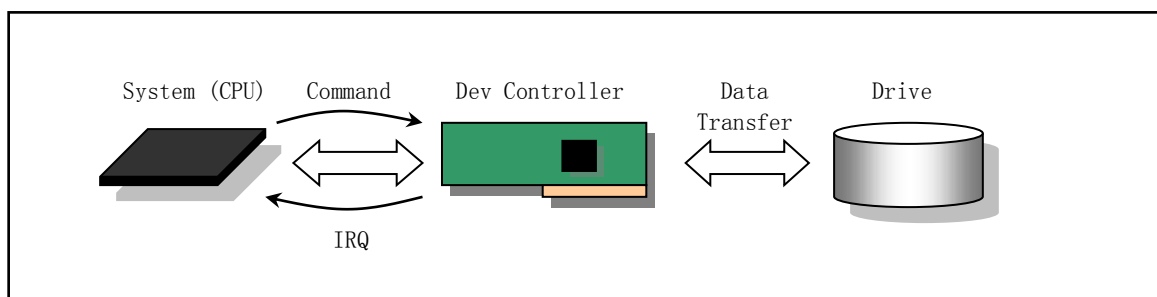


Figure 9-2 System, block device controllers, and drives

For disk write operations, the system needs to wait for the controller to give a response to allow data to be written to the controller after issuing a write command (using `hd_out()`), ie, waiting for the data request service flag DRQ of the controller status register to be set. Once DRQ is set, the system can send a sector of data to the controller buffer.

When all data is written to the drive (or an error occurs), the controller also generates an interrupt request signal to execute the pre-set C function (`write_intr()`) during interrupt processing. This function will check if there is still data to write. If so, the system will transfer the data of one sector to the controller buffer, and then wait for the interrupt generated by the controller to write the data to the drive again, and repeat this. If all the data has been written to the drive at this time, the C function performs the finishing work after the end of this write: Wake up the relevant process waiting for the request item data, wake up the process waiting for the request item, release the current request item and delete the request item from the linked list and release the locked buffer. Finally, the request item operation function is called to execute the next read/write disk request

item (if any).

For the disk read operation, the system waits for the controller to generate an interrupt signal after transmitting a command including information such as the start position of the sector, the number of sectors, and the like to the controller. When the controller passes the specified sector data from the drive to its own buffer as required by the read command, an interrupt request is issued. This will execute the C function (read\_intr()) that was previously set for the read operation. This function first puts the data of one sector in the controller buffer into the buffer of the system, and then decrements the number of sectors to be read. If there is still data to read (the decrement result value is not 0), continue to wait for the controller to send the next interrupt signal. If all the required sectors have been read into the system buffer at this time, the same end operation as the above write operation is performed.

For a virtual disk device, since its read and write operations do not involve a synchronous operation with an external device, there is no interrupt processing described above. The read and write operations of the current request item to the virtual device are completely implemented in do\_rd\_request().

One thing to be reminded is that after sending a read/write or other command to the hard disk or floppy disk controller, the function that sends the command does not wait for the execution of the issued command, but immediately returns to the program that called it. And finally return to the other program that invokes the block device function ll\_rw\_block () to wait for the completion of the block device IO. For example, the read block device function bread() in the cache manager (fs/buffer.c line 267), after calling ll\_rw\_block(), it invokes the wait function wait\_on\_buffer() to let the process go to sleep immediately. The state is awakened in the end\_request() function until the end of the associated block device IO.

## 9.2 blk.h

### 9.2.1 Function Description

This is the header file for block device parameters such as hard disks. Because it is only used for block devices, it is placed in the same place as the block device source file. The data structure request of the request item in the request queue is mainly defined; the elevator search algorithm is defined by the macro statement. For the virtual disk, floppy disk and hard disk, the corresponding constant values are defined according to their respective major device numbers.

### 9.2.2 Code annotation

---

Program 9-1 linux/kernel/blk\_drv/blk.h

---

```
1 #ifndef BLK\_H
2 #define BLK\_H
3
4 #define NR\_BLK\_DEV      7           // nr of device types
5 /*
6  * NR_REQUEST is the number of entries in the request-queue.
7  * NOTE that writes may use only the low 2/3 of these: reads
8  * take precedence.
9  *
10 * 32 seems to be a reasonable number: enough to get some benefit
```

```

11  * from the elevator-mechanism, but not so much as to lock a lot of
12  * buffers when they are in the queue. 64 seems to be too many (easily
13  * long pauses in reading when heavy writing/syncing is going on)
14  */
15 #define NR_REQUEST      32
16
17 /*
18  * Ok, this is an expanded form so that we can use the same
19  * request for paging requests when that is implemented. In
20  * paging, 'bh' is NULL, and 'waiting' is used to wait for
21  * read/write completion.
22  */
    // Below is the structure of the request item in the request queue. If the field dev = -1, it
    // means that the item in the queue is not used. The field cmd can be a constant of READ(0)
    // or WRITE(1) (defined in include/linux/fs.h). In addition, the kernel does not use the waiting
    // pointer. Instead, the kernel uses the wait queue of the buffer block. Because waiting for
    // a buffer block is equivalent to waiting for the completion of the request.
23 struct request {
24     int dev;                /* -1 if no request */ // device requested.
25     int cmd;                /* READ or WRITE */
26     int errors;             // count of error.
27     unsigned long sector;   // start sector no.
28     unsigned long nr_sectors; // nr sectors needed.
29     char * buffer;          // data buffer.
30     struct task_struct * waiting; // waiting queue of tasks.
31     struct buffer_head * bh; // Buffer header (include/linux/fs.h, 73).
32     struct request * next;   // points to the next request.
33 };
34
35 /*
36  * This is used in the elevator algorithm: Note that
37  * reads always go before writes. This is natural: reads
38  * are much more time-critical than writes.
39  */
    // The values of the parameters s1 and s2 in the macro below are pointers to the request structure.
    // The macro is used to determine the order of the two request item s1 and s2 in the request
    // queue based on the information in them (command cmd (READ or WRITE), device number dev, and
    // sector number sector being operated). This order will be used as the order in which the
    // request items are executed when accessing the block device. This macro will be used in the
    // function add_request() (blk_drv/ll_rw_blk.c, line 96). This macro partially implements the
    // I/O scheduling function, which implements the sorting function of the request item (the other
    // is request item merging).
40 #define IN_ORDER(s1,s2) \
41 ((s1)->cmd < (s2)->cmd || (s1)->cmd == (s2)->cmd && \
42 ((s1)->dev < (s2)->dev || ((s1)->dev == (s2)->dev && \
43 (s1)->sector < (s2)->sector)))
44
    // Block device structure.
45 struct blk_dev_struct {
46     void (*request_fn)(void); // request handling function.
47     struct request * current_request; // current request item.
48 };
49

```

```

// Block device table (array). Each block device occupies one item, a total of 7 items. The
// index of the array is the major device number. The next is an array of request item structures,
// a total of 32 items. wait_for_request is a process queue header waiting for an idle request.
50 extern struct blk_dev_struct blk_dev[NR_BLK_DEV];
51 extern struct request request[NR_REQUEST];
52 extern struct task_struct * wait_for_request;
53
// An array of pointers to the total number of device data blocks. Each pointer entry points
// to a total number of blocks array hd_sizes[] (blk_drv/hd.c, line 62) of the specified major
// device. Each item of the array corresponds to the total number of data blocks owned by one
// minor device (1 block size = 1 KB).
54 extern int * blk_size[NR_BLK_DEV];
55
// In a block device driver (such as hd.c) that contains this header file, you must first define
// the major device number of the device being processed. Thus, the correct macro definition
// can be given to the driver in the following 63-90 lines.
56 #ifdef MAJOR_NR // current major number used.
57
58 /*
59  * Add entries as needed. Currently the only block devices
60  * supported are hard-disks and floppies.
61  */
62
// If device is RAM disk, the following constants and macros are used.
63 #if (MAJOR_NR == 1)
64 /* ram disk */
65 #define DEVICE_NAME "ramdisk"
66 #define DEVICE_REQUEST do_rd_request // request handler.
67 #define DEVICE_NR(device) ((device) & 7) // Sub-device nr (0 - 7).
68 #define DEVICE_ON(device) // Turn on (ram disk doesn't need this).
69 #define DEVICE_OFF(device)
70
// If device is floppy driver, the following constants and macros are used.
71 #elif (MAJOR_NR == 2)
72 /* floppy */
73 #define DEVICE_NAME "floppy"
74 #define DEVICE_INTR do_floppy // device interrupt handler.
75 #define DEVICE_REQUEST do_fd_request // request handler.
76 #define DEVICE_NR(device) ((device) & 3) // Sub-device nr (0 - 3).
77 #define DEVICE_ON(device) floppy_on(DEVICE_NR(device))
78 #define DEVICE_OFF(device) floppy_off(DEVICE_NR(device))
79
// If device is hard disk, the following constants and macros are used.
80 #elif (MAJOR_NR == 3)
81 /* harddisk */
82 #define DEVICE_NAME "harddisk"
83 #define DEVICE_INTR do_hd // device interrupt handler.
84 #define DEVICE_TIMEOUT hd_timeout // device timeout.
85 #define DEVICE_REQUEST do_hd_request // request handler.
86 #define DEVICE_NR(device) (MINOR(device)/5) // harddisk no (0,1).
87 #define DEVICE_ON(device) // the harddisk always running.
88 #define DEVICE_OFF(device)
89

```

```

90 #elif
91 /* unknown blk device */
92 #error "unknown blk device"
93
94 #endif
95
96 // For ease of programming, two macros are defined here: CURRENT is the current request item
97 // pointer, and CURRENT_DEV is the device number in the current request item CURRENT.
98 #define CURRENT (blk_dev[MAJOR_NR].current_request)
99 #define CURRENT_DEV DEVICE_NR(CURRENT->dev)
100
101 // If a device interrupt handling symbol is defined, it is declared as a function pointer and
102 // defaults to NULL.
103 #ifdef DEVICE_INTR
104 void (*DEVICE_INTR)(void) = NULL;
105 #endif
106
107 // If a device timeout symbol is defined, the same variable is defined with a value equal to
108 // 0, and the SET_INTR() macro is defined.
109 #ifdef DEVICE_TIMEOUT
110 int DEVICE_TIMEOUT = 0;
111 #define SET_INTR(x) (DEVICE_INTR = (x), DEVICE_TIMEOUT = 200)
112 #else
113 #define SET_INTR(x) (DEVICE_INTR = (x))
114 #endif
115
116 // Declare symbol DEVICE_REQUEST is a static function pointer with no arguments & no return.
117 static void (DEVICE_REQUEST)(void);
118
119 // Unlock the specified buffer block. If the specified buffer block bh is not locked, a warning
120 // message is displayed. Otherwise the buffer block is unlocked and the process waiting for
121 // the buffer block is woken up. This is an inline function, but is used as a "macro". The argument
122 // is the buffer block header pointer.
123 extern inline void unlock_buffer(struct buffer_head * bh)
124 {
125     if (!bh->b_lock)
126         printk(DEVICE_NAME " : free buffer being unlocked\n");
127     bh->b_lock=0;
128     wake_up(&bh->b_wait);
129 }
130
131 // End request processing "macro". The parameter uptodate is the update flag.
132 // First turn off the specified block device and then check if the read/write buffers are valid.
133 // If valid, the buffer data update flag is set according to the parameter value and the buffer
134 // is unlocked. If the parameter uptodate value is 0, it indicates that the operation of the
135 // request item has failed, so the related block device IO error message is displayed. Finally,
136 // the processes waiting for the request item and the processes waiting for the idle request
137 // item are awake, and the request item is released and deleted from the request list, and the
138 // current request item pointer is pointed to the next request.
139 extern inline void end_request(int uptodate)
140 {
141     DEVICE_OFF(CURRENT->dev); // turn off the device.
142     if (CURRENT->bh) { // the current request item pointer.
143         CURRENT->bh->b_uptodate = uptodate; // set flag.
144         unlock_buffer(CURRENT->bh);
145     }
146 }

```

```

124     }
125     if (!uptodate) {                                     // uptodate flag not set...
126         printk(DEVICE_NAME " I/O error\n\r");
127         printk("dev %04x, block %d\n\r", CURRENT->dev,
128             CURRENT->bh->b_blocknr);
129     }
130     wake_up(&CURRENT->waiting);                          // Wake up process waiting for the request.
131     wake_up(&wait_for_request);                          // Wake up process waiting for idle request.
132     CURRENT->dev = -1;                                    // Release the request item.
133     CURRENT = CURRENT->next;                              // Point to the next request item.
134 }
135
136 // If the device timeout symbol DEVICE_TIMEOUT is defined, the CLEAR_DEVICE_TIMEOUT symbol is
137 // defined as "DEVICE_TIMEOUT = 0". Otherwise only define CLEAR_DEVICE_TIMEOUT.
138 #ifdef DEVICE_TIMEOUT
139 #define CLEAR_DEVICE_TIMEOUT DEVICE_TIMEOUT = 0;
140 #else
141 #define CLEAR_DEVICE_TIMEOUT
142 #endif
143
144 // If the device interrupt symbol DEVICE_INTR is defined, the CLEAR_DEVICE_INTR symbol is defined
145 // as "DEVICE_INTR = 0", otherwise it is defined as empty.
146 #ifdef DEVICE_INTR
147 #define CLEAR_DEVICE_INTR DEVICE_INTR = 0;
148 #else
149 #define CLEAR_DEVICE_INTR
150 #endif
151
152 // A macro that defines the initialization of a request item. Since the initialization of the
153 // request items at the beginning of the block device driver is similar, a uniform initialization
154 // macro is defined for them. This macro is used to make some validity judgments on the current
155 // request item. The work done is as follows:
156 // If the current request item of the device is empty (NULL), it means that the device has no
157 // request items yet to be processed. Then the work is skipped and the corresponding function
158 // is exited. Otherwise, if the major device number in the current request item is not equal
159 // to the major number defined by the driver, the request item queue is garbled, and the kernel
160 // displays an error message and stops. Otherwise, if the buffer block used in the request item
161 // is not locked, it also indicates that there is a problem with the kernel code, so an error
162 // message is displayed and the machine is stopped too.
163 #define INIT_REQUEST \
164 repeat: \
165     if (!CURRENT) { \                                     // no more requests need to processed.
166         CLEAR_DEVICE_INTR \
167         CLEAR_DEVICE_TIMEOUT \
168         return; \
169     } \
170     if (MAJOR(CURRENT->dev) != MAJOR_NR) \                // major nr error.
171         panic(DEVICE_NAME ": request list destroyed"); \
172     if (CURRENT->bh) { \
173         if (!CURRENT->bh->b_lock) \                        // buffer error.
174             panic(DEVICE_NAME ": block not locked"); \
175     }

```

```
162 #endif  
163  
164 #endif  
165
```

---

## 9.3 hd.c

### 9.3.1 Function description

The hd.c program is the hard disk controller driver. It provides read and write operations to the hard disk controller and block devices, as well as hard disk initialization processing. All functions in the program can be divided into five categories according to their functions:

- Functions that initialize the hard disk and set the data structure information used by the hard disk, such as `sys_setup()` and `hd_init()`;
- The function `hd_out()` that sends the command to the hard disk controller;
- The function `do_hd_request()` that handles the current request item of the hard disk;
- C functions called during hard disk interrupt handling, such as `read_intr()`, `write_intr()`, `bad_rw_intr()`, and `recal_intr()`. The `do_hd_request()` function will also be called in `read_intr()` and `write_intr()`;
- The hard disk controller operates auxiliary functions such as `controller_ready()`, `drive_busy()`, `win_result()`, `hd_out()`, and `reset_controller()`.

The `sys_setup()` function uses the information provided by the `boot/setup.s` to set the parameters of the hard drive included in the system. Then read the hard disk partition table and try to copy the root fs image on the boot disk to the memory virtual disk. If successful, the root file system in the virtual disk is loaded, otherwise the normal root file system loading operation is continued.

The `hd_init()` function is used to set the hard disk controller interrupt descriptor during kernel initialization and reset the hard disk controller interrupt mask to allow the hard disk controller to send an interrupt request signal.

`Hd_out()` is the hard disk controller operation command send function. This function takes a C function pointer parameter that is called during an interrupt. Before sending a command to the controller, it first uses this parameter to preset the function pointer (do\_hd, such as `read_intr()`) that is called during the interrupt. It then sends the command parameter block to the hard disk controller (ports 0x1f0 to 0x1f7) in the prescribed manner. The function returns immediately and does not wait for the hard disk controller to execute the read/write commands. In addition to the controller diagnostics (WIN\_DIAGNOSE) and the establishment of the drive parameters (WIN\_SPECIFY), the hard disk controller will send an interrupt request signal to the CPU after receiving any other commands and executing the commands. This causes the system to perform the hard disk interrupt processing (in `system_call.s`, line 235).

`do_hd_request()` is the operation function of the hard disk request item. The operation process is as follows:

- (1) First determine if the current request item exists. If the current request pointer is empty, it indicates that the current hard disk block device has no pending request items, so the program is immediately exited. This is the statement executed in the macro `INIT_REQUEST`. Otherwise continue processing the current request item.
- (2) verifying the reasonableness of the device number specified in the current request item and the

- requested disk start sector number;
- (3) calculating the disk track number, head number and cylinder number of the request data according to the information provided by the current request item;
  - (4) If the reset flag has been set, the hard disk recalibration flag is also set, and the hard disk is reset, and the "create drive parameter" command (WIN\_SPECIFY) is resent to the controller. This command does not cause a hard disk interrupt;
  - (5) If the recalibration flag is set, send the hard disk recalibration command (WIN\_RESTORE) to the controller, and pre-set the C function (recal\_intr()) that needs to be executed in the interrupt caused by the command, and drop out. The main function of the recal\_intr() is to re-execute this function when the controller terminates the command and raises an interrupt.
  - (6) If the current request item specifies a write operation, first set the C function to write\_intr() to send a command parameter block of the write operation to the controller. The controller's status register is queried cyclically to determine if the request service flag (DRQ) is set. If the flag is set, it indicates that the controller has "agreed" to receive the data, and then the data in the buffer pointed to by the request item is written into the data buffer of the controller. If the flag is still not set after the loop query times out, the operation failed. The bad\_rw\_intr() function is then called, and it is determined whether to abandon the current request item or to set a reset flag according to the number of occurrences of the error to continue reprocessing the current request item.
  - (7) If the current request item is a read operation, set the C function to read\_intr(), and send a read operation command to the controller.

write\_intr() is a C function that is set to be called during the interrupt when the current request item is a write operation. When the controller completes the write command, it will immediately send an interrupt request signal to the CPU, so the function will be called immediately after the controller write operation is completed.

The function first calls the win\_result(), which reads the controller's status register to determine if an error has occurred. If an error occurs during the write operation, bad\_rw\_intr() is called, and it is determined whether to abort to continue processing the current request according to the number of errors occurring in processing the current request, or whether a reset flag needs to be set to continue to reprocess the current request item. If no error occurs, it is determined whether all the data required has been written to the disk according to the total number of sectors to be written indicated in the current request item. If there is still data to write to the disk, use the port\_write() function to copy the data of one sector to the controller buffer. If the data has all been written, the end\_request() is called to handle the end of the current request: Wake up the process waiting for the completion of the request, wake up the process waiting for the idle request item (if any), set the buffer data updated flag by the current request item, and release the current request (remove the item from the block device list) . Finally, continue to call the do\_hd\_request() function to continue processing other request items on the hard disk device.

read\_intr() is a C function that is set to be called during an interrupt when the current request item is a read operation. After the controller reads the specified sector data from the hard disk drive into its own buffer, it immediately sends an interrupt request signal. The main purpose of this function is to copy the data in the controller to the buffer specified by the current request.

The same as when write\_intr() starts, the function first calls the win\_result() to read the controller's status register to determine if an error has occurred. If an error occurs while reading the disk, the same processing as write\_intr() is performed. If no errors have occurred, use the port\_read() function to copy the data of one sector from the controller buffer to the buffer specified by the request. Then, according to the total number of sectors



The diagram illustrates the process of reading data from a hard disk, showing the interaction between the Program, Controller, and Interrupt handler over time. The vertical axis represents time, with events numbered 1 through 7.

**Legend:**

- Processing Request
- ▨ Interrupt handler
- ▨ Controller processing
- Write Command
- ← Read Status
- ← Transfer Data

**Sequence of Events:**

- ①** The Program issues a read request (red bar) and sends a write command (magenta arrow) to the Controller.
- ②** The Controller processes the request (green hatched bar).
- ③** The Interrupt handler reads the controller status (blue hatched bar) and receives a read status signal (brown arrow) from the Controller.
- ④** The Interrupt handler transfers data (green arrow) from the Controller to the Program.
- ⑤** The Controller processes the next sector data (green hatched bar).
- ⑥** The Interrupt handler reads the controller status (blue hatched bar) and receives a read status signal (brown arrow) from the Controller.
- ⑦** The Interrupt handler transfers data (green arrow) from the Controller to the Program.

453

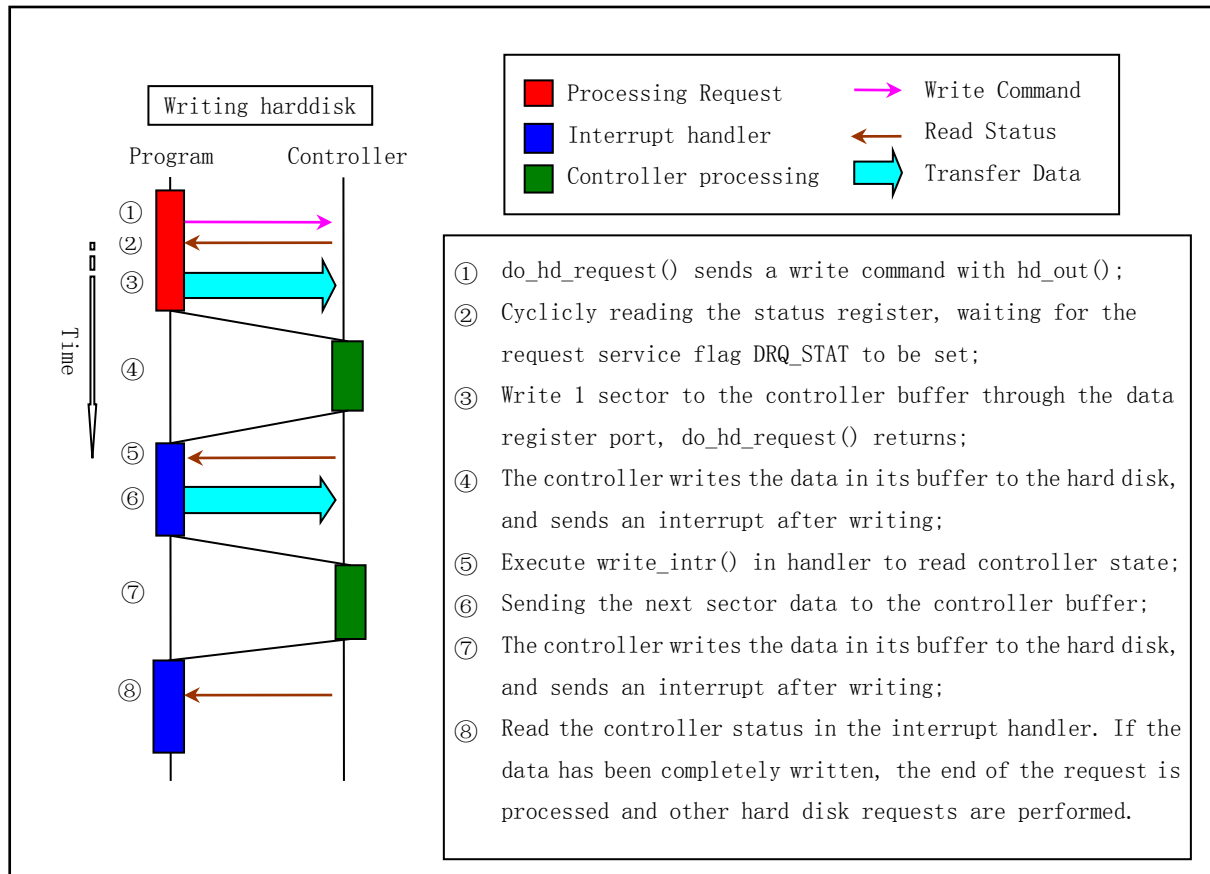


Figure 9-4 Timing relationship of hard disk write data operations

As can be seen from the above analysis, the four most important functions in this program are `hd_out()`, `do_hd_request()`, `read_intr()`, and `write_intr()`. Understand the role of these four functions and understand the operation process of the hard disk drive.

It is worth noting again that after using `hd_out()` to send a read/write or other command to the hard disk controller, the `hd_out()` function does not wait for the execution of the issued command, but immediately returns to the program that called it, for example `do_hd_request()`. The `do_hd_request()` function will immediately return to the function that called it (`add_request()`), and finally return to the other program that calls the block device read/write function `ll_rw_block()` (for example, the `bread()` function in `fs/buffer.c`), waiting for the completion of the block device IO.

### 9.3.2 Code annotation

Program 9-2 linux/kernel/blk\_drv/hd.c

```

1  /*
2  *  linux/kernel/hd.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  This is the low-level hd interrupt support. It traverses the
9  *  request-list, using interrupts to jump between functions. As
10 *  all the functions are called within interrupts, we may not

```

```

11 * sleep. Special care is recommended.
12 *
13 * modified by Drew Eckhardt to check nr of hd's from the CMOS.
14 */
15
// <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk
// type (HD_TYPE) options.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
// of the initial task 0, and some embedded assembly function macro statements about the
// descriptor parameter settings and acquisition.
// <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,
// m_inode, etc.).
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
// used functions of the kernel.
// <linux/hdreg.h> Hard disk parameter header file. Define access to the hard disk register port,
// status code, partition table and other information.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
// descriptors/interrupt gates, etc. is defined.
// <asm/io.h> Io header file. Defines the function that operates on the io port in the form of
// a macro's embedded assembler.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
// segment register operations.
16 #include <linux/config.h>
17 #include <linux/sched.h>
18 #include <linux/fs.h>
19 #include <linux/kernel.h>
20 #include <linux/hdreg.h>
21 #include <asm/system.h>
22 #include <asm/io.h>
23 #include <asm/segment.h>
24
// Define the hard disk major number symbol constant. In the driver, the major device number
// must be defined before the blk.h file is included. This symbol is used in the blk.h file
// to determine some other symbol constants and macros associated with it.
25 #define MAJOR_NR 3 // harddisk major no is 3.
26 #include "blk.h"
27
// Read the CMOS parameter macro function. This macro reads the hard drive information in CMOS.
// outb_p, inb_p are port input and output macros defined in include/asm/io.h. It is exactly
// the same macro used to read CMOS clock information in init/main.c.
28 #define CMOS_READ(addr) ({ \
29 outb_p(0x80|addr,0x70); \ // 0x70 is write port, 0x80|addr is CMOS addr.
30 inb_p(0x71); \ // 0x71 is read port.
31 })
32
33 /* Max read/write errors/sector */
34 #define MAX_ERRORS 7
35 #define MAX_HD 2 // nr of hard disks supported by the system.
36
// Recalibration. The recalibration function (line 311) called in the hard disk interrupt
// handler during reset operation.
37 static void recal_intr(void);
// A function that handles hard disk read and write failures.

```

```

// End the processing of this request item, or set the reset flag to perform the hard disk
// controller reset operation and then try again (242 lines).
38 static void bad\_rw\_intr(void);
39
// Recalibration flag. When this flag is set, recal_intr() is called in the program to move
// the head to the 0 cylinder.
40 static int recalibrate = 0;
// Reset flag. This flag is set when a read or write error occurs and the associated reset function
// is called to reset the hard disk and controller.
41 static int reset = 0;
42
43 /*
44  * This struct defines the HD's and their types.
45  */
// Harddisk information struct.
// The fields are: the nr of heads, the nr of sectors per track, the nr of cylinders, the
// pre-compensation cylinder nr before writing, the cylinder landing nr of the head, and the
// control byte. See the instructions following the program list for their meaning.
46 struct hd\_i\_struct {
47     int head, sect, cyl, wpcom, lzone, ctl;
48 };

// If the symbol constant HD_TYPE has been defined in the include/linux/config.h file, the
// parameter defined therein is taken as the data in the hard disk information array hd_info[].
// Otherwise, the default value is set to 0 first, and will be reset in the setup() function.
49 #ifdef HD_TYPE
50 struct hd\_i\_struct hd\_info[] = { HD_TYPE };
51 #define NR\_HD ((sizeof (hd\_info))/(sizeof (struct hd\_i\_struct))) // count hd nr.
52 #else
53 struct hd\_i\_struct hd\_info[] = { {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0} };
54 static int NR\_HD = 0;
55 #endif
56
// Define the hard disk partition structure. Give the physical start sector number and the total
// number of partition sectors for each partition from the 0 track of the hard disk. Items in
// multiples of 5 (such as hd[0] and hd[5], etc.) represent parameters of the entire hard disk.
57 static struct hd\_struct {
58     long start_sect; // The partition start sector in the hard disk.
59     long nr_sects; // The total nr of sectors in the partition.
60 } hd[5*MAX\_HD]={{0, 0},};
61
// An array of the total number of data blocks in each partition of the hard disk.
62 static int hd\_sizes[5*MAX\_HD] = {0, };
63
// Read port inline assembly macros. Read the port, read nr word, and save it in buf.
64 #define port\_read(port, buf, nr) \
65 __asm__ ("cld;rep;insw"::"d" (port), "D" (buf), "c" (nr):"cx", "di")
66
// Write port inline macros. Write port, write nr words, take data from buf.
67 #define port\_write(port, buf, nr) \
68 __asm__ ("cld;rep;outsw"::"d" (port), "S" (buf), "c" (nr):"cx", "si")
69

```

```

70 extern void hd\_interrupt(void);           // Harddisk interrupt handler (sys_call.s, line 235).
71 extern void rd\_load(void);                 // The ram disk load function (ramdisk.c, line 71).
72
73 /* This may be used only once, enforced by 'static int callable' */
// System setup syscall function.
// The function parameter BIOS is set by the init subroutine in the initialization program
// init/main.c to point to the hard disk parameter table. The hard disk parameter table structure
// contains the contents of two hard disk parameter tables (32 bytes in total), which are copied
// from the memory 0x90080. The information at 0x90080 is obtained by the setup.s program using
// the ROM BIOS function. For a description of the hard disk parameter table, see Table 6-4
// in Section 6.3.3. The main purpose of this function is to read the CMOS hard disk information,
// used to set the hard disk partition structure hd, and try to load the RAM virtual disk and
// the root file system.
74 int sys\_setup(void * BIOS)
75 {
76     static int callable = 1;                // used to limit invocation only once.
77     int i, drive;
78     unsigned char cmos_disks;
79     struct partition *p;
80     struct buffer head * bh;
81
// First set the callable flag so that this function can only be called once. Then set the hard
// disk information array hd_info[]. If the symbol HD_TYPE is already defined in the
// include/linux/config.h file, it means that the hd_info[] array has been set on line 49 above.
// Otherwise, you need to read the hard disk parameter table at memory 0x90080. The setup.s
// program will store one or two hard disk parameter tables in the memory here.
82     if (!callable)
83         return -1;
84     callable = 0;
85 #ifndef HD_TYPE                                // Read if HD_TYPE is not defined.
86     for (drive=0 ; drive<2 ; drive++) {
87         hd\_info[drive].cyl = *(unsigned short *) BIOS;        // cylinders.
88         hd\_info[drive].head = *(unsigned char *) (2+BIOS);    // headers.
89         hd\_info[drive].wpcom = *(unsigned short *) (5+BIOS);  // write pre-com.
90         hd\_info[drive].ctl = *(unsigned char *) (8+BIOS);     // control byte.
91         hd\_info[drive].lzone = *(unsigned short *) (12+BIOS); // landing zone.
92         hd\_info[drive].sect = *(unsigned char *) (14+BIOS);   // sectors/track.
93         BIOS += 16;                                         // Each hd parameter table is 16 bytes long.
94     }
// When the setup.s program takes the BIOS hard disk parameter table information, if there is
// only one hard disk in the system, the 16 bytes corresponding to the second hard disk will
// be cleared. Therefore, if you check whether the number of the second hard disk cylinder is
// 0, you can know if there is a second hard disk.
95     if (hd\_info[1].cyl)
96         NR\_HD=2;                                           // The nr of hard disks is set to 2.
97     else
98         NR\_HD=1;
99 #endif

// At this point, the hard disk information array hd_info[] has been set, and the number of
// hard disks NR_HD is determined. Now set up the hard disk partition structure array hd[]. Items
// 0 and 5 of the array represent the overall parameters of the two hard disks, while items
// 1-4 and 6-9 represent the parameters of the four partitions of the two hard disks,

```

```

// respectively. Only two items (items 0 and 5) indicating the overall information of the hard
// disk are set here.
100     for (i=0 ; i<NR_HD ; i++) {
101         hd[i*5].start_sect = 0;                                // starting sector number.
102         hd[i*5].nr_sects = hd_info[i].head*
103                             hd_info[i].sect*hd_info[i].cyl; // total nr of sectors.
104     }
105
106     /*
107         We query CMOS about hard disks : it could be that
108         we have a SCSI/ESDI/etc controller that is BIOS
109         compatable with ST-506, and thus showing up in our
110         BIOS table, but not register compatable, and therefore
111         not present in CMOS.
112
113         Furthurmore, we will assume that our ST-506 drives
114         <if any> are the primary drives in the system, and
115         the ones reflected as drive 1 or 2.
116
117         The first drive is stored in the high nibble of CMOS
118         byte 0x12, the second in the low nibble. This will be
119         either a 4 bit drive type or 0xf indicating use byte 0x19
120         for an 8 bit type, drive 1, 0x1a for drive 2 in CMOS.
121
122         Needless to say, a non-zero value means we have
123         an AT controller hard disk for that drive.
124
125     */
126
127     // According to the above principle, here is to check whether the hard disk is compatible with
128     // the AT controller. See Section 7.1.3 for a description of CMOS information. Here, the hard
129     // disk type byte is read from the CMOS offset address 0x12. If the low nibble value (the second
130     // hard disk type) is not 0, it means that the system has two hard disks, otherwise the system
131     // has only one hard disk. If the value read from 0x12 is 0, it means there is no AT compatible
132     // hard disk in the system.
133     if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
134         if (cmos_disks & 0x0f)
135             NR_HD = 2;
136         else
137             NR_HD = 1;
138     else
139         NR_HD = 0;
140
141     // If NR_HD = 0, the two hard disks are not compatible with the AT controller, and the data
142     // structures of the two hard disks are all cleared. If NR_HD = 1, the parameter of the second
143     // hard disk is cleared.
144     for (i = NR_HD ; i < 2 ; i++) {
145         hd[i*5].start_sect = 0;
146         hd[i*5].nr_sects = 0;
147     }
148
149     // Ok, so far we have clearly determined the number of hard disks NR_HD contained in the system.
150     // Now let's read the partition table information in the first sector of each hard disk, which
151     // is used to set the information of each partition of the hard disk in the partition structure

```

```

// array hd[]. First, read the first data block of the hard disk (fs/buffer.c, line 267) by
// using the read block function bread(). The first parameter (0x300, 0x305) is the device number
// of the two hard disks, and the second parameter (0) is the block number to be read. If the
// read operation is successful, the data will be stored in the data area of the buffer block
// bh. If the buffer block head pointer bh is 0, it means that the read operation failed, the
// error message is displayed and the machine stops. Otherwise, we judge the validity of the
// data according to whether the last two bytes of the first sector are 0xAA55, so that it can
// be known whether the partition table at the beginning of the offset 0x1BE in the sector is
// valid. If it is valid, the hard disk partition table information is stored in the hard disk
// partition structure array hd[]. Finally release the bh buffer.
139     for (drive=0 ; drive<NR_HD ; drive++) {
140         if (!(bh = bread(0x300 + drive*5, 0))) {           // 0x300, 0x305 is dev no.
141             printk("Unable to read partition table of drive %d\n\r",
142                 drive);
143             panic("");
144         }
145         if (bh->b_data[510] != 0x55 || (unsigned char)
146             bh->b_data[511] != 0xAA) {                     // check flag 0xAA55
147             printk("Bad partition table on drive %d\n\r", drive);
148             panic("");
149         }
150         p = 0x1BE + (void *)bh->b_data; // partition table is located at 0x1BE.
151         for (i=1; i<5; i++, p++) {
152             hd[i+5*drive].start_sect = p->start_sect;
153             hd[i+5*drive].nr_sects = p->nr_sects;
154         }
155         brelse(bh); // Release the buffer.
156     }

// Now count the total number of data blocks in each partition and save it in the hard disk
// partition total data block array hd_sizes[]. Then let the block device item (3) of the block
// size array point to this array.
157     for (i=0 ; i<5*MAX_HD ; i++)
158         hd_sizes[i] = hd[i].nr_sects>>1 ;
159     blk_size[MAJOR_NR] = hd_sizes; // MAJOR_NR = 3

// Now we have finally completed the task of setting the hard disk partition structure array
// hd[]. If a hard disk does exist and its information has been read into its partition table,
// the ok message is displayed. Then try to load the root fs image (blk_drv/ramdisk.c, line
// 71) into the memory ram disk. That is, if the system is provided with a virtual ram disk,
// it is determined whether the boot disk also contains the image data of the root fs. If there
// is (the boot disk is called the integrated disk at this time), try to load and store the
// image into the ram disk, and then modify the root fs device number ROOT_DEV to the device
// number of the ram disk. The switching device is then initialized. Finally install the root
// file system.
160     if (NR_HD)
161         printk("Partition table%s ok. \n\r", (NR_HD>1)? "s": "");
162     rd_load(); // blk_drv/ramdisk.c, line 71.
163     init_swapping(); // mm/swap.c, line 199.
164     mount_root(); // fs/super.c, line 241.
165     return (0);
166 }
167
//// Check and cycle to wait for the hard disk controller to be ready.

```

```

// Read the hard disk controller status register port HD_STATUS (0x1f7), loop to detect if the
// drive ready bit (bit 6) is set, and if the controller busy bit (bit 7) is reset. If the return
// value retries is 0, it means that the time waiting for the controller to idle has timed out
// and an error occurs; if the return value is not 0, the controller returns to the idle state
// during the waiting (loop) time period, OK!
// In fact, we only need to check if the status register busy bit (bit 7) is 1 to determine
// if the controller is busy. Whether the drive is ready (ie, bit 6 is 1) is independent of
// the state of the controller. So we can rewrite the line 172 statement as:
// while (--retries && (inb_p(HD_STATUS)&0x80));
// In addition, since the current PC speed is very fast, we can increase the number of waiting
// cycles, for example, 10 times more!
168 static int controller\_ready(void)
169 {
170     int retries = 100000;
171
172     while (--retries && (inb_p(HD_STATUS)&0xc0)!=0x40);
173     return (retries);
174 }
175
// Checks the status of hard disk after executing a command (win stands for Winchester hd)
// Read the status of the command execution result in the status register. Returning 0 means
// normal; 1 means error. If the execution command is wrong, you need to read the error register
// HD_ERROR (0x1f1).
176 static int win\_result(void)
177 {
178     int i=inb_p(HD_STATUS); // get status.
179
180     if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT))
181         == (READY_STAT | SEEK_STAT))
182         return(0); /* ok */
183     if (i&1) i=inb(HD_ERROR); // if ERR_STAT is set, read HD_ERROR.
184     return (1);
185 }
186
//// Send command block to the hard disk controller.
// Params: drive - harddisk no (0-1); nsect - nr of read/write sectors;
// sect - starting sector; head - head number; cyl - cylinder number;
// cmd - command code (see controller cmd list); intr_addr() - C function pointer.
// After the hard disk controller is ready, the function sets the global function pointer variable
// do_hd to point to the C handler that will be called in the hard disk interrupt handler, and
// then sends the hard disk control byte and the 7-byte parameter command block. The harddisk
// interrupt handler is located at line235 in file kernel/sys_call.s.
// Line 191 defines a register variable __res. This variable will be saved in a register for
// quick access. If you want to specify a register (such as eax), you can write the sentence
// as "register char __res asm("ax");".
187 static void hd\_out(unsigned int drive,unsigned int nsect,unsigned int sect,
188                 unsigned int head,unsigned int cyl,unsigned int cmd,
189                 void (*intr_addr)(void))
190 {
191     register int port asm("dx"); // define a register variable.
192
// First check the validity of the parameters. If the drive number is greater than 1 (only 0,
// 1 is valid) or the head number is greater than 15, the program does not support, stop. Otherwise

```



```

// check and loop waiting for the drive to be ready. If it is not ready after waiting for a
// while, it indicates that the hard disk controller is faulty and also stops.
193     if (drive>1 || head>15)
194         panic("Trying to write bad sector");
195     if (!controller\_ready())
196         panic("HD controller not ready");

// Next, we set the C function pointer do_hd that will be called when the hard disk interrupt
// occurs (the function pointer is defined between lines 56-109 in file blk.h, please pay special
// attention to lines 83-100). Then send a control byte to the hard disk controller cmd port
// (0x3f6) to establish the control mode of the specified drive. This control byte is the ctl
// field in the array of hard disk information structures. After that, a 7-byte parameter command
// block is sent to controller port 0x1f1-0x1f7.
197     SET\_INTR(intr_addr);                // do_hd = intr_addr
198     outb\_p(hd\_info[drive].ctl, HD\_CMD);    // out control byte.
199     port=HD\_DATA;                        // (0x1f0)
200     outb\_p(hd\_info[drive].wpcom>>2, ++port); // Param: write pre-comp. (divided by 4)
201     outb\_p(nsect, ++port);                // Param: total nr of r/w sectors.
202     outb\_p(sect, ++port);                 // Param: starting sector.
203     outb\_p(cyl, ++port);                  // Param: The cylinder nr low 8 bits.
204     outb\_p(cyl>>8, ++port);               // Param: The cylinder nr high 8 bits.
205     outb\_p(0xA0 | (drive<<4) | head, ++port); // Param: drive no + head no.
206     outb(cmd, ++port);                   // Param: Hard disk command.
207 }
208
////// Wait for the drive to be ready.
// This function loops to wait for the main status register busy flag to be reset. If the ready
// or seek finished flag is set, the hard disk is ready and if it is successful, it returns
// 0. Returns 1 if it is still busy after a while.
209 static int drive\_busy(void)
210 {
211     unsigned int i;
212     unsigned char c;
213
// The master status register HD_STATUS of the controller is cyclically read, waiting for the
// ready flag bit to be set, and the busy bit is reset. Then detect the busy bit, ready bit,
// and the seek end bit. If only the ready or seek end flag is set, it indicates that the hard
// disk is ready and returns 0. Otherwise, it means that the timeout has expired at the end
// of the loop, so the warning message is displayed and 1 is returned.
214     for (i = 0; i < 50000; i++) {
215         c = inb\_p(HD\_STATUS);                // get main status byte.
216         c &= (BUSY\_STAT | READY\_STAT | SEEK\_STAT);
217         if (c == (READY\_STAT | SEEK\_STAT))
218             return 0;
219     }
220     printk("HD controller times out\n\r");
221     return(1);
222 }
223
////// Diagnostic reset of the hard disk controller.
// The enable reset (4) control byte is first sent to the control register port (0x3f6), and
// then waits for a period of time for the controller to perform a reset operation. The normal
// control byte (allow retry, re-read) is then sent to the port , and wait for the hard disk

```

```

// is ready. If you wait for the hard disk ready timeout, a busy warning message is displayed.
// Then read the contents of the error register. If it is not equal to 1 (1 means no error),
// the hard disk controller reset failure message is displayed.
224 static void reset_controller(void)
225 {
226     int    i;
227
228     outb(4,HD_CMD);           // 4 is reset byte.
229     for(i = 0; i < 1000; i++) nop(); // wait a while.
230     outb(hd_info[0].ctl & 0x0f ,HD_CMD); // normal control byte(allow retry, reread).
231     if (drive_busy())
232         printk("HD-controller still busy\n\r");
233     if ((i = inb(HD_ERROR)) != 1)
234         printk("HD-controller reset failed: %02x\n\r",i);
235 }
236
///// Hard disk reset operation.
// First reset the hard disk controller and then send the hard disk controller command "Create
// Drive Parameters". This function is called again in the hard disk interrupt handler caused
// by this command. At this point, the function will determine whether to perform error processing
// or continue to perform the request item processing based on the result of executing the
// command.
237 static void reset_hd(void)
238 {
239     static int i;
240
// If the reset flag is set, the reset hard disk controller operation is performed after the
// reset flag is cleared. Then send the "Create Drive Parameters" command to the controller
// for the i-th hard disk. When the controller executes the command, it will issue a hard disk
// interrupt signal. At this point, this function will be executed again by the interrupt handler.
// Since the reset flag has been reset at this time, the statement starting at line 246 will
// be executed first to determine whether the command execution is ok. If an error still occurs,
// the bad_rw_intr() function is called to count the number of errors and determine if the reset
// flag is set again based on the error times. If the reset flag is set again, it will jump
// to the label repeat to re-execute this function. If the reset operation is ok, the "Create
// Drive Parameter" command is sent for the next hard disk, and the same processing as described
// above is performed. If all hard disks in the system have executed the sent command normally,
// the do_hd_request() function is called again to start processing the request item.
241 repeat:
242     if (reset) {
243         reset = 0;
244         i = -1; // initialize the current hd number.
245         reset_controller();
246     } else if (win_result()) {
247         bad_rw_intr();
248         if (reset)
249             goto repeat;
250     }
251     i++; // handling next hd.
252     if (i < NR_HD) {
253         hd_out(i, hd_info[i].sect, hd_info[i].sect, hd_info[i].head-1,
254             hd_info[i].cyl, WIN_SPECIFY, &reset_hd);
255     } else

```

```

256         do\_hd\_request();                // request item processing.
257     }
258
259     ///// The default function called by an unexpected hard disk interrupt.
260     // This function is the default C function called in the hard disk interrupt handler when an
261     // unexpected hard disk interrupt occurs. This function is invoked when the called function
262     // pointer is NULL. See kernel/sys_call.s, line 256. The function sets the reset flag after
263     // displaying the warning message, and then continues to invoke the request item function
264     // go_hd_request() and performs a reset processing operation therein.
265     void unexpected\_hd\_interrupt(void)
266     {
267         printk("Unexpected HD interrupt\n\r");
268         reset = 1;
269         do\_hd\_request();
270     }
271
272     ///// The processing function for handling hard disk read/write failure.
273     // If the number of errors in the read sector operation is greater than or equal to 7, the current
274     // request item is terminated and the process waiting for the request is awake, and the
275     // corresponding buffer update flag is reset, indicating that the data is not updated. If the
276     // number of errors in reading/writing a sector operation has been greater than 3 times, it
277     // is required to perform a controller reset operation (set reset flag).
278     static void bad\_rw\_intr(void)
279     {
280         if (++CURRENT->errors >= MAX\_ERRORS)
281             end\_request(0);
282         if (CURRENT->errors > MAX\_ERRORS/2)
283             reset = 1;
284     }
285
286     ///// The read sector function called in the interrupt.
287     // This function will be called during the hard disk interrupt that is triggered at the end
288     // of the hard disk read command. The controller generates an interrupt request signal after
289     // the read command is executed and triggers the execution of the interrupt handler. At this
290     // point, the C function pointer do_hd in interrupt handler has pointed to read_intr(), so the
291     // function will be executed after a read sector operation is completed(or an error).
292     static void read\_intr(void)
293     {
294         // This function first determines if the read command operation is in error. If the controller
295         // is still busy after the command is finished, or the command execution error occurs, the hard
296         // disk operation failure problem is handled, and then the hard disk is again requested to perform
297         // reset processing and continue to execute other request items, and then returns.
298         //
299         // In the function bad_rw_intr(), each read operation error will accumulate the number of errors
300         // in the current request item. If the number of errors is less than half of the maximum number
301         // of allowed, the hard disk reset operation will be performed first, and then the request
302         // processing will be executed. If the number of errors has been greater than or equal to the
303         // maximum number of allowed errors MAX_ERRORS (7 times), the processing of this request item
304         // is ended, and the next request item in the queue is processed.
305         // In function do_hd_request(), it is determined whether the reset, ecalibration, etc. need
306         // to be performed according to the specific flag status at that time, and then the next request
307         // is continued or processed.
308         if (win\_result()) {                // if there is error...

```

```

277         bad_rw_intr();           // r/w failure handling.
278         do_hd_request();          // continue handling request items.
279         return;
280     }
    // If there is no error in the read operation, the data of one sector is read from the data
    // register port into the buffer of the request, and the sector number to be read is decremented.
    // If it is not equal to 0 after decrementing, it means that there is data to be read in this
    // request, so the interrupt C function pointer sets to point to read_intr() again and return
    // directly, waiting for the hard disk to interrupt again after reading another sector data.
    // Note 1: Number 256 in line 281 refers to the memory word, which is 512 bytes.
    // Note 2: The statement on line 262 again sets the do_hd pointer to read_intr() because the
    // hard disk interrupt handler will set the function pointer to NULL each time do_hd is called.
See lines 251-253 in kernel/sys_call.s file.
281     port_read(HD_DATA, CURRENT->buffer, 256);    // put in buffer.
282     CURRENT->errors = 0;                          // Clear the number of errors.
283     CURRENT->buffer += 512;                        // adjust the buffer pointer.
284     CURRENT->sector++;                             // increase sectors read.
285     if (--CURRENT->nr_sectors) {                   // if there is still data to read
286         SET_INTR(&read_intr);                    // points to read_intr() again.
287         return;
288     }
    // Execution here, indicating that all sector data of this request item has been read. The
    // end_request() function is then called to handle the end of the request. Finally, call
    // do_hd_request() again to process other hard disk requests.
289     end_request(1);                               // set data updated flag.
290     do_hd_request();                              // Perform other request operations.
291 }
292
    /// The write sector function called in the interrupt.
    // This function will be called during the hard disk interrupt that is triggered at the end
    // of the hard disk write command. This function is handled similarly to read_intr(). After
    // the write command is executed, a hard disk interrupt signal is generated and the hard disk
    // interrupt handler is executed. At this point, the C function pointer do_hd called in the
    // interrupt handler has already pointed to write_intr(), so the function will be executed after
    // a write sector operation is completed (or an error).
293 static void write_intr(void)
294 {
    // This function first determines if the write command operation is in error. If the controller
    // is still busy after the command is finished, or the command execution error occurs, the hard
    // disk operation failure problem is handled, and then the hard disk is again requested to perform
    // reset processing and continue to execute other request items, and then returns.
295     if (win_result()) {                          // If controller returns error messages,
296         bad_rw_intr();                          // r/w error handling.
297         do_hd_request();                        // continue handling request items.
298         return;
299     }
    // At this point, it indicates that the write one sector operation is successful, so the number
    // of sectors to be written is decremented by one. If it is not 0, it means that there are still
    // sectors to write, so the current request start sector number is +1, and the request data
    // buffer pointer is adjusted to point to the next piece of data to be written. Then reset the
    // C function pointer do_hd in the hard disk interrupt handler to point to this function. The
    // 512-byte data is then written to the controller data port, and the function returns to wait
    // for the interrupt generated after the operation is completed.

```

```

300     if (--CURRENT->nr_sectors) {           // If there are still sectors to write...
301         CURRENT->sector++;
302         CURRENT->buffer += 512;
303         SET_INTR(&write_intr);           // do_hd points write_intr() again.
304         port_write(HD_DATA, CURRENT->buffer, 256);
305         return;
306     }
    // If all the sector data of this request item has been written, the end_request() function
    // is called to process the end of the request item. Finally, call do_hd_request() again to
    // process other requests.
307     end_request(1);                       // set data updated flag.
308     do_hd_request();                     // Perform other request operations.
309 }
310
    /// The hard disk recalibration (reset) function called in the interrupt.
    // If the hard disk controller returns an error message, the function first performs a read/write
    // failure process, and then requests the hard disk to perform a corresponding (reset) process.
    // In function do_hd_request(), it is determined whether the reset, ecalibration, etc. need
    // to be performed according to the specific flag status at that time, and then the next request
    // is continued or processed.
311 static void recal_intr(void)
312 {
313     if (win_result())                     // if error invoke bad_rw_intr()
314         bad_rw_intr();
315     do_hd_request();
316 }
317
    // Hard disk timeout processing function.
    // This function will be called in do_timer() (kernel/sched.c, line 340). After sending a command
    // to the hard disk controller, if the controller has not issued any interrupt signal after
    // hd_timeout ticks, the operation of the controller (or hard disk) has timed out. At this point
    // do_timer () will call this function to set the reset flag, and call do_hd_request () to perform
    // the reset process. If the hard disk controller issues a hard disk interrupt request signal
    // within a predetermined time (200 ticks) and starts executing the hard disk interrupt handler,
    // the ht_timeout value is set to 0 in the handler. At this point do_timer() will skip this
    // function.
318 void hd_times_out(void)
319 {
    // If there is currently no request item to process (the request item pointer is NULL), there
    // is no timeout and it can be returned directly. Otherwise, the warning message is displayed
    // first, and then it is judged whether the number of errors occurring during the execution
    // of the current request item is greater than the value MAX_ERRORS (7). If yes, the processing
    // of this request item is terminated in a failure form (data updated flag is not set), then
    // the C function pointer do_hd is set to NULL, and the reset flag is set. The reset operation
    // is then performed in the request item handler do_hd_request().
320     if (!CURRENT)
321         return;
322     printk("HD timeout");
323     if (++CURRENT->errors >= MAX_ERRORS)
324         end_request(0);
325     SET_INTR(NULL);                     // let do_hd = NULL, time_out = 200
326     reset = 1;
327     do_hd_request();

```

```

328 }
329
330 // Perform hard disk read/write request operations.
331 // The function first calculates the cylinder number on the hard disk, the sector number in
332 // the current track, the head number, etc. according to the device number and the starting
333 // sector number information in the current request item, and then according to the command
334 // in the request item (READ/ WRITE) sends a read or write command to the hard disk. If the
335 // controller reset flag or the recalibration flag has been set, then a reset or recalibration
336 // operation will be performed first.
337 // If the request item is the first one of the block device (the device is originally idle),
338 // the current request item pointer will directly point to the request item (see ll_rw_blk.c,
339 // line 84), and the function will be called to execute the read/write operation. Otherwise,
340 // in the handling of hard disk interrupt caused by the completion of a read/write, if there
341 // is still a request item to be processed, this function will also be called during the interrupt
342 // handling. See kernel/sys_call.s, line 235.
343 void do_hd_request(void)
344 {
345     int i,r;
346     unsigned int block,dev;
347     unsigned int sec,head,cyl;
348     unsigned int nsect;
349
350     // The function first checks the validity of the request item. Exit if there are no requests
351     // in the request queue (see blk.h, line 148). Then take the sub-device number in the device
352     // number and the starting sector in the current request item. The sub-device number corresponds
353     // to each partition on the hard disk. If the sub-device number does not exist or the starting
354     // sector is greater than the partition sector number -2, the request is terminated and jumps
355     // to the label repeat (defined in macro INIT_REQUEST, line 149 in blk.h). Since one block of
356     // data (2 sectors, ie 1024 bytes) is required to be read/written at a time, the requested sector
357     // number cannot be greater than the last penultimate sector number in the partition. Then,
358     // by adding the starting sector number of the partition corresponding to the sub-device number,
359     // the block to be read/written is mapped to the absolute sector number of the entire hard disk.
360     // The sub-device number is divided by 5 to get the corresponding hard disk number. For the
361     // description of the hard disk device number, see Table 6-1 in Section 6.2.3.
362     INIT_REQUEST;
363     dev = MINOR(CURRENT->dev);
364     block = CURRENT->sector; // starting sector.
365     if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
366         end_request(0);
367         goto repeat; // in blk.h, line 149.
368     }
369     block += hd[dev].start_sect;
370     dev /= 5; // dev is now the hd no (0, or 1).
371
372     // Then, based on the obtained absolute sector number 'block' and hard disk number 'dev',
373     // we can calculate the sector number (sec), the cylinder number (cyl), and the head number
374     // (head) in the hard disk. The following inline assembly code is used to calculate these data
375     // according to the number of sectors per track and the number of heads in the hard disk
376     // information structure. The calculation method is:
377     // The statement on line 346 indicates that EAX is the sector number 'block' and "0" in EDX.
378     // The DIVL instruction divides the sector number EDX:EAX by the number of sectors per track
379     // (hd_info[dev].sect), the resulting quotient is in EAX, and the remainder is in EDX. Among
380     // them, EAX is the total number of tracks to the specified position (all head faces), and EDX

```

```

// is the sector number on the current track.
// The statement on line 348 indicates that EAX is the calculated total number of tracks, and
// EDX is set to zero. The DIVL instruction divides the total number of tracks EDX:EAX by the
// total number of heads (hd_info[dev].head). The integer division value obtained in EAX is
// the cylinder number (cyl), and the remainder obtained in EDX is the head number (head).
346   __asm__ ("divl %4": "=a" (block), "=d" (sec): "0" (block), "1" (0),
347           "r" (hd_info[dev].sect));
348   __asm__ ("divl %4": "=a" (cyl), "=d" (head): "0" (block), "1" (0),
349           "r" (hd_info[dev].head));
350   sec++; // current track sector number is adjusted.
351   nsect = CURRENT->nr_sectors; // The number of sectors to read/write.

// Now we have the cylinder number (cyl) corresponding to the starting sector 'block' to be
// read/written, the sector number (sec) on the current track, the head number (head), and the
// total sector number (nsect) to be read or written. . Then we can send I/O operation commands
// to the hard disk controller based on this information. But before sending, we also need to
// see if there are flags to reset the controller state and recalibrate the hard disk. It is
// usually necessary to recalibrate the hard disk head position after the reset operation. If
// these flags have been set, it may indicate some problems with the previous hard disk operation,
// or it is now the first hard disk read and write operation of the system, so we need to reset
// the hard disk or controller and recalibrate the hard disk.
//
// If the reset flag is set at this time, a reset operation is required. Then reset the hard
// disk and controller, and set the flag that hard disk needs to be recalibrated, and return.
// The function reset_hd() will first send a reset (recalibration) command to the hard disk
// controller and then send the command "Create Drive Parameter".
352   if (reset) {
353       recalibrate = 1; // need to recalibrate.
354       reset_hd();
355       return;
356   }
// If the recalibrate flag is set at this time, the flag is first reset and then a recalibration
// command is sent to the controller. This command performs a track seek operation to move the
// head from anywhere to the 0 cylinder.
357   if (recalibrate) {
358       recalibrate = 0;
359       hd_out(dev, hd_info[CURRENT_DEV].sect, 0, 0, 0,
360             WIN_RESTORE, &recal_intr);
361       return;
362   }
// If none of the above two flags are set, then we can start sending real data read/write
// operations to the hard disk controller. If the current request is a write sector operation,
// a write command is sent, and then the status register information is cyclically read and
// it is determined whether the request service flag DRQ_STAT is set. DRQ_STAT is the request
// service bit of the hard disk status register, indicating that the drive is ready to transfer
// one word or one byte of data between the host and the data port. Exit the loop if the request
// service DRQ is set. If it is not set after the end of the loop, it means that the request
// to write the hard disk command failed, so jump to deal with the problem or continue to execute
// the next hard disk request. Otherwise, we can write 1 sector of data to the hard disk controller
// data register port HD_DATA.
363   if (CURRENT->cmd == WRITE) {
364       hd_out(dev, nsect, sec, head, cyl, WIN_WRITE, &write_intr);
365       for(i=0 ; i<10000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)

```

```

366             /* nothing */ ;
367         if (!r) {
368             bad_rw_intr();
369             goto repeat;           // label is in blk.h, line 149.
370         }
371         port_write(HD_DATA, CURRENT->buffer, 256);

// If the current request is to read hard disk data, a read sector command is sent to the
// controller. Stop if the command is invalid.
372     } else if (CURRENT->cmd == READ) {
373         hd_out(dev, nsect, sec, head, cyl, WIN_READ, &read_intr);
374     } else
375         panic("unknown hd-command");
376 }
377

// Hard disk system initialization.
// The function sets the hard disk interrupt descriptor and enable the controller to send
// interrupt request signal. It sets the request handler of the hard disk to do_hd_request(),
// and then sets the hard disk interrupt gate descriptor. hd_interrupt is the address of its
// interrupt handler (kernel/sys_call.s, line 235). The hard disk interrupt is INT 0x2E (46),
// which corresponds to the interrupt request signal IRQ14 of the 8259A chip. The mask bit of
// INT2 on the master chip 8259A is then reset, allowing interrupt request signal to be issued
// from slave chip. Then reset the interrupt mask bit of the hard disk (on the slave), allowing
// the hard disk controller to send an interrupt request signal. The macro set_intr_gate() of
// the interrupt gate descriptor in IDT is in include/asm/system.h.
378 void hd_init(void)
379 {
380     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;    // do_hd_request().
381     set_intr_gate(0x2E, &hd_interrupt);              // set interrupt handler
382     outb_p(inb_p(0x21)&0xfb, 0x21);                  // reset mask bit 2 on master chip.
383     outb(inb_p(0xA1)&0xbf, 0xA1);                     // reset mask bit 6 on slave chip.
384 }
385

```

## 9.3.3 Information

### 9.3.3.1 AT Hard Disk Interface Register

The programming register port description for the AT hard disk controller is shown in Table 9–3. Also see the description in the include/linux/hdreg.h header file.

Table 9-3 AT hard disk controller register port and its use

Port	Name	Read operation	Write operation
0x1f0	HD_DATA	Data Register - Sector data (read, write and format)	
0x1f1	HD_ERROR, HD_PRECOMP	Error Register (HD_ERROR)	Write Precomp Register (HD_PRECOMP)
0x1f2	HD_NSECTOR	Total Sectors Register - Total number of sectors (read, write, check and format)	
0x1f3	HD_SECTOR	Sector Number Register - Start sector number (read, write and check)	
0x1f4	HD_LCYL	Cylinder Number Register - Cylinder number low byte (read, write, check and format)	
0x1f5	HD_HCYL	Cylinder Number Register - Cylinder number high byte (read, write, check and format)	
0x1f6	HD_CURRENT	Drive/Head Register - Drive/Head No. (101dhhhh, d=Drive No, h=Head No)	
0x1f7	HD_STATUS, HD_COMMAND	Main Status Register (HD_STATUS)	Command Register (HD_COMMAND)



0x3f6	HD_CMD	---	Harddisk Control Register (HD_CMD)
0x3f7		Digital Input Register (1.2MB Floppy)	---

The port registers are described in detail below.

◆Data register (HD\_DATA, 0x1f0)

This is a pair of 16-bit high speed PIO data transmitters for sector read, write and track formatting operations. The CPU writes to the hard disk through the data register or reads data of one sector from the hard disk, that is, repeats reading/writing  $cx=256$  words using the command 'rep outsw' or 'rep insw'.

◆Error register (read) / write precompensation register (write) (HD\_ERROR, 0x1f1)

When reading, this register holds an 8-bit error status. However, the data in this register is valid only when Bit 0 of the main status register (HD\_STATUS, 0x1f7) is set. The meaning when executing the controller diagnostic command is different from that of other commands. See Table 9-4.

This register acts as a write pre-compensation register during a write operation. It records the write pre-compensation starting cylinder number. Corresponds to a word at 0x05 of the basic parameter table of the hard disk, which is divided by 4 and output. But most of the current hard drives ignore this parameter. For a description of the hard disk parameter table, see Table 6-4 in Section 6.3.3.

What is write pre-compensation?

Early hard disks have a fixed number of sectors per track, and since each sector has a fixed 512 bytes, the physical track length occupied by each sector is shorter as it gets closer to the center of the disk. The ability to cause magnetic media to store data is reduced. Therefore, for the hard disk head, it is necessary to take certain measures to put the data of one sector into a relatively small sector at a relatively high density. The common method used is the Write Precompensation technique. That is, starting from the edge of the disc to a position near a track (cylinder) in the center of the disc, the write current in the head is adjusted in some way.

The specific adjustment method is as follows: the representation of the binary data 0, 1 on the disk is recorded by a magnetic recording and encoding method (for example, FM, MFM, etc.). If the adjacent recording bits are magnetized and flipped twice, magnetic field overlap may occur. Therefore, the peak value of the corresponding electrical waveform will drift when the data is read at this time. If the recording density is increased, the degree of peak drift is increased, and sometimes the data bits cannot be separated and recognized, resulting in read data errors. The way to overcome this problem is to use pre-write compensation or post-read compensation techniques. Pre-write compensation means that the pulse compensation is written in advance in the opposite direction to the peak drift of the readout before the write data is sent to the driver. If the peak value of the signal drifts forward when read, the signal is delayed to be written; if the signal drifts backward when read, the signal is written in advance. Thus, when reading, the position of the peak can be close to the normal position.

Table 9-4 Hard disk controller error register

Value	Dianostic command	Other commands
0x01	No error	Data flag missing
0x02	Controller error	Track 0 error
0x03	Sector buffer error	
0x04	ECC part error	Command abandon
0x05	Control processor error	
0x10		ID not found
0x40		ECC error
0x80		Bad sector

◆Sector Number Register (HD\_NSECTOR, 0x1f2)

This register holds the number of sectors specified by the read, write, verify, and format commands. When used for multi-sector operation, the register is automatically decremented by one each time a sector operation is completed, until it is zero. If the initial value is 0, it means that the maximum number of sectors is 256.

◆Begin Sector No Register (HD\_SECTOR, 0x1f3)

This register holds the sector start number specified by the read, write, and verify operation commands. In multi-sector operation, the starting sector number is stored, and the operation is automatically incremented by 1 for each sector operation.

◆Cylinder number register (HD\_LCYL, HD\_HCYL, 0x1f4, 0x1f5)

The two cylinder registers respectively store the lower 8 bits and the upper 2 bits of the cylinder number.

◆Driver/head register (HD\_CURRENT, 0x1f6)

This register holds the drive and head numbers specified by the read, write, verify, seek, and format commands. Its bit format is 101dhhhh. Where 101 indicates that the ECC check code is used and 512 bytes per sector; d indicates the selected drive (0 or 1); hhhh indicates the selected head, as shown in Table 9–5.

Table 9-5 Drive/head register

Bit	Name	Description
0	HS0	Head number lowest bit
1	HS1	
2	HS2	
3	HS3	Head number highest bit
4	DRV	Select drive, 0 - Select drive 0; 1 - Select drive 1
5	Reserved	Always 1
6	Reserved	Always 0
7	Reserved	Always 1

◆Main Status Register (Read) / Command Register (Write) (HD\_STATUS/HD\_COMMAND, 0x1f7)

At the time of reading, it corresponds to an 8-bit main status register. Reflects the operating state of the hard disk controller before and after executing the command. The meaning of each bit in this register is shown in Table 9–6.

Table 9-6 8-bit main status register

Bit	Name	Mask	Description
0	ERR_STAT	0x01	Command execution error. When this bit is set, the previous command ends with an error. At this point, the bits in the error register and status register contain some information that caused the error.
1	INDEX_STAT	0x02	Received the index. This bit is set when an index flag is encountered while the disk is spinning.
2	ECC_STAT	0x04	ECC checksum error. This bit is set when a recoverable data error is encountered and has been corrected. This situation does not interrupt a multi-sector read operation.
3	DRQ_STAT	0x08	Data request service. When this bit is set, it indicates that the drive is ready to transfer one word or one byte of data between the host and the data port.
4	SEEK_STAT	0x10	The drive seek ends. When this bit is set, it indicates that the seek operation has been

			completed and the head has stopped on the specified track. This bit does not change when an error occurs. This bit will again indicate the completion status of the current seek after the host has read the status register.
5	WRERR_STAT	0x20	Drive failure (write error). This bit does not change when an error occurs. This bit will again indicate the error status of the current write operation only after the host has read the status register.
6	READY_STAT	0x40	The drive is ready. Indicates that the drive is ready to receive commands. This bit does not change when an error occurs. This bit will again indicate the current drive ready state after the host has read the status register. At power-on, this bit should be reset until the drive speed is normal and the command can be received.
7	BUSY_STAT	0x80	<p>The controller is busy. This bit is set when the drive is operating by the controller of the drive. At this point, the host cannot send a command block. A read of any of the command registers will return the value of the status register. This bit will be set under the following conditions:</p> <p>(*) It is within 400 nanoseconds after the machine reset signal RESET becomes negative or the SRST of the device control register is set. The set state of this bit is required to be no longer than 30 seconds after a machine reset.</p> <p>(*) The host is within 400 nanoseconds of writing commands such as recalibration, read, read buffer, initialization of drive parameters, and execution of diagnostics.</p> <p>(*) Within 5 microseconds of 512 bytes of data transferred during a write operation, write buffer, or format track command.</p>

When a write operation is performed, the port corresponds to a command register. It accepts hard disk control commands from the CPU. There are 8 commands, as shown in Table 9–7. The last column is used to describe the actions taken by the controller after the end of the corresponding command (causing an interrupt or doing nothing).

Table 9-7 AT hard disk controller command list

Command Name		Command Code Byte		Default value	Command End Form
		High 4 bits	D3 D2 D1 D0		
WIN_RESTORE	Drive Recalibration (Reset)	0x1	R R R R	0x10	Interrupt
WIN_READ	Read Sector	0x2	0 0 L T	0x20	Interrupt
WIN_WRITE	Write Sector	0x3	0 0 L T	0x30	Interrupt
WIN_VERIFY	Sector Check	0x4	0 0 0 T	0x40	Interrupt
WIN_FORMAT	Format track	0x5	0 0 0 0	0x50	Interrupt
WIN_INIT	Controller Initialization	0x6	0 0 0 0	0x60	Interrupt
WIN_SEEK	Seek Track	0x7	R R R R	0x70	Interrupt
WIN_DIAGNOSE	Controller Diagnostic	0x9	0 0 0 0	0x90	Interrupt or Idle
WIN_SPECIFY	Build Drive Parameters	0x9	0 0 0 1	0x91	Interrupt

The lower 4 bits of the command code byte in the table are additional parameters, which means:

R is the step rate. When R=0, the step rate is 35us; R=1 is 0.5ms, which is incremented by this amount. The default R=0 in the program.

L is the data mode. L = 0 indicates that the read/write sector is 512 bytes; L = 1 indicates that the read/write sector is 512 plus 4 bytes of ECC code. The default value in the program is L=0.

T is the retry mode. T=0 indicates that retry is allowed; T=1 prohibits retry. Take T=0 in the kernel program.

These commands are described in detail below.

(1) 0x1X -- (WIN\_RESTORE), drive recalibrate command

This command moves the read/write head from any position on the disk to the 0 cylinder. When the command is received, the drive sets the BUSY\_STAT flag and issues a 0 cylinder seek command. The driver then waits for the end of the seek operation, then updates the state, resets the BUSY\_STAT flag, and generates an interrupt.

(2) 0x20 -- (WIN\_READ) retryable read sector; 0x21 -- no retry read sector.

The read sector command can read from 1 to 256 sectors starting from the specified sector. If the sector count in the specified command block (see Table 9-9) is 0, it means that 256 sectors are read. When the drive accepts the command, the BUSY\_STAT flag will be set and execution of the command will begin. For a single sector read operation, if the track position of the head is not correct, the driver implicitly performs a seek operation. Once the head is on the correct track, the drive head is positioned to the corresponding ID field in the track address field.

For the no retry sector read command, if the specified ID field cannot be read correctly before the two index pulses occur, the drive will give an error message that the ID is not found in the error register. For a retryable read sector command, the drive will retry multiple times when it encounters a problem in the read ID field. The number of retries is set by the drive manufacturer.

If the drive correctly reads the ID field, it needs to identify the Data Address Mark in the specified number of bytes, otherwise it will report an error that the data address mark did not find. Once the head finds the data address mark, the drive reads the data in the data field into the sector buffer. If an error occurs, the drive sets the error bit, sets DRQ\_STAT, and generates an interrupt. Regardless of whether an error occurs, the drive always sets DRQ\_STAT after reading the sector. After the command is completed, the command block register will contain the cylinder number, head number and sector number of the last sector read.

For multi-sector read operations, each time the drive is ready to send a sector of data to the host, DRQ\_STAT is set, the BUSY\_STAT flag is cleared, and an interrupt is generated. When the sector data transfer ends, the drive resets the DRQ\_STAT and BUSY\_STAT flags, but sets the BUSY\_STAT flag after the last sector transfer is complete. At the end of the command, the command block register will contain the cylinder number, head number and sector number of the last sector read.

If an uncorrectable error occurs in a multi-sector read operation, the read operation will terminate at the sector where the error occurred. Similarly, the command block register will contain the cylinder number, head number and sector number of the error sector. Regardless of whether the error can be corrected, the drive puts the data into the sector buffer.

(3) 0x30 -- (WIN\_WRITE) retryable write sector; 0x31 -- no retry write sector.

The Write Sector command can write from 1 to 256 sectors starting from the specified sector. If the sector count in the specified command block (see Table 9-9) is 0, it means that 256 sectors are to be written. When the drive accepts the command, it sets DRQ\_STAT and waits for the sector buffer to be filled with data. There is no interruption when starting to add data to the sector buffer for the first time. Once the data is full, the drive resets the DRQ, sets the BUSY\_STAT flag, and begins executing the command.

For operations that write data for one sector, the drive sets DRQ\_STAT when it receives the command and waits for the host to fill the sector buffer. Once the data has been transferred, the drive sets BUSY\_STAT and resets DRQ\_STAT. As with the read sector operation, if the track position of the head is incorrect, the drive implicitly performs a seek operation. Once the head is on the correct track, the drive head is positioned to the corresponding ID field in the track address field.

If the ID field is correctly read, the data in the sector buffer, including the ECC byte, is written to disk. When the drive has processed a sector, the BUSY\_STAT flag is cleared and an interrupt is generated. At this point the host can read the status register. At the end of the command, the command block register will contain the cylinder number, head number and sector number of the last sector written.

During a multi-sector write operation, in addition to the operation of the first sector, when the drive is ready to receive data for one sector from the host, DRQ\_STAT is set, the BUSY\_STAT flag is cleared, and an interrupt is generated. Once a sector has been transferred, the drive resets the DRQ and sets the BUSY flag. When the last sector is written to disk, the drive clears the BUSY\_STAT flag and generates an interrupt (at this point DRQ\_STAT has been reset). At the end of the write command, the command block register will contain the cylinder number, head number and sector number of the last sector written.

If an error occurs in a multi-sector write operation, the write operation will terminate at the sector where the error occurred. Similarly, the command block register will contain the cylinder number, head number and sector number of the error sector.

(4) 0x40 -- (WIN\_VERIFY) retryable sector read verification; 0x41 -- no retry sector verification.

The execution of this command is the same as the read sector operation, but this command does not cause the drive to set DRQ\_STAT and does not transfer data to the host. When the read verification command is received, the drive sets the BUSY\_STAT flag. When the specified sector is verified, the drive resets the BUSY\_STAT flag and generates an interrupt. At the end of the command, the command block register will contain the cylinder number, head number and sector number of the last verified sector.

If an error occurs in the multi-sector verification operation, the verification operation will terminate at the sector where the error occurred. Similarly, the command block register will contain the cylinder number, head number and sector number of the error sector.

(5) 0x50 -- (WIN\_FORMAT) Format the track command.

The track address is specified in the sector count register. When the drive accepts the command, it sets the DRQ\_STAT bit and then waits for the host to fill the sector buffer. When the buffer is full, the drive clears DRQ\_STAT, sets the BUSY\_STAT flag, and begins execution of the command.

(6) 0x60 -- (WIN\_INIT) controller initialization.

(7) 0x7X -- (WIN\_SEEK) seek operation.

The seek operation command moves the selected head in the command block register to the specified track. When the host issues a seek command, the drive sets the BUSY flag and generates an interrupt. The drive does not set SEEK\_STAT (DSC - seek completion) until the seek operation is completed. The seek operation may not have completed before the drive generates an interrupt. If the host issues a new command to the drive while the seek operation is in progress, then BUSY\_STAT will remain set until the end of the seek. Then the drive starts executing the new command.

(8) 0x90 -- (WIN\_DIAGNOSE) drive diagnostic command.

This command performs the diagnostic test process implemented inside the drive. Driver 0 sets the BUSY\_STAT bit within 400 ns of the command.

If the system contains a second drive, drive 1, then both drives perform diagnostic operations. Drive 0 will wait for drive 1 to perform a diagnostic operation for 5 seconds. If drive 1 diagnostics fails, drive 0

appends 0x80 to its diagnostic status. If the host detects that the diagnostic operation of the drive 1 has failed while reading the state of the drive 0, it sets the drive select bit (bit 4) of the drive/head register (0x1f6) and then reads the state of the drive 1. If the drive 1 passes the diagnostic test or the drive 1 does not exist, the drive 0 directly loads its own diagnostic status into the error register. If drive 1 does not exist, then drive 0 only reports its own diagnostic results and generates an interrupt after resetting the BUSY\_STAT bit.

(9) 0x91 -- (WIN\_SPECIFY) Create a drive parameter command.

This command is used to allow the host to set the head swap and sector count loop values for multi-sector operations. When the command is received, the drive sets the BUSY\_STAT bit and generates an interrupt. This command uses only the values of the two registers. One is the sector count register, which is used to specify the number of sectors; the other is the driver/head register, which is used to specify the number of heads, and the drive select bit (bit 4) is set according to the specifically selected driver.

This command does not verify the selected sector count value and number of heads. If these values are invalid, the drive will not report an error until another command uses these values and invalidates an access error.

#### ◆ Hard disk control register (write) (HD\_CMD, 0x3f6)

This register is write-only and is used to store the hard disk control byte and control the reset operation. Its definition is the same as the byte description at shift 0x08 of the hard disk basic parameter table, as shown in Table 9-8.

Table 9-8 The meaning of the hard disk control byte

Offset	Bit	Control Byte Description (drive step selection)
0x08	0	Not used
	1	Reserved (0) (Close IRQ)
	2	Allow reset
	3	Set if the number of heads is greater than 8
	4	Not used (0)
	5	If there is a manufacturer's bad area map at the cylinders +1, set 1
	6	Prohibit ECC retry
	7	Prohibit access retry

### 9.3.3.2 AT hard disk controller programming

When operating the hard disk controller, you need to send parameters and commands at the same time. The command format is shown in Table 9-9. First, you need to send a 6-byte parameter, and finally issue a 1-byte command code. No matter what command, you need to completely output the 7-byte command block, and write to port 0x1f1 -- 0x1f7 in turn. Once the command block register is loaded, the command begins execution.

Table 9-9 Hard disk controller command format

Port	Description
0x1f1	Write pre-compensation start cylinder number
0x1f2	Number of sectors

0x1f3	Starting sector number
0x1f4	Cylinder number low byte
0x1f5	Cylinder number high byte
0x1f6	Drive number / head number
0x1f7	Command code

First, the CPU outputs a control byte to the control register port (HD\_CMD, 0x3f6) to establish a corresponding hard disk control mode. After the mode is established, parameters and commands can be sent in the above order. The steps are:

1. Detect controller idle state: The CPU reads the main status register. If bit 7 (BUSY\_STAT) is 0, the controller is idle. If the controller is always busy within the specified time, it is judged as a timeout error. See the controller\_ready() function on line 168 in hd.c.
2. Check if the drive is ready: The CPU determines if the main status register bit 6 (READY\_STAT) is 1 to see if the drive is ready. If 1, the CPU can output parameters and commands. See the drive\_busy() function on line 209 in hd.c.
3. Output command block: Outputs parameters and commands to the corresponding ports in sequence. See the hd\_out() function starting with line 187 in hd.c.
4. CPU waits for interrupt generation: After the command is executed, the hard disk controller will generate an interrupt request signal (IRQ14 - corresponding int46) or set the controller state to idle, indicating the end of the operation or the request for sector transfer (multi-sector read/write). The function called in the program hd.c during the interrupt processing is shown in the code 237--293. There are 5 functions corresponding to 5 cases: hard disk reset, unexpected interrupt, bad read/write interrupt, read interrupt and write interrupt.
5. Detection operation result: The CPU reads the main status register again. If bit 0 is equal to 0, the command execution is successful, otherwise it fails. If it fails, you can further query the error register (HD\_ERROR) to get the error code. See the win\_result() function on line 176 of hd.c.

### 9.3.3.3 硬盘分区表

If the PC starts the operating system from the hard disk, the ROM BIOS program reads the first sector into the memory 0x7c00 after executing the machine self-test diagnostic program, and gives execution control to the code on the sector to continue executing. This particular sector is referred to as the Master Boot Record (MBR), and its content structure is shown in Table 9-10.

Table 9-10 The structure of the hard disk master boot sector MBR

Offset	Name	Size (Byte)	Description
0x000	MBR Code	446	Boot program code and data.
0x1BE	Partition table entry 1	16	The first partition table entry, a total of 16 bytes.
0x1CE	Partition table entry 2	16	The second partition table entry, 16 bytes.
0x1DE	Partition table entry 3	16	The third partition table entry, 16 bytes.
0x1EE	Partition table entry 4	16	The fourth partition table entry, 16 bytes.
0x1FE	Boot flag	2	Valid boot sector flags: 0x55, 0xAA

In addition to the initial 446-byte boot executable code, the MBR also contains a hard disk partition table with a total of four entries. The partition table is stored at the 0x1BE--0x1FD offset position of the 1st sector of

the 0 cylinder 0 of the hard disk. In order to enable multiple operating systems to share hard disk resources, the hard disk can logically divide all sectors into 1--4 partitions. The sector numbers between each partition are contiguous. Each entry in the partition table has 16 bytes and is used to describe the characteristics of a partition. The size of the partition, the cylinder number, the track number and the sector number of the starting and ending are stored, as shown in Table 9-11.

Table 9-11 Hard disk partition table entry structure

Offset	Name	Size	Description
0x00	boot_ind	1 byte	Boot index. Only one partition of the 4 partitions can be booted at a time. 0x00 - Do not boot from this partition; 0x80 - Boot from this partition.
0x01	head	1 byte	Partition start head number. The head number ranges from 0 to 255.
0x02	sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the beginning of the partition.
0x03	cyl	1 byte	The lower 8 bits of the cylinder number at the starting of the partition.
0x04	sys_ind	1 byte	Partition type. 0x0b - DOS; 0x80 - Old Minix; 0x83 - Linux . . .
0x05	end_head	1 byte	The head number at the end of the partition. It ranges from 0 to 255.
0x06	end_sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the end of the partition .
0x07	end_cyl	1 byte	The lower 8 bits of the cylinder number at the end of the partition.
0x08-0x0b	start_sect	4 byte	The physical sector number at the beginning of the partition. It counts from 0 in the order of the sector number of the entire hard disk.
0x0c-0x0f	nr_sects	4 byte	The number of sectors occupied by the partition.

The fields 'head', 'sector', and 'cyl' in the table represent the head number, sector number, and cylinder number at the beginning of a partition, respectively. The range of the 'head' number ranges from 0 to 255. The lower 6 bits in the 'sector' byte field represent the sector number counted in the current cylinder. The sector number count range is 1-63. The upper 2 bits of the 'sector' field form a 10-bit cylinder number with the 'cyl' field, which ranges from 0 to -1023. Similarly, the 'end\_head', 'end\_sector', and 'end\_cyl' fields in the table indicate the head number, sector number, and cylinder number at the end of the partition, respectively. Therefore, if we use C to indicate the cylinder number, H for the head number, and S for the sector number, the starting CHS value of the partition can be expressed as:

---

```

H = head;
S = sector & 0x3f;
C = (sector & 0xc0) << 2) + cyl;

```

---

The 'start\_sect' field in the table is the 4-byte partition start physical sector number. It represents the sector number of the entire hard disk compiled from 0. The encoding method is: starting from CHS for 0 cylinder, 0 head and 1 sector (0, 0, 1), first encode the sectors in the current cylinder in order, and then encode the head from 0 to the maximum head number. Finally, the cylinder is counted. If the total number of heads of a hard disk is MAX\_HD and the total number of sectors per track is MAX\_SECT, then the physical sector number phy\_sector of the hard disk corresponding to a CHS value is:



---

$$\text{phy\_sector} = (\text{C} * \text{MAX\_HEAD} + \text{H}) * \text{MAX\_SECT} + \text{S} - 1$$

---

The first sector of the hard disk (0 cylinder 0 header 1 sector) has the same purpose as the first sector (boot sector) on the floppy disk except for one partition table. Only its code will move itself from 0x7c00 to 0x6000 during execution to free up space at 0x7c00, and then find out which active partition is based on the information in the partition table. Then load the first sector of the active partition to 0x7c00 to execute. A partition from which cylinder, head and sector of the hard disk are recorded in the partition table. Therefore, it is possible to know from the partition table where the first sector of an active partition (ie, the boot sector of the partition) is on the hard disk.

#### 9.3.3.4 Relationship between absolute sector and current cylinder, sector, head.

Assume that the number of sectors per track of the hard disk is `track_secs`, the total number of heads is `dev_heads`, and the total number of tracks is `tracks`. For the specified sequential sector number `sector`, the corresponding current cylinder number is `cyl`, the sector number on the current track is `sec`, and the current head number is `head`. Then, if you want to convert from the specified sequential sector number to the corresponding current cylinder number, the current track sector number, and the current head number, you can use the following steps:

- `sector / track_secs` = The quotient is `tracks`, and the remainder is `sec`;
- `tracks / dev_heads` = the quotient is `cyl`, and the remainder is `head`;
- On the current track, the sector number starts from 1, so you need to increase `sec` by 1.

If you want to convert the specified current `cyl`, `sec`, and `head` to the sequential sector number from the hard disk, the process is exactly the opposite. The conversion formula is exactly the same as given above, namely:

---

$$\text{sector} = (\text{cyl} * \text{dev\_heads} + \text{head}) * \text{track\_secs} + \text{sec} - 1$$

---

## 9.4 ll\_rw\_blk.c

### 9.4.1 Function Description

This program is mainly used to perform low-level block device read/write operations. It is the interface program between all block devices (hard drive, floppy drive and virtual ram disk) in this chapter and other parts of the system. By calling the program's read/write function `ll_rw_block()`, other programs in the system can asynchronously read and write data from the block device. The main purpose of this function is to create block device read and write request items for other programs and insert them into the specified block device request queue. The actual read and write operations are done by the request handling function `request_fn()` of the device. For hard disk operations, the function is `do_hd_request()`; for floppy operations, the function is `do_fd_request()`; for virtual disks it is `do_rd_request()`.

If `ll_rw_block()` builds a request item for a block device and determines that the device is idle by checking that the current request item pointer of the block device is `NULL`, the newly created request item is set as the current request, and the `request_fn()` is directly invoked. Otherwise, the elevator request algorithm will be used

to insert the newly created request item into the request linked list queue of the device for processing. When request\_fn() ends processing, the request item is removed from the linked list.

Since request\_fn() completes the processing of a request item, it will call request\_fn() itself again through the interrupt callback C function (mainly read\_intr() and write\_intr()) to process the remaining request items in the linked list. Therefore, as long as there are unprocessed request items in the linked list (or called queues), they will be processed one after another until the linked list of request items of the device is empty. When the linked list of request items is empty, request\_fn() will no longer send commands to the drive controller, but will exit immediately. Therefore, the loop call to the request\_fn() function ends, as shown in Figure 9-5.

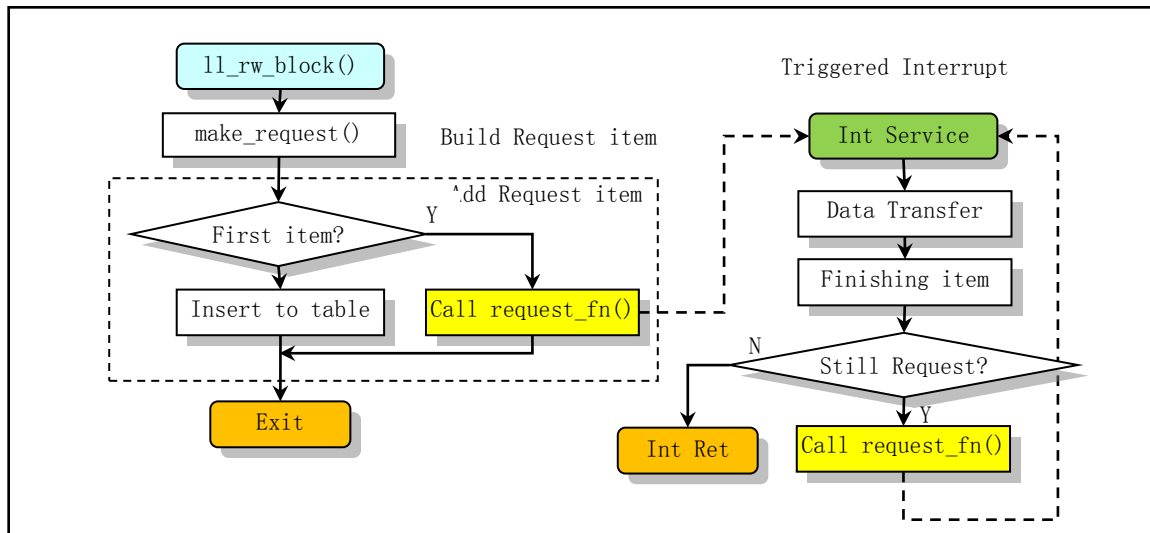


Figure 9-5 ll\_rw\_block call sequence

For the virtual disk device, since its read and write operations do not involve the above-mentioned synchronous operation with the external hardware device, there is no interrupt processing described above. The read and write operations of the current request item to the virtual device are completely implemented in do\_rd\_request().

## 9.4.2 Code Annotation

Program 9-3 linux/kernel/blk\_drv/ll\_rw\_blk.c

```

1  /*
2  *  linux/kernel/blk_dev/ll_rw.c
3  *
4  *  (C) 1991 Linus Torvalds
5  */
6
7  /*
8  *  This handles all read/write requests to block devices
9  */
10 // <errno.h> Error number header file. Contains various error numbers in the system.
11 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
12 //   of the initial task 0, and some embedded assembly function macro statements about the
13 //   descriptor parameter settings and acquisition.
14 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly

```

```

//      used functions of the kernel.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//      descriptors/interrupt gates, etc. is defined.
10 #include <errno.h>
11 #include <linux/sched.h>
12 #include <linux/kernel.h>
13 #include <asm/system.h>
14
15 #include "blk.h"          // request structure, linked list queue.
16
17 /*
18  * The request-struct contains all necessary data
19  * to load a nr of sectors into memory
20  */
// request item array queue. NR_REQUEST = 32
21 struct request request[NR_REQUEST];
22
23 /*
24  * used to wait on when there are no free requests
25  */
26 struct task_struct * wait_for_request = NULL;
27
28 /* blk_dev_struct is:
29  *      do_request-address
30  *      next-request
31  */
// An array of block devices. This array uses the major device number as the index. The actual
// content will be filled in at the initialization of each device driver. For example, when
// the hard disk driver is initialized (hd.c, line 378), the first statement is used to set
// the contents of blk_dev[3]. See the file blk_drv/blk.h, line 45, line 50.
32 struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
33     { NULL, NULL },          /* no_dev */
34     { NULL, NULL },          /* dev mem */
35     { NULL, NULL },          /* dev fd */
36     { NULL, NULL },          /* dev hd */
37     { NULL, NULL },          /* dev ttyx */
38     { NULL, NULL },          /* dev tty */
39     { NULL, NULL }           /* dev lp */
40 };
41
42 /*
43  * blk_size contains the size of all block-devices:
44  *
45  * blk_size[MAJOR][MINOR]
46  *
47  * if (!blk_size[MAJOR]) then no minor size checking is done.
48  */
// An array of pointers to the total number of device data blocks. Each pointer item points
// to an array of the total number of blocks for the specified major device number. The total
// number of blocks in the array corresponds to the total number of data blocks owned by a
// sub-device determined by the sub-device number (1 block size = 1 KB).
49 int * blk_size[NR_BLK_DEV] = { NULL, NULL, }; // nr of block devices, NR_BLK_DEV = 7.
50

```

```

// Locks the specified buffer block.
// If the specified buffer block has been locked by another task, sleep itself (uninterruptedly
// waiting) until the task of performing the unlock explicitly wakes up the task.
51 static inline void lock_buffer(struct buffer_head * bh)
52 {
53     cli();                                // disable int.
54     while (bh->b_lock)                      // sleeps if locked, until buffer is unlocked.
55         sleep_on(&bh->b_wait);
56     bh->b_lock=1;                          // locked the buffer immediately.
57     sti();                                // enable int.
58 }
59
// Unlock the locked buffer.
// This function is identical to the function of the same name in the blk.h file, but the
// implementation in blk.h is used as a macro.
60 static inline void unlock_buffer(struct buffer_head * bh)
61 {
62     if (!bh->b_lock)                        // if not locked...
63         printk("ll_rw_block.c: buffer not locked\n\r");
64     bh->b_lock = 0;                        // reset lock flag.
65     wake_up(&bh->b_wait);                  // wake up the task waiting for this buffer.
66 }
67
68 /*
69  * add-request adds a request to the linked list.
70  * It disables interrupts so that it can muck with the
71  * request-lists in peace.
72  *
73  * Note that swapping requests always go before other requests,
74  * and are done in the order they appear.
75  */
// Add a request item to the linked list.
// The parameter dev is a pointer to the specified block device structure (blk.h, line 45),
// which has a request function pointer and a current request item pointer; req is a request
// item structure pointer with the content set.
// This function adds the already set request item req to the linked list of request of the
// specified device. If the device's current request pointer is NULL, then req can be set to
// the current request item and the device request item handler can be called immediately.
// Otherwise, the req request item is inserted into the linked list of the request item.
76 static void add_request(struct blk_dev_struct * dev, struct request * req)
77 {
78     struct request * tmp;
79
// First, the pointers and flags of the request items provided by the parameters are further
// set. The next request item pointer in the request is set to NULL. Disable the interrupt
// and clear the request-related buffer dirty flag.
80     req->next = NULL;
81     cli();
82     if (req->bh)
83         req->bh->b_dirt = 0;                // clear the buffer dirty flag.
// Then check if the specified device has current request item, that is, check if the device
// is busy. If the current request item (current_request) field of the specified device dev
// is NULL, it means that the device has no request item at present, this time is the first

```

```

// request item, and is the only one. Therefore, the block device current request pointer can
// be directly pointed to the request item, and the request function of the corresponding device
// is immediately executed.
84     if (!(tmp = dev->current_request)) {
85         dev->current_request = req;
86         sti(); // Enable int.
87         (dev->request_fn)(); // runs request function, ie.do_hd_request().
88         return;
89     }
// If the device currently has current request item in process, the elevator algorithm is first
// used to search for the best insertion location, and then the request item is inserted into
// the request list. During the search course, if it is determined that the buffer block pointer
// to be inserted is NULL, that is, there is no buffer block, then an item needs to be found,
// which already has a buffer block available. Therefore, if the free entry buffer block header
// pointer at the current insertion position (after tmp) is not empty, this position is selected.
// Then exit the loop and insert the request item here. Finally enable the interrupt and exit
// the function. The role of the elevator algorithm is to minimize the movement distance of
// the disk head, thereby reducing hard disk access time.
//
// The following statement in the loop is used to compare the request item referred to by req
// with the existing request item in the request queue, and find out the correct position order
// in which the req is inserted into the queue. Then break the loop and insert req into the
// correct position of the queue.
90     for ( ; tmp->next ; tmp=tmp->next) {
91         if (!req->bh)
92             if (tmp->next->bh)
93                 break;
94             else
95                 continue;
96         if ((IN_ORDER(tmp, req) || // blk.h, line 40.
97             !IN_ORDER(tmp, tmp->next)) &&
98             IN_ORDER(req, tmp->next))
99             break;
100     }
101     req->next=tmp->next;
102     tmp->next=req;
103     sti();
104 }
105
//// Create a request and insert it into the request queue.
// The parameter major is the major device number; rw is the specified command; bh is the buffer
// header pointer for storing data.
106 static void make_request(int major, int rw, struct buffer_head * bh)
107 {
108     struct request * req;
109     int rw_ahead;
110
111     /* WRITEA/READA is special case - it is not really needed, so if the */
112     /* buffer is locked, we just forget about it, else it's a normal read */
// Here the suffix 'A' character after 'READ' and 'WRITE' represents the word Ahead, indicating
// the pre-read/write block of data. This function first does some processing on the case of
// the READA/WRITEA command. These two commands discard the read/write request for the case
// where the specified buffer is in use and has been locked. Otherwise, it operates as a normal

```

```

// READ/WRITE command. In addition, if the command given by the parameter is neither READ nor
// WRITE, it means that the kernel program is faulty, then displays an error message and stops
// the kernel. Note that the flag rw_ahead has been set here for the parameter whether it is
// a prefetch/write command before modifying the command.
113     if (rw_ahead = (rw == READA || rw == WRITEA)) {
114         if (bh->b_lock)
115             return;
116         if (rw == READA)
117             rw = READ;
118         else
119             rw = WRITE;
120     }
121     if (rw!=READ && rw!=WRITE)
122         panic("Bad block dev command, must be R/W/RA/WA");
123     lock\_buffer(bh);
124     if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
125         unlock\_buffer(bh);
126         return;
127     }
128 repeat:
129 /* we don't allow the write-requests to fill up the queue completely:
130  * we want some room for reads: they take precedence. The last third
131  * of the requests are only for reads.
132  */
// Ok, now we have to generate and add read/write request items for this function. First we
// need to find a free item (slot) in the request array to store the new request item. The search
// process begins at the end of the request array. According to the above requirements, for
// the read command request, we start the search directly from the end of the queue, and for
// the write request, we can only fill in the empty entry from the queue 2/3 to the head of
// the queue. So we start searching from the back. When the device field dev of the request
// structure request is -1, it means that the item is idle. If none of the items are free, that
// is, the request item array has been searched over the header, then check whether the request
// is read/write in advance (READA or WRITEA), and if so, discard the request operation.
// Otherwise, let the request operation sleep first (to wait for the queue to free up the empty
// item), and then search the request queue after a while.
133     if (rw == READ)
134         req = request+NR\_REQUEST;           // for read, search from the end.
135     else
136         req = request+((NR\_REQUEST*2)/3); // for write, search from 2/3 backward.
137 /* find an empty request */
138     while (--req >= request)
139         if (req->dev<0)
140             break;
141 /* if none found, sleep on new requests: check for rw_ahead */
142     if (req < request) {                     // no free item ...
143         if (rw_ahead) {                     // exit if it's read/write ahead.
144             unlock\_buffer(bh);
145             return;
146         }
147         sleep\_on(&wait\_for\_request);
148         goto repeat;                        // line 128.
149     }
150 /* fill up the request-info, and add it to the queue */

```

```

// OK, we have found an idle request. So after we set the new request, we call add_request()
// to add it to the request queue and exit immediately. See blk_drv/blk.h for the request
// structure, line 23. Where req->sector is the starting sector number of the read/write
// operation, and req->buffer is the buffer in which the request item stores data.
151     req->dev = bh->b_dev;           // device no.
152     req->cmd = rw;                 // command (READ/WRITE).
153     req->errors=0;                 // error count.
154     req->sector = bh->b_blocknr<<1; // start sector, (lblock = 2 sectors).
155     req->nr_sectors = 2;           // number of sectors read/written.
156     req->buffer = bh->b_data;       // data buffer location.
157     req->waiting = NULL;          // waiting list for the operation.
158     req->bh = bh;                  // buffer header.
159     req->next = NULL;             // points to next request.
160     add_request(major+blk_dev, req); // add_request(blk_dev[major], req).
161 }
162
////// Low-level page read and write function (Low-Level Read Write Page).
// The block device data is accessed in units of pages (4K), that is, 8 sectors are read/written
// each time. See the ll_rw_blk() function below.
163 void ll_rw_page(int rw, int dev, int page, char * buffer)
164 {
165     struct request * req;
166     unsigned int major = MAJOR(dev);
167
// First check the legality of the function parameters. If the device's major number does not
// exist or the device's request handling function does not exist, an error message is displayed
// and returned. If the command given by the parameter is neither READ nor WRITE, it means that
// the kernel program is faulty, displays an error message and stops the kernel.
168     if (major >= NR_BLK_DEV || !(blk_dev[major].request_fn)) {
169         printk("Trying to read nonexistent block-device\n\r");
170         return;
171     }
172     if (rw!=READ && rw!=WRITE)
173         panic("Bad block dev command, must be R/W");
// After the parameter check is complete, we now need to create a request for this operation.
// First we need to find a free item (slot) in the request array to hold the new request item.
// The search action begins at the end of the request array. So we started searching from the
// back. When the device field dev of the request structure is less than 0, it means that the
// item is idle. If none of the items are idle, let the request operation sleep first (to wait
// for the idle item), and then search the request queue after a while.
174 repeat:
175     req = request+NR_REQUEST;           // points to the end.
176     while (--req >= request)
177         if (req->dev<0)
178             break;
179     if (req < request) {
180         sleep_on(&wait_for_request);      // sleep at wait_for_request.
181         goto repeat;
182     }
183 /* fill up the request-info, and add it to the queue */
// OK, we have found an idle request. So we set up the new request item, put the current process
// into an uninterruptible sleep state, then call add_request() to add it to the request queue,
// and then directly call the scheduler to let the current process sleep, waiting page read

```

```

// from the switching device. Here, instead of exiting the function directly like the
// make_request(), schedule() is called here. This is because the make_request() only reads
// 2 sectors, but here it takes 8 sectors to read/write to the switching device, which takes
// a long time. So the current process definitely needs to wait and sleep. So let the process
// go to sleep directly, saving the need to perform these judgments elsewhere in the program.
184     req->dev = dev;                // device no.
185     req->cmd = rw;                // command.
186     req->errors = 0;              // error count.
187     req->sector = page<<3;        // starting sector.
188     req->nr_sectors = 8;          // nr of sectors read/written.
189     req->buffer = buffer;         // data buffer.
190     req->waiting = current;      // waiting queue.
191     req->bh = NULL;              // buffer header.
192     req->next = NULL;           // points to next request.
193     current->state = TASK_UNINTERRUPTIBLE;
194     add_request(major+blk_dev, req); // add to request queue.
195     schedule();
196 }
197
//// Low-level data block read and write functions (Low Level Read Write Block).
// This function is an interface between the block device driver and the rest of the system.
// Usually called in the fs/buffer.c program. Its main purpose is to create block device read
// and write request items and insert them into the specified block device request queue. The
// actual read and write operations are done by the device's request_fn() function, and for
// hard disk, the function is do_hd_request(); for floppy the function is do_fd_request(); for
// virtual disks it is do_rd_request(). In addition, before calling this function, the caller
// needs to first save the information of the read/write block device in the buffer block header
// structure, such as the device number and block number. Parameters: rw - is the command READ,
// READA, WRITE or WRITEA; bh - data buffer block header pointer.
198 void ll_rw_block(int rw, struct buffer_head * bh)
199 {
200     unsigned int major;           // major device no ( 3 for hard disk).
201
// If the device's major number does not exist or the device's request action function does
// not exist, an error message is displayed and returned. Otherwise create a request and insert
// into the request queue.
202     if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
203         !(blk_dev[major].request_fn)) {
204         printk("Trying to read nonexistent block-device\n\r");
205         return;
206     }
207     make_request(major, rw, bh);
208 }
209
//// The block device initialization function, called by the initialization program main.c.
// Initializes the request array and sets all request items as free items (dev = -1). There
// are 32 items (NR_REQUEST = 32).
210 void blk_dev_init(void)
211 {
212     int i;
213
214     for (i=0 ; i<NR_REQUEST ; i++) {
215         request[i].dev = -1;

```



```

216         request[i].next = NULL;
217     }
218 }
219

```

## 9.5 ramdisk.c

### 9.5.1 Function

This file is a virtual Ram Disk driver created by Theodore Ts'o. A ram disk device is a way to use physical memory to simulate the actual disk storage data. Its purpose is mainly to improve the speed of reading and writing of "disk" data. In addition to taking up some valuable memory resources, the main disadvantage is that once the system is shut down or crashes, all the data in the virtual disk will disappear. Therefore, the virtual disk usually only stores some common tool programs or temporary data such as system commands, rather than important input documents.

When the symbol RAMDISK is defined in the linux/Makefile, the kernel initializer draws a memory area of a specified size in memory for the virtual disk data. The virtual ram disk capacity is equal to the value of RAMDISK (KB). If RAMDISK=512, the virtual disk size is 512KB. The specific location of the virtual disk in physical memory is determined during the kernel initialization phase (init/main.c, line 150), which is located between the kernel cache and the main memory area. If the running machine contains 16MB of physical memory, the kernel code will set the virtual ram disk area at the beginning of 4MB of memory. At this time, the memory allocation is shown in Figure 9-6.

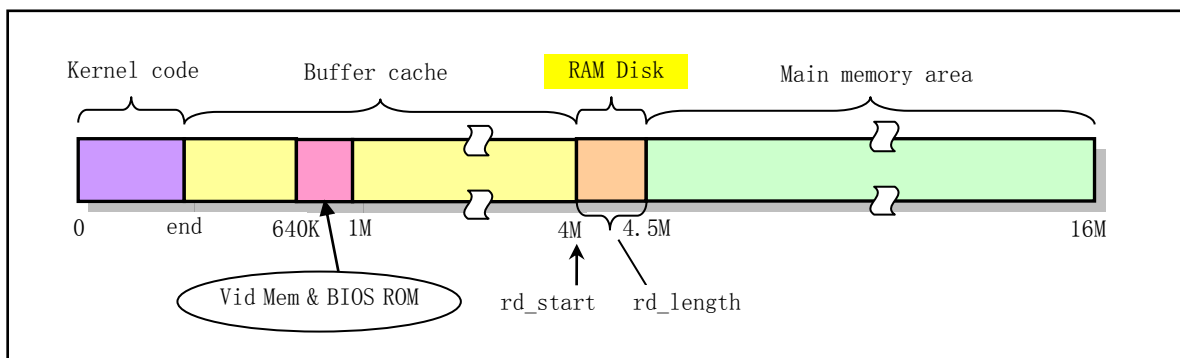


Figure 9-6 The specific location of the ram disk in the 16MB memory system

The read and write access operations to the ram disk are in principle the same as those for ordinary disks, and they need to be operated according to the access mode of the block device. Since the implementation does not involve synchronization with an external controller or device, its implementation is relatively simple. For data transfer between the system and the device, it is only necessary to perform an in-memory block copy operation.

The ramdisk.c file contains 3 functions. The rd\_init() function is called by the init/main.c program during system initialization to determine the specific location and size of the virtual disk in physical memory; do\_rd\_request() is the request function of the virtual disk device to implement virtual disk data access operation; rd\_load() is a virtual disk root file system load function. During the system initialization phase, the rd\_load() function is used to attempt to load a root file system into the virtual disk from the disk block location specified

on the boot disk. In this function, this starting boot disk block location is set to 256. Of course, you can also modify this value according to your specific requirements, as long as the disk capacity specified by this value can accommodate the kernel image file. In this way, a "two-in-one" disk composed of a kernel boot image file (Bootimage) plus a root file system image file (Rootimage) can boot a Linux system just like a DOS system disk. The experimental creation of this type of combination disk (integrated disk) can be found in Chapter 17.

Before the root file system image is loaded from disk using the normal method, the system first executes the `rd_load()` function, trying to read the root file system super block from block 257 of the disk. If successful, the root file image file is read into the memory virtual disk, and the root file system device flag `ROOT_DEV` is set to the virtual disk device (0x0101). Otherwise, `rd_load()` is exited, and the system continues to load the root file system from other devices in the normal way. The operation process of loading the root file system to the ram disk is shown in Figure 9-7.

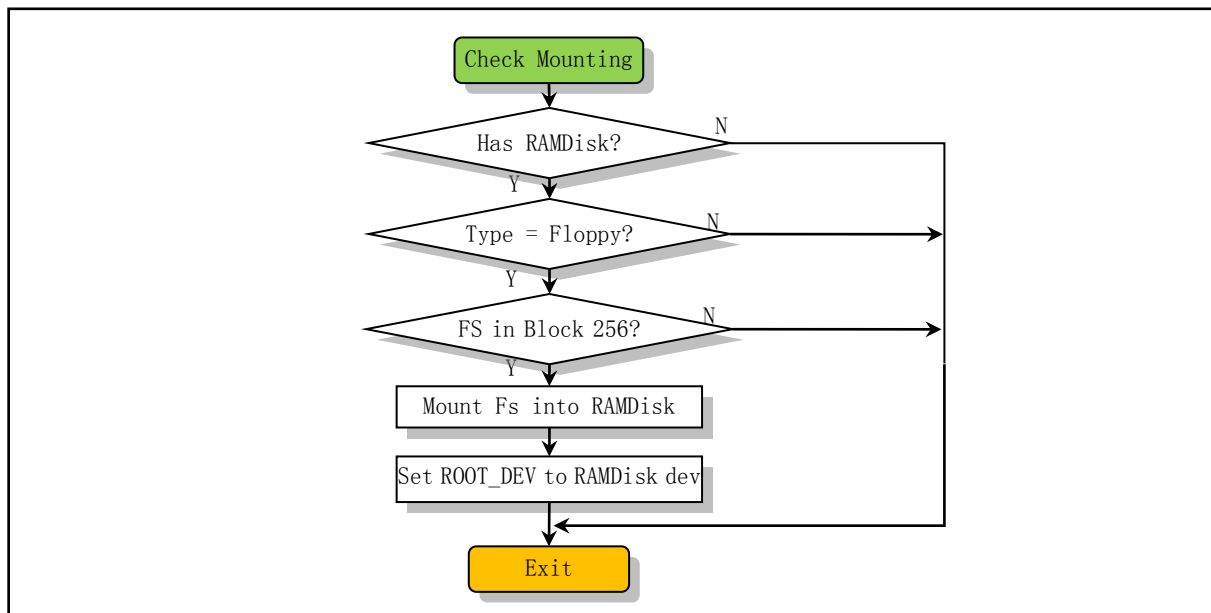


Figure 9-7 Flowchart for loading the root file system to the ram disk

If the symbol `RAMDISK` and its size are defined in the `linux/Makefile` configuration file when compiling the Linux 0.12 kernel source code, then after booting and initializing the `RAMDISK` area, it will first try to check the location at the 256th disk block on the disk, Is there a root file system? The detection method is to determine whether there is a valid file system super block in the 257th disk block. If so, the file system is loaded into the `RAMDISK` area in memory and used as the root file system. So we can use a boot disk that integrates the root file system to boot the system to the shell command prompt. If a valid root file system is not stored at the specified disk block location (256th disk block) on the boot disk, the kernel will prompt to insert the root file system disk. After the user presses the Enter key to confirm, the kernel will read the root file system on a separate disk into the virtual disk area for execution.

On a 1.44MB kernel boot boot disk, put a basic root file system at the beginning of the 256th block on the disk, you can combine to form an integrated disk, the layout is shown in Figure 9-8.

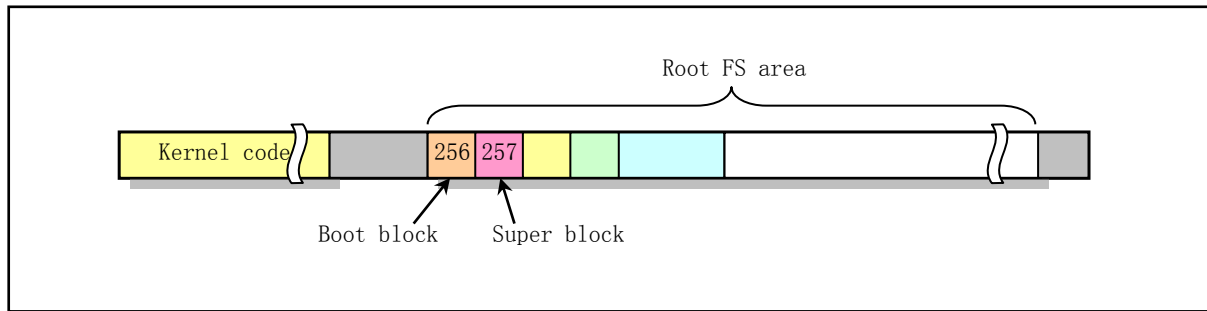


Figure 9-8 Data block layout on the integrated disk

## 9.5.2 Code Annotation

Program 9-4 linux/kernel/blk\_drv/ramdisk.c

```

1  /*
2  *  linux/kernel/blk_drv/ramdisk.c
3  *
4  *  Written by Theodore Ts'o, 12/2/91
5  */
// Theodore Ts'o (Ted Ts'o) is a famous figure in the Linux community. The popularity of Linux
// in the world also has his great contribution. As early as the Linux operating system came
// out, he provided a maillist service for the development of Linux with great enthusiasm, and
// established a Linux ftp server site (tsx-11.mit.edu) in North America. One of his biggest
// contributions to Linux was to propose and implement the ext2 file system. This file system
// has become the de facto file system standard in the Linux world. Recently, he introduced
// the ext3 and ext4 file systems, which greatly improved the stability, recoverability and
// access efficiency of the file system. To pay tribute to him, the Linux Journal's 97th issue
// (May 2002) interviewed him and used him as a cover person. Now he is a staff engineer working
// at Google, where he still work on file system and storage. His homepage:thunk.org/tytso/
6
// <string.h> String header file. Defines some embedded functions about string operations.
// <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk
// type (HD_TYPE) options.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
// of the initial task 0, and some embedded assembly function macro statements about the
// descriptor parameter settings and acquisition.
// <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,
// m_inode, etc.).
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
// used functions of the kernel.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
// descriptors/interrupt gates, etc. is defined.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
// segment register operations.
// <asm/memory.h> Memory copy header file. Contains memcpy() embedded assembly macro functions.
7 #include <string.h>
8
9 #include <linux/config.h>
10 #include <linux/sched.h>
11 #include <linux/fs.h>
12 #include <linux/kernel.h>

```

```

13 #include <asm/system.h>
14 #include <asm/segment.h>
15 #include <asm/memory.h>
16
    // Define the RAM disk major number symbol constant. The major device number must be defined
    // in the driver before the blk.h file is included, as this symbolic constant value is used
    // in the blk.h file to determine a range of other constant symbols and macros.
17 #define MAJOR_NR 1
18 #include "blk.h"
19
    // The starting position of the virtual disk in memory, which is determined in the initialization
    // function rd_init() on line 52. See kernel initialization program init/main.c, line 151.
20 char    *rd_start;                // the starting address of the ram disk in memory.
21 int     rd_length = 0;            // memory size (in bytes) occupied by the ram disk.
22
    // The ram disk current request operation function.
    // The structure of this function is similar to the do_hd_request() of the hard disk driver
    // (see hd.c, line 330). After the low-level block device interface function ll_rw_block()
    // establishes the request item of the ram disk (rd) and adds it to the linked list of rd, this
    // function is called to process the current request item of rd. The function first calculates
    // the starting addr of the memory corresponding to the ram disk of the starting sector specified
    // in the current request item and the bytes length len corresponding to the required number
    // of sectors, and then operates according to the command in the request item. If the command
    // is WRITE, the data in the buffer pointed to by the request is directly copied to the memory
    // location addr. If it is a read operation, the reverse is true. After the data is copied,
    // you can directly call end_request() to end the request. Then jump to the beginning of the
    // function and then process the next request item. Exit if there is no request left.
23 void do_rd_request(void)
24 {
25     int     len;
26     char    *addr;
27
    // First check the legality of the request item, and exit if there is no request item (see blk.h,
    // line 148). Then calculate the address addr corresponding to the starting sector in the virtual
    // disk and the occupied memory byte length len. The following sentence is used to obtain the
    // memory start position and memory length corresponding to the starting sector in the request
    // item, where 'sector << 9 represents sector * 512', which is converted into bytes. CURRENT
    // is defined as '(blk_dev[MAJOR_NR].current_request)' in blk.h.
28     INIT_REQUEST;
29     addr = rd_start + (CURRENT->sector << 9);
30     len = CURRENT->nr_sectors << 9;
    // If the minor-device number in the current request is not 1 or the corresponding memory start
    // position is greater than the end of the ram disk, the request item is ended, and the jump
    // to the repeat is performed to process the next virtual disk request item. The label 'repeat'
    // is defined in the macro INIT_REQUEST, see line 149 of the blk.h file.
31     if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd_start+rd_length)) {
32         end_request(0);
33         goto repeat;
34     }
    // Then perform the actual read and write operations. If it is a write command (WRITE), the
    // contents of the buffer in the request are copied to the address 'addr', and the length is
    // 'len' bytes. If it is a read command (READ), the memory content started by 'addr' is copied
    // into the request item buffer, and the length is 'len' bytes. Otherwise it will print that

```

```

// the command does not exist and crashes.
35     if (CURRENT->cmd == WRITE) {
36         (void) memcpy(addr,
37                        CURRENT->buffer,
38                        len);
39     } else if (CURRENT->cmd == READ) {
40         (void) memcpy(CURRENT->buffer,
41                        addr,
42                        len);
43     } else
44         panic("unknown ramdisk-command");
// Then, after the request item is successfully processed, the update flag is set, and the next
// request item of the device is processed.
45     end_request(1);
46     goto repeat;
47 }
48
49 /*
50  * Returns amount of memory which needs to be reserved.
51  */
// Virtual ram disk initialization function.
// The function first sets the request handler pointer of the virtual disk to point to
// do_rd_request(), then determines the starting address, byte length of the virtual disk in
// physical memory, and clears the entire virtual extent. Finally return the length of the ram
// disk. When the RAMDISK value is set to non-zero in the linux/Makefile, it means that the
// virtual ram disk device will be created in the system. In this case, the kernel initialization
// process calls this function (init/main.c, line 151). The second parameter 'length' is assigned
// to RAMDISK * 1024 in bytes.
52 long rd_init(long mem_start, int length)
53 {
54     int    i;
55     char   *cp;
56
57     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST; // do_rd_request()
58     rd_start = (char *) mem_start; // 4MB for the 16MB machine.
59     rd_length = length; // size of the ram disk.
60     cp = rd_start;
61     for (i=0; i < length; i++) // cleared.
62         *cp++ = '\0';
63     return(length);
64 }
65
66 /*
67  * If the root device is the ram disk, try to load it.
68  * In order to do this, the root device is originally set to the
69  * floppy, and we later change it to be ram disk.
70  */
///// Try to load the root file system into the ram disk.
// This function will be called in the kernel setup function setup() (hd.c, line 162). The
// variable 'block=256' on line 75 indicates that the root file system image file is located
// at the beginning of the 256th disk block on the boot disk. (1 disk block = 1024 bytes).
71 void rd_load(void)
72 {

```

```

73     struct buffer head *bh;           // cache buffer head pointer.
74     struct super block      s;
75     int          block = 256;    /* Start at block 256 */
76     int          i = 1;
77     int          nblocks;       // The amount of file system disk blocks.
78     char         *cp;          /* Move pointer */
79
// First check the validity and integrity of the ram disk. If the length of the ramdisk is zero,
// then exit. Otherwise, the size of the ramdisk and the starting position of the memory are
// displayed. If the root file device is not a floppy device at this time, it also exits.
80     if (!rd\_length)
81         return;
82     printk("Ram disk: %d bytes, starting at 0x%x\n", rd\_length,
83          (int) rd\_start);
84     if (MAJOR(ROOT\_DEV) != 2)
85         return;
// Then read the basic parameters of the root file system, that is, read the floppy disk blocks
// 256+1, 256 and 256+2. Here block+1 refers to the super block of the root file system in the
// virtual disk. The function breada() is used to read the specified data block from the floppy
// disk, mark the block that still needs to be read, and then return the buffer pointer containing
// the data block (fs/buffer.c, line 322). Then copy the disk superblock in the buffer to the
// s variable (d_super_block is the superblock structure) and release the buffer. Then we begin
// to check the validity of the super block. If the fs magic number in the super block is
// incorrect, it means that the loaded data block is not the MINIX file system, so it exits.
// See the File System chapter for the structure of the MINIX superblock.
86     bh = breada(ROOT\_DEV, block+1, block, block+2, -1);
87     if (!bh) {
88         printk("Disk error while looking for ramdisk!\n");
89         return;
90     }
91     *((struct d super block *) &s) = *((struct d super block *) bh->b_data);
92     brelse(bh);
93     if (s.s_magic != SUPER\_MAGIC)
94         /* No ram disk image present, assume normal floppy boot */
95         return;
// Then we try to read the entire root file system into the memory virtual disk extent. For
// a file system, the amount of logical blocks (or number of zones) is stored in the s_nzones
// field of the superblock structure. The number of data blocks contained in a logical block
// is specified by the field s_log_zone_size. Therefore, the total number of data blocks nblocks
// in a file system is equal to (the amount of logical blocks * 2^ (the power of each block)),
// that is, nblocks = (s_nzones * 2^s_log_zone_size). If the amount of data blocks in the file
// system is greater than the number of blocks that the ram disk can hold, the load operation
// cannot be performed, but only the error message is displayed and returned.
96     nblocks = s.s_nzones << s.s_log_zone_size;
97     if (nblocks > (rd\_length >> BLOCK\_SIZE\_BITS)) {
98         printk("Ram disk image too big! (%d blocks, %d avail)\n",
99              nblocks, rd\_length >> BLOCK\_SIZE\_BITS);
100         return;
101     }
// Otherwise, if the virtual disk can hold the total number of file system blocks, we display
// the load block information and let 'cp' point to the beginning of the virtual disk in memory.
// Then start the loop operation to load the root file system image file on the floppy disk
// to the ram disk. In the course of operation, if the number of disk blocks that need to be

```

```
// loaded at one time is greater than 2, we use the advanced read-ahead function breada(),
// otherwise we use the bread() function for single-block reading. If an I/O error occurs during
// the disk reading process, you can only abandon the loading process and return. The read disk
// block is copied from the cache to the corresponding location of the memory virtual disk using
// the memcpy() function, and the number of loaded blocks is displayed. The octal number '\010'
// in the display string indicates that a tab is displayed.
102     printk("Loading %d bytes into ram disk... 0000k",
103           nblocks << BLOCK_SIZE_BITS);
104     cp = rd_start;
105     while (nblocks) {
106         if (nblocks > 2)                                // need read ahead ?
107             bh = breada(ROOT_DEV, block, block+1, block+2, -1);
108         else                                              // read one block each time.
109             bh = bread(ROOT_DEV, block);
110         if (!bh) {
111             printk("I/O error on block %d, aborting load\n",
112                   block);
113             return;
114         }
115         (void) memcpy(cp, bh->b_data, BLOCK_SIZE);      // copy to location cp.
116         brelse(bh);
117         printk("\010\010\010\010\010\010%4dk", i);        // nr of blocks loaded.
118         cp += BLOCK_SIZE;                                // next disk block.
119         block++;
120         nblocks--;
121         i++;
122     }
// When the entire root file system starting from the 256 disk block in the boot disk is loaded,
// we display "done", and the current root fs device number is changed to the virtual disk
// device number 0x0101, and finally returned.
123     printk("\010\010\010\010\010\010done \n");
124     ROOT_DEV=0x0101;
125 }
126
```

---

## 9.6 floppy.c

### 9.6.1 Function description

This program is a floppy disk controller driver. Like other block device drivers, the program also uses the request item operation function (`do_fd_request()` for the floppy disk drive) to perform read and write operations on the floppy disk. The main difference from the hard disk driver is that the floppy disk driver uses more timing functions and operations.

Considering that the floppy disk drive does not normally rotate when it is not working, we need to wait for the motor of the drive to start and reach the normal operating speed before the actual floppy disk can be read or written. Compared to the speed of the computer, this period of time is very long, usually takes about 0.5 seconds. In addition, when reading and writing to a disk is completed, we also need to stop the drive motor to reduce the head's friction on the disk surface. But we can't stop it after the disk is finished, because it may need to be read

and written right away. Therefore, after a drive has not been operated, it is necessary to idle the drive motor for a period of time to wait for possible read and write operations. If the drive does not operate for a long time, the program stops it from rotating. The time to maintain the rotation can be set to about 3 seconds. Furthermore, when a disk read/write operation fails, or some other condition causes a drive's motor to not be turned off, we also need to have the system automatically turn it off after a certain amount of time. The Linux kernel sets this delay value to 100 seconds.

It can be seen that many delay (timer) operations are used when operating the floppy disk drive, so more timing processing functions are involved in the driver. There are also several functions that are closely related to the timer are placed in kernel/sched.c (lines 215-281). This is the biggest difference between a floppy disk driver and a hard disk driver, and it is also a reason why a floppy disk driver is more complicated than a hard disk driver.

Although the program is more complicated, the working principle of floppy disk read/write operations is the same as other block devices. The program also uses the request item and the linked list structure of the request to handle all read/write operations to the floppy disk, so the request item function `do_fd_request()` is still one of the important functions in the program. This function can be expanded as a main line while reading. In addition, the programming and operation of the floppy disk controller is complicated, involving many controller execution states and flags. Therefore, you need to refer to the instructions behind the program and the header file `include/linux/fdreg.h`. This header file defines all floppy controller parameter constants and explains the meaning of these constants.

## 9.6.2 Code Annotation

Program 9-5 linux/kernel/blk\_drv/floppy.c

---

```
1  /*
2  *  linux/kernel/floppy.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  02.12.91 - Changed to static variables to indicate need for reset
9  *  and recalibrate. This makes some things easier (output_byte reset
10 *  checking etc), and means less interrupt jumping in case of errors,
11 *  so the code is hopefully easier to understand.
12 */
13
14 /*
15 *  This file is certainly a mess. I've tried my best to get it working,
16 *  but I don't like programming floppies, and I have only one anyway.
17 *  Urgel. I should check for more errors, and do more graceful error
18 *  recovery. Seems there are problems with several drives. I've tried to
19 *  correct them. No promises.
20 */
21
22 /*
23 *  As with hd.c, all routines within this file can (and will) be called
24 *  by interrupts, so extreme caution is needed. A hardware interrupt
25 *  handler may not sleep, or a kernel panic will happen. Thus I cannot
26 *  call "floppy-on" directly, but have to set a special timer interrupt
```



```

27  * etc.
28  *
29  * Also, I'm not certain this works on more than 1 floppy. Bugs may
30  * abound.
31  */
32
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the
//      data of the initial task 0, and some embedded assembly function macro statements
//      about the descriptor parameter settings and acquisition.
// <linux/fs.h> File system header file. Define the file table structure (file,
//      buffer_head, m_inode, etc.).
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//      commonly used functions of the kernel.
// <linux/fdreg.h> Floppy disk file. Contains some definitions of floppy disk controller
//      parameters.
// <asm/system.h> System header file. An embedded assembly macro that defines or
//      modifies descriptors/interrupt gates, etc. is defined.
// <asm/io.h> Io header file. Defines the function that operates on the io port in the
//      form of a macro's embedded assembler.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//      for segment register operations.
33 #include <linux/sched.h>
34 #include <linux/fs.h>
35 #include <linux/kernel.h>
36 #include <linux/fdreg.h>
37 #include <asm/system.h>
38 #include <asm/io.h>
39 #include <asm/segment.h>
40
// Define the floppy drive major device number symbol. In the driver, the major device number
// must be defined before the blk.h file is included, because this symbolic constant is used
// in the blk.h file to determine some other related symbol constants and macros.
41 #define MAJOR_NR 2 // floppy drive major number.
42 #include "blk.h" // block dev header file. requests, queues are defined.
43
44 static int recalibrate = 0; // flag: recalibrate head position (return to zero).
45 static int reset = 0; // flag: reset operation needed.
46 static int seek = 0; // flag: perform a seek operation.
47
// The current digital output register (DOR), defined in kernel/sched.c, line 223, the default
// value is 0x0C. This variable contains important flags from the floppy drive operation,
// including select floppy drive, control motor start, start reset floppy disk controller, and
// enable/disable DMA and interrupt requests. See the description of the DOR register after
// the program listing.
48 extern unsigned char current_DOR;
49
// Byte direct output (inline assembly macro). Output the value 'val' to the port.
50 #define immoutb_p(val,port) \
51 __asm__( "outb %0,%1\n\tjmp 1f\n1:\tjmp 1f\n1::\"a\" ((char) (val)), \"i\" (port))
52
// These two macros are defined to calculate the device number of the floppy drive. The parameter
// x is the minor device number. minor device number = TYPE*4 + DRIVE. The calculation method
// is shown after the program listing.

```

```

53 #define TYPE(x) ((x)>>2)          // Type of floppy drive (2--1.2Mb, 7--1.44Mb).
54 #define DRIVE(x) ((x)&0x03)       // The floppy drive number (0--3 corresponds to A--D).
55 /*
56  * Note that MAX_ERRORS=8 doesn't imply that we retry every bad read
57  * max 8 times - some types of errors increase the errorcount by 2,
58  * so we might actually retry only 5-6 times before giving up.
59  */
60 #define MAX_ERRORS 8
61
62 /*
63  * globals used by 'result()'
64  */
65 // See the include/linux/fdreg.h header file for the meaning of the bits in these status bytes.
66 // See also the instructions at the end of the program list.
67 #define MAX_REPLIES 7              // FDC returns up to 7 bytes of results.
68 static unsigned char reply_buffer[MAX_REPLIES]; // used to store the response results.
69 #define ST0 (reply_buffer[0])      // result status byte 0.
70 #define ST1 (reply_buffer[1])      // result status byte 1.
71 #define ST2 (reply_buffer[2])      // result status byte 2.
72 #define ST3 (reply_buffer[3])      // result status byte 3.
73
74 /*
75  * This struct defines the different floppy types. Unlike minix
76  * linux doesn't have a "search for right type"-type, as the code
77  * for that is convoluted and weird. I've got enough problems with
78  * this driver as it is.
79  *
80  * The 'stretch' tells if the tracks need to be boubled for some
81  * types (ie 360kB diskette in 1.2MB drive etc). Others should
82  * be self-explanatory.
83  */
84 // Define the floppy drvie data structure. The floppy disk parameters are:
85 // size          number of sectors;
86 // sect          sectors per track;
87 // head          number of heads;
88 // track          number of tracks;
89 // stretch      flag, if the tracks need to be handled specifically;
90 // gap           Sector gap length (bytes);
91 // rate          Data transfer rate;
92 // spec1         Parameters (high 4-bit step rate, low 4-bit head unloading time).
93 static struct floppy_struct {
94     unsigned int size, sect, head, track, stretch;
95     unsigned char gap, rate, spec1;
96 } floppy_type[] = {
97     { 0, 0, 0, 0, 0, 0x00, 0x00, 0x00 }, /* no testing */
98     { 720, 9, 2, 40, 0, 0x2A, 0x02, 0xDF }, /* 360kB PC diskettes */
99     { 2400, 15, 2, 80, 0, 0x1B, 0x00, 0xDF }, /* 1.2 MB AT-diskettes */
100    { 720, 9, 2, 40, 1, 0x2A, 0x02, 0xDF }, /* 360kB in 720kB drive */
101    { 1440, 9, 2, 80, 0, 0x2A, 0x02, 0xDF }, /* 3.5" 720kB diskette */
102    { 720, 9, 2, 40, 1, 0x23, 0x01, 0xDF }, /* 360kB in 1.2MB drive */
103    { 1440, 9, 2, 80, 0, 0x23, 0x01, 0xDF }, /* 720kB in 1.2MB drive */
104    { 2880, 18, 2, 80, 0, 0x1B, 0x00, 0xCF }, /* 1.44MB diskette */
105 };

```

```

95
96 /*
97  * Rate is 0 for 500kb/s, 2 for 300kbps, 1 for 250kbps
98  * Spec1 is 0xSH, where S is stepping rate (F=1ms, E=2ms, D=3ms etc),
99  * H is head unload time (1=16ms, 2=32ms, etc)
100  *
101  * Spec2 is (HLD<<1 | ND), where HLD is head load time (1=2ms, 2=4 ms etc)
102  * and ND is set means no DMA. Hardcoded to 6 (HLD=6ms, use DMA).
103  */
104
    // floppy_interrupt is the floppy drive interrupt handler label in the kernel/sys_call.s
    // program. It will be used in the floppy disk initialization function floppy_init() (line 469)
    // to initialize the interrupt trap gate descriptor.
105 extern void floppy_interrupt(void);
    // This is the temporary floppy buffer defined at line 132 of boot/head.s. If the buffer of
    // the request item is somewhere above 1MB in memory, you need to set the DMA buffer at the
    // temporary buffer area. Because the 8237A chip can only be addressed within the 1MB address
    // range.
106 extern char tmp_floppy_area[1024];
107
108 /*
109  * These are global variables, as that's the easiest way to give
110  * information to interrupts. They are the data used for the current
111  * request.
112  */
    // These so-called "global variables" refer to the variables used by the C functions called
    // in the floppy disk interrupt handler. Of course, these C functions are all within the program.
113 static int cur_spec1 = -1;                // current spec1.
114 static int cur_rate = -1;
115 static struct floppy_struct * floppy = floppy_type; // floppy point to the floppy_type[].
116 static unsigned char current_drive = 0;
117 static unsigned char sector = 0;
118 static unsigned char head = 0;
119 static unsigned char track = 0;
120 static unsigned char seek_track = 0;
121 static unsigned char current_track = 255;
122 static unsigned char command = 0;        // read/write command.
123 unsigned char selected = 0;              // drive selected flag.
124 struct task_struct * wait_on_floppy_select = NULL; // wait floppy queue.
125
    //// Deselect a floppy drive.
    // If the floppy drive nr specified by the function parameter is not currently selected, a warning
    // message is displayed. Then reset the floppy drive selected flag and wake up the task waiting
    // to select the floppy drive. The lower 2 bits of the Digital Output Register (DOR) are used
    // to specify the selected floppy drive (0-3 to A-D).
126 void floppy_deselect(unsigned int nr)
127 {
128     if (nr != (current_DOR & 3))
129         printk("floppy_deselect: drive not selected\n|r");
130     selected = 0;
131     wake_up(&wait_on_floppy_select);
132 }
133

```

```

134 /*
135  * floppy-change is never called from an interrupt, so we can relax a bit
136  * here, sleep etc. Note that floppy-on tries to set current_DOR to point
137  * to the desired drive, but it will probably not survive the sleep if
138  * several floppies are used at the same time: thus the loop.
139  */
140 // Check the floppy disk replacement in the specified floppy drive.
141 // The parameter 'nr' is the floppy drive number. Returns 1 if the floppy disk is replaced,
142 // otherwise returns 0. The function first selects the specified floppy drive 'nr' and then
143 // tests the controller's digital input register (DIR) to determine if the floppy disk in the
144 // drive has been replaced. This function is called by the check_disk_change() in the program
145 // fs/buffer.c (line 119).
146 int floppy_change(unsigned int nr)
147 {
148     // First, let the floppy disk in the floppy drive spin up and reach the normal working speed.
149     // This takes a certain amount of time. The method is to use the floppy timer function
150     // do_floppy_timer () (kernel / sched.c, line 264) for a certain delay processing. The
151     // floppy_on() function (sched.c, line 251) is used to determine if the delay has expired
152     // (mon_timer[nr]==0?). If not, let the current process continue to sleep. If the delay has
153     // expired, do_floppy_timer() will wake up the current process.
154     repeat:
155         floppy_on(nr); // Start and wait for the specified floppy drive nr.
156         // After the floppy disk is started (rotated), let's check to see if the currently selected
157         // floppy drive is drive 'nr' specified by the function parameter. If the currently selected
158         // floppy drive is not the specified floppy drive nr and other floppy drives have been selected,
159         // then the current task is put into an uninterruptible wait state, waiting for other floppy
160         // drives to be deselected, see floppy_deselect() above. If no other floppy drive is currently
161         // selected, or if the other floppy drive is deselected and the current floppy drive is still
162         // not the specified floppy drive nr when the current task is woken up, then jump to the beginning
163         // of the function and re-circulate.
164         while ((current_DOR & 3) != nr && selected)
165             sleep_on(&wait_on_floppy_select);
166         if ((current_DOR & 3) != nr)
167             goto repeat;
168         // Now the floppy controller has selected the floppy drive 'nr' we specified. The value of the
169         // digital input register DIR is then taken. If its highest bit (bit 7) is set, it means the
170         // floppy disk has been replaced, then the motor can be turned off and 1 will exit. Otherwise,
171         // the motor is turned off and 0 is exited, indicating that the disk has not been replaced.
172         if (inb(FD_DIR) & 0x80) {
173             floppy_off(nr);
174             return 1;
175         }
176         floppy_off(nr);
177         return 0;
178 }
179 // Copy 1024 bytes of data from the memory address 'from' to the address 'to'.
180 #define copy_buffer(from, to) \
181 __asm__( "cld ; rep ; movsl" \
182         : : "c" (BLOCK_SIZE/4), "S" ((long)(from)), "D" ((long)(to)) \
183         : "cx", "di", "si" )
184 // Setup (initialize) the floppy disk DMA channel.

```

```

// Data access operations on floppy disk are performed using DMA. Therefore, it is necessary
// to set the channel 2 dedicated to floppy drive on the DMA chip before each data transmission.
// See the information after the program list for the DMA programming method.
161 static void setup\_DMA(void)
162 {
163     long addr = (long) CURRENT->buffer;    // current request buffer address.
164
165     // First check the location of the buffer for the request item. If the buffer is somewhere above
166     // 1MB in memory, you need to set the DMA buffer in the temporary buffer area (tmp_floppy_area).
167     // Because the 8237A chip can only be addressed within the 1MB address range. If it is a write
168     // disk command, you also need to copy the data from the request item buffer to the temporary
169     // area.
170     cli();
171     if (addr >= 0x100000) {
172         addr = (long) tmp\_floppy\_area;
173         if (command == FD\_WRITE)
174             copy\_buffer(CURRENT->buffer, tmp\_floppy\_area);
175     }
176
177     // Next we start to set up DMA channel 2, but you need to mask the channel before you start
178     // setting up. The single channel mask register port is 10. Bits 0-1 specify the DMA channel
179     // (0-3), bits 2:1 indicate masking, and 0 indicates that the request is allowed. The mode
180     // word is then written to DMA controller ports 12 and 11 (read is 0x46 and write is 0x4A).
181     // Write the buffer address 'addr' and the number of bytes to be transferred 0x3ff (0-1023).
182     // Finally, the mask of DMA channel 2 is reset, and the DREQ signal requested by DMA2 is opened.
183
184     /* mask DMA 2 */
185     immoutb\_p(4|2, 10);    // port 10
186     /* output command byte. I don't know why, but everyone (minix, */
187     /* sanches & canton) output this twice, first to 12 then to 11 */
188     // The following inline assembly code writes the mode word to the "clear sequence trigger" port
189     // 12 and mode register port 11 of the DMA controller (0x46 when the disk is read and 0x4A when
190     // the disk is written).
191     // Since the address and count registers of each channel are 16 bits, it is necessary to operate
192     // in two steps when setting them, one low byte and one high byte. Which byte is actually written
193     // is determined by the state of the trigger. When the trigger is 0, the low byte is accessed;
194     // when the trigger is 1, the high byte is accessed. The state of the trigger changes once per
195     // visit. Writing to port 12 sets the flip-flop to a 0 state, so that the setting of the 16-bit
196     // register starts from the low byte.
197     __asm__ ("outb %a1, $12\n\tjmp 1f\n1:\tjmp 1f\n1:\t"
198             "outb %a1, $11\n\tjmp 1f\n1:\tjmp 1f\n1:":::
199             "a" ((char) ((command == FD\_READ)?DMA_READ:DMA_WRITE)));
200     /* 8 low bits of addr */
201     // Write the base/current address register (port 4) to DMA channel 2.
202     immoutb\_p(addr, 4);
203     addr >>= 8;
204     /* bits 8-15 of addr */
205     immoutb\_p(addr, 4);
206     addr >>= 8;
207     /* bits 16-19 of addr */
208     // The DMA can only be addressed in 1MB of memory, and its upper 16-19 bits are placed in the
209     // page register (port 0x81).
210     immoutb\_p(addr, 0x81);
211     /* low 8 bits of count-1 (1024-1=0x3ff) */
212     // Write the base/current byte counter value (port 5) to DMA channel 2.

```

```

187     immoutb\_p(0xff, 5);
188     /* high 8 bits of count-1 */
189     // A total of 1024 bytes (two sectors) are transmitted at a time.
190     immoutb\_p(3, 5);
191     /* activate DMA 2 */
192     immoutb\_p(0|2, 10);
193     sti();
194 }
195
196     //// Output a byte command or parameter to the floppy drive controller.
197     // Before sending a byte to the controller, the controller needs to be in a ready state, and
198     // the data transfer direction must be set from CPU to FDC, so the function needs to read the
199     // controller state information first. The loop query method is used here for proper delay.
200     // If an error occurs, the reset flag is set.
201     static void output\_byte(char byte)
202     {
203         int counter;
204         unsigned char status;
205
206         // First, the state of the main state controller FD_STATUS (0x3f4) is cyclically read. If the
207         // read status is STATUS_READY and the direction bit STATUS_DIR = 0 (CPU  $\diamond$  FDC), the specified
208         // byte is output to the data port.
209         if (reset)
210             return;
211         for(counter = 0 ; counter < 10000 ; counter++) {
212             status = inb\_p(FD\_STATUS) & (STATUS\_READY | STATUS\_DIR);
213             if (status == STATUS\_READY) {
214                 outb(byte, FD\_DATA);
215                 return;
216             }
217         }
218         // If it cannot be sent after the end of the cycle of 10,000 times, the reset flag is set and
219         // an error message is printed.
220         reset = 1;
221         printk("Unable to send byte to FDC\n|r");
222     }
223
224     //// Read the execution result information of the FDC.
225     // The result message is up to 7 bytes and is stored in the array reply_buffer[].Returns the
226     // number of result bytes read in. If the return value = -1, it indicates an error. The program
227     // is handled in a similar way to the above function.
228     static int result(void)
229     {
230         int i = 0, counter, status;
231
232         // If the reset flag is set, exit immediately to perform the reset operation in the subsequent
233         // program. Otherwise, the state of the main state controller FD_STATUS (0x3f4) is cyclically
234         // read. If the read controller status is READY, indicating that no data is available, then
235         // the number of bytes i read is returned. If the controller status is: The direction flag is
236         // set (CPU  $\leftarrow$  FDC), ready, busy, it indicates that data is readable. The result data in the
237         // controller is then read into the response result array. The maximum number of bytes read
238         // is MAX_REPLIES(7).
239         if (reset)

```

```

218         return -1;
219     for (counter = 0 ; counter < 10000 ; counter++) {
220         status = inb_p(FD_STATUS) & (STATUS_DIR | STATUS_READY | STATUS_BUSY);
221         if (status == STATUS_READY)
222             return i;
223         if (status == (STATUS_DIR | STATUS_READY | STATUS_BUSY)) {
224             if (i >= MAX_REPLIES)
225                 break;
226             reply buffer[i++] = inb_p(FD_DATA);
227         }
228     }
    // If it cannot be read after the end of the cycle of 10,000 times, the reset flag is set and
    // an error message is printed.
229     reset = 1;
230     printk("Getstatus times out\n\r");
231     return -1;
232 }
233
    /// Floppy disk read/write error handling function.
    // This function determines the further action that needs to be taken based on the number of
    // floppy disk read and write errors. If the number of currently processed request errors is
    // greater than the specified maximum number of errors, MAX_ERRORS (8 times), no further
    // operational attempts are made for the current request. If the number of read/write errors
    // has exceeded MAX_ERRORS/2, then the floppy drive needs to be reset, so the reset flag is
    // set. Otherwise, if the number of errors is less than half of the maximum value, then only
    // the head position needs to be recalibrated, so the recalibration flag is set. The actual
    // reset and recalibration process will be performed in subsequent programs.
234 static void bad_flp_intr(void)
235 {
    // First increase the number of errors in the current request item by one. If the current request
    // item has more errors than the maximum allowed, the current floppy drive is deselected and
    // the request is terminated (the buffer contents are not updated).
236     CURRENT->errors++;
237     if (CURRENT->errors > MAX_ERRORS) {
238         floppy_deselect(current_drive);
239         end_request(0);
240     }
    // If the number of errors in the current request item is greater than half of the maximum number
    // of allowed errors, set the reset flag to reset the floppy drive afterwards, and try again.
    // Otherwise, the floppy drive needs to be recalibrated and try again.
241     if (CURRENT->errors > MAX_ERRORS/2)
242         reset = 1;
243     else
244         recalibrate = 1;
245 }
246
247 /*
248  * Ok, this interrupt is called after a DMA read/write has succeeded,
249  * so we check the results, and copy any buffers.
250  */
    /// The floppy disk read/write function called in the interrupt.
    // This function is called during the interrupt handling process that is initiated after the
    // floppy drive controller operation ends. The function first reads the status information of

```

```

// the operation result, and accordingly determines whether the operation has a problem and
// handles it accordingly. If the read/write operation is successful, then if the request is
// a read operation and its buffer is in memory above 1MB, then the data needs to be copied
// from the floppy temporary buffer to the buffer of the request.
251 static void rw\_interrupt(void)
252 {
// First read the result information of the FDC execution. If the number of returned result
// bytes is not equal to 7, or there is an error flag in status byte 0, 1, or 2, if a write
// protection error occurs, an error message is displayed, the current drive is released, and
// the current request is terminated. Otherwise, the error counting is performed, and then the
// floppy request item operation is continued. See the fdreg.h file for the meaning of the
// following states.
// ( 0xf8 = ST0_INTR | ST0_SE | ST0_ECE | ST0_NR )
// ( 0xbf = ST1_EOC | ST1_CRC | ST1_OR | ST1_ND | ST1_WP | ST1_MAM ), should be 0xb7
// ( 0x73 = ST2_CM | ST2_CRC | ST2_WC | ST2_BC | ST2_MAM )
253     if (result() != 7 || (ST0 & 0xf8) || (ST1 & 0xbf) || (ST2 & 0x73)) {
254         if (ST1 & 0x02) { // 0x02 = ST1_WP - Write Protected.
255             printk("Drive %d is write protected\n\r", current\_drive);
256             floppy\_deselect(current\_drive);
257             end\_request(0);
258         } else
259             bad\_flp\_intr();
260         do\_fd\_request();
261         return;
262     }
// If the buffer of the current request item is above the 1MB address, the content of the floppy
// disk read operation is still placed in the temporary buffer and needs to be copied into the
// buffer of the current request item. Finally release the current floppy drive (deselected)
// and execute the current request item end processing: wake up the process waiting for the
// request item, wake up the process waiting for the idle request item (if any), delete the
// request item from the linked list of requests. Then continue to perform other floppy request
// operations.
263     if (command == FD_READ && (unsigned long)(CURRENT->buffer) >= 0x100000)
264         copy\_buffer(tmp\_floppy\_area, CURRENT->buffer);
265     floppy\_deselect(current\_drive);
266     end\_request(1);
267     do\_fd\_request();
268 }
269
//// Set DMA channel 2 and output commands and parameters to the floppy disk controller
// (1-byte command + 0~7 byte parameter).
// If the reset flag is not set, then a floppy disk interrupt will be generated and the floppy
// disk interrupt handler will be executed after the function exits and the floppy disk controller
// performs the corresponding read/write operation.
270 inline void setup\_rw\_floppy(void)
271 {
272     setup\_DMA(); // Initialize the floppy disk DMA channel.
273     do\_floppy = rw\_interrupt; // set function called in the int.
274     output\_byte(command); // send command.
275     output\_byte(head<<2 | current\_drive); // param: head no + drive no.
276     output\_byte(track); // param: track no.
277     output\_byte(head); // param: head no.
278     output\_byte(sector); // param: start sector no.

```



```

279     output_byte(2);          /* sector size = 512 */
280     output_byte(floppy->sect); // param: sectors per track.
281     output_byte(floppy->gap);  // param: gap between sectors.
282     output_byte(0xFF);        /* sector size (0xff when n!=0 ?) */
    // If any of the above output_byte() operations fail, the reset flag is set. The reset processing
    // code in do_fd_request() is executed immediately.
283     if (reset)
284         do_fd_request();
285 }
286
287 /*
288  * This is the routine called after every seek (or recalibrate) interrupt
289  * from the floppy controller. Note that the "unexpected interrupt" routine
290  * also does a recalibrate, but doesn't come here.
291  */
    /// The C function called during the interrupt process after the seek operations.
    // First, the detection interrupt status command is sent, and the status information ST0 and
    // the track information of the head are obtained. If an error occurs, the error count detection
    // process is executed or the floppy operation request item is canceled. Otherwise, set the
    // current track variable according to the status information, then call the function
    // setup_rw_floppy() to set the DMA and output the read/write commands and parameters.
292 static void seek_interrupt(void)
293 {
    // The check interrupt status command is sent first to obtain the result of the seek operation
    // execution. This command takes no arguments. The returned result is two bytes: ST0 and the
    // current track number of the head. Then read the result information of the FDC execution.
    // If the number of returned result bytes is not equal to 2, or ST0 is not the end of the seek,
    // or the track on which the head is located (ST1) is not equal to the set track, an error has
    // occurred. Then, the error counting is processed, and then the execution of the floppy disk
    // request item or the execution of the reset processing is continued. Note that the sense
    // interrupt status command (FD_SENSEI) should return 2 result bytes, that is the result() return
    // value should equal 2.
294 /* sense drive status */
295     output_byte(FD_SENSEI);
296     if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {
297         bad_flp_intr();
298         do_fd_request();
299         return;
300     }
    // If the seek operation is successful, the floppy disk operation of the current request is
    // continued, that is, the command and parameters are sent to the floppy disk controller.
301     current_track = ST1;        // set current track.
302     setup_rw_floppy();          // set DMA, output floppy commands and parameters.
303 }
304
305 /*
306  * This routine is called when everything should be correctly set up
307  * for the transfer (ie floppy motor is on and the correct floppy is
308  * selected).
309  */
    /// Read/write data transfer function.
310 static void transfer(void)
311 {

```

```

// First check if the current drive parameter is the one of the specified drive. If not, send
// the set drive parameter command and the corresponding parameters (param1: high 4-bit step
// rate, low 4-bit head unloading time; param2: head loading time). Then it is judged whether
// the current data transmission rate is consistent with the specified drive, and if not, the
// rate of the specified floppy drive is sent to the data transmission rate control register
// (FD_DCR).
312     if (cur_spec1 != floppy->spec1) {           // check the current parameters.
313         cur_spec1 = floppy->spec1;
314         output_byte(FD SPECIFY);                // send set disk parameters command.
315         output_byte(cur_spec1);                  /* hut etc */
316         output_byte(6);                          /* Head load time =6ms, DMA */
317     }
318     if (cur_rate != floppy->rate)                // check current rate.
319         outb_p(cur_rate = floppy->rate, FD_DCR);
// If any of the above output_byte() operations fail, the reset flag will be set. So here we
// need to check the reset flag. If the reset is actually set, the reset processing code in
// do_fd_request() is executed immediately.
320     if (reset) {
321         do_fd_request();
322         return;
323     }
// If the seek flag is zero at this time (ie no seek is required), the DMA is set and the
// corresponding operation command and parameters are sent to the floppy disk controller and
// returned. Otherwise, the seek processing is performed, so the function called in the floppy
// interrupt is first set to the seek track function. If the starting track number is not equal
// to zero, the head seek command and parameters are sent. The parameter used is the global
// variable value set on line 112--121. If the starting track number seek_track is 0, a
// recalibration command is executed to return the head to the zero track.
324     if (!seek) {
325         setup_rw_floppy();                        // Send command & parameter block.
326         return;
327     }
328     do_floppy = seek_interrupt;                  // set invoked function.
329     if (seek_track) {                             // start track.
330         output_byte(FD SEEK);                     // send seek command.
331         output_byte(head<<2 | current_drive);    // param: head + current drive.
332         output_byte(seek_track);                 // param: track no.
333     } else {
334         output_byte(FD RECALIBRATE);              // send recalibrate command.
335         output_byte(head<<2 | current_drive);    // param: head + current drive.
336     }
// Similarly, if any of the above output_byte() operations fail, the reset flag will be set
// and the reset processing code in do_fd_request() is executed immediately.
337     if (reset)
338         do_fd_request();
339 }
340
341 /*
342  * Special case - used after a unexpected interrupt (or reset)
343  */
//// The floppy drive recalibration function called in the interrupt.
// The check interrupt status command (no parameter) is sent first. If the return result
// indicates an error, the reset flag is set, otherwise the recalibration flag is cleared. Then

```

```

// execute the floppy disk request item processing function to perform the corresponding
// operation. Note that the sense interrupt status command (FD_SENSEI) will return 2 result
// bytes, that is the result() return value should equal 2.
344 static void recal_interrupt(void)
345 {
346     output_byte(FD_SENSEI);           // send sense interrupt status cmd.
347     if (result()!=2 || (ST0 & 0xE0) == 0x60) // reset if there are errors
348         reset = 1;
349     else
350         recalibrate = 0;
351     do_fd_request();
352 }
353
//// The unexpected interrupt handling function called in the floppy interrupt.
// The check interrupt status command (no parameter) is sent first. If the return result
// indicates an error, the reset flag is set, otherwise the recalibration flag is set.
354 void unexpected_floppy_interrupt(void)
355 {
356     output_byte(FD_SENSEI);           // send sense interrupt status cmd.
357     if (result()!=2 || (ST0 & 0xE0) == 0x60) // reset if there are errors.
358         reset = 1;
359     else
360         recalibrate = 1;
361 }
362
//// The floppy disk recalibration function.
// The recalibration flag is first reset and a recalibration command and its parameters are
// sent to the floppy disk controller (FDC). When the controller executes the recalibration
// command, it calls the recal_interrupt() function in the floppy disk interrupt it raises.
363 static void recalibrate_floppy(void)
364 {
365     recalibrate = 0;
366     current_track = 0;
367     do_floppy = recal_interrupt;      // point to recal function.
368     output_byte(FD_RECALIBRATE);      // cmd: recalibrate.
369     output_byte(head<<2 | current_drive); // param: head no + drive no.
// Similarly, if any of the above output_byte() operations fail, the reset flag will be set
// and the reset processing code in do_fd_request() is executed immediately.
370     if (reset)
371         do_fd_request();
372 }
373
//// The floppy disk controller FDC reset handling function called in the interrupt.
// First send the sense interrupt status command (no parameters) and read the returned result
// byte. Then send the set floppy drive parameter command and its related parameters, and finally
// call the request processing function do_fd_request() again to perform the request item or
// error processing operation.
374 static void reset_interrupt(void)
375 {
376     output_byte(FD_SENSEI);           // send sense interrupt status cmd.
377     (void) result();
378     output_byte(FD_SPECIFY);          // send drive param setting cmd.
379     output_byte(cur_spec1);           /* hut etc */

```

```

380     output_byte(6);                /* Head load time =6ms, DMA */
381     do_fd_request();
382 }
383
384 /*
385  * reset is done by pulling bit 2 of DOR low for a while.
386  */
387 static void reset_floppy(void)
388 {
389     int i;
390
391     reset = 0;
392     cur_spec1 = -1;                // invalidated.
393     cur_rate = -1;
394     recalibrate = 1;              // set recalibration flag.
395     printk("Reset-floppy called\n\r");
396     cli();
397     do_floppy = reset_interrupt;   // point reset function.
398     outb_p(current_DOR & ~0x04, FD_DOR); // do reset command.
399     for (i=0 ; i<100 ; i++)        // delay for a while.
400         __asm__("nop");
401     outb(current_DOR, FD_DOR);      // enable controller again.
402     sti();
403 }
404
405 static void floppy_on_interrupt(void) // floppy_on() interrupt.
406 {
407     /* We cannot do a floppy-select, as that might sleep. We just force it */
408     // If the current drive is different from the DOR, you will need to reset the DOR to the currently
409     // specified drive. After outputting the current DOR value to the DOR register, the timer is
410     // used to delay 2 ticks to allow the command to be executed, and then the floppy disk read/write
411     // transfer function transfer() is called. If the current drive matches the DOR, then the floppy
412     // disk read and write transfer function can be called directly.
413     selected = 1;                  // set drive selected flag.
414     if (current_drive != (current_DOR & 3)) {
415         current_DOR &= 0xFC;       // clear selected drive.
416         current_DOR |= current_drive; // set current drive.
417         outb(current_DOR, FD_DOR); // send current DOR.
418         add_timer(2, &transfer);   // add timer and related function.
419     } else

```

```

415         transfer();
416     }
417
418     /// Floppy disk read/write request item processing function.
419     // This is the main function in the floppy driver. Its main uses are: (1) Processing the case
420     // where the reset flag or the re-correction flag is set; (2) Obtaining the parameter block
421     // of the floppy drive by the request item using the device number in the request item; (3)
422     // Starting the floppy disk read/write operation by using the kernel timer.
423     void do fd request(void)
424     {
425         unsigned int block;
426
427         // First check if there is a reset flag or a recalibration flag. If one of them exists, the
428         // function returns immediately after processing the relevant flag.
429         seek = 0; // reset seek flag.
430         if (reset) {
431             reset floppy();
432             return;
433         }
434         if (recalibrate) {
435             recalibrate floppy();
436             return;
437         }
438
439         // The important aspects of this function start here. First use the INIT_REQUEST macro in the
440         // blk.h file to check the validity of the request item, and exit if there is no request. Then
441         // use the device number in the request item to get the parameter block of the specified floppy
442         // drive. This parameter block will be used below to set the global variable parameter block
443         // used by the floppy disk operation (see lines 112 - 122). The floppy disk type
444         // (MINOR(CURRENT->dev)>>2) in the request item number is used as the index value of the disk
445         // type array floppy_type[] to get the parameter block of the specified floppy drive.
446         INIT REQUEST;
447         floppy = (MINOR(CURRENT->dev)>>2) + floppy\_type;
448
449         // The following code begins to set the global variable parameter value on lines 112-122. If
450         // the current drive 'current_drive' is not the drive specified in the request item, the flag
451         // seek is set to indicate that the drive needs to perform seek processing before performing
452         // the read/write operation. Then set the current drive to the drive specified in the request.
453         if (current\_drive != CURRENT\_DEV) // the drive specified in the request.
454             seek = 1;
455         current\_drive = CURRENT\_DEV;
456
457         // Next, start setting the read/write start sector block. Since each read and write is in block
458         // units (1 block is 2 sectors), the starting sector needs to be at least 2 sectors smaller
459         // than the total number of sectors of the disk. Otherwise, the request item parameter is invalid,
460         // and the floppy disk request item is terminated to execute the next request item. Then
461         // calculate: the sector number, head number, track number, and seek track number (for
462         // floppy drives to read discs of different formats).
463         block = CURRENT->sector;
464         if (block+2 > floppy->size) {
465             end request(0);
466             goto repeat;
467         }
468         sector = block % floppy->sect; // the sector number on the track.

```

```

442     block /= floppy->sect;           // track number.
443     head = block % floppy->head;     // head no.
444     track = block / floppy->head;    // track no.
445     seek_track = track << floppy->stretch; // seek track number related to drive type.

// Then see if we still need to perform the seek operation first. If the seek number is different
// from the track number of the current head, a seek operation is required, and the seek flag
// is required. Finally we set up the floppy command to be executed.
446     if (seek_track != current_track)
447         seek = 1;
448     sector++;                       // sectors count from 1.
449     if (CURRENT->cmd == READ)
450         command = FD_READ;
451     else if (CURRENT->cmd == WRITE)
452         command = FD_WRITE;
453     else
454         panic("do_fd_request: unknown command");
// After setting all the global variable values on lines 112-122, we can start the request item
// operation. Here, the operation is started using a timer. Because it is necessary to start
// the drive motor first and reach the normal running speed, the floppy drive can be read and
// written, which takes a certain amount of time. So here ticks_to_floppy_on() is used to
// calculate the start delay time, and then use this delay to set a timer. The function
// floppy_on_interrupt() is called when the time expires.
455     add_timer(ticks_to_floppy_on(current_drive), &floppy_on_interrupt);
456 }
457
// The total number of data blocks contained in various types of floppy disk.
458 static int floppy_sizes[] = {           // initial data for array blk_size[].
459     0, 0, 0, 0,
460     360, 360, 360, 360,
461     1200, 1200, 1200, 1200,
462     360, 360, 360, 360,
463     720, 720, 720, 720,
464     360, 360, 360, 360,
465     720, 720, 720, 720,
466     1440, 1440, 1440, 1440
467 };
468
///// Floppy disk system initialization function.
// Set the processing function do_fd_request() of the floppy device request and set the floppy
// disk interrupt gate (int 0x26, corresponding to the hardware interrupt request signal IRQ6).
// The masking of the interrupt signal is then reset to allow the floppy disk controller FDC
// to send an interrupt request signal. The setting macro set_trap_gate() of the trap gate
// descriptor in the interrupt descriptor table IDT is defined in the header file
// include/asm/system.h.
469 void floppy_init(void)
470 {
// Set the floppy disk interrupt gate descriptor. floppy_interrupt is its interrupt handler,
// see kernel/sys_call.s, line 267. The interrupt number is int 0x26 (38), corresponding to
// the 8259A chip interrupt request signal IRQ6.
471     blk_size[MAJOR_NR] = floppy_sizes;
472     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST; // = do_fd_request().
473     set_trap_gate(0x26, &floppy_interrupt);

```

```

474         outb(inb_p(0x21)&~0x40, 0x21);           // reset floppy int mask bit.
475     }
476

```

### 9.6.3 Information

#### 9.6.3.1 Device number of the floppy disk drive

In Linux, the floppy drive's major number is 2, and the minor device number is determined by the floppy drive type and the floppy drive sequence number, which is:

---


$$\text{FD Minor No.} = \text{TYPE} * 4 + \text{DRIVE}$$


---

Among them, DRIVE is 0-3, corresponding to floppy drive A, B, C or D; TYPE is the type of floppy drive, for example, 2 means 1.2M floppy drive, 7 means 1.44M floppy drive, as shown in Table 9-12. That is, it is the index value of the floppy type array (floppy\_type[]) defined in floppy.c, line 85.

Table 9-12 Floppy drive type

Type	Description
0	Not used.
1	360KB PC Floppy drive.
2	1.2MB AT Floppy drive.
3	360kB Floppy disk used in the 720kB drive.
4	3.5" 720kB Floppy drive.
5	360kB Floppy disk used in the 1.2MB drive.
6	720kB Floppy disk used in the 1.2MB drive.
7	1.44MB Floppy drive.

For example, type 7 indicates a 1.44MB drive, drive number 0 indicates an A drive, because  $7*4 + 0 = 28$ , so (2, 28) refers to the 1.44M drive A, the device number is 0x021C, and the corresponding device file name is /dev/fd0 or /dev/PS0. Similarly, type 2 represents a 1.22MB drive, then  $2*4 + 0 = 8$ , so (2,8) refers to the 1.2M drive A, the device number is 0x0208, and the corresponding device file name is /dev/at0.

#### 9.6.3.2 Floppy Drive Controller

Since it is necessary to select a floppy drive, wait for the motor to reach a certain speed before reading and writing the floppy disk, and when the data block is transmitted, it needs to be realized by means of the DMA controller, it is cumbersome to program the floppy disk controller (FDC). When programming the FDC, it usually needs to access 4 ports, corresponding to one or more registers on the floppy disk controller. For the 1.2M floppy disk controller there are some of the ports shown in Table 9-13.

Table 9-13 Floppy disk controller ports

I/O port	Name	Reed/Write	Register Name
0x3f2	FD_DOR	Write only	Digital Output Register (DOR)
0x3f4	FD_STATUS	Read only	Main Status Register (STATUS)

0x3f5	FD_DATA	Read only	Result Register(RESULT)
		Write only	Data Register(DATA)
0x3f7	FD_DIR	Read only	Digital Input Register (DIR)
	FD_DCR	Write only	Drive Control Register (DCR)(Transfer Rate)

The digital output register (DOR) port is an 8-bit register that controls the driver motor turn-on, drive select, start/reset FDC, and enable/disable DMA and interrupt requests. The meaning of each bit of this register is shown in Table 9-14.

Table 9-14 Digital output register definition

Bit	Name	Description
7	MOT_EN3	Motor control for drive D: 1- Start motor; 0-Stop motor.
6	MOT_EN2	Motor control for drive C: 1- Start motor; 0-Stop motor.
5	MOT_EN1	Motor control for drive B: 1- Start motor; 0-Stop motor.
4	MOT_EN0	Motor control for drive A: 1- Start motor; 0-Stop motor.
3	DMA_INT	DMA and IRQ channel: 1-Enabled; 0-Disabled.
2	RESET	Controller reset: 1-Controller enabled; 0-Reset controller.
1	DRV_SEL1	00-11 is used to select floppy drive A-D respectively.
0	DRV_SEL0	

The FDC's Main Status Register (MSR) is also an 8-bit register that reflects the basic state of the floppy disk controller FDC and floppy disk drive FDD operation. Typically, the status bits of the main status register are read before the CPU sends a command to the FDC or before the FDC obtains the result of the operation to determine if the current FDC data register is ready and to determine the direction of data transfer. The definition of each bit of the MSR is shown in Table 9-17.

Table 9-15 The definition of each bit of the MSR

Bit	Name	Description
7	RQM	Data port ready: FDC data register is ready.
6	DIO	Transfer direction: 1- FDC -> CPU; 0 - CPU -> FDC
5	NDM	Non DMA mode: 1 - Controller not in DMA; 0 - DMA mode
4	CB	Controller busy: FDC is busy in executing a command.
3	DDB	Drive D is busy.
2	DCB	Drive C is busy.
1	DBB	Drive B is busy.
0	DAB	Drive A is busy.

The data port of the FDC corresponds to multiple registers: a write-only command and parameter register and a read-only result register. Only one register can appear on data port 0x3f5 at a time. When accessing a write-only register, the direction bit DIO of the main status register must be 0 (CPU -> FDC), and vice versa when accessing a read-only register. When accessing the write-only register to send commands and parameters, the command is 1 byte and the related parameter is 1-8 bytes. When accessing the read-only register to read the result, the result is only read after the FDC is not busy, and usually the result data has a maximum of 7 bytes.



For the data input register (DIR), only bit 7 (D7) is valid for the floppy disk, which is used to indicate the disk replacement status, and the remaining seven bits are used for the hard disk controller interface.

The write-only disk control register (DCR) is used to select the data transfer rate that the disc uses on different types of drives. Only the lower 2 bits (D1D0) are used, 00 means 500 kbps, 01 means 300 kbps, and 10 means 250 kbps.

In the Linux 0.12 kernel, the data transfer between the driver and the disk in the floppy drive is implemented by the DMA controller. Therefore, before performing read and write operations, you need to initialize the DMA controller first and program the floppy drive controller. For a 386-compatible PC, the floppy drive controller uses the hardware interrupt request signal IRQ6 (corresponding to interrupt descriptor 0x26) and uses channel 2 of the DMA controller. See the following sections for details on DMA control processing.

### 9.6.3.3 Floppy disk controller command

The floppy disk controller can accept a total of 15 commands, each of which goes through three phases: the command phase, the execution phase, and the result phase.

The command phase is that the CPU sends command byte and parameter bytes to the FDC. The first byte is always the command byte (command code). This is followed by parameter of 0-8 bytes. These parameters are usually the drive number, head number, track number, sector number, and total number of read/write sectors.

The execution phase is the operation specified by the FDC command. During the execution phase, the CPU does not intervene on the FDC. Generally, the FDC issues an interrupt request to let the CPU know the end of the command execution. If the FDC command sent by the CPU is to transfer data, the FDC can operate in an interrupt mode or a DMA mode. Each time the interrupt mode is transmitted by 1 byte, the DMA mode can transfer a large amount of data at a time. Under the management of the DMA controller, data is transferred between the FDC and the memory until all data has been transferred. At this time, the DMA controller notifies the FDC of the transmission byte count termination signal, and finally the FDC issues an interrupt request signal to inform the CPU that the execution phase is over.

The result phase is that the CPU reads the FDC data register (result register) return value to obtain the result of the FDC command execution. The result data returned is 0--7 bytes in length. For commands that do not return result data, we should send a detection interrupt status command to the FDC to get the status of the operation.

Since only six of the 15 commands are used in the Linux 0.12 floppy driver, only the commands used are described here.

#### 1. Recalibration command (FD\_RECALIBRATE)

This command is used to return the head to track 0. Usually used to recalibrate the head when the floppy disk operation is in error. The command code is 0x07, and the parameter is the specified drive letter (0-3).

The command has no result phase, and the program needs to obtain the execution result by executing "detect interrupt status" command. The format of this command is shown in Table 9-16.

Table 9-16 Format of recalibration command (FD\_RECALIBRATE)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	0	0	0	0	0	1	1	1	Recalibration command code: 0x07
Parameter	1	0	0	0	0	0	0	US1	US2	Disk drive number.
Execution										The head moves to 0 track
Result		None.								You need to use the command to get the

			execution result.
--	--	--	-------------------

### 2. Head seek command (FD\_SEEK)

This command moves the head of the selected drive to the specified track. The first parameter specifies the drive number and head number, bits 0-1 are the drive number, bit 2 is the head number, and other bits are useless. The second parameter specifies the track number.

The command has no result phase, and the program needs to obtain the execution result by executing "detect interrupt status" command. The format of this command is shown in Table 9-17.

Table 9-17 Format of head seek command (FD\_SEEK)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	0	0	0	0	1	1	1	1	Head seek command code: 0x0F
Parameter	1	0	0	0	0	0	HD	US1	US2	Head number, Drive number.
	2	C								Track number.
Execution										The head moves to the specified track.
Result		None.								You need to use the command to get the execution result.

### 3. Read sector data command (FD\_READ)

This command is used to read the sector starting from the specified location on the disk and transfer it to the system memory buffer via DMA controller. Whenever a sector is read, parameter 4 (R) is automatically incremented by one to continue reading the next sector until the DMA controller sends a transmission count termination signal to the floppy disk controller. This command usually begins after the head seek command is executed and the head is already on the specified track. The format of the command is shown in Table 9-18.

In the returned result, the track number C and the sector number R are the positions at which the current head is located. Since the starting sector number R is automatically incremented by one after reading one sector, the R value in the result is the next unread sector number. If the last sector on a track (ie EOT) is read, the track number is also incremented by one and the R value is reset to one.

Table 9-18 Format of read sector data command (FD\_READ)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	MT	MF	SK	0	0	1	1	0	Read sector command code: 0xE6 (MT=MF=SK=1)
Parameters	1	0	0	0	0	0	0	US1	US2	Drive number
	2	C								Track number (Cylinder address, 0 to 255)
	3	H								Head number (Head address, 0 or 1)
	4	R								Start sector number (Sector address)
	5	N								Sector size code (N=0..7: 128,256,512,,16KB)
	6	EOT								Final sector number. of the track (End of Track)
	7	GPL								Length of gap between sectors (3)
	8	DTL								The number of bytes in sector, when N=0
Execution										Data is transferred from disk to system
Result	1	ST0								Status byte 0

2	ST1	Status byte 1
3	ST2	Status byte 2
4	C	Track number
5	H	Header number
6	R	Sector number
7	N	Sector size code (0..7: 128,256,512,....,16KB)

Among them, the meanings of MT, MF and SK are:

- MT represents multi-track operation. MT = 1 indicates that two heads are allowed to operate continuously on the same track.
- MF indicates the recording method. MF=1 means to select the MFM recording mode, otherwise it is the FM recording mode.
- SK indicates whether to skip the sector with the delete flag. SK=1 means skip.

The meanings of the returned three status bytes ST0, ST1, and ST2 are shown in Table 9–19, Table 9–20, and Table 9–21, respectively.

Table 9-19 Status byte 0 (ST0)

Bit	Name	Description
7	ST0_INTR	Reason for interrupt. 00 - Normal termination of command; 01 - Abnormal termination of command; 10 - Invalid command; 11 - Abnormal termination caused by Polling.
6		
5	ST0_SE	Seek End. The controller completed a SEEK or RECALIBRATE command, or a READ/WRITE with implied seek command.
4	ST0_ECE	Equip. Check Error. Recalibration track 0 error.
3	ST0_NR	Not Ready. Floppy disk drive not ready.
2	ST0_HA	Head address. The current head number when interrupt occurs.
1	ST0_DS	Drive Select. Drive number when interrupt occurs. 00 - 11 corresponds to drive 0 - 3 respectively
0		

Table 9-20 Status byte 1 (ST1)

Bit	Name	Description
7	ST1_EOC	End of Cylinder. Tried to access a sector beyond the final sector of the track.
6		Unused. This bit is always 0.
5	ST1_CRC	The controller detected a CRC error in ID field or the Data field of a sector.
4	ST1_OR	Over Run. Data transfer timeout, DMA controller failure
3		Unused (0).
2	ST1_ND	No Data. The specified sector was not found.
1	ST1_WP	Write Protect.
0	ST1_MAM	Missing Address Mask. Sector ID address mark not found.

Table 9-21 Status byte 2 (ST2)

Bit	Name	Description
-----	------	-------------

7		Unused (0).
6	ST2_CM	Control Mark. When SK=0, the read data encounters the delete flag.
5	ST2_CRC	CRC error. The sector data field CRC check error.
4	ST2_WC	Wrong Cylinder. The track number C of the sector ID info does not match.
3	ST2_SEH	Scan Equal Hit. The scanning conditions meet the requirements.
2	ST2_SNS	Scan Not Satisfied: Scanning conditions do not meet the requirements
1	ST2_BC	Bad Cylinder. The track C = 0xFF in the sector ID info, the track is bad.
0	ST2_MAM	Missing Address Mask. The sector ID data address mark not found.

#### 4. Write sector data command (FD\_WRITE)

This command is used to write data from the memory buffer to disk. In the DMA transfer mode, the floppy drive controller serially writes the data in the memory to the specified sector of the disk. Each time a sector is written, the starting sector number is automatically incremented by one and continues to write one sector until the floppy drive controller receives the count termination signal from the DMA controller. The format of the command is shown in Table 9-22. The abbreviated name has the same meaning as in the read command.

Table 9-22 Format of write sector data command (FD\_WRITE)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	MT	MF	0	0	0	1	0	1	Write data command code: 0xC5 (MT=MF=1)
Parameters	1	0	0	0	0	0	0	US1	US2	Floppy drive number.
	2	C								Track number.
	3	H								Head number.
	4	R								Start sector number
	5	N								Sector size code.
	6	EOT								Final sector number. of the track
	7	GPL								Length of gap between sectors (3)
	8	DTL								The number of bytes in sector, when N=0
Execution										Data is transferred from the system to the disk
Result	1	ST0								Status byte 0
	2	ST1								Status byte 1
	3	ST2								Status byte 2
	4	C								Track number
	5	H								Head number
	6	R								Sector number
	7	N								Sector size code.

#### 5. Check interrupt status command (FD\_SENSEI)

After sending this command, the floppy controller will immediately return the normal results 1 and 2 (ie, state ST0 and the track number PCN where the head is located). They are the result states after the controller executes the previous command. An interrupt signal is usually sent to the CPU after the execution of a command. For interrupts caused by read/write sectors, read/write tracks, read/write delete flags, read ID field, format and scan commands, and commands in non-DMA transfer mode, the cause of the interrupt can be known directly based on the flag of the main status register. For the interrupt caused by the drive's ready signal change,

seek and recalibration (head return to zero), because there is no return result, you need to use this command to read the status information after the controller executes the command. The format of this command is shown in Table 9-23.

Table 9-23 Format of check interrupt status command (FD\_SENSEI)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	0	0	0	0	1	0	0	0	Detection interrupt status cmd code: 0x08
Execution										
Result	1	ST0								Status byte 0
	2	C								Track number of the current head.

#### 6. Set drive parameter command (FD\_SPECIFY)

This command is used to set the three initial timer values and the selected transmission mode inside the floppy disk controller, that is, the drive motor step rate (SRT), head loading/unloading (HLT/HUT) time, and whether to use DMA mode for transmission, are sent to the floppy drive controller. The format of this command is shown in Table 9-24. The time unit is the value when the data transmission rate is 500 KB/S. In addition, in the Linux 0.12 kernel, the parameter byte 1 of the command phase is the spec1 described in the original comment on lines 96-103 in floppy.c file; the parameter byte 2 is the spec2. From the original comment and the program statement on line 316, spec2 is fixedly set to a value of 6 (ie, HLT = 3, ND = 0), indicating that the head load time is 6 milliseconds, and the DMA mode is used.

Table 9-24 Format of setting drive parameter command (FD\_SPECIFY)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	0	0	0	0	0	0	1	1	Set parameter command code: 0x03
Parameters	1	SRT (Unit: 1ms)				HUT (Unit: 16ms)				Motor step rate, head unloading time
	2	HLT (Unit: 2ms)							ND	Head load time, non-DMA mode
Execution										Set the controller, no interrupt issued.
Result		None.								None.

### 9.6.3.4 Floppy disk controller programming

In a PC, the floppy disk controller generally uses a compatible chip of NEC PD765 or Intel 8287A, such as Intel's 82078. Since the driver of the floppy disk is relatively complicated, the programming method of the floppy disk controller composed of such a chip is described in detail below. Typical disk operations include not only sending commands and waiting for controllers to return results. The control of a floppy disk drive is a low-level operation that requires the program to interfere with its execution at different stages.

#### 1. Interaction between command and result phases

Before the above disk operation commands and parameters are sent to the floppy disk controller, the controller's main status register (MSR) must be queried first to know the ready state of the drive and the data transfer direction. The floppy disk driver uses an output\_byte(byte) function to do this. The equivalent block diagram of this function is shown in Figure 9-9.

This function is looped until the data port ready flag RQM of the main status register is 1, and the direction flag DIO is 0 (CPU -> FDC), at which point the controller is ready to accept command and parameter bytes. The loop statement starts from the timeout counting function to cope with the case where the controller does not

respond. In this driver, the number of loops is set to 10000 times. The choice of the number of loops needs to be careful to avoid the program making an incorrect timeout. In the Linux kernel version 0.1x to 0.9x, it is often encountered that the number of cycles needs to be adjusted, because the PCs used at that time have different speeds (16MHz - 40MHz), so the actual delay caused by the loop will be very different. This can be seen from the discussion of many articles in the early Linux mailing list. To completely solve this problem, it is best to use the system hardware clock to generate a fixed frequency delay value.

For the result phase of reading the result bytes of the controller, it is also necessary to take the same operation method as the send command, except that the data transfer direction flag request is set (FDC -> CPU). The corresponding function in this program is `result()`, which stores the result status bytes into the `reply_buffer[]` byte array.

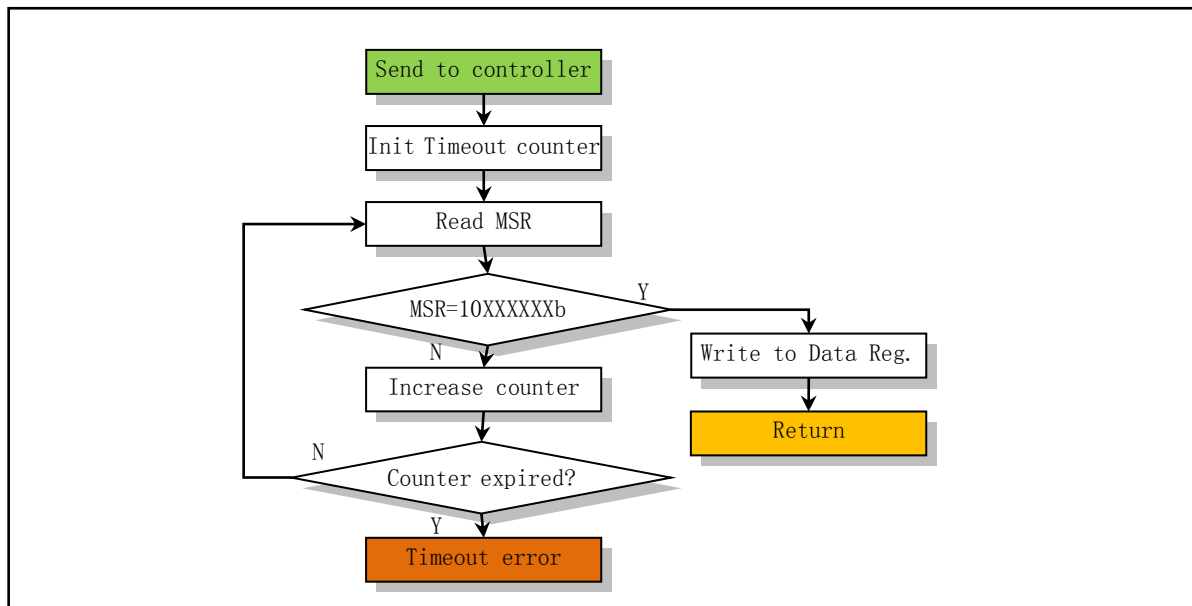


Figure 9-9 Send command or parameter bytes to the floppy controller

## 2. Floppy disk controller initialization

Initializing the floppy disk controller involves configuring the appropriate parameters for the drive after the controller is reset. The controller reset operation is to set the FDC reset flag (bit 2 of DOR) to 0 (reset) and then 1. After the FDC reset, the value set by the "Specify Drive Parameters" command SPECIFY is no longer valid and needs to be re-established. In the `floppy.c` program, the reset operation is included in the function `reset_floppy()` and the interrupt handler C function `reset_interrupt()`. The previous function is used to modify bit 2 of the DOR register to reset the controller. The latter function is used to re-establish the driver parameters in the controller using the SPECIFY command after the controller is reset. During the data transfer preparation phase, if it is detected that the current drive parameters in the FDC are different from the actual disk specifications, it will be additionally reset at the beginning of the transfer function `transfer()`.

After the controller is reset, the specified transfer rate should also be sent to the digital control register DCR to reinitialize the data transfer rate. If the machine performs a reset operation (such as a warm boot), the data transfer rate will become the default value of 250Kpbs. However, the reset operation issued to the controller through the digital output register DOR does not affect the data transfer rate.

## 3. Drive recalibration and head seek

Driver recalibration (FD\_RECALIBRATE) and head seek (FD\_SEEK) are two head positioning commands. The recalibration command moves the head to zero track, and the head seek command moves the head to the specified track. These two head positioning commands are different from typical read/write commands because they have no result phase. Once one of these two commands is issued, the controller will immediately return to the Ready state in the Main Status Register (MSR) and perform head positioning operations in the background. When the positioning operation is completed, the controller generates an interrupt request service. At this point, you should send a "Detect Interrupt Status" command to end the interrupt and read the status after the positioning operation. Since the drive and motor enable signals are directly controlled by the digital output register (DOR), if the drive or motor has not been started, the operation of writing DOR must be performed before the positioning command is issued. A related flow chart is shown in Figure 9-10.

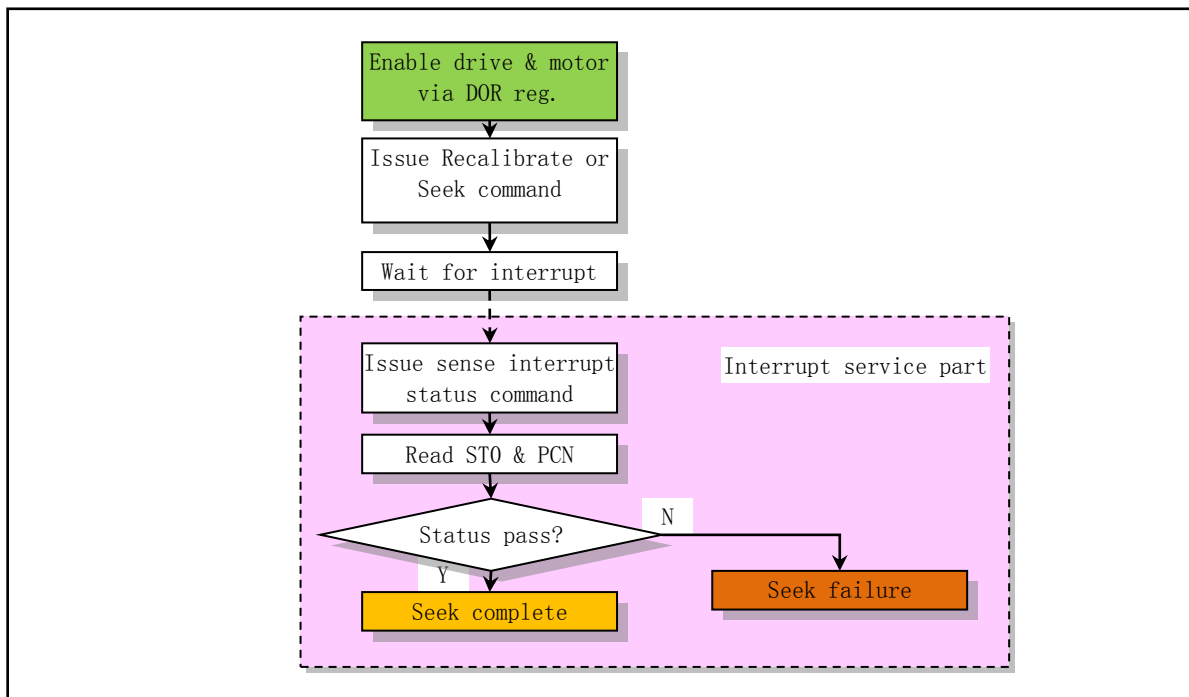


Figure 9-10 Recalibration and seek operation

#### 4. Data read/write operations

Data read or write operations take a few steps to complete. First the drive motor needs to be turned on and the head is positioned on the correct track, then the DMA controller is initialized and finally a data read or write command is sent. In addition, you need to determine the processing plan when an error occurs. A typical operational flow chart is shown in Figure 9-11.

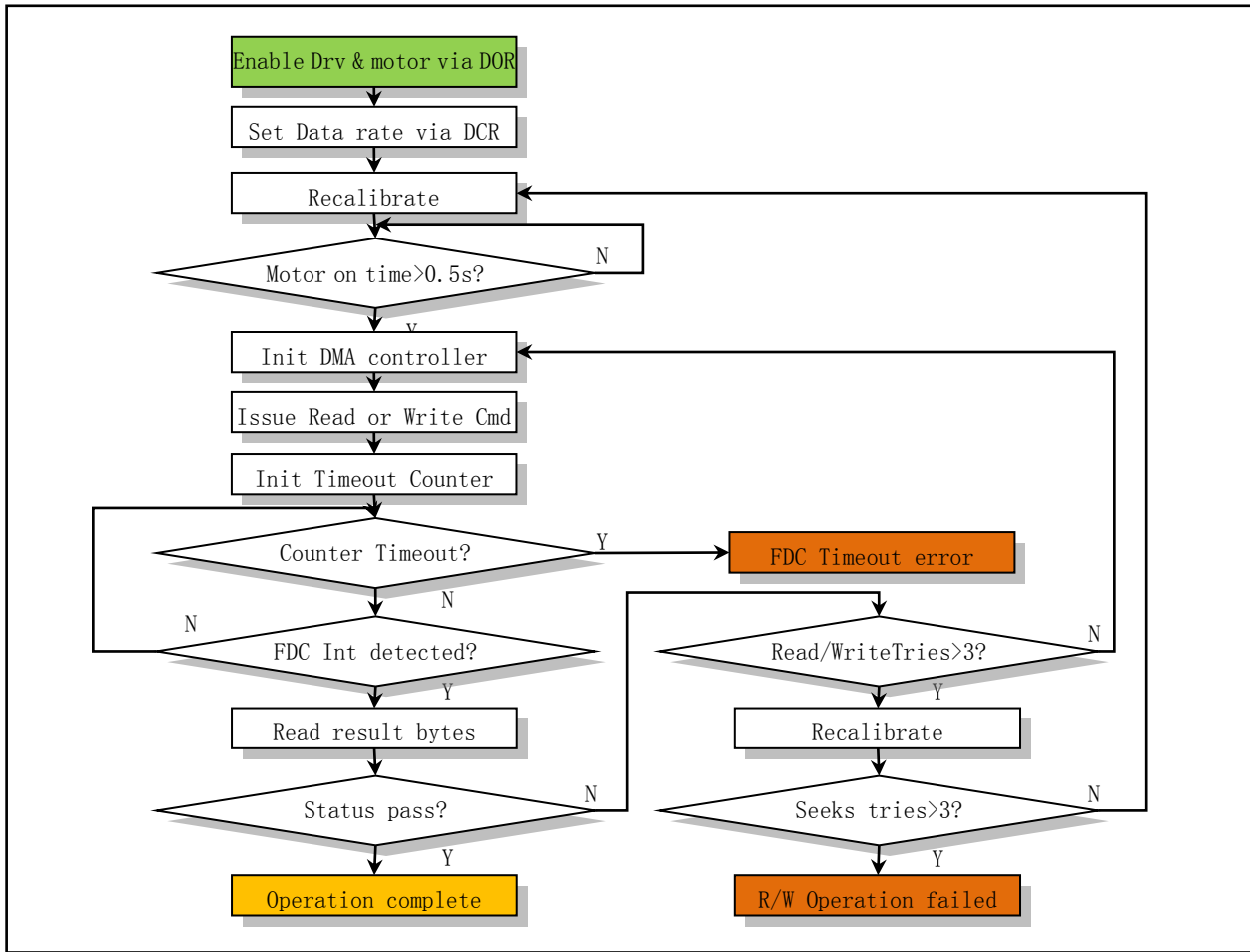


Figure 9-11 Data read/write operation flow diagram

The disk drive's motor must first reach normal operating speed before the disk can start transferring data. For most 3 1/2-inch floppy drives, this boot time takes approximately 300ms, while the 5 1/4-inch floppy drive takes approximately 500ms. This startup delay time was set to 500ms in the floppy.c program.

After the motor is started, it is necessary to use the digital control register DCR to set the data transfer rate that matches the current floppy disk medium.

If the implicit seek mode is not enabled, then the seek command `FD_SEEK` needs to be sent to position the head on the correct track. After the seek operation is completed, the head also takes a period of loading time. For most drives, this delay takes at least 15ms. When the implicit seek mode is used, the head load time (HLT) specified by the "Specify Drive Parameters" command can be used to determine the minimum head arrival time. For example, in the case where the data transmission rate is 500 Kbps, if  $HLT = 8$ , the effective head in-position time is 16 ms. Of course, if the head is already in place on the correct track, it will not take this time.

Then the DMA controller is initialized and the read and write commands are executed. Usually, after the data transfer is completed, the DMA controller will issue a termination count (TC) signal, at which point the floppy disk controller will complete the current data transfer and issue an interrupt request signal indicating that the operation has reached the result stage. If an error occurs during operation or the last sector number equals the last sector of the track (EOT), the floppy disk controller will immediately enter the result phase.

According to the above flow diagram, if an error is found after reading the result status byte, the data read or write operation command is restarted by reinitializing the DMA controller. A continuous error usually indicates that the seek operation did not cause the head to reach the specified track, and the recalibration of the



head should be repeated multiple times and the seek operation should be performed again. If there is still an error after that, the controller will report the read or write operation failure to the driver.

#### 5. Disk format operations

Although the Linux 0.12 kernel does not implement the format operation of the floppy disk, as a reference, the disk formatting operation is briefly described here. The disk formatting operation involves positioning the heads on each track and creating a fixed format field for composing the data field.

After the motor has started and the correct data transfer rate is set, the head will return to zero track. At this point, the disk needs to reach a normal and stable operating speed within a 500ms delay time.

The identification field (ID field) established on the disk during the formatting operation is provided by the DMA controller during the execution phase. The DMA controller is initialized to provide values for the track (C), head (H), sector number (R), and sector bytes for each sector identification field. For example, for a disk with 9 sectors per track, each sector size is 2 (512 bytes). If the track 7 is formatted with head 1, the DMA controller should be programmed to transmit 36 bytes (9 sectors x 4 bytes per sector), the data fields should be: 7,1,1,2,7,1,2,2,7,1,3,2,...,7, 1,9,2. Because the data provided by the floppy disk controller is recorded directly on the disk as an identification field during the execution of the format command, the content of the data can be arbitrary. So some people use this feature to prevent protected disk copying.

After each head on one track has performed the formatting operation, it is necessary to perform a seek operation to advance the head to the next track and repeat the formatting operation. Because the Formatted Track command does not contain an implicit seek operation, the seek command SEEK must be used. Similarly, the head in-position time discussed above also needs to be set after each seek.

### 9.6.3.5 DMA Controller Programming

The primary purpose of the Direct Memory Access (DMA) controller is to enhance the system's data transfer performance by allowing external devices to transfer data directly to the memory. Usually it is implemented by the Intel 8237A chip or its compatible chip on the machine. By programming the DMA controller, the data transfer between the peripheral and the memory can be performed independently of the CPU. Therefore, the CPU can do other things during data transfer. The working process of the DMA controller to transfer data is as follows:

#### 1. Initialization of the DMA controller.

The program initializes it through the DMA controller port, which includes: (1) sending control commands to the DMA controller; (2) sending memory start address for transfer; (3) sending data length. The command sent indicates whether the DMA channel used for the transfer, whether the memory is transferred to the peripheral (write) or the peripheral data is transferred to the memory, whether it is a single-byte transfer or a bulk (block) transfer. For PCs, the floppy disk controller is designated to use DMA channel 2. In the Linux 0.12 kernel, the floppy disk driver uses a single-byte transfer mode. Since the Intel 8237A chip has only 16 address pins (eight of which are used in conjunction with the data lines), only 64KB of memory space can be addressed. In order to allow it to access 1MB of address space, the PC uses a LS670 chip as a DMA page register, and divides 1MB of memory into 16 pages for operation, as shown in Table 9-25. Therefore, the transferred memory start address needs to be converted into the DMA page value and the offset address in the page, and the data length of each transmission cannot exceed 64 KB.

From this we can see that the transfer buffer we set in memory must be within 1MB address space. However, if the actual data buffer (such as the user buffer) is outside the 1MB space, we need to setup a temporary transfer buffer in the memory 1MB address area for use by the DMA, and copy the transferred

data between the temporary buffer and the actual user buffer. This is exactly the way the Linux 0.12 kernel is used, see the function `setup_DMA()` (program `floppy.c`, line 106 and line 161).

Table 9-25 DRAM page corresponding memory address range

DMA page	Address range (64KB)
0x00	0x00000 - 0x0FFFF
0x01	0x10000 - 0x1FFFF
0x02	0x20000 - 0x2FFFF
0x03	0x30000 - 0x3FFFF
0x04	0x40000 - 0x4FFFF
0x05	0x50000 - 0x5FFFF
0x06	0x60000 - 0x6FFFF
0x07	0x70000 - 0x7FFFF
0x08	0x80000 - 0x8FFFF
0x09	0x90000 - 0x9FFFF
0x0A	0xA0000 - 0xAFFFF
0x0B	0xB0000 - 0xBFFFF
0x0C	0xC0000 - 0xCFFFF
0x0D	0xD0000 - 0xDFFFF
0x0E	0xE0000 - 0xEFFFF
0x0F	0xF0000 - 0xFFFFF

## 2. Data transmission

After the initialization is completed, the mask register of the DMA controller is modified, and DMA channel 2 is enabled, so that the DMA controller starts data transmission.

## 3. End of transmission

When all the data to be transferred is transferred, the DMA controller will generate an "End of Process" (EOP) signal to the floppy controller. At this point, the floppy disk controller can perform the end operation: turn off the drive motor and send an interrupt request signal to the CPU.

In the PC/AT machine, the DMA controller has 8 independent channels available, of which the last 4 channels are 16 bits. The floppy disk controller is designated to use DMA channel 2. You must set it first before using it. This involves operations on three ports: the page register port, the (offset) address register port, and the data count register port. Since the DMA register is 8-bit and the address and count values are 16 bits, they need to be sent twice. The low byte is sent first, then the high byte is sent. The port address corresponding to each channel is shown in Table 9-26.

Table 9-26 Page, address, and count register ports used by each DMA channel

DMA Channel	Page register	Base address register	Word count register
0	0x87	0x00	0x01
1	0x83	0x02	0x03
2	0x81	0x04	0x05
3	0x82	0x06	0x07

4	0x8F	0xC0	0xC2
5	0x8B	0xC4	0xC6
6	0x89	0xC8	0xCA
7	0x8A	0xCC	0xCE

For normal DMA applications, there are five common registers that control the operation and state of the DMA controller. These are the Command Register, the Request Register, the Single Mask Register, the Mode Register, and the Clear Pre/Post Pointer Trigger, as shown in Table 9–27. The Linux 0.12 kernel mainly uses three shaded register ports (0x0A, 0x0B, 0x0C) in the table.

Table 9-27 registers commonly used in DMA programming

Register Name	Read/Write	Port address	
		Channel 0 - 3	Channel 4 - 7
Status register/Command register	Read/Write	0x08	0xD0
Request register	Write	0x09	0xD2
Single Channel mask register	Write	0x0A	0xD4
Mode register	Write	0x0B	0xD6
Clear First/Last Flip-Flop	Write	0x0C	0xD8
Temporary register/Mater Clear	Read/Write	0x0D	0xDA
Clear Mask register	Write	0x0E	0xDC
Full Mask register	Write	0x0F	0xDE

**Command Register** -- The command register is used to specify the operational requirements of the DMA controller chip and set the overall state of the DMA controller. Usually it does not need to change after boot initialization. In the Linux 0.12 kernel, the floppy driver directly uses the ROM BIOS settings after booting. For reference, the meaning of the bits of the command register is listed here, as shown in Table 9–28. Note that when reading the same port, you will get the information of the DMA controller status register.

Table 9-28 DMA command register format

Bit	Description
7	DMA response peripheral signal DACK: 0-DACK active low; 1-DACK active high.
6	Peripheral request DMA signal DREQ: 0-DREQ active low; 1-DREQ active high.
5	Write mode selection: 0-select late write; 1-select extended write; X-if bit 3=1.
4	DMA channel priority mode: 0-fixed priority; 1-rotating priority.
3	DMA cycle selection: 0 - normal timing cycle (5); 1- compression timing cycle (3); X - if bit 0 = 1.
2	Start DMA controller: 0 - enable controller; 1 - disables the controller.
1	Channel 0 address hold: 0 - disable channel 0 address hold; 1 - allow channel 0 address to hold; X - if bit 0 = 0.
0	Memory transfer mode: 0 - disable memory to memory; 1 - enable memory to memory.

**Request Register** -- The 8237A can respond to requests for DMA service which are initiated by software as well as by a DREQ. The request register is used to record the request service signal DREQ of the peripheral to the channel, one bit for each channel. When DREQ is active, it corresponds to position 1, which is set to 0 when

the DMA controller responds to it. If the DMA request signal DREQ pin is not used, the DMA controller's service can also be requested by directly setting the request bit of the corresponding channel by programming. In the PC, the floppy disk controller has a direct request signal DREQ connection to channel 2 of the DMA controller, so there is no need to operate the register in the Linux kernel. For reference, the byte format of the request channel service is listed here, as shown in Table 9-29.

Table 9-29 The meaning of each bit of the DMA request register

Bit	Description
7 -3	Not used.
2	Set flag. 0 - Request bit is set; 1 - Request bit is reset (set to 0).
1	Channel selection. 00-11 selects channel 0-3.
0	

Mask Register -- The port of the single mask register is 0x0A (0xD4 for 16-bit channels). A channel is masked, meaning that the DMA request signal DREQ issued by the peripheral using the channel does not get the response from the DMA controller, therefore, the DMA controller cannot be operated on the channel. Each channel has associated with it a mask bit which can be set to disable the incoming DREQ. The meaning of each bit of this register is shown in Table 9-30.

Table 9-30 The meaning of each bit of the DMA single mask register

Bit	Description
7 -3	Not used.
2	mask flag. 1 - set mask bit; 0 - Clear mask bit.
1	Channel selection. 00-11 selects channel 0-3.
0	

Mode Register -- The mode register is used to specify how a DMA channel operates. The meaning of each bit of this register is shown in Table 9-31. In the Linux 0.12 kernel, two setting modes, read disk (0x46) and write disk (0x4A), are used. According to Table 9-31, use DMA channel 2, read disk (write memory) transfer, disable autoinitialization, select address increment and use single byte mode; 0x4A means set mode: use DMA channel 2, write disk (read memory) transfer, disable autoinitialization, select address increment and use single byte mode.

Table 9-31 The meaning of each bit of the DMA mode register

Bit	Description
7	Select the transfer mode: 00-request mode; 01-single byte mode; 10-block byte mode; 11-cascade mode.
6	
5	Address method. 0 - address increment; 1 - address decrement.
4	Autoinitialization. 0-Autoinitialization disable; 1-Autoinitialization enable.
3	Transmission type: 00-verify transfer; 01-write memory transfer; 10-read memory transfer; 11- illegal; XX - if bits 6-7=11.
2	
1	Channel selection. 00-11 selects channel 0-3 respectively.
0	

Since the channel address and count registers can read and write 16-bit data, you need to perform two write operations, one low byte and one high byte at a time. Which byte is actually written is determined by the state of the software command clear first/last flip-flop. This command must be executed prior to writing or reading new address or word count information to the 8237A. This initializes the flip-flop to a known state so that subsequent accesses to register contents by the CPU will address upper and lower bytes in the correct sequence. Port 0x0C is used to initialize the byte-order flip-flop to the default state before reading or writing the address or count information in the DMA controller. When the clear byte first/last flip-flop is 0, the low byte is accessed; when the it is 1, the high byte is accessed. The flip-flop changes once per visit. The 0x0C port can be written to set the clear first/last flip-flop to the 0 state.

When using the DMA controller, it usually needs to follow a certain step. The following uses the DMA controller method to describe the DMA's brief programming steps:

1. Turn off the interrupt to eliminate any interference;
2. Modify the mask register (port 0x0A) to mask the DMA channel that needs to be used. For the floppy disk driver is channel 2;
3. Write to the 0x0C port and set "clear first/last flip-flop" to the default state;
4. Write mode register (port 0x0B) to set the operation mode word of the specified channel;
5. Write the address register (port 0x04) to set the offset address in the memory page used by the DMA. Write the low byte first, then write the high byte;
6. Write the page register (port 0x81) to set the memory page used by the DMA;
7. Write the count register (port 0x05), set the number of bytes for DMA transfer, which should be the transfer length -1. We also need to write once for the high and low bytes. In this book, the length of the floppy disk driver required by the DMA controller is 1024 bytes, so the length of the write DMA controller should be 1023 (ie 0x3FF);
8. Modify the mask register (port 0x0A) again to enable the DMA channel;
9. Finally, enable the interrupt to allow the floppy controller to issue an interrupt request to the system after the transfer is complete.

## 9.7 Summary

This chapter firstly introduces the main data structures such as request items and request queues used by the block device driver, and then introduces the general working method of the block device from the relationship between the system processor, the device controller and the specific block device. It is worth reminding again that the access to the block device data uses the interrupt handler method (except the virtual memory disk), after the device issues a read and write command, it can directly return to the calling program, and enter the non-interruptible sleep waiting queue, waiting for the block device operation to complete. In addition, the access of the upper layer program to the block device data is realized by a unified low-level block device read/write function `ll_rw_block()`.

In the next chapter, let's learn about another different type of device: character devices, such as terminal console devices. It is one of the interactive device we communicate directly with the system.

