









# 10 Character Device Driver

In the Linux 0.12 kernel, character devices mainly include control terminal devices and serial terminal devices. The code in this chapter is used to operate the input and output of these devices. For the basic working principle of the terminal driver, we can refer to the book "The Design of the UNIX Operating System" by Mr. Maurice J. Bach or other related books. This chapter contains a total of seven source code files, as shown in Listing 10-1, two of which are programmed in assembly language. These files are located in the kernel/chr\_drv/ directory.

List 10-1 linux/kernel/chr\_drv

	Filename	Size	Last modified date(GMT)	Desc.
	<a href="#">Makefile</a>	3618 bytes	1992-01-12 19:49:17	
	<a href="#">console.c</a>	23327 bytes	1992-01-12 20:28:33	
	<a href="#">keyboard.S</a>	13020 bytes	1992-01-12 15:30:51	
	<a href="#">pty.c</a>	1186 bytes	1992-01-10 23:56:45	
	<a href="#">rs_io.s</a>	2733 bytes	1992-01-08 06:27:08	
	<a href="#">serial.c</a>	1412 bytes	1992-01-08 06:17:01	
	<a href="#">tty_io.c</a>	12282 bytes	1992-01-11 16:18:46	
	<a href="#">tty_ioctl.c</a>	6325 bytes	1992-01-11 04:02:37	

## 10.1 Main Functions

The programs in this chapter can be divided into three parts. The first part is about the RS-232 serial line driver, including the programs rs\_io.s and serial.c; The second part is about the console driver, which includes the keyboard interrupt driver keyboard.S and the console display driver console.c; The third part is the interface part between the terminal driver and the upper layer program, including the terminal input and output program tty\_io.c and the terminal control program tty\_ioctl.c. Below we first outline the basic principles of terminal control driver implementation, and then explain these basic functions in three parts. After that, each source file will be explained and annotated in detail.

### 10.1.1 Basic Principles of Terminal Drivers

The terminal driver is used to control the terminal device, transfer data between the terminal device and the process, and perform certain processing on the transmitted data. The raw data typed by the user on the keyboard is transferred to a receiving process after being processed by the terminal program. The data sent by the process to the terminal is displayed on the terminal screen after being processed by the terminal program, or transmitted to the remote terminal through the serial communication line. The terminal operating mode can be divided into two types according to the manner in which the terminal program treats input or output data. One is the canonical mode, in which the data passing through the terminal program will be transformed and then sent out. For example, the TAB character is expanded to 8 space characters, and the typed backspace is used to control

the deletion of the previously typed characters. The processing functions used are generally referred to as line disciplines or line discipline modules. The other is a non-canonical mode or raw mode. In this mode, the line discipline program transfers data only between the terminal and the process without performing a canonical mode conversion on the data.

In the terminal driver, the code can be divided into a direct driver of the character device and an interface program directly connected to the upper layer according to their relationship with the device and the position in the execution flow. We can use Figure 10-1 to illustrate this control relationship.

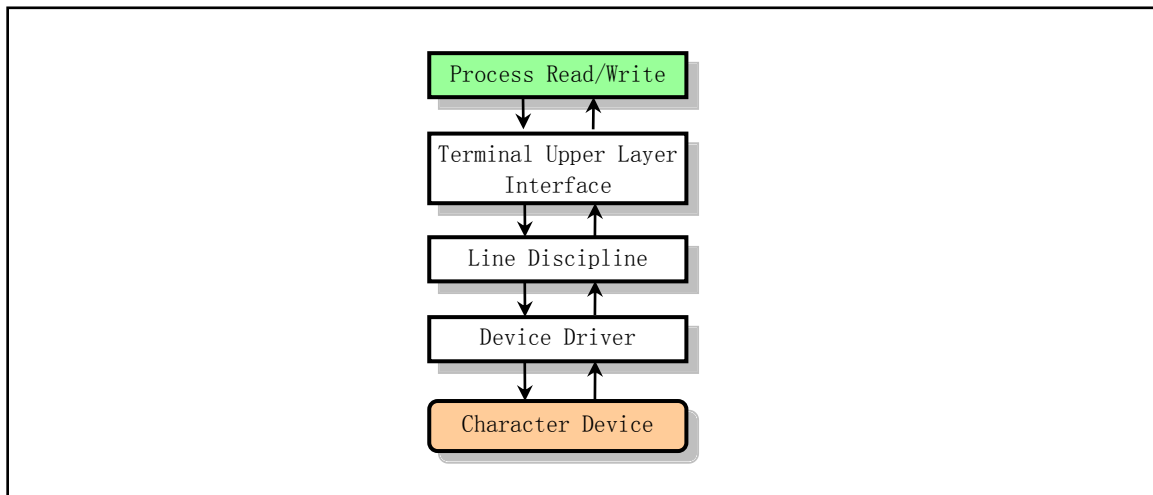


Figure 10-1 Terminal driver control flow

### 10.1.2 Types of Terminal Devices Supported by Linux

A terminal is a character type device that has many types. We usually use `tty` to refer to various types of terminal devices. `tty` is the abbreviation of Teletype, which is the earliest terminal device produced by Teletype, which looks like a teletypewriter. In the Linux 0.1x system device file directory `/dev/`, it usually contains the following terminal device files:

crw-rw-rw-	1 root	tty	5,	0 Jul 30 1992	tty	// Control terminal
crw--w--w-	1 root	tty	4,	0 Jul 30 1992	tty0	// virtual terminal alias.
crw--w--w-	1 root	tty	4,	1 Jul 30 1992	console	// Console
crw--w--w-	1 root	other	4,	1 Jul 30 1992	tty1	// virtual terminal 1
crw--w--w-	1 root	tty	4,	2 Jul 30 1992	tty2	
crw--w--w-	1 root	tty	4,	3 Jul 30 1992	tty3	
crw--w--w-	1 root	tty	4,	4 Jul 30 1992	tty4	
crw--w--w-	1 root	tty	4,	5 Jul 30 1992	tty5	
crw--w--w-	1 root	tty	4,	6 Jul 30 1992	tty6	
crw--w--w-	1 root	tty	4,	7 Jul 30 1992	tty7	
crw--w--w-	1 root	tty	4,	8 Jul 30 1992	tty8	
crw-rw-rw-	1 root	tty	4,	64 Jul 30 1992	ttys1	// Serial port terminal 1
crw-rw-rw-	1 root	tty	4,	65 Jul 30 1992	ttys2	
crw--w--w-	1 root	tty	4,	128 Jul 30 1992	ptyp0	// primary pseudo terminal.
crw--w--w-	1 root	tty	4,	129 Jul 30 1992	ptyp1	
crw--w--w-	1 root	tty	4,	130 Jul 30 1992	ptyp2	
crw--w--w-	1 root	tty	4,	131 Jul 30 1992	ptyp3	
crw--w--w-	1 root	tty	4,	192 Jul 30 1992	ttyp0	// slave pty

crw--w--w-	1 root	tty	4, 193 Jul 30 1992	ttyp1
crw--w--w-	1 root	tty	4, 194 Jul 30 1992	ttyp2
crw--w--w-	1 root	tty	4, 195 Jul 30 1992	ttyp3

---

These terminal device files can be divided into the following types:

1. Serial port terminal (/dev/ttySn)

A serial port terminal is a terminal device that is connected using a computer serial port. The computer treats each serial port as a character device. For some time these serial port devices are often referred to as terminal devices because its primary use at that time was to connect to a terminal. The device file names corresponding to these serial ports are /dev/ttyS0, /dev/ttyS1, etc., and the device numbers are (4, 64), (4, 65), etc., respectively corresponding to COM1 and COM2 under the DOS system. To send data to a port, you can redirect the standard output to these special file names on the command line. For example, typing "echo test > /dev/ttyS1" at the command prompt will send the word "test" to the device connected to the ttyS1 port.

2. Pseudo terminal (/dev/ptyp, /dev/ttyp)

A pseudo terminal (or Pseudo - TTY, abbreviated as PTY) is a device that functions like a general terminal, but the device is not related to any terminal hardware. The pseudo terminal device is used to provide a terminal-like interface for other programs, and is mainly used to provide a terminal interface for the network server and the login shell program when logging in to the host through the network, or to provide a terminal style interface for the terminal program running in the X Window window. Of course, we can also use the pseudo terminal to establish a data read and write channel between any two programs that use the terminal interface. In order to provide terminal style interfaces for two applications or processes, the pseudo terminals are paired, and one is called a master terminal or a pseudo terminal master, and the other is called a slave terminal or a pseudo terminal slave. For paired pseudo-terminal logical devices like ptyp1 and ttyp1, ptyp1 is the master or control terminal, and ttyp1 is the slave. Data written to any of the pseudo terminals is received directly by the paired pseudo terminal through the kernel. For example, for the master device /dev/ptyp3 and the slave device /dev/ttyp3, if a program treats ttyp3 as a serial port device, its read/write operations on that port are reflected in the corresponding logical terminal device ptyp3, and ptyp3 is another logical device used by programs for read and write operations. In this way, two programs can communicate with each other through this logical device, and one of the programs using the slave device ttyp3 considers that it is communicating with a serial port. This is much like a pipeline operation between pairs of logical devices.

For a pseudo-terminal slave device, any program designed to use a serial port device can use the logic device, but for a program using the master device, it is specifically designed to use the pseudo-terminal master device. For example, if someone connects to your computer using a telnet program on the Internet, the telnet program may start to connect to the pseudo terminal master device ptyp2, and a getty program should run on the corresponding ttyp2 port. When telnet retrieves a character from the far end, the character is passed to the getty program via ptyp2 and ttyp2, and the getty program sends the "login:" string information to the network via the ttyp2, ptyp2, and telnet programs. In this way, the login program and the telnet program can communicate through the "pseudo terminal". By using the appropriate software, we can connect two or more pseudo terminal devices to the same physical port.

Previous Linux systems only had a maximum of 16 pairs of ttyp (ttyp0-ttypf) device file names, but nowadays Linux systems usually use the "prm-pty master" naming scheme, such as /dev/ptm3. Its corresponding end is automatically created as /dev/pts/3, so that a pty pseudo terminal can be dynamically provided when needed. The directory /dev/pts on current Linux systems is a file system of type devpts.

Although the "file" `/dev/pts/3` appears to be one of the device file systems, it is actually a different file system.

### 3. Control terminal (`/dev/tty`)

The character device file `/dev/tty` is an alias for the Controlling Terminal. Its major device number is 5 and the minor device number is 0. If the current process has a control terminal, `/dev/tty` is the device file of the current process control terminal. We can use the command `"ps -ax"` to see which control terminal the process is connected to. For the login shell, `/dev/tty` is the terminal we use, and its device number is (5,0). We can use the command `"tty"` to see which actual terminal device it corresponds to. In fact `/dev/tty` is somewhat similar to a link to an actual terminal device.

If an terminal user executes a program but does not want the control terminal (such as a background server program), the process can try to open the `/dev/tty` file first. If the opening is successful, the process has a control terminal. At this point we can use the `ioctl()` call with the `TIOCNOTTY` (Terminal IO Control NO TTY) parameter to abandon the control terminal.

### 4. Console (`/dev/ttyN`, `/dev/console`)

In a Linux system, computer monitor is often referred to as console terminal or console. It emulates a VT200 or Linux type terminal (`TERM=Linux`) and has some character device files associated with it: `tty0`, `tty1`, `tty2`, and so on. When we log in on the console, we are using `tty1`. In addition, using `Alt+[F1—F6]`, we can switch to `tty2`, `tty3`, etc. `Tty1 – tty6` is called a virtual terminal, and `tty0` is an alias for the currently used virtual terminal. Information generated by the Linux system is sent to `tty0`, so no matter which virtual terminal is currently being used, system information is sent to our screen.

You can log in to different virtual terminals, so you can have several different sessions at the same time. However, only the system or superuser root can write to `/dev/tty0`, and sometimes `/dev/console` will also connect to the terminal device. However, in Linux 0.12 systems, `/dev/console` is usually connected to the first virtual terminal `tty1`.

### 5. Other types of terminals

Today's Linux systems also have many other types of terminal device special files for many different character devices. For example, the `/dev/ttyIn` terminal device for ISDN devices. I won't go into details here.

## 10.1.3 Terminal data structures

Each terminal device has a `tty_struct` data structure, which is mainly used to store information such as the current parameter settings of the terminal device, the foreground process group ID and the character IO buffer queue. This structure is defined in the `include/linux/tty.h` file and its structure is as follows:

---

```
struct tty_struct {
    struct termios termios;           // Terminal io properties and control chars.
    int pgrp;                          // The process group to which it belongs.
    int stopped;
    void (*write)(struct tty_struct * tty); // pointer to tty write function.
    struct tty_queue read_q;          // tty read queue.
    struct tty_queue write_q;         // tty write queue.
    struct tty_queue secondary;       // tty auxiliary queue (or canonical queue).
};
extern struct tty_struct tty_table[]; // tty structure array.
```

---

The Linux kernel uses the array `tty_table[]` to store information about each terminal device in the system.

Each array item is a data structure `tty_struct` that corresponds to a terminal device in the system. The Linux 0.12 kernel supports a total of three terminal devices, one for the console device and the other two for the serial terminal devices that use the two serial ports on the system.

The `termios` structure is used to store the io attribute of the corresponding terminal device. A detailed description of the structure is described below. `pgrp` is the process group ID, which indicates the process group in the foreground in a session, that is, the process group that currently owns the terminal device. `pgrp` is mainly used for job control operations of processes. `stopped` is a flag indicating whether the corresponding terminal device has been discontinued. The function pointer `*write()` is the output processing function of the terminal device. The function pointer `*write()` is the output processing function of the terminal device. For the console terminal, it is responsible for driving the display hardware, displaying characters and other information on the screen, and for the serial terminal connected through the system serial port, it is responsible for outputting Characters are sent to the serial port.

The data processed by the terminal is stored in the character buffer queues (or character lists) of the three `tty_queue` structures. The structure is shown below:

---

```
struct tty_queue {
    unsigned long data;                // Current data statistics.
                                         // Serial port address for serial terminals.
    unsigned long head;                // The header of the data in the buffer.
    unsigned long tail;                // The data tail pointer.
    struct task_struct * proc_list;    // Processes waiting for this buffer queue.
    char buf[1024];                    // The buffer of the queue.
};
```

---

The buffer length of each `tty` character queue is 1K bytes. The read buffer queue `read_q` is used to temporarily store the original character sequence input from the keyboard or the serial terminal; the write buffer queue `write_q` is used to store data written to the console display or the serial terminal; The auxiliary queue secondary is used to store the data taken from the `read_q` and processed (filtered) by the canonical mode program (based on the `ICANON` flag), and the data also referred to as cooked mode data. This is the canonical input data after the canonical program converts the special characters in the original data, such as the backspace character, and it is read and used by the program in the character line unit. The upper terminal read function `tty_read()` is used to read the characters in the secondary queue.

When reading in the data typed by the user, the interrupt handler is only responsible for putting the original character data into the input buffer queue, and the `C` function (`copy_to_cooked()`) called during the interrupt processing handles the character conversion. For example, when a process writes data to a terminal, the terminal driver calls the canonical mode function `copy_to_cooked()`, writes all the data in the user buffer to the buffer queue, and sends the data to the terminal for display. When a key is pressed on the terminal, the triggered keyboard interrupt handler puts the character corresponding to the key scan code into the read queue `read_q`, and calls the canonical mode handler to process the characters in `read_q` and put them into the auxiliary queue secondary. At the same time, if the echo flag (`L_ECHO`) of the terminal device is set, the character is also placed in the write queue `write_q`, and the terminal write function is called to display the character on the screen. Usually, the echo flag is set except for typing a password or other special requirements. We can change these flag values by modifying the information in the `termios` structure of the terminal.

The above `tty_struct` structure also includes a `termios` structure defined in the `include/termios.h` header file,

the contents of which are as follows:

---

```
struct termios {
    unsigned long c_iflag;          /* input mode flags */
    unsigned long c_oflag;          /* output mode flags */
    unsigned long c_cflag;          /* control mode flags */
    unsigned long c_lflag;          /* local mode flags */
    unsigned char c_line;           /* line discipline */           // or speed
    unsigned char c_cc[NCCS];      /* control characters */
};
```

---

Where `c_iflag` is the input mode flag set. The Linux 0.12 kernel implements all 11 input flags defined by POSIX.1, as described in the `termios.h` header file. The terminal device driver uses these flags to control how the characters entered by the terminal are transformed (filtered). For example, do you need to convert the entered newline character (NL) into a carriage return (CR), whether you need to convert the input uppercase characters to lowercase characters (because some terminal devices can only input uppercase characters before). In the Linux 0.12 kernel, the associated handler is `copy_to_cooked()` in the `tty_io.c` file. See also lines 86 - 99 of the `include/termios.h` file.

`c_oflag` is the set of output mode flags. The terminal device driver uses these flags to control how characters are output to the terminal, primarily in the `tty_write()` function of `tty_io.c`. See lines 102-132 of the `termios.h` file.

`c_cflag` is a control mode flag set, which is mainly used to define serial terminal transmission characteristics, including baud rate, number of character bits, and number of stop bits. See lines 135 -- 166 in the `termios.h` file.

`c_lflag` is a local mode flag set, mainly used to control the driver's interaction with the user. For example, if you need to echo (Echo) characters, whether you need to display the deleted characters directly on the screen, and whether you need to make the control characters typed on the terminal generate signals. These operations are mainly used in the `copy_to_cooked()` function and `tty_read()`. For example, if the ICANON flag is set, it indicates that the terminal is in the canonical mode input state, otherwise the terminal is in the non-canonical mode. If the ISIG flag is set, it means that the system needs to generate a corresponding signal when receiving the control characters INTR, QUIT, and SUSP issued by the terminal. See lines 169 -- 183 in the `termios.h` file.

The above four types of flag sets are all unsigned long, and each bit can represent a flag, so each flag set can have up to 32 input flags. All of these flags and their meanings can be found in the `termios.h` header file.

The `c_cc[NCCS]` array contains all the special characters that can be modified by the terminal. Where NCCS (=17) is the length value of the array. For example, you can modify the break character (^C) to be generated by other keys. The default value of the terminal's `c_cc[]` array is defined in the `include/linux/tty.h` file. The array item symbol names are defined when the program references the items in the array. These names begin with the letter V, such as VINTR, VMIN. See lines 67-83 of the `termios.h` file.

Therefore, using the system call `ioctl` or using the correlation function (`tcsetattr()`), we can change the terminal's setting parameters by modifying the information in the `termios` structure. The canonical mode function operates on these setup parameters. For example, the control terminal should echo the typed characters, set the baud rate of the serial terminal transmission, clear the read buffer queue, and write the buffer queue.

When the user modifies the terminal parameters and resets the canonical mode flag, the terminal can be set to work in the raw mode. At this point, the canonical mode handler will transfer the data entered by the user to the user intact, and the carriage return is treated as a normal character. Therefore, when the user uses the

system-call read, some decision-making scheme should be made to determine when the system-call read is completed and returned. This will be determined by the VTIME and VMIN control characters in the terminal termios structure. These two are the timeout timing values for the read operation. VMIN indicates the minimum number of characters to read in order to satisfy the read operation; VTIME is a read operation wait timing value.

We can use the command `stty` to view the settings of the flags in the current terminal device termios structure. Typing the `stty` command at the Linux 0.1x system command line prompt displays the following information:

---

```
[/root]# stty
-----Characters-----
INTR:  '^C'  QUIT:  '^\'  ERASE:  '^H'  KILL:  '^U'  EOF:    '^D'
TIME:   0    MIN:   1    SWTC:  '^@'  START:  '^Q'  STOP:   '^S'
SUSP:  '^Z'  EOL:   '^@'  EOL2:  '^@'  LNEXT:  '^V'
DISCARD: '^O'  REPRINT: '^R'  RWERASE: '^W'
-----Control Flags-----
-CSTOPB  CREAD -PARENB -PARODD  HUPCL -CLOCAL -CRTSCTS
Baud rate: 9600 Bits: CS8
-----Input Flags-----
-IGNBRK -BRKINT -IGNPAR -PARMRK -INPCK -ISTRIP -INLCR -IGNCR
ICRNL -IUCLC  IXON  -IXANY  IXOFF -IMAXBEL
-----Output Flags-----
OPOST -OLCUC  ONLCR -OCRNL -ONOCR -ONLRET -OFILL -OFDEL
Delay modes: CR0 NLO TAB0 BS0 FFO VTO
-----Local Flags-----
ISIG ICANON -XCASE  ECHO -ECHOE -ECHOK -ECHONL -NOFLSH
-TOSTOP ECHOCTL ECHOPRT ECHOKE -FLUSHO -PENDIN -IEXTEN
rows 0 cols 0
```

---

Among them, the minus sign indicates that there is no setting. Also for current Linux systems, you need to type '`stty -a`' to display all of this information, and the display format is different.

The above main data structures used by the terminal program and the relationship between them can be seen in Figure 10-2.

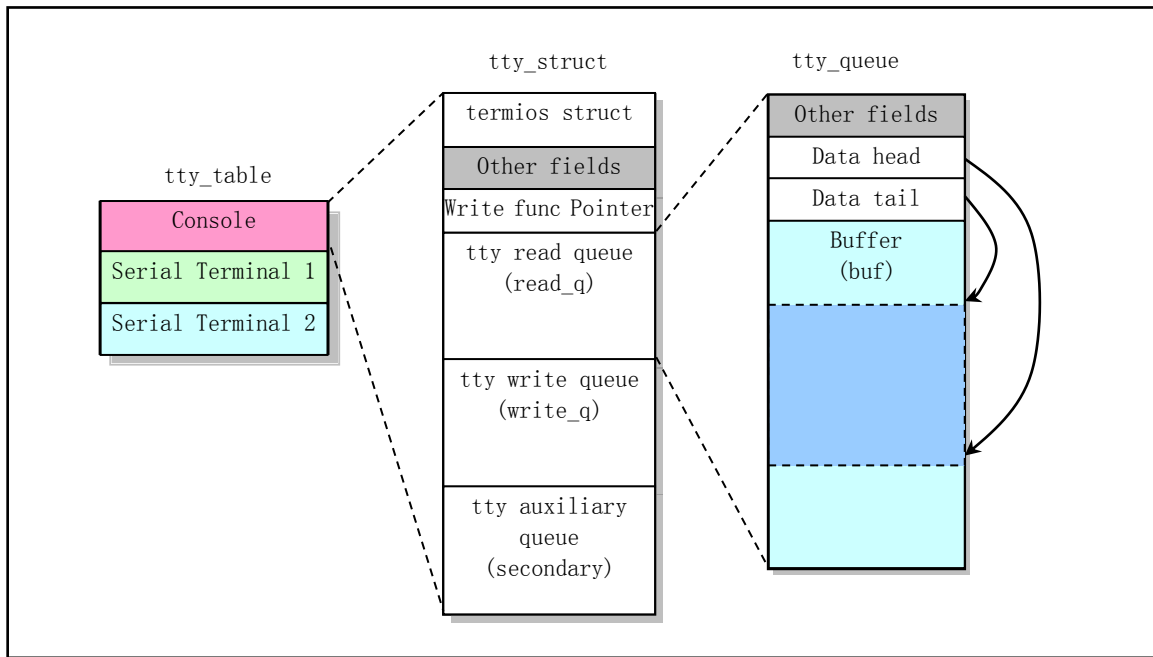


Figure 10-2 Data structures in the terminal program

### 10.1.4 Canonical mode and non-canonical mode

As can be seen from the foregoing, according to the way in which the terminal program processes the input or output data, we can divide the terminal working mode into a canonical mode and a non-canonical mode or a raw mode. For the canonical mode, the data passing through the terminal device will be modified and transformed according to relevant rules or standards, and then sent out. The transformation function used is often referred to as a line discipline or a line procedure module. For the non-canonical mode, the line discipline program only transfers data between the terminal and the process, and does not perform the transformation processing of the data. Below we detail the processing rules used in these two modes, as well as the implementation in the kernel code.

#### 10.1.4.1 Canonical mode

When the `ICANON` flag in `c_lflag` is set, the terminal input data is processed in accordance with the canonical mode. The input characters are now assembled into lines, and the process reads as a line of characters. When a line of characters is entered, the terminal driver will return immediately. The delimiters for the line are `NL`, `EOL`, `EOL2`, and `EOF`. In addition to the last `EOF` (end of file) will be deleted by the handler, the remaining four characters will be returned to the caller as the last character of a line.

In canonical mode, the following characters entered by the terminal will be processed: `ERASE`, `KILL`, `EOF`, `EOL`, `REPRINT`, `WERASE`, and `EOL2`.

`ERASE` is a backspace. In canonical mode, when the `copy_to_cooked()` function encounters this character, the last character entered in the buffer queue is deleted. If the last character in the queue is the character of the previous line (for example, `NL`), no processing is done. This character is then ignored and not placed in the buffer queue.

`KILL` is a line delete character. It deletes the last line of characters in the queue. This character is then ignored.

`EOF` is the end of file. This character and the end-of-line characters `EOL` and `EOL2` will be treated as carriage returns in the `copy_to_cooked()` function. The character encountered in the read operation function will



return immediately. EOF characters are not placed in the queue but ignored.

REPRINT and WERASE are characters recognized in extended canonical mode. REPRINT will cause all unread inputs to be output. WERASE is used to remove words (skip whitespace characters). In Linux 0.12, the program ignores the recognition and processing of these two characters.

#### 10.1.4.2 Non-Canonical mode

If the ICANON flag in the `c_iflag` set is in reset state, the terminal program operates in the non-canonical mode. At this time, the terminal program does not process the above characters, but treats them as ordinary characters, and the input data does not have the concept of a line. When the terminal program returns to the read process is determined by the values of MIN and TIME (VMIN, VTIME). These two variables are variables in the `c_cc[]` array. By modifying them you can change how the process reads characters in non-canonical mode.

MIN indicates the minimum number of characters to read for a read operation; TIME specifies the timeout value for which characters are waiting to be read (the unit of measure is 1/10th of a second). According to their values, they can be described in four cases.

1. MIN>0, TIME>0

At this time TIME is a character interval timeout timing value, which will take effect after receiving the first character. If a MIN character is received before the timeout, the read operation returns immediately. If a timeout occurs before the MIN characters are received, the read operation returns the number of characters that have been received. At least one character can be returned at this time. Therefore, if the secondary is empty before receiving a character, the read process will be blocked (sleep).

2. MIN>0, TIME=0

At this time, the read operation returns only when MIN characters are received. Otherwise, wait indefinitely (blocking).

3. MIN=0, TIME>0

At this time TIME is a read operation timeout timing value. When a character is received or has timed out, the read operation returns immediately. If it is timed back, the read operation returns 0 characters.

4. MIN=0, TIME=0

With this setting, if there is data in the queue that can be read, the read operation reads the number of characters required. Otherwise, it returns 0 characters immediately.

In the above four cases, MIN only indicates the minimum number of characters read. If the process requires more characters than MIN, then the current requirements of the process may be met as long as there are characters in the queue. For the read operation of the terminal device, see the `tty_read()` function in the program `tty_io.c`.

### 10.1.5 Console Terminal and Serial Terminal Devices

Two types of terminals can be used in the Linux 0.12 system, one is the console terminal on the host, and the other is the serial hardware terminal device. The console terminal is operated and managed by the keyboard interrupt handler program `keyboard.s` and the display control program `console.c` in the kernel. It receives the display characters or control messages passed by the upper `tty_io.c` program and controls the display of characters on the host screen. At the same time, the console (host) transfers the code generated by the keyboard to the `tty_io.c` program via `keyboard.s`. The serial terminal device is connected to the serial port of the computer through a line, and directly exchanges information with `tty_io.c` through the serial program `rs_io.s` in the kernel.

The two programs, `keyboard.s` and `console.c`, are actually much like emulation programs that use a monitor and keyboard to emulate a hardware terminal device in a Linux system host. Just because it is on the host, we call this simulated terminal environment a console terminal, or directly called a console. The functions

implemented by these two programs are equivalent to the role of the terminal processing program in the ROM of a serial terminal device (except for the communication part), and also like a terminal emulation software on a normal PC. So, although the two programs are in the kernel, we can still look at them independently. The main difference between this "simulated terminal" and the ordinary hardware terminal device is that there is no need to communicate the driver through the serial line. Therefore, the `keyboard.s` and `console.c` programs must be able to simulate all the hardware processing functions of an actual terminal device (such as DEC's VT100 terminal), that is, all processing functions except the communication part in the terminal device firmware. See Figure 10-3 for the differences in processing structure and similarities between the console terminal and the serial terminal device. Therefore, if we have a certain understanding of the working principle of general hardware terminal equipment or terminal emulation program, then reading these two programs will not encounter any difficulties.

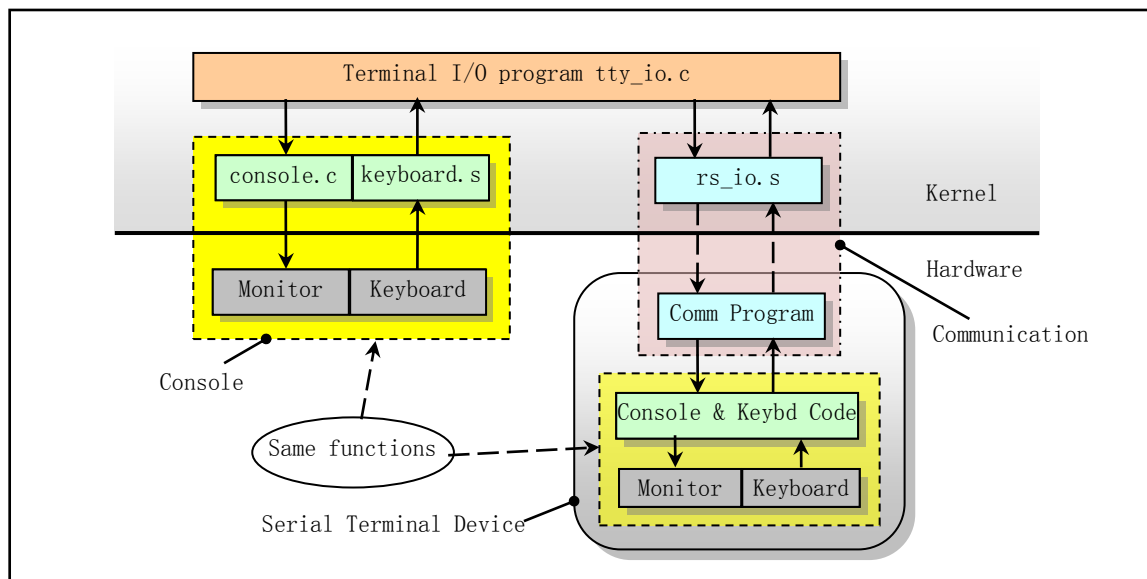


Figure 10-3 Console terminal and serial terminal device diagram

#### 10.1.5.1 Console Driver

In the Linux 0.12 kernel, the terminal console driver involves the `keyboard.S` and `console.c` programs. `keyboard.S` is used to process the characters typed by the user, put them into the read buffer queue `read_q`, and call the `copy_to_cooked()` function to read the characters in `read_q`, and then convert them into the auxiliary buffer queue `secondary`. The `console.c` program implements the output display processing of the code received by the console terminal.

For example, when the user types a character on the keyboard, it causes a keyboard interrupt response (interrupt request signal `IRQ1`, corresponding to interrupt number `INT 33`) to execute the handler. At this point, the keyboard interrupt handler reads the corresponding keyboard scan code from the keyboard controller, and then translates the corresponding character according to the used keyboard scan code mapping table into the `tty` read queue `read_q`. Then call the C function `do_tty_interrupt()` of the interrupt handler, which directly calls the canonical mode function `copy_to_cooked()` to filter the character and put it into the `tty` auxiliary queue 'secondary', and put the character into the `tty` write queue `write_q`. Then call the write console function `con_write()` for console output (display) processing. At this time, if the `echo` attribute of the terminal is set, the character will be displayed on the screen. The `do_tty_interrupt()` and `copy_to_cooked()` functions are implemented in `tty_io.c`. The entire operation process is shown in Figure 10-4.

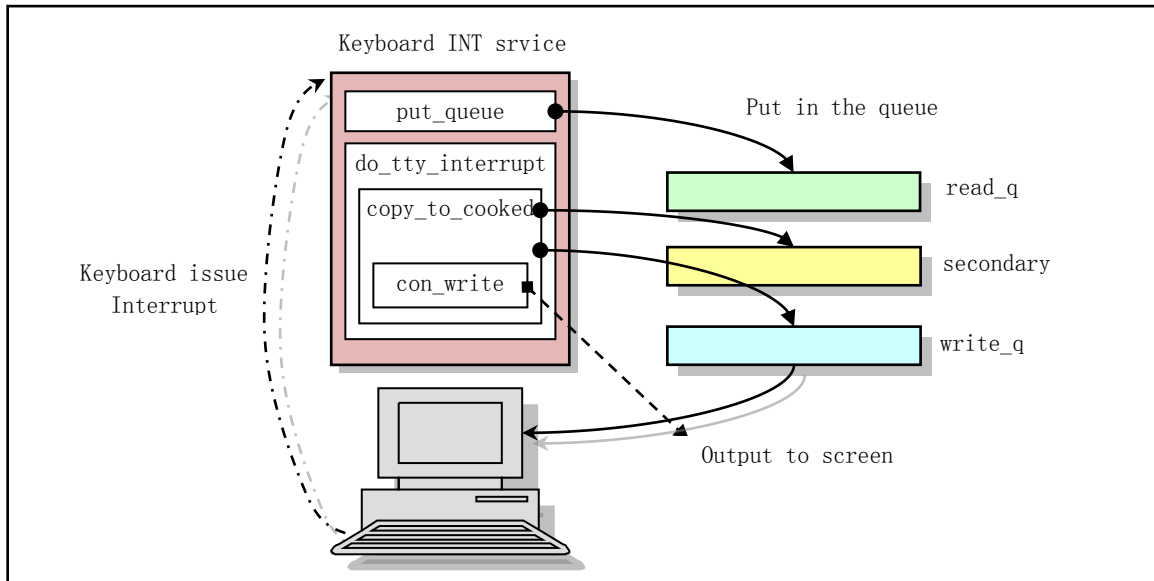


Figure 10-4 Console keyboard interrupt service

For the process to perform tty write operations, the terminal driver handles the characters one by one. When the write buffer queue `write_q` is not full, it takes a character from the user buffer and processes it and puts it in `write_q`. When the user data is all placed in the `write_q` queue or the `write_q` is full at this time, the write function specified in the terminal structure `tty_struct` is called, and the data in the `write_q` buffer queue is output to the console screen. For console terminals, the write function is `con_write()`, which is implemented in the `console.c` program.

Therefore, in general, the keyboard interrupt handler `keyboard.S` of the console terminal is mainly used to put the characters typed by the user into the `read_q` buffer queue. Its on-screen display handler, `console.c`, is used to retrieve characters from the `write_q` queue and display them on the screen. The relationship between all three character buffer queues and the above functions or files can be clearly shown in Figure 10-5.

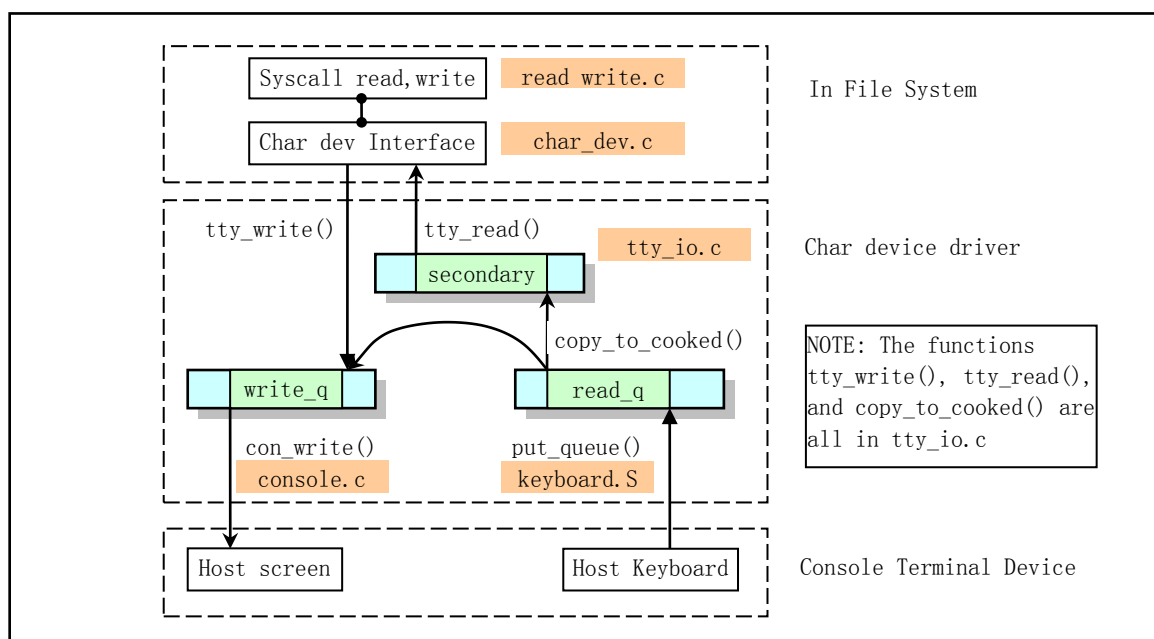


Figure 10-5 Relationships between console character queues and functions

### 10.1.5.2 Serial Terminal Driver

The programs that handle serial terminal operations are `serial.c` and `rs_io.s`. The `serial.c` program is responsible for initializing the serial port and enabling the serial interrupt transmit character operation by unmasking the transmit hold register empty interrupt. The `rs_io.s` program is a serial interrupt handler. It is mainly processed according to the four reasons for causing serial interrupts.

The serial interrupts caused by the system are: (1) due to changes in the modem state; (2) due to changes in line status; (3) due to received characters; (4) due to the requirement to send characters (The flag of transmitter holding register empty is set). The processing of the first two cases that cause an interrupt is reset by reading the corresponding status register. For the case of receiving a character, the program first needs to put the character into the read buffer queue `read_q`, and then call the `copy_to_cooked()` function to convert it into a canonical mode character in character line units and place it in the auxiliary queue 'secondary'. For the case where a character needs to be sent, the program first takes out a character from the location pointed by a tail pointer of the write buffer queue `write_q` and sends it out, and then checks whether the write buffer queue is empty, and if there are still have characters, it performs transmission operation cyclically.

For terminals that are accessed through the system serial port, in addition to processing similar to the console, input/output handling operations for serial communication are required. The reading of data is put into the read queue `read_q` by the serial interrupt handler, and then the same operation as the console terminal is performed.

For example, for a terminal connected to serial port 1, the typed characters will first be transmitted to the host over the serial line, causing host serial port 1 to issue an interrupt request. At this point, the serial port interrupt handler puts the character into the tty read queue `read_q` of serial terminal 1, and then calls the C function `do_tty_interrupt()` of the interrupt handler. The function will directly call the row rule function `copy_to_cooked()` to filter the characters and put them into the tty auxiliary queue `secondary`. At the same time, the character is placed in the tty write queue `write_q`, and the function `rs_write()` of the serial terminal 1 is called. The function, in turn, sends the character back to the serial terminal, and if the echo attribute of the terminal is set, the character is displayed on the screen of the serial terminal.

When the process needs to write data to a serial terminal, the operation process is similar to that of the write terminal, except that the write function in the `tty_struct` data structure of the terminal is the serial terminal write function `rs_write()`. This function unmask the "Transmit Holding Register Empty Enable Interrupt", causing a serial interrupt to occur when the transmit holding register is empty. The serial interrupt process takes a character from the `write_q` write buffer queue and puts it into the transmit holding register for character transmission according to the cause of the interrupt. The operation process also sends a character for one interrupt, and finally, when the `write_q` is empty, the transmit hold register empty enable interrupt bit is masked again, thereby preventing such an interrupt from being generated again.

The serial terminal write function `rs_write()` is implemented in the `serial.c` program, and the serial interrupt program is implemented in `rs_io.s`. See Figure 10-6 for the relationship between the three buffer queues of serial terminals and functions.

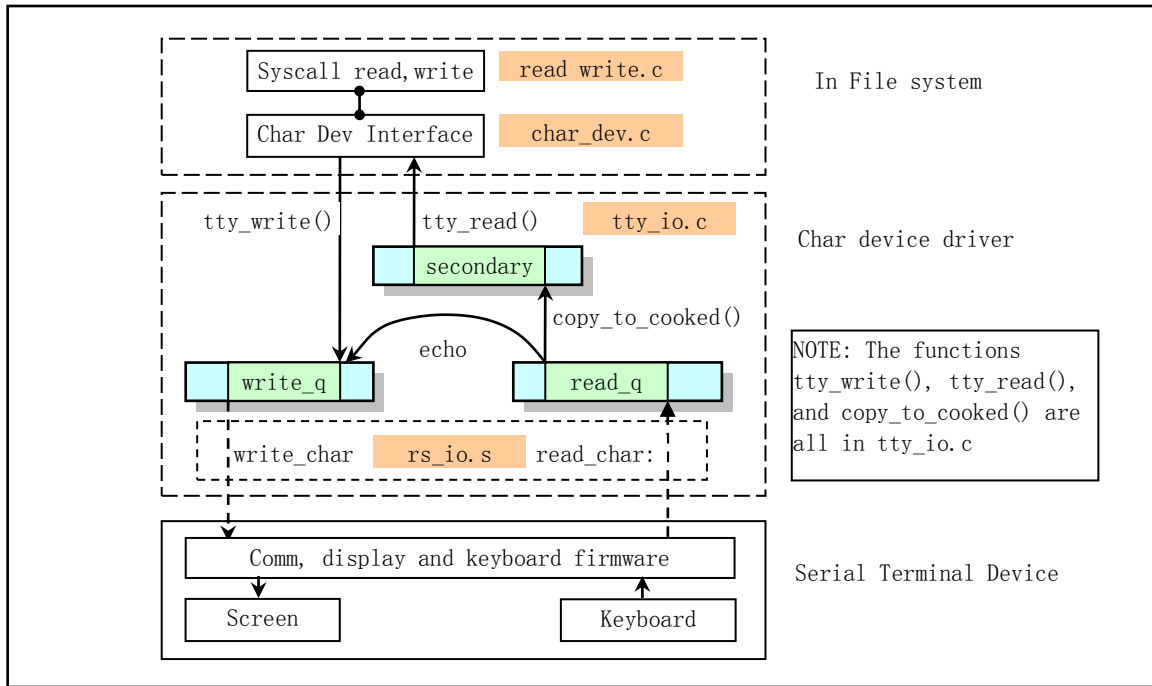


Figure 10-6 The relationship between character queues and functions of serial terminal devices

As can be seen from Figure 10-6, the main difference between the serial terminal and the console is that the serial terminal replaces the console operation display and keyboard program console.c and keyboard.S with the program rs\_io.s, and the rest processing method is exactly the same.

### 10.1.6 Terminal Driver Interface

Typically, users interact with devices through the file system. Each device has a file name and, accordingly, an index node (i node) in the file system. But the file type in the i-node is the device type to distinguish it from other regular files. So users can access the device directly using file system calls. The terminal driver also provides a call interface function to the file system for this purpose. The interface between the terminal driver and other programs in the system is implemented using the generic functions in the tty\_io.c file, which implements the read terminal function tty\_read() and the write terminal function tty\_write(), and the input line rule function copy\_to\_cooked(). In addition, in the tty\_ioctl.c program, an input/output control function (or system-call) tty\_ioctl() that modifies terminal parameters is implemented. The setting parameters of the terminal are placed in the termios structure in the terminal data structure. There are many parameters and complex, please refer to the description in the include/termios.h file.

For different terminal devices, there can be different canonical mode programs to match. However, there is only one canonical mode function in Linux 0.12, so the row rule field 'c\_line' in the termios structure does not work and is set to 0.

## 10.2 keyboard.S

### 10.2.1 Function descriptions

The keyboard.S program mainly includes a keyboard interrupt handler, and the related C function called is

do\_tty\_interrupt(). The program first sets the state flag variable 'mode' to be used later in the program according to the state of the special keys on the keyboard (for example, Alt, Shift, Ctrl, and Caps). Then, according to the key scan code that causes the keyboard interrupt, the corresponding scan code processing subroutine that has been arranged into the jump table is used to put the character associated with the scan code into the read character queue (read\_q). Next, call the C function do\_tty\_interrupt() (tty\_io.c, line 397), which contains only one call to the canonical function copy\_to\_cooked(). The main purpose of this canonical mode function is to put the characters in the read\_q read buffer queue into the normal mode queue (secondary queue) after proper processing, and if the corresponding terminal device sets the echo flag, then the program will also place the character directly into the write queue (write\_q), and the write tty function will be called to display the characters on the terminal screen.

For the scan code of the AT keyboard, when the key is pressed, the scan code of the key is sent, but when the key is released, two bytes will be sent, the first one is 0xf0, and the second is the same scan code as when the key pressed. For backward compatibility, the PC designer converted the scan code from the AT keyboard to the scan code of the old PC/XT standard keyboard, so the program only needs to process the PC/XT scan code. For a description of the keyboard scan code, see the description in section 10.3 of the program listing. Here is also a reminder that in keyboard-related terms, 'make' means that the key is pressed; 'break' means that the key is released.

In addition, the file name of this program is different from other gas assembly language programs, and its suffix is in uppercase '.S'. Using such a suffix allows the assembler to use the GNU C compiler's preprocessor CPP, which means that many C language directives can be used in your assembly language program. For example, "#include", "#if", etc., refer to the specific usage of the relevant code in the program.

### 10.2.2 Code Annotation

Program 10-1 linux/kernel/chr\_drv/keyboard.S

---

```
1  /*
2  *  linux/kernel/keyboard.S
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *      Thanks to Alfred Leung for US keyboard patches
9  *      Wolfgang Thiel for German keyboard patches
10 *      Marc Corsini for the French keyboard
11 */
12
13 /* KBD_FINNISH for Finnish keyboards
14 * KBD_US for US-type
15 * KBD_GR for German keyboards
16 * KBD_FR for Frech keyboard
17 */
18 // Define the type of keyboard used for later selection of character mapping code table.
19 #define KBD_FINNISH          // Linus is a Finnish man ☺
20 .text
21 // Declared as a global variable, used to set the keyboard interrupt descriptor at init.
22 .globl _keyboard_interrupt    // same as keyboard_interrupt().
```

```

22
23 /*
24  * these are for the keyboard read functions
25  */
26 // Size is the keyboard buffer queue length in bytes.
27 size = 1024 /* must be a power of two ! And MUST be the same
                as in tty_io.c !!!! */

// The following is the offset of each field in the keyboard buffer queue data structure tty_queue
// (include/linux/tty.h, line 22).
28 head = 4 // Offset of the head field in the tty_queue structure.
29 tail = 8 // Offset of the tail field.
30 proc_list = 12 // the wait process field offset (wait for this buffer).
31 buf = 16 // offset of the buffer field in the tty_queue structure.
32

// The three flag bytes (mode, leds, e0) are used in this program. The meaning of each bit in
// each byte flag is as follows:
// (1) 'mode' is the press status flag of the special keys. Used to indicate the state of the
// case conversion keys (caps), alternate keys (alt), control keys (ctrl), and shift keys.
// bit 7 - caps key pressed; bit 3 - right ctrl key pressed;
// bit 6 - caps key status; bit 2 - left ctrl key pressed;
// bit 5 - right alt key pressed; bit 1 - right shift key pressed;
// bit 4 - left alt key pressed; bit 0 - left shift key pressed.
// (2) 'leds' is a status flag for indicating the keyboard indicator, that is, the status of
// the LED tube indicating the num-lock, caps-lock, and scroll-lock.
// bit 7-3 all zeros, not used.; bit 1 num-lock (Initial setting 1);
// bit 2 caps-lock; bit 0 scroll-lock.
// (3) 'e0' This flag is set when the scan code is 0xe0 or 0xe1. It indicates that it is followed
// by a one or two character scan code. Usually, if the scan code 0xe0 is received, it means
// that there is still one character following it; if the scan code 0xe1 is received, it means
// that it is followed by 2 characters. See the description after the program list.
// bit 1 = 1 Received the 0xe1; bit 0 = 1 Received the 0xe0.
33 mode: .byte 0 /* caps, alt, ctrl and shift mode */
34 leds: .byte 2 /* num-lock, caps, scroll-lock mode (nom-lock on) */
35 e0: .byte 0
36
37 /*
38  * con_int is the real interrupt routine that reads the
39  * keyboard scan-code and converts it into the appropriate
40  * ascii character(s).
41  */ // Note: The original comment above is obsolete.

///// Keyboard interrupt handler entry point.
// When the keyboard controller receives a key pressed operation of the user, it sends a interrupt
// request signal IRQ1 to the interrupt controller. This keyboard interrupt handler is executed
// when the CPU responds to the request. The interrupt handler reads the key scan code from
// the keyboard controller port (0x60) and calls the corresponding scan code subroutine for
// processing.
// First, read the scan code of the current button from port 0x60, and then determine whether
// the scan code is 0xe0 or 0xe1. If so, it immediately responds to the keyboard controller
// and sends an End of Interrupt (EOI) signal to the interrupt controller to allow the keyboard
// controller to continue generating interrupt signals, allowing us to receive subsequent
// characters. If the two special scan codes are not received, we invoke the corresponding key
// processing subroutine in the key jump table key_table according to the scan code value, and

```

```
// put the character corresponding to the scan code into the read character buffer queue read_q.  
// Then, after responding to the keyboard controller and sending the EOI signal, the function  
// do_tty_interrupt() is called (actually calling copy_to_cooked()) to process the characters  
// in read_q and place them in the secondary auxiliary queue.
```

```
42 _keyboard_interrupt:  
43     pushl %eax  
44     pushl %ebx  
45     pushl %ecx  
46     pushl %edx  
47     push %ds  
48     push %es  
49     movl $0x10,%eax      // Set ds and es to kernel data segment.  
50     mov %ax,%ds  
51     mov %ax,%es  
52     movl _blankinterval,%eax  
53     movl %eax,_blankcount // preset black screen time count (blankinterval).  
54     xorl %eax,%eax       /* %eax is scan code */  
55     inb $0x60,%al        // Read scan code to al.  
56     cmpb $0xe0,%al       // jump to set_e0 if scan code is 0xe0  
57     je set_e0  
58     cmpb $0xe1,%al       // jump to set_e1 if scan code is 0xe1  
59     je set_e1  
60     call key_table(,%eax,4) // key handler: key_table + eax*4 (see line 513).  
61     movb $0,e0           // Reset e0 flag after return.
```

```
// The following code (lines 62-72) performs a hardware reset operation on the PC standard  
// keyboard circuit using the 8255A. Port 0x61 is the address of the 8255A output port B. Bit  
// 7 (PB7) of this port is used to disable and enable handling of keyboard data. This program  
// is used to respond to the received scan code by first disabling the keyboard and then  
// immediately re-enabling the keyboard to work.
```

```
62 e0_e1:  inb $0x61,%al      // Get port B state, PB7 is used to enable/disable keyboard.  
63         jmp lf             // delay for awhile.  
64 1:      jmp lf  
65 1:      orb $0x80,%al      // set bit7 of al.  
66         jmp lf  
67 1:      jmp lf  
68 1:      outb %al,$0x61     // set port B bit7 to disable keyboard.  
69         jmp lf  
70 1:      jmp lf  
71 1:      andb $0x7F,%al     // reset bit7 of al.  
72         outb %al,$0x61     // reset port B bit7 to enable keyboard.  
73         movb $0x20,%al     // send EOI signal to 8259A chip.  
74         outb %al,$0x20  
75         pushl $0           // The console tty = 0, and pushed as a parameter.  
76         call _do_tty_interrupt // converted to canonical mode  
77         addl $4,%esp       // Discard the parameters of the stack.  
78         pop %es  
79         pop %ds  
80         popl %edx  
81         popl %ecx  
82         popl %ebx  
83         popl %eax  
84         iret
```



```

85 set_e0: movb $1,e0          // When 0xe0 is received, bit0 of e0 flag is set.
86         jmp e0_e1
87 set_e1: movb $2,e0          // When 0xe1 is received, bit1 of e0 flag is set.
88         jmp e0_e1
89
90 /*
91  * This routine fills the buffer with max 8 bytes, taken from
92  * %ebx:%eax. (%ebx is high). The bytes are written in the
93  * order %al,%ah,%eal,%eah,%bl,%bh ... until %eax is zero.
94  */
// First, take the console's read buffer queue read_q address from the buffer queue address
// table table_list (tty_io.c, line 81), then copy the characters in the al register to the
// read queue header pointer and move the head pointer forward by 1 byte. If the head pointer
// moves out of the end of the read buffer, let it wrap around to the beginning of the buffer,
// and then see if the buffer queue is full at this time, that is, compare the queue head pointer
// with the tail pointer (equal means full). If it is full, discard the remaining characters
// that may be in ebx:eax. If the buffer is not full, transfer the data in ebx:eax to the right
// by 8 bits (ie move the ah value to al, bl -> 0ah, bh -> bl), and then repeat the above process
// for al. Until all characters have been processed, the current head pointer value is saved,
// and then check if there is a process waiting for the read queue, and if so, wake up.
95 put_queue:
96     pushl %ecx
97     pushl %edx              // Get read buffer queue pointer.
98     movl _table_list,%edx   # read-queue for console
99     movl head(%edx),%ecx    // Take the queue head pointer -> ecx.
100 1:   movb %al,buf(%edx,%ecx) // put char in al into the head pointer position.
101     incl %ecx              // moved forward by 1 byte.
102     andl $size-1,%ecx      // Adjust the head pointer.
103     cmpl tail(%edx),%ecx   # buffer full - discard everything
104     je 3f
105     shrdl $8,%ebx,%eax     // Move ebx right 8 bits into eax, ebx is unchanged.
106     je 2f                  // Are there still characters? if no then jump.
107     shr1 $8,%ebx           // Move ebx value to the right by 8 bits
108     jmp 1b
109 2:   movl %ecx,head(%edx)   // If all chars in queue, the head pointer is saved.
110     movl proc_list(%edx),%ecx // Waiting process pointer for this queue?
111     testl %ecx,%ecx        // Check if there are processes waiting for this queue.
112     je 3f                  // No, then jump (to line 114).
113     movl $0,(%ecx)         // Yes, the process is awake.
114 3:   popl %edx
115     popl %ecx
116     ret
117
// From here on, it is the subroutine of each key corresponding to the pointer in the jump table
// key_table, and called from the statement in line 60 above. The jump table key_table starts
// at line 513.
// The following code sets the corresponding bit in the mode flag according to the scan code
// of ctrl or alt. If the 0xe0 scan code is received before the scan code (the e0 flag is set),
// it means that the ctrl or alt key on the right side of the keyboard is pressed, and the bit
// in the mode flag is set correspondingly to ctrl or alt.
118 ctrl: movb $0x04,%al      // 0x04 (bit2), the left ctrl key in mode.
119         jmp 1f
120 alt:   movb $0x10,%al      // 0x10 (bit4), the left alt key in mode.

```

```

121 1:      cmpb $0,e0                // e0 set? (key pressed is right ctrl/alt key)?
122      je 2f                      // no, jump (to line 124).
123      addb %al,%al                // yes, change to right-click flag (bit3 or bit5).
124 2:      orb %al,mode              // Set the corresponding bit in the mode flag.
125      ret
// This code handles the scan code when the ctrl or alt key is released, and resets the
// corresponding bit in the mode flag. In the operation, it is necessary to judge whether it
// is the ctrl or alt key on the right side of the keyboard according to whether the e0 flag
// is set.
126 unctrl: movb $0x04,%al           // bit2 is for the left ctrl key in mode.
127      jmp 1f
128 unalt:  movb $0x10,%al           // 0x10 is the bit 4 for the left alt key in mode.
129 1:      cmpb $0,e0                // e0 set? (Is the right ctrl/alt key released?)?
130      je 2f                      // no, jump (to line 132)
131      addb %al,%al                // yes, then change to reset flag bit (bit3 or bit5).
132 2:      notb %al                 // Reset corresponding bit in mode flag.
133      andb %al,mode
134      ret
135
// This code handles the scan codes when the left or right shift keys are pressed or released
// to set and reset the corresponding bits in mode.
136 lshift:
137      orb $0x01,mode              // left shift key is pressed, set mode bit0.
138      ret
139 unlshift:
140      andb $0xfe,mode             // left shift key is released, reset mode bit 0.
141      ret
142 rshift:
143      orb $0x02,mode              // right shift key is pressed, set mode bit1.
144      ret
145 unrshift:
146      andb $0xfd,mode             // right shift key is released, reset mode bit1.
147      ret
148
// This code handles caps scan code. It is known by mode bit7 that the caps key is currently
// in the pressed state. If yes, return, otherwise flip the bit 6 (caps key pressed) of mode flag and
// the caps-lock bit (bit 2) of the leds flag, and set the caps key pressed flag bit (bit 7).
149 caps:  testb $0x80,mode          // bit7 set ? (caps pressed?)
150      jne 1f                      // yes, jump to line 169.
151      xorb $4,leds                // flip caps-lock(bit2) in leds flag.
152      xorb $0x40,mode             // flip caps state (bit6) in mode flag.
153      orb $0x80,mode              // set bit7 (caps pressed) in mode flag.
// This code turns the LED indicator on or off according to the leds flag.
154 set_leds:
155      call kb_wait                 // Wait for controller input buffer to be empty.
156      movb $0xed,%al              /* set leds command */
157      outb %al,$0x60              // Send kbd command 0xed to port 0x60.
158      call kb_wait
159      movb leds,%al               // get leds flag as parameter.
160      outb %al,$0x60
161      ret
162 uncaps: andb $0x7f,mode           // caps released, then reset bit7 in mode flag.
163      ret

```

```

164 scroll:
165     testb $0x03,mode           // ctrl key pressed too?
166     je 1f                     // no, jump (to line 169).
167     call _show_mem            // yes, display memory info (mm/memory.c, line 457)
168     jmp 2f
169 1:    call _show_state         // no, show process state info (kernel/sched.c, 45)
170 2:    xorb $1,leds             // flip bit0 of leds if scroll key pressed.
171     jmp set_leds              // turn on or off LED indicator based on leds flag.
172 num:  xorb $2,leds             // flip bit1 of leds flag if num key pressed.
173     jmp set_leds              // turn on or off LED indicator based on leds flag.
174
175 /*
176  * curosr-key/numeric keypad cursor keys are handled here.
177  * checking for numeric keypad etc.
178 */
// The following code first determines if the scan code is issued by the right numeric keypad
// buttons on the keyboard. If not, exit the this subroutine. If the last key Del(0x53) on the
// numeric keypad is pressed, then it is determined whether or not the "Ctrl-Alt-Del" key
// combination is pressed. If the combination key is pressed, jump to the system reboot code.
// Please refer to the XT keyboard scan code table given in section 10.2.3 after this program
// list, or browse the first set of keyboard scan code table in the appendix. As can be seen
// from the table, the scan code range of the keys on the numeric keypad are [0x47 - 0x53],
// and when they are used as direction keys, they will have the preamble code e0.
179 cursor:
180     subb $0x47,%al            // If scan code not in range [0x47 - 0x53], it ret (198).
181     jb 1f                     // That is, scan code is not generated from numeric keypad.
182     cmpb $12,%al              // (0x53 - 0x47 = 12)
183     ja 1f
184     jne cur2                  /* check for ctrl-alt-del */
// If it is equal to 12, it indicates that the 'del' key has been pressed, so it continues to
// check whether 'ctrl' and 'alt' are also pressed simultaneously.
185     testb $0x0c,mode          // ctrl key pressed? jump if not.
186     je cur2
187     testb $0x30,mode          // alt key pressed?
188     jne reboot                // yes, then jump to reboot system (line 594).

// Next, the code determines whether the numeric keypad keys are used as direction (page up/dn ,
// insert, delete, etc.) keys. If the pressed button is indeed on the keypad, and if the leading
// scan code e0 is received at this time, it means that the button on the keypad is used as
// the direction key. Then jump to handle the cursor movement or insert/delete button. If e0
// is not set, first check if the num-lock LED is on. If it is not lit, also perform cursor
// movement processing. However, if the num-lock light is on (indicating that the keypad is
// used as a numeric key) and the shift key is also pressed, then we will treat the keypad key
// at this time as a cursor movement operation.
189 cur2: cmpb $0x01,e0           /* e0 forces cursor movement */ // e0 flag set ?
190     je cur                    // jump to cur if e0 is set.
191     testb $0x02,leds          /* not num-lock forces cursor */ // test leds flag.
192     je cur                    // if num's LED is on, do cursor movement operation.
193     testb $0x03,mode          /* shift forces cursor */
194     jne cur                    // if shift is pressed, also do cursor movement operation.

// Finally, the numeric keypad keys are used as numeric keys. Then, the small number table
// num_table is queried according to the scan code, and the numeric characters corresponding

```

```

// to the keys are taken out and placed in the buffer queue. Since the number of characters
// placed in the character queue at a time requires <=8, it is necessary to clear ebx before
// jumping to put_queue to execute. See the comment on line 95.
195     xorl %ebx,%ebx
196     movb num_table(%eax),%al      // obtain the digital char with eax as index.
197     jmp put_queue                // parameter: 1-8 chars in ebx:eax.
198 1:     ret
199
// This code handles cursor movement or insert/delete buttons. Here, the representative
// character of the corresponding key in the cursor character table is first obtained. If the
// character is <='9' (5, 6, 2 or 3), this means that it is one of a Page Up, Page Dn, Insert,
// or Delete key, and the character '~' is need to be added to the function character sequence.
// However, this kernel does not recognize and process them. The code then moves the contents
// of ax into the eax high word, putting 'esc[' into ax, and with the characters in the high
// word of eax to form a moving sequence. Finally, the character sequence is placed in the
// character queue.
200 cur:   movb cur_table(%eax),%al  // get cursor char to al.
201       cmpb $'9',%al             // if <='9' (5,6,2 or 3) it's page up/dn or Ins/Del key
202       ja ok_cur                 // it needs to add char '~'.
203       movb $'~',%ah
204 ok_cur: shll $16,%eax            // move content of ax to high word of eax.
205       movw $0x5b1b,%ax          // put 'esc [' in ax
206       xorl %ebx,%ebx
207       jmp put_queue
208
209 #if defined(KBD_FR)
210 num_table:
211     .ascii "789 456 1230."      // ASCII code associated to keys on numeric keypad.
212 #else
213 num_table:
214     .ascii "789 456 1230,"
215 #endif
216 cur_table:
217     .ascii "HA5 DGC YB623"      // associated to the arrow, Ins, Del keys on keypad.
218
219 /*
220  * this routine handles function keys
221  */
// The subroutine converts the function key scan code into an escape character sequence and
// stores it in the read queue. The range of function keys to be checked by the code is F1--F12.
// The scan codes of the function keys F1--F10 are 0x3B-0x44, and the scan codes of F11 and
// F12 are 0x57, 0x58. First, F1--F12 is first converted into the corresponding sequence number
// 0--11, and then the function key table func_table is queried to obtain the corresponding
// escape character sequence and placed in the character queue. If the Alt key is pressed while
// a function key is pressed, only the switching operation of the control terminal is performed.
222 func:
223     subb $0x3B,%al              // F1 scan code is 0x3B
224     jb end_func                 // ret if not a function key
225     cmpb $9,%al                 // F1--F10 ?
226     jbe ok_func                 // jump if yes
227     subb $18,%al                // key F11, F12 ?
228     cmpb $10,%al                // key F11 ?
229     jb end_func                 // ret if order no less than F11's.

```

```

230      cmpb $11,%al          // key F12 ?
231      ja end_func          // ret if order no great than F12's.
232 ok_func:
233      testb $0x10,mode      // left alt pressed ?
234      jne alt_func         // jump to change console if yes.
235      cmpl $4,%ecx          /* check that there is enough room */
236      jl end_func          // ret if not enough space (4 bytes).
237      movl func_table(,%eax,4),%eax    // get escape character sequence of the key.
238      xorl %ebx,%ebx
239      jmp put_queue
// The following code handles the alt + Fn key combination to change the virtual control
// terminal. At this time, eax is the function key index or order number (F1 -- 0), corresponding
// to the virtual control terminal number.
240 alt_func:
241      pushl %eax            // push the index number as virtual console number.
242      call _change_console  // chr_dev/tty_io.c, line 87.
243      popl %eax            // discard the parameter.
244 end_func:
245      ret
246
247 /*
248  * function keys send F1:'esc [ [ A' F2:'esc [ [ B' etc.
249  */
250 func_table:
251      .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
252      .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
253      .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b
254
// Scan code - ASCII character mapping table.
// The scan code of the corresponding key is mapped to ASCII characters according to the keyboard
// type defined previously (FINNISH, US, GERMEN, FRANCH).
255 #if defined(KBD_FINNISH)
256 key_map:
257      .byte 0,27           // for scan code 0x00, 0x01;
258      .ascii "1234567890+' " // for scan code 0x02,...0x0c,0x0d. Bellow is similar.
259      .byte 127,9
260      .ascii "qwertyuiop}"
261      .byte 0,13,0
262      .ascii "asdfghjkl|{"
263      .byte 0,0
264      .ascii "'zxcvbnm,.-"
265      .byte 0,'*,0,32      /* 36-39 */
266      .fill 16,1,0        /* 3A-49 */
267      .byte '-,0,0,0,0,0+ /* 4A-4E */
268      .byte 0,0,0,0,0,0,0 /* 4F-55 */
269      .byte '<'
270      .fill 10,1,0
271
272 shift_map:              // mapping table when shift is pressed at the same time.
273      .byte 0,27
274      .ascii "!\"#$%&/()=?`"
275      .byte 127,9
276      .ascii "QWERTYUIOP]^"

```

```

277     .byte 13,0
278     .ascii "ASDFGHJKL\[\"
279     .byte 0,0
280     .ascii "*ZXCVBNM;:_\"
281     .byte 0,'*',0,32      /* 36-39 */
282     .fill 16,1,0          /* 3A-49 */
283     .byte '-',0,0,0,'+'   /* 4A-4E */
284     .byte 0,0,0,0,0,0,0   /* 4F-55 */
285     .byte '>'
286     .fill 10,1,0
287
288 alt_map:                  // mapping table when alt key is pressed at the same time.
289     .byte 0,0
290     .ascii "\0@\0$\0\0{[]}\0\"
291     .byte 0,0
292     .byte 0,0,0,0,0,0,0,0,0,0
293     .byte '~',13,0
294     .byte 0,0,0,0,0,0,0,0,0,0
295     .byte 0,0
296     .byte 0,0,0,0,0,0,0,0,0,0
297     .byte 0,0,0,0        /* 36-39 */
298     .fill 16,1,0        /* 3A-49 */
299     .byte 0,0,0,0,0      /* 4A-4E */
300     .byte 0,0,0,0,0,0,0  /* 4F-55 */
301     .byte '|'
302     .fill 10,1,0
303
304 #elif defined(KBD_US)    // mapping table for us keyboard.
305
306 key_map:
307     .byte 0,27
308     .ascii "1234567890-=\""
309     .byte 127,9
310     .ascii "qwertyuiop[]"
311     .byte 13,0
312     .ascii "asdfghjkl;'"
313     .byte '`',0
314     .ascii "\\zxcvbnm,./\"
315     .byte 0,'*',0,32      /* 36-39 */
316     .fill 16,1,0          /* 3A-49 */
317     .byte '-',0,0,0,'+'   /* 4A-4E */
318     .byte 0,0,0,0,0,0,0   /* 4F-55 */
319     .byte '<'
320     .fill 10,1,0
321
322
323 shift_map:
324     .byte 0,27
325     .ascii "!@#$$%^&*()_+\"
326     .byte 127,9
327     .ascii "QWERTYUIOP{}\"
328     .byte 13,0
329     .ascii "ASDFGHJKL:\\"

```

```

330     .byte '~', 0
331     .ascii "|ZXCVBNM<>?"
332     .byte 0, '*', 0, 32          /* 36-39 */
333     .fill 16, 1, 0              /* 3A-49 */
334     .byte '-', 0, 0, 0, '+'     /* 4A-4E */
335     .byte 0, 0, 0, 0, 0, 0, 0   /* 4F-55 */
336     .byte '>'
337     .fill 10, 1, 0
338
339 alt_map:
340     .byte 0, 0
341     .ascii "~\0@\0$\0\0{[]}\0"
342     .byte 0, 0
343     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
344     .byte '~', 13, 0
345     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
346     .byte 0, 0
347     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
348     .byte 0, 0, 0, 0          /* 36-39 */
349     .fill 16, 1, 0          /* 3A-49 */
350     .byte 0, 0, 0, 0, 0     /* 4A-4E */
351     .byte 0, 0, 0, 0, 0, 0, 0 /* 4F-55 */
352     .byte '|'
353     .fill 10, 1, 0
354
355 #elif defined(KBD_GR)          // mapping table for german keyboard.
356
357 key_map:
358     .byte 0, 27
359     .ascii "1234567890\\' "
360     .byte 127, 9
361     .ascii "qwertzuiop@+"
362     .byte 13, 0
363     .ascii "asdfghjkl[]^"
364     .byte 0, '#'
365     .ascii "yxcvbnm,.-"
366     .byte 0, '*', 0, 32      /* 36-39 */
367     .fill 16, 1, 0          /* 3A-49 */
368     .byte '-', 0, 0, 0, '+' /* 4A-4E */
369     .byte 0, 0, 0, 0, 0, 0, 0 /* 4F-55 */
370     .byte '<'
371     .fill 10, 1, 0
372
373
374 shift_map:
375     .byte 0, 27
376     .ascii "!\"#$%&/()=?`"
377     .byte 127, 9
378     .ascii "QWERTZUIOP\\*"
379     .byte 13, 0
380     .ascii "ASDFGHJKL{}~"
381     .byte 0, ''
382     .ascii "YXCVBNM;:_"
```

```

383     .byte 0, '*', 0, 32          /* 36-39 */
384     .fill 16, 1, 0              /* 3A-49 */
385     .byte '-', 0, 0, 0, '+'     /* 4A-4E */
386     .byte 0, 0, 0, 0, 0, 0, 0  /* 4F-55 */
387     .byte '>'
388     .fill 10, 1, 0
389
390 alt_map:
391     .byte 0, 0
392     .ascii "\0@\0$\0\0{[]}\0\0"
393     .byte 0, 0
394     .byte '@', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
395     .byte '~', 13, 0
396     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
397     .byte 0, 0
398     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
399     .byte 0, 0, 0, 0          /* 36-39 */
400     .fill 16, 1, 0          /* 3A-49 */
401     .byte 0, 0, 0, 0, 0     /* 4A-4E */
402     .byte 0, 0, 0, 0, 0, 0 /* 4F-55 */
403     .byte '|'
404     .fill 10, 1, 0
405
406
407 #elif defined(KBD_FR)          // mapping table for France keyboard.
408
409 key_map:
410     .byte 0, 27
411     .ascii "&{\\" ( - ) _ / @ ) = "
412     .byte 127, 9
413     .ascii "azertyuiop^$"
414     .byte 13, 0
415     .ascii "qsd fghjklm | "
416     .byte '`', 0, 42          /* coin sup gauche, don't know, [*|mu] */
417     .ascii "wxcvbn, ; : ! "
418     .byte 0, '*', 0, 32      /* 36-39 */
419     .fill 16, 1, 0          /* 3A-49 */
420     .byte '-', 0, 0, 0, '+' /* 4A-4E */
421     .byte 0, 0, 0, 0, 0, 0, 0 /* 4F-55 */
422     .byte '<'
423     .fill 10, 1, 0
424
425 shift_map:
426     .byte 0, 27
427     .ascii "1234567890] + "
428     .byte 127, 9
429     .ascii "AZERTYUIOP<>"
430     .byte 13, 0
431     .ascii "QSDFGHJKLM%"
432     .byte '~', 0, '#'
433     .ascii "WXCVCBN?. / \ \ "
434     .byte 0, '*', 0, 32      /* 36-39 */
435     .fill 16, 1, 0          /* 3A-49 */

```



```

436     .byte '-', 0, 0, 0, '+'      /* 4A-4E */
437     .byte 0, 0, 0, 0, 0, 0, 0    /* 4F-55 */
438     .byte '>'
439     .fill 10, 1, 0
440
441 alt_map:
442     .byte 0, 0
443     .ascii "\0~#{[|`\\`@]}"
444     .byte 0, 0
445     .byte '@, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
446     .byte '~', 13, 0
447     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
448     .byte 0, 0
449     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
450     .byte 0, 0, 0, 0             /* 36-39 */
451     .fill 16, 1, 0              /* 3A-49 */
452     .byte 0, 0, 0, 0, 0         /* 4A-4E */
453     .byte 0, 0, 0, 0, 0, 0, 0   /* 4F-55 */
454     .byte '|'
455     .fill 10, 1, 0
456
457 #else
458 #error "KBD-type not defined"
459 #endif
460 /*
461  * do_self handles "normal" keys, ie keys that don't change meaning
462  * and which have just one character returns.
463  */
464 // The code first selects a corresponding character mapping table (alt_map, shift_map or key_map)
465 // according to the mode flag, and then searches the mapping table according to the scan code
466 // of the button to obtain the corresponding character (ASCII code). Then, according to whether
467 // the current character is pressed with the ctrl or alt button at the same time and the character
468 // ASCII code value, a certain conversion is performed. Finally, the converted result characters
469 // are stored in the read buffer queue.
470 // The following code first selects one of the alt_map, shift_map, or key_map mapping tables
471 // based on the mode flag.
472 do_self:
473     lea alt_map,%ebx             // alt_map table address -> ebx (use alt_map table).
474     testb $0x20,mode            /* alt-gr */ // right alt key pressed ?
475     jne 1f                      // jump to label 1 if yes.
476     lea shift_map,%ebx          // otherwise, use shift_map table.
477     testb $0x03,mode            // any shift key pressed ?
478     jne 1f                      // jump to label 1 if yes.
479     lea key_map,%ebx            // otherwise, use key_map table.
480
481 // Now you have chosen the mapping table to use. Next, the corresponding character in the mapping
482 // table is obtained according to the scan code value. If there is no corresponding character
483 // in the mapping table, it returns (turns to none, line 495).
484 1:     movb (%ebx,%eax),%al       // get the corresponding char from mapping table.
485     orb %al,%al                 // is it a non-zero ASCII code char ?
486     je none                     // jump to none and return if the code is zero.
487
488 // If the ctrl key is pressed or the caps key is locked and the character is in the range 'a'--'z'

```

```

// [0x61--0x7D], it is converted to the corresponding character in range of [0x41--0x5D]
// by decrementing 0x20. Otherwise jump to label 2 to continue execution.
475     testb $0x4c,mode          /* ctrl or caps */ // ctrl pressed or caps lock ?
476     je 2f                    // jump to label 2 if not.
477     cmpb $'a',%al            // check to see if the char is in range of 'a' -- '}'
478     jb 2f                    // jump to label 2 if not.
479     cmpb $'}',%al
480     ja 2f
481     subb $32,%al              // The char ASCII code value is decremented by 0x20.

// If the ctrl key has been pressed and the character is within the range '@' -- '_' [0x40--0x5F],
// it is subtracted from 0x40 and converted to a value range [0x00--0x1F]. The characters in
// this range are control characters. This means that characters in the range ctrl + ['@' -- '_']
// can produce corresponding control characters in the range [0x00-0x1F]. For example, pressing
// ctrl + 'M' will generate the corresponding carriage return control character.
482 2:     testb $0x0c,mode       /* ctrl */ // ctrl key pressed ?
483     je 3f                    // jump forward to label 3 if not.
484     cmpb $64,%al             // check to see if char is in range [0x40--0x5F].
485     jb 3f                    // jump to label 3 if not.
486     cmpb $64+32,%al
487     jae 3f
488     subb $64,%al             // convert to control char [0x00--0x1f].

// If the left alt key is pressed simultaneously, bit 7 of the character is set. That is,
// characters in the extended character set whose value is greater than 0x7f can be generated
// at this time.
489 3:     testb $0x10,mode       /* left alt */ // left alt pressed ?
490     je 4f                    // jump to label 4 if not
491     orb $0x80,%al            // set bit 7 of the char.

// Finally, we put the character in al into the read buffer queue.
492 4:     andl $0xff,%eax        // only one character
493     xorl %ebx,%ebx
494     call put_queue
495 none:   ret
496
497 /*
498 * minus has a routine of it's own, as a 'E0h' before
499 * the scan code for minus means that the numeric keypad
500 * slash was pushed.
501 */
// Note that for Finnish and German keyboards, scan code 0x35 corresponds to the '-' key.
// See lines 264 and 365.
502 minus:  cmpb $1,e0            // e0 flag is set to 0x01 ? (e0 received ?)
503     jne do_self               // jump to normal processing of char if no.
504     movl $'/',%eax            // otherwise replace '-' with '/'
505     xorl %ebx,%ebx
506     jmp put_queue
507
508 /*
509 * This table decides which routine to call when a scan-code has been
510 * gotten. Most routines just call do_self, or none, depending if
511 * they are make or break.

```

```

512 */      // Note that 'make' means a key is pressed, and 'break' means released.
513 key_table:
514     .long none, do_self, do_self, do_self      /* 00-03 s0 esc 1 2 */
515     .long do_self, do_self, do_self, do_self    /* 04-07 3 4 5 6 */
516     .long do_self, do_self, do_self, do_self    /* 08-0B 7 8 9 0 */
517     .long do_self, do_self, do_self, do_self    /* 0C-0F + ' bs tab */
518     .long do_self, do_self, do_self, do_self    /* 10-13 q w e r */
519     .long do_self, do_self, do_self, do_self    /* 14-17 t y u i */
520     .long do_self, do_self, do_self, do_self    /* 18-1B o p } ^ */
521     .long do_self, ctrl, do_self, do_self        /* 1C-1F enter ctrl a s */
522     .long do_self, do_self, do_self, do_self    /* 20-23 d f g h */
523     .long do_self, do_self, do_self, do_self    /* 24-27 j k l | */
524     .long do_self, do_self, lshift, do_self      /* 28-2B { para lshift , */
525     .long do_self, do_self, do_self, do_self    /* 2C-2F z x c v */
526     .long do_self, do_self, do_self, do_self    /* 30-33 b n m , */
527     .long do_self, minus, rshift, do_self        /* 34-37 . - rshift * */
528     .long alt, do_self, caps, func              /* 38-3B alt sp caps fl */
529     .long func, func, func, func                /* 3C-3F f2 f3 f4 f5 */
530     .long func, func, func, func                /* 40-43 f6 f7 f8 f9 */
531     .long func, num, scroll, cursor              /* 44-47 f10 num scr home */
532     .long cursor, cursor, do_self, cursor        /* 48-4B up pgup - left */
533     .long cursor, cursor, do_self, cursor        /* 4C-4F n5 right + end */
534     .long cursor, cursor, cursor, cursor         /* 50-53 dn pgdn ins del */
535     .long none, none, do_self, func              /* 54-57 sysreq ? < f11 */
536     .long func, none, none, none                /* 58-5B f12 ? ? ? */
537     .long none, none, none, none                /* 5C-5F ? ? ? ? */
538     .long none, none, none, none                /* 60-63 ? ? ? ? */
539     .long none, none, none, none                /* 64-67 ? ? ? ? */
540     .long none, none, none, none                /* 68-6B ? ? ? ? */
541     .long none, none, none, none                /* 6C-6F ? ? ? ? */
542     .long none, none, none, none                /* 70-73 ? ? ? ? */
543     .long none, none, none, none                /* 74-77 ? ? ? ? */
544     .long none, none, none, none                /* 78-7B ? ? ? ? */
545     .long none, none, none, none                /* 7C-7F ? ? ? ? */
546     .long none, none, none, none                /* 80-83 ? br br br */
547     .long none, none, none, none                /* 84-87 br br br br */
548     .long none, none, none, none                /* 88-8B br br br br */
549     .long none, none, none, none                /* 8C-8F br br br br */
550     .long none, none, none, none                /* 90-93 br br br br */
551     .long none, none, none, none                /* 94-97 br br br br */
552     .long none, none, none, none                /* 98-9B br br br br */
553     .long none, unctrl, none, none              /* 9C-9F br unctrl br br */
554     .long none, none, none, none                /* A0-A3 br br br br */
555     .long none, none, none, none                /* A4-A7 br br br br */
556     .long none, none, unlshift, none            /* A8-AB br br unlshift br */
557     .long none, none, none, none                /* AC-AF br br br br */
558     .long none, none, none, none                /* B0-B3 br br br br */
559     .long none, none, unrshift, none            /* B4-B7 br br unrshift br */
560     .long unalt, none, uncaps, none             /* B8-BB unalt br uncaps br */
561     .long none, none, none, none                /* BC-BF br br br br */
562     .long none, none, none, none                /* C0-C3 br br br br */
563     .long none, none, none, none                /* C4-C7 br br br br */
564     .long none, none, none, none                /* C8-CB br br br br */

```

```
565      .long none, none, none, none          /* CC-CF br br br br */
566      .long none, none, none, none          /* D0-D3 br br br br */
567      .long none, none, none, none          /* D4-D7 br br br br */
568      .long none, none, none, none          /* D8-DB br ? ? ? */
569      .long none, none, none, none          /* DC-DF ? ? ? ? */
570      .long none, none, none, none          /* E0-E3 e0 e1 ? ? */
571      .long none, none, none, none          /* E4-E7 ? ? ? ? */
572      .long none, none, none, none          /* E8-EB ? ? ? ? */
573      .long none, none, none, none          /* EC-EF ? ? ? ? */
574      .long none, none, none, none          /* F0-F3 ? ? ? ? */
575      .long none, none, none, none          /* F4-F7 ? ? ? ? */
576      .long none, none, none, none          /* F8-FB ? ? ? ? */
577      .long none, none, none, none          /* FC-FF ? ? ? ? */
578
579 /*
580  * kb_wait waits for the keyboard controller buffer to empty.
581  * there is no timeout - if the buffer doesn't empty, we hang.
582  */
583 kb_wait:
584     pushl %eax
585 1:    inb $0x64,%al                        // read status of kbd controller.
586     testb $0x02,%al                      // test if input buffer is empty (0)
587     jne 1b                               // jump to label 1 if not empty.
588     popl %eax
589     ret
590 /*
591  * This routine reboots the machine by asking the keyboard
592  * controller to pulse the reset-line low.
593  */
594 // This subroutine writes the value 0x1234 to physical memory address 0x472. This location is
595 // the reboot mode flag. During the boot process, the ROM BIOS reads the reboot mode flag and
596 // directs the next execution based on its value. If the value is 0x1234, the BIOS will skip
597 // the memory detection process and perform the warm-boot process. If the value is 0, a cold-boot
598 // process is performed.
599 reboot:
600     call kb_wait                          // wait controller buffer to empty.
601     movw $0x1234,0x472                    /* don't do memory check */
602     movb $0xfc,%al                        /* pulse reset and A20 low */
603     outb %al,$0x64
604 die:  jmp die
```

---

## 10.2.3 Information

### 10.2.3.1 PC/AT keyboard interface programming

The keyboard interface on the motherboard of a PC is a dedicated interface that can be seen as a simplified version of the regular synchronous serial port. The interface circuit is called a keyboard controller, which receives the scan code data sent by the keyboard using a serial communication protocol. The keyboard controller used on the motherboard is the Intel 8042 chip or its compatible one. The schematic diagram is shown in Figure 10-7. The independent 8042 chip is not included on the current motherboard, but other integrated circuits on the motherboard will simulate the function of the 8042 chip for compatibility purposes. Therefore, the programming method of the keyboard controller is still applicable to the current motherboard. In addition, the chip output port P2 are used for other purposes. Bit 0 (P20 pin) is used to implement the reset operation of

the CPU, and bit 1 (P21 pin) is used to control whether the A20 signal line is turned on or not. When the output port bit 1 is 1, the A20 signal line is turned on (strobe), and 0 is the A20 signal line is disabled.

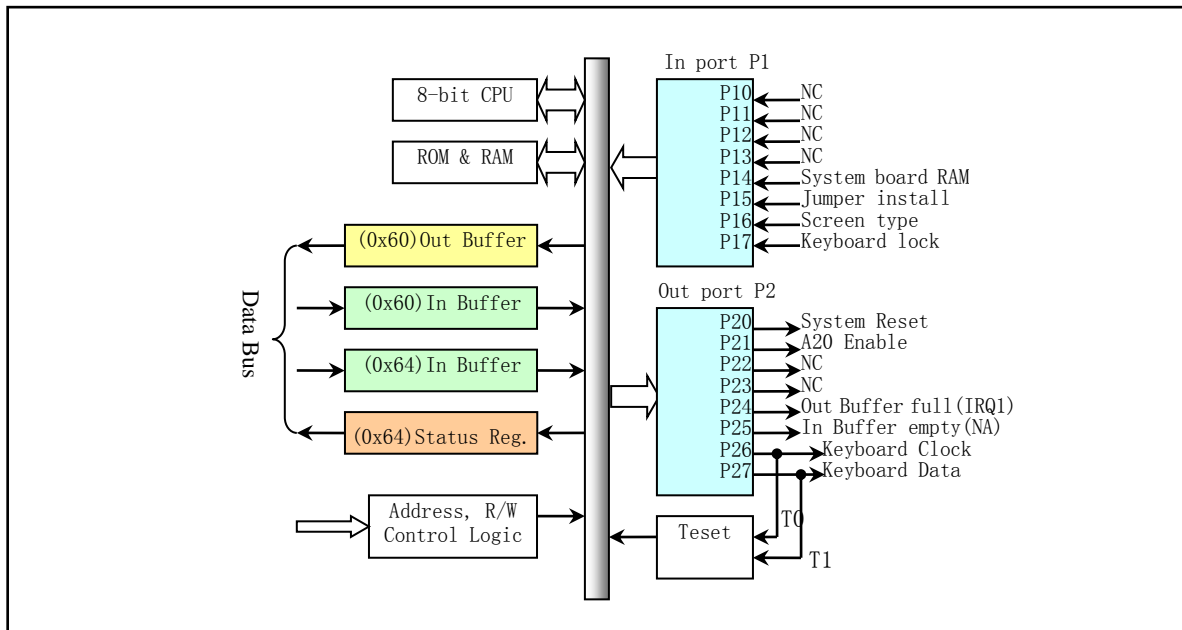


Figure 10-7 Keyboard controller 804X schematic diagram

The range of IO ports assigned to the keyboard controller is 0x60-0x6f, but in fact IBM PC/AT uses only 0x60 and 0x64 port addresses (0x61, 0x62 and 0x63 for XT compatible purposes). See Table 10-1. In addition, the meaning of the read and write operations on the port is different, so there are mainly four different operations. Programming the keyboard controller will involve the status registers, input buffers, and output buffers in the chip.

Table 10-1 Keyboard controller 804X ports

Port	R/W	Name	Purposes
0x60	Read	Data port or Output Buffer	Is an 8-bit read-only register. When the keyboard controller receives a scan code or command response from the keyboard, it sets the status register bit 0 = 1 on the one hand and generates an interrupt IRQ1 on the other hand. Normally it should only be read when the status port bit 0 = 1.
0x60	Write	Input Buffer	Used to send commands and/or subsequent parameters to the keyboard, or to write parameters to the keyboard controller. There are more than 10 keyboard commands, see the instructions after the table. Normally it should only be written when the status port bit 1 = 0.
0x61	Read/Write		This port is the address of the 8255A output port B (P2) and is hardware reset processing for the PC standard keyboard circuit using the 8255A. This port is used to respond to the received scan code. The method is to first disable the keyboard and then immediately re-enabled the keyboard. The data being manipulated is: Bit 7 = 1 Disables the keyboard; = 0 allows the keyboard; Bit 6 = 0 Forces the keyboard clock to be low, so the keyboard cannot send any

			data.; Bits 5-0 These bits are independent of the keyboard and are used for Programmable Parallel Interface (PPI).
0x64	Read	Status Register	The port is an 8-bit read-only register whose bit field meanings are: Bit 7=1 parity error of the transmitted data from the keyboard (should be 0, odd parity); Bit 6=1 Receive timeout (IRQ1 is not generated by keyboard transfer); Bit 5=1 Transmit timeout (the keyboard is not responding); Bit 4=0 Inhibit Switch. Indicates that the keyboard interface is inhibited; Bit 3=1 The data written to the input buffer is a command (via port 0x64); =0 The data written to the input buffer is a parameter (via port 0x60); Bit 2 System Flag Status: 0 = Power-on or Reset; 1 = Self-test passed; Bit 1=1 Input buffer is full (0x60/64 port has data for 8042); Bit 0 = 1 Output buffer is full (data port 0x60 has data for the system).
0x64	Write	Input Buffer	Write commands to the keyboard controller. It can take one parameter and the parameter is written from port 0x60. There are 12 keyboard controller commands, see the instructions after the table.

### 10.2.3.2 Keyboard commands

The system writes 1 byte to port 0x60, which is to send a keyboard command. The keyboard should respond within 20 ms of receiving the command, that is, return a command response. Some commands also need to follow a parameter (also written to the port). The command list is shown in Table 10-2. Note that all commands are returned with a 0xfa response code (ACK) if not indicated otherwise.

Table 10-2 Keyboard command list

Command	Has Params	Description
0xed	Yes	Set/reset mode indicator. Set to 1 to open and 0 to close. Parameter byte: Bits 7-3 are all reserved as 0; Bit 2 = caps-lock key; Bit 1 = num-lock key; Bit 0 = scroll-lock key.
0xee	No	Diagnostic response. The keyboard should return 0xee.
0xef		Reserved.
0xf0	Yes	Read/set the scan code set. The parameter byte is equal to 0x00 - select the current scan code set; 0x01 - Select scan code set 1 (for PCs, PS/2 30, etc.); 0x02 - Select scan code set 2 (for AT, PS/2, which is the default); 0x03 - Select scan code set 3.
0xf1		Reserved.
0xf2	No	Read the keyboard identification number (read 2 bytes). The AT keyboard returns the response code 0xfa.
0xf3	Yes	Set the rate and delay time when the scan code is sent continuously. The meaning of the parameter byte is:

		Bit 7 is reserved as 0; Bit 6-5 Delay value: Let C=bit 6-5, then the formula: delay value = (1 + C) * 250ms; Bit 4-0 The rate at which the scan code is continuously transmitted; let B = Bit 4-3; A = Bit 2-0, then a formula: Rate = 1 / ((8 + A) * 2 ^ B * 0.00417). The default value of the parameter is 0x2c.
0xf4	No	Enable keyboard.
0xf5	No	Disable keyboard.
0xf6	No	Set keyboard default parameters.
0xf7-0xfd		Reserved.
0xfe	No	Resend the scan code. This command is issued when the system detects that the keyboard transmits data incorrectly.
0xff	No	Performing a keyboard power-on reset operation is called a Basic Assured Test (BAT). The operation process is: 1. The keyboard responds by sending the command 0xfa immediately after receiving the command; 2. The keyboard controller sets the keyboard clock and data lines high; 3. The keyboard begins to perform BAT operations; 4. If it is completed normally, the keyboard sends 0xaa; otherwise it sends 0xfd and stops scanning.

### 10.2.3.3 Keyboard Controller Commands

The system writes 1 byte to the input buffer (port 0x64), which sends a keyboard controller command. It can take one parameter, but the parameter is sent by writing to port 0x60, as shown in Table 10-3.

Table 10-3 Keyboard controller command list

Command	Has Params	Description
0x20	No	The last command byte to keyboard controller is placed on port 0x60 for system read.
0x21-0x3f	No	Reads the command in the internal RAM of the controller specified by the lower 5 bits of the command.
0x60-0x7f	Yes	Write the keyboard controller command byte. The parameter byte is: (default is 0x5d) Bit 7 is reserved as 0; Bit 6 IBM PC compatibility mode (parity, conversion to system scan code, single-byte PC BREAK code); Bit 5 PC mode (no parity check for scan code; no conversion to system scan code); Bit 4 disables keyboard operation (making the keyboard clock low); Bit 3 disables override and does not work for keyboard lock conversion; Bit 2 system flag; 1 indicates that the controller is working correctly; Bit 1 is reserved as 0; Bit 0 allows an interrupt to be generated when the output register is full.
0xaa	No	Initialize the keyboard controller self test. Returns 0x55 on success; 0xfc on failure.
0xab	No	Initialize the keyboard interface test. The meaning of the returned byte is: 0x00 is error free;

		0x01 keyboard clock line is low (always low, low stuck); 0x02 The keyboard clock line is high; 0x03 keyboard data line is low; 0x04 The keyboard data line is high.
0xac	No	Diagnostic dump. The 804x 16-byte RAM, output port, and input port status are sequentially output to the system.
0xad	No	Disable keyboard operation (set bit 4 of command byte = 1).
0xae	No	Enable keyboard operation (reset command byte bit 4 = 0).
0xc0	No	Read the input port P1 of 804X and put it at 0x60 for reading;
0xd0	No	Read the output port P2 of 804X and put it at 0x60 for reading;
0xd1	Yes	Write 804X output port P2, the original IBM PC uses the output port bit 2 to control the A20 gate. Note that bit 0 (system reset) should always be set.
0xe0	No	The input of the test terminals T0 and T1 is sent to the output buffer for reading by the system. Bit 1 - Keyboard Data; Bit 0 - Keyboard Clock.
0xed	Yes	Control the status of keyboard LEDs. Set to 1 to open and 0 to close. Parameter byte: Bits 7-3 are all reserved as 0; Bit 2 = Caps-lock key; Bit 1 = Num-lock key; Bit 0 = Scroll-lock key.
0xf0-0xff	No	Send a pulse to the output port. This command sequence controls the output port P20-23 line, see the schematic diagram of the keyboard controller. If you want to output a negative pulse (6 microseconds), set this bit to 0. That is, the lower 4 bits of the command control the output of the negative pulse, respectively. For example, to reset the system, you need to issue the command 0xfe (P20 low).

#### 10.2.3.4 Keyboard Scan Code

The PCs are all non-encoded keyboards. Each key on the keyboard has a position number, from left to right, from top to bottom, and the position code of the PC XT and AT machine keyboard are very different. The microprocessor in the keyboard sends the scan code corresponding to the button to the system. When the key is pressed, the scan code output by the keyboard is called a 'make' scan code, and when the key is released, it is called a 'break' scan code. The scan codes of the keys of the XT keyboard are shown in Table 10-4.

Table 10-4 XT keyboard scan code table

F1	F2	1 2 3 4 5 6 7 8 9 0 - = \ BS														ESC	NUML	SCRL	SYSR
3B	3C	29 02 03 04 05 06 07 08 09 0A 0B 0C 0D 2B 0E														01	45	46	**
F3	F4	TAB Q W E R T Y U I O P [ ]														Home	↑	PgUp	PrtSc
3D	3E	0F 10 11 12 13 14 15 16 17 18 19 1A 1B														47	48	49	37
F5	F6	CNTL A S D F G H J K L ; ' ENTER														←	5	→	-
3F	40	1D 1E 1F 20 21 22 23 24 25 26 27 28 1C														4B	4C	4D	4A
F7	F8	LSHFT Z X C V B N M , . / RSHFT														End	↓	PgDn	+
41	42	2A 2C 2D 2E 2F 30 31 32 33 34 35 36														4F	50	51	4E
F9	F10	ALT Space CAPLOCK														Ins		Del	
43	44	38 39 3A														52		53	

Each button on the keyboard has a corresponding scan code contained in the lower 7 bits (bits 6-0) of the byte, and the highest bit (bit 7) indicates whether the button is pressed or released. Bit 7 = 0 indicates the scan



code just pressed the key, and bit 7 = 1 indicates the scan code after the key is released. For example, if someone just pressed the ESC key, the scan code transmitted to the system would be 1 (1 is the scan code for the ESC key), and when the key is released, a 1+0x80=129 scan code will be generated.

For the standard 83-key keyboard of PC, PC/XT, the scan code is the same as the key number (the position code of the key) and is represented by 1 byte. For example, the "A" key, the key position number is 30, the make code and the scan code are also 30 (0x1e), and the break code is the make code plus 0x80, that is, 0x9e.

The situation is somewhat different for some "extended" keys. When an extended key is pressed, an interrupt will be generated and the keyboard will output an extended scan code prefix of 0xe0, while in the next interrupt an "extended" scan code will be given. For example, for the PC/XT standard keyboard, the scan code of the left control key ctrl is 29 (0x1d), and the "extended" control key ctrl on the right has an extended scan code sequence 0xe0, 0x1d. This rule is also suitable for alt, directional arrow keys.

In addition, there are two keys that are very special, the PrtScn key and the Pause/Break key. Pressing the PrtScn button will send 2 extended characters, 42 (0x2a) and 55 (0x37), to the keyboard interrupt routine, so the actual byte sequence will be 0xe0, 0x2a, 0xe0, 0x37, and when the button is repeatedly generated, the extended character 0xaa is also sent, that is, the sequence 0xe0, 0x2a, 0xe0, 0x37, 0xe0, 0xaa is generated. When the key is released, two extended codes plus 0x80 (0xe0, 0xb7, 0xe0, 0xaa) are resent. When the PrtScn key is pressed, if the shift or ctrl key is also pressed, only 0xe0, 0x37 is sent, and only 0xe0, 0xb7 are sent when released. If the alt key is pressed at the same time, the PrtScn key is like a normal key with scan code 0x54.

For the Pause/Break key, if you press any of the control keys ctrl while pressing the key, the line will be like the extended key 70 (0x46), and in other cases it will send the character sequence 0xe1, 0x1d, 0x45, 0xe1, 0x9d, 0xc5. Pressing the button all the way does not produce a duplicate scan code, and releasing the button does not produce any scan code. Therefore, we can look at and handle this way: scan code 0xe0 means that there is one more character to follow, and scan code 0xe1 means 2 characters followed.

The scan code of the AT keyboard is slightly different from that of PC/XT. When the key is pressed, the scan code of the corresponding key is sent, but when the key is released, two bytes will be sent, the first one is 0xf0, and the second one is the same key scan code. Keyboard designers now use the 8049 as the input processor for the AT keyboard, and for the downward compatibility, the scan code from the AT keyboard is converted to the scan code of the old PC/XT standard keyboard before being sent to the system.

## 10.3 console.c

### 10.3.1 Function description

console.c is one of the longest programs in the kernel, but its functionality is relatively simple. All of the subroutines are used to implement the terminal screen write function `con_write()` and the control operation of the terminal screen display.

The `con_write()` function is called when a write to a console device is performed. This function manages all control and escape character sequences that provide the entire screen management operation for the application. The implemented escape sequence uses the vt102 terminal specification, which means that when you connect to a non-Linux host using a telnet program, your environment variable should have `TERM=vt102`. However, the best option for local operations is to set `TERM=console` because the Linux console provides a superset of vt102 functionality.

The function `con_write()` is mainly composed of a conversion statement for interpreting the finite-length state automatic escape sequence of one character at a time. In normal mode, the display characters are written

directly to the display memory using the current attributes. The function will take a character or sequence of characters from the write buffer queue `write_q` of the terminal `tty_struct` data structure, and then display the characters on the terminal screen according to the nature of the characters (normal characters, control characters, escape sequences or control sequences), or perform some screen control operations such as cursor movement and character erasure.

The terminal screen initialization function `con_init()` sets some basic parameter values about the screen according to the system information obtained when the system starts initialization, and is used for the operation of the `con_write()` function.

For a description of the terminal device character buffer queue, see the `include/linux/tty.h` header file, which shows the data structure `tty_queue` of the character buffer queue, the data structure `tty_struct` of the terminal, and some control character values. There are also some macro definitions that operate on the buffer queue. See Figure 10-14 for a schematic of the buffer queue and its operation.

### 10.3.2 Code annotation

Program 10-2 `linux/kernel/chr_drv/console.c`

---

```

1  /*
2   *  linux/kernel/console.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *      console.c
9   *
10 * This module implements the console io functions
11 *      'void con_init(void)'
12 *      'void con_write(struct tty_queue * queue)'
13 * Hopefully this will be a rather complete VT102 implementation.
14 *
15 * Beeping thanks to John T Kohl.
16 *
17 * Virtual Consoles, Screen Blanking, Screen Dumping, Color, Graphics
18 * Chars, and VT100 enhancements by Peter MacDonald.
19 */
20
21 /*
22 * NOTE!!! We sometimes disable and enable interrupts for a short while
23 * (to put a word in video IO), but this will work even for keyboard
24 * interrupts. We know interrupts aren't enabled when getting a keyboard
25 * interrupt, as we use trap-gates. Hopefully all is well.
26 */
27
28 /*
29 * Code to check for different video-cards mostly by Galen Hunt,
30 * <g-hunt@ee.utah.edu>
31 */
32
33 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
34 //      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/config.h> Kernel configuration header file. Define keyboard language and hard
//      disk type (HD_TYPE) options.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//      commonly used functions of the kernel.
// <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/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.
// <string.h> String header file. Defines some embedded functions about string operations.
// <errno.h> Error number header file. Contains various error numbers in the system.
33 #include <linux/sched.h>
34 #include <linux/tty.h>
35 #include <linux/config.h>
36 #include <linux/kernel.h>
37
38 #include <asm/io.h>
39 #include <asm/system.h>
40 #include <asm/segment.h>
41
42 #include <string.h>
43 #include <errno.h>
44
// This symbolic constant defines the default data for the terminal IO structure. For the
// symbolic constants, please refer to the include/termios.h file.
45 #define DEF_TERMIOS \
46 (struct termios) { \
47     ICRNL, \
48     OPOST | ONLCR, \
49     0, \
50     IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, \
51     0, \
52     INIT_C_CC \
53 }
54
55
56 /*
57  * These are set up by the setup-routine at boot-time:
58  */
// Please refer to the comments on boot/setup.s and the system parameter table read and saved
// by the setup program.
59
60 #define ORIG_X                (*(unsigned char *)0x90000)        // original cursor colum no.
61 #define ORIG_Y                (*(unsigned char *)0x90001)        // original cursor row no.
62 #define ORIG_VIDEO_PAGE        (*(unsigned short *)0x90004)      // current video page.
63 #define ORIG_VIDEO_MODE        (((*(unsigned short *)0x90006) & 0xff) // display mode.
64 #define ORIG_VIDEO_COLS        (((*(unsigned short *)0x90006) & 0xff00) >> 8) // screen columns.
65 #define ORIG_VIDEO_LINES        (((*(unsigned short *)0x9000e) & 0xff) // screen rows.
66 #define ORIG_VIDEO_EGA_AX        (*(unsigned short *)0x90008)      // current function.

```

```

67 #define ORIG_VIDEO_EGA_BX (*(unsigned short *)0x9000a)    // disp mem size, color mode.
68 #define ORIG_VIDEO_EGA_CX (*(unsigned short *)0x9000c)    // video card properties.
69
// Defines the monochrome/color display mode type symbol constant.
70 #define VIDEO_TYPE_MDA 0x10    /* Monochrome Text Display */
71 #define VIDEO_TYPE_CGA 0x11    /* CGA Display */
72 #define VIDEO_TYPE_EGAM 0x20    /* EGA/VGA in Monochrome Mode */
73 #define VIDEO_TYPE_EGAC 0x21    /* EGA/VGA in Color Mode */
74
75 #define NPAR 16    // The max nr of arguments in the escape sequence.
76
77 int NR_CONSOLES = 0;    // The nr of virtual consoles that the system supports.
78
79 extern void keyboard_interrupt(void);    // Keyboard interrupt handler (in keyboard.S).
80
// The following static variables are some of the global variables used in this file.
81 static unsigned char video_type;    /* Type of display being used */
82 static unsigned long video_num_columns;    /* Number of text columns */
83 static unsigned long video_mem_base;    /* Base of video memory */
84 static unsigned long video_mem_term;    /* End of video memory */
85 static unsigned long video_size_row;    /* Bytes per row */
86 static unsigned long video_num_lines;    /* Number of test lines */
87 static unsigned char video_page;    /* Initial video page */
88 static unsigned short video_port_reg;    /* Video register select port */
89 static unsigned short video_port_val;    /* Video register value port */
90 static int can_do_colour = 0;    // flag, can use color function.
91
// The virtual console structure is defined below. It contains all the current information for
// a virtual console. vc_origin and vc_scr_end are the display memory locations corresponding
// to the start line and the last line used by the virtual console currently being processed
// to perform a fast scroll operation. vc_video_mem_start and vc_video_mem_end are the display
// memory areas used by the current virtual console.
92 static struct {
93     unsigned short vc_video_erase_char;    // erase char attributes & char (0x0720)
94     unsigned char vc_attr;    // char attribute.
95     unsigned char vc_def_attr;    // default attribute.
96     int vc_bold_attr;    // bold char attribute.
97     unsigned long vc_ques;    // question mark char.
98     unsigned long vc_state;    // state of the escape or control sequence.
99     unsigned long vc_restate;    // next state of escape or control sequence.
100     unsigned long vc_checkin;
101     unsigned long vc_origin;    /* Used for EGA/VGA fast scroll */
102     unsigned long vc_scr_end;    /* Used for EGA/VGA fast scroll */
103     unsigned long vc_pos;    // The mem location of the current cursor.
104     unsigned long vc_x,vc_y;    // current cursor column, row value.
105     unsigned long vc_top,vc_bottom;    // The top & bottom line nr when scrolling.
106     unsigned long vc_npar,vc_par[NPAR];    // escape sequence param nr and array.
107     unsigned long vc_video_mem_start;    /* Start of video RAM */
108     unsigned long vc_video_mem_end;    /* End of video RAM (sort of) */
109     unsigned int vc_saved_x;    // The saved cursor column number.
110     unsigned int vc_saved_y;    // The saved cursor row number.
111     unsigned int vc_iscolor;    // The color display flag.
112     char * vc_translate;    // The character set used.

```

```

113 } vc_cons [MAX_CONSOLES];
114
    // For ease of reference, the following defines the symbols of the console currently being
    // processed, with the same meaning as above. Where currcons is the current virtual terminal
    // number in the function argument using the vc_cons[] structure.
115 #define origin          (vc_cons[currcons].vc_origin)
116 #define scr_end        (vc_cons[currcons].vc_scr_end)
117 #define pos            (vc_cons[currcons].vc_pos)
118 #define top            (vc_cons[currcons].vc_top)
119 #define bottom         (vc_cons[currcons].vc_bottom)
120 #define x              (vc_cons[currcons].vc_x)
121 #define y              (vc_cons[currcons].vc_y)
122 #define state          (vc_cons[currcons].vc_state)
123 #define restate        (vc_cons[currcons].vc_restate)
124 #define checkin       (vc_cons[currcons].vc_checkin)
125 #define npar           (vc_cons[currcons].vc_npar)
126 #define par           (vc_cons[currcons].vc_par)
127 #define ques          (vc_cons[currcons].vc_ques)
128 #define attr          (vc_cons[currcons].vc_attr)
129 #define saved_x       (vc_cons[currcons].vc_saved_x)
130 #define saved_y       (vc_cons[currcons].vc_saved_y)
131 #define translate     (vc_cons[currcons].vc_translate)
132 #define video_mem_start (vc_cons[currcons].vc_video_mem_start)
133 #define video_mem_end  (vc_cons[currcons].vc_video_mem_end)
134 #define def_attr      (vc_cons[currcons].vc_def_attr)
135 #define video_erase_char (vc_cons[currcons].vc_video_erase_char)
136 #define iscolor      (vc_cons[currcons].vc_iscolor)
137
138 int blankinterval = 0;          // Black screen interval.
139 int blankcount = 0;            // Black screen time count.
140
141 static void sysbeep(void);      // System beep function.
142
143 /*
144  * this is what the terminal answers to a ESC-Z or csi0c
145  * query (= vt100 response).
146  */                                // csi - Control Sequence Introducer.
    // The host requests the terminal to answer a device attribute control sequence by sending a
    // device attribute (DA) control sequence with no parameters or parameter 0 ('ESC [c' or
    // 'ESC [0c') (ESC-Z acts the same). The terminal sends the following sequence to respond to
    // the host. This sequence (ie 'ESC [?1;2c') indicates that the terminal is a VT100 compatible
    // terminal with advanced video capabilities.
147 #define RESPONSE "\033[?1;2c"
148
    // Define the character set to use. The upper part is the normal 7-bit ASCII code, which is
    // the US character set. The lower part corresponds to the line character in the VT100 terminal
    // device, that is, the character set showing the chart line.
149 static char * translations[] = {
150 /* normal 7-bit ascii */
151     " !\"#$%&'()*+,-./0123456789:;<=>?"
152     "@ABCDEFGHIJKLMNPOQRSTUVWXYZ[\ ]^_`"
153     "`abcdefghijklmnopqrstuvwxyz{|}~",
154 /* vt100 graphics */

```

```

155     " !\"#$%&'()*+,-./0123456789:;<=>?"
156     "@ABCDEFGHIJKLMNPQRSTUVWXYZ[\ ]^`"
157     "\004\261\007\007\007\007\370\361\007\007\275\267\326\323\327\304"
158     "\304\304\304\304\307\266\320\322\272\363\362\343|\007\234\007 "
159 };
160
161 #define NORM_TRANS (translations[0])
162 #define GRAF_TRANS (translations[1])
163
164 // Tracks the current position of the cursor.
165 // Parameters: currcons - the current virtual terminal; new_x - the cursor column number;
166 // new_y - the cursor line number.
167 // This function is used to update the current cursor position variable x, y and correct the
168 // corresponding position pos of the cursor in the display memory. This function first checks
169 // the validity of the parameters. Exit if the given cursor column number exceeds the maximum
170 // number of columns in the display, or if the cursor line number is not lower than the maximum
171 // number of rows displayed. Otherwise, the current cursor variable and the new cursor position
172 // are updated to correspond to the position pos in the display memory. Note that all variables
173 // in the function are actually the corresponding fields in the vc_cons[currcons] structure,
174 // and the following functions are the same.
175 /* NOTE! gotoxy thinks x==video_num_columns is ok */
176 static inline void gotoxy(int currcons, int new_x, unsigned int new_y)
177 {
178     if (new_x > video_num_columns || new_y >= video_num_lines)
179         return;
180     x = new_x;
181     y = new_y;
182     pos = origin + y*video_size_row + (x<<1); // One col needs 2 bytes, so x<<1.
183 }
184
185 // Set the scroll start display memory address.
186 // First check whether the display card type is a color card, and judge whether the console
187 // specified by the parameter is the front console, and exit if the two conditions are not met.
188 // Otherwise, the scroll start address is output to the register on the display card.
189 static inline void set_origin(int currcons)
190 {
191     // First determine the type of display card. For EGA/VGA cards, we can specify the on-screen
192     // range (area) for scrolling, while the MDA monochrome display card can only perform full-screen
193     // scrolling. Therefore, only the EGA/VGA card needs to set the display memory address of the
194     // start line of the scrolling (the starting line is the line corresponding to origin), otherwise
195     // it will exit directly. In addition, we only operate on the foreground console, so only when
196     // the current console currcons is the foreground console, we need to set the memory start
197     // position corresponding to the scroll start line.
198     if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
199         return;
200     if (currcons != fg_console)
201         return;
202     // Then we output 12 to the "display register select port" video_port_reg, which selects the
203     // display control data register r12 and then writes the scroll start address high byte to it.
204     // Shifting 9 bits to the right actually means shifting 8 bits to the right and dividing by
205     // 2 (1 character on the screen is represented by 2 bytes). Then select the display control
206     // data register r13 and write the low byte of the scroll start address. Shifting 1 bit to the
207     // right means dividing by 2, which also means that 1 character on the screen is represented

```

```

// by 2 bytes of memory. The output value operates relative to the default display memory start
// position video_mem_base, for example, for EGA/VGA color mode, video_mem_base = physical
// memory address 0xb8000.
180     cli();
181     outb_p(12, video_port_reg); // Select data register r12 to output high byte.
182     outb_p(0xff & ((origin - video_mem_base) >> 9), video_port_val);
183     outb_p(13, video_port_reg); // Select r13 to output the low byte.
184     outb_p(0xff & ((origin - video_mem_base) >> 1), video_port_val);
185     sti();
186 }
187
//// The content scrolls up one line.
// Move the screen scroll window down one line and add a space character to the new line that
// appears at the bottom of the screen scroll area. The scrolling area must be greater than
// one line. See section 10.3.3.2 for the principle of scrolling operation.
188 static void scrup(int curcons)
189 {
// The scrolling area must have at least 2 lines. If the top line number of the scrolling area
// is greater than or equal to the bottom line number, the condition for the rolling operation
// is not satisfied. In addition, for the EGA/VGA card, we can specify the on-screen range (area)
// for scrolling, while the MDA monochrome display card can only perform full-screen scrolling.
// This function handles the EGA and MDA display types separately. If the display type is EGA,
// it is also divided into full-screen window movement and intra-area window movement. Here
// the code first deals with the case where the display card is of the EGA/VGA type.
190     if (bottom <= top)
191         return;
192     if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
193     {
// If the move start line top=0, the bottom line = video_num_lines = 25, it means that the entire
// screen window moves down one line. Therefore, the starting memory position origin
// corresponding to the upper left corner of the entire screen window is adjusted to the memory
// position shifted downward by one line, and the memory position corresponding to the current
// cursor and the position of the character pointer scr_end at the end of the screen end are
// also tracked. Finally, the new screen window memory start position origin is written into
// the display controller.
194         if (!top && bottom == video_num_lines) {
195             origin += video_size_row;
196             pos += video_size_row;
197             scr_end += video_size_row;
// If the display memory pointer scr_end corresponding to the end of the screen window exceeds
// the end of the actual display memory, the memory data corresponding to all the lines except
// the first line of the screen content is moved to the start position video_mem_start of the
// display memory, and fill in the space character on a new line that appears after the entire
// screen window has been moved down. Then, according to the situation that the screen memory
// data is moved, the start pointer corresponding to the current screen, the cursor position
// pointer, and the corresponding memory pointer scr_end at the end of the screen are readjusted.
//
// This embedded assembly code first moves the memory data corresponding to the (screen character
// line number - 1) line to the display memory start position video_mem_start, and then adds
// a line of space (erase) character data at the subsequent memory location.
// %0 - eax (erase character + attribute); %1 - ecx ((the number of lines of the screen - 1)
// corresponds to the number of characters / 2, moving in long words); %2 - edi (displays memory
// start position video_mem_start); %3 - esi (screen window memory start position origin).

```

```

// Direction of movement: [edi] -> [esi], move ecx long words.
198         if (scr\_end > video\_mem\_end) {
199             __asm__("cld\n\t"          // clear direction.
200                   "rep\n\t"          // repeat move data.
201                   "movsl\n\t"
202                   "movl \_video\_num\_columns,%1\n\t"
203                   "rep\n\t"          // fill a line of spaces.
204                   "stosw"
205                   ::"a" \(video\\_erase\\_char\),
206                   "c" \(\(video\\_num\\_lines-1\)\*video\\_num\\_columns>>1\),
207                   "D" \(video\\_mem\\_start\),
208                   "S" \(origin\)
209                   : "cx", "di", "si"\);
210             scr\_end -= origin-video\_mem\_start;
211             pos -= origin-video\_mem\_start;
212             origin = video\_mem\_start;
// If the memory pointer scr_end at the end of the adjusted screen does not exceed the end of
// the display memory video_mem_end, simply fill in the erase line (space character) on the
// new line.
// %0 - eax (erase character + attribute); %1 - ecx (number of screen lines); %2 - edi
// (corresponding memory location at the beginning of the last line).
213         } else {
214             __asm__("cld\n\t"
215                   "rep\n\t"          // fill a line of spaces.
216                   "stosw"
217                   ::"a" \(video\\_erase\\_char\),
218                   "c" \(video\\_num\\_columns\),
219                   "D" \(scr\\_end-video\\_size\\_row\)
220                   : "cx", "di"\);
221         }
// Then write the new screen window memory start position origin into the display controller.
222         set\_origin(currcons);

// Otherwise it means that it is not a full screen move. That is to say, all the lines from
// the specified line top to bottom area are moved up by one line, and the specified line top
// is deleted. At this time, the display memory data corresponding to all the lines of the screen
// from the specified line top to the bottom is directly moved up by one line, and the erasing
// character is filled in the newly appearing line.
// %0 - eax (erase character + attribute); %1 - ecx (the number of long words from the top+1
// line to the bottom line); %2 - edi (the memory location where the top row is located); %3
// - esi (the memory location where the top+1 row is located).
223         } else {
224             __asm__("cld\n\t"
225                   "rep\n\t"          // repeat from top+1 to bottom.
226                   "movsl\n\t"
227                   "movl \_video\_num\_columns,%ecx\n\t"
228                   "rep\n\t"          // fill blank char in new line.
229                   "stosw"
230                   ::"a" \(video\\_erase\\_char\),
231                   "c" \(\(bottom-top-1\)\*video\\_num\\_columns>>1\),
232                   "D" \(origin+video\\_size\\_row\\*top\),
233                   "S" \(origin+video\\_size\\_row\\*\\(top+1\\)\)
234                   : "cx", "di", "si"\);

```



```

235     }
236 }
// If the display type is not EGA (but MDA), perform the following move operation. Because the
// MDA display card can only scroll through the entire screen, and it will automatically adjust
// beyond the display memory range (that is, the pointer will be automatically scrolled), so
// the memory corresponding to the screen content is not processed separately than the display
// memory range. The processing method is exactly the same as the EGA non-full screen movement.
237     else                /* Not EGA/VGA */
238     {
239         __asm__ ("cld\n\t"
240                 "rep\n\t"
241                 "movsl\n\t"
242                 "movl _video_num_columns, %%ecx\n\t"
243                 "rep\n\t"
244                 "stosw"
245                 ":: \"a\" (video_erase_char),
246                 "c\" ((bottom-top-1)*video_num_columns>>1),
247                 "D\" (origin+video_size_row*top),
248                 "S\" (origin+video_size_row*(top+1))
249                 : \"cx\", \"di\", \"si\");
250     }
251 }
252
//// The display content scrolls down one line.
// Move the screen scroll window up one line, and the corresponding screen scroll area content
// moves down one line. A new line will appear above the start line of the move. The processing
// method is similar to scrup() except that the data overlay does not occur when moving the
// memory data, and the copy operation is reversed. That is, copy from the last character on
// the second line of the countdown to the last line, then copy the characters on the third
// line of the countdown to the second line of the last, and so on.
253 static void scrdown(int currcons)
254 {
// Similarly, the scrolling area must have at least 2 lines. If the top line number of the
// scrolling area is greater than or equal to the bottom line number of the area, the condition
// for the rolling operation is not satisfied. In addition, for the EGA/VGA card, we can specify
// the on-screen range (area) for scrolling, while the MDA monochrome display card can only
// perform full-screen scrolling. Since the window moves up to the start position of the display
// area memory at most, the display memory pointer scr_end corresponding to the end of the screen
// window does not exceed the actual display memory end, so only the normal memory data movement
// needs to be handled here.
255     if (bottom <= top)
256         return;
257     if (video\_type == VIDEO\_TYPE\_EGAC || video\_type == VIDEO\_TYPE\_EGAM)
258     {
// %0 - eax (erase character + attribute); %1 - ecx (the number of long words corresponding
// to the number of lines from top to bottom-1); %2 - edi (the last long word position in the
// lower right corner of the window); %3 - esi (the last long word position in the second line
// of the window countdown).
// Direction of movement: [esi] -> [edi], move ecx long words.
259         __asm__ ("std\n\t"                // set direction flag!!
260                 "rep\n\t"
261                 "movsl\n\t"
262                 "addl $2, %%edi\n\t" /* %edi has been decremented by 4 */

```

```

263         "movl _video_num_columns, %%ecx\n\t"
264         "rep\n\t"           // fill in blanks at above line.
265         "stosw"
266         ":: \"a\" (video_erase_char),
267         \"c\" ((bottom-top-1)*video_num_columns>>1),
268         \"D\" (origin+video_size_row*bottom-4),
269         \"S\" (origin+video_size_row*(bottom-1)-4)
270         : \"ax\", \"cx\", \"di\", \"si\");
271     }
    // If it is not the EGA display type, do the following (same as above).
272     else        /* Not EGA/VGA */
273     {
274         __asm__ ( "std\n\t"
275                 "rep\n\t"
276                 "movsl\n\t"
277                 "addl $2, %%edi\n\t"    /* %edi has been decremented by 4 */
278                 "movl _video_num_columns, %%ecx\n\t"
279                 "rep\n\t"
280                 "stosw"
281                 ":: \"a\" (video_erase_char),
282                 \"c\" ((bottom-top-1)*video_num_columns>>1),
283                 \"D\" (origin+video_size_row*bottom-4),
284                 \"S\" (origin+video_size_row*(bottom-1)-4)
285                 : \"ax\", \"cx\", \"di\", \"si\");
286     }
287 }
288
    /// The cursor moves down one line at the same column position.
    // If the cursor is not on the last line, directly modify the cursor current line variable y++,
    // and adjust the cursor corresponding display memory position pos (plus the memory length
    // corresponding to one line of characters). Otherwise you need to move the contents of the
    // screen window up one line. The function name lf (line feed) refers to the processing control
    // character LF.
289 static void lf(int currcons)
290 {
291     if (y+1<bottom) {
292         y++;
293         pos += video_size_row;    // Plus the nr of bytes occupied by one line.
294         return;
295     }
296     scrup(currcons);    // Move the contents up one line.
297 }
298
    /// The cursor moves up one row in the same column.
    // If the cursor is not on the first line of the screen, directly modify the cursor current
    // line variable y--, and adjust the cursor corresponding display memory position pos, minus
    // the number of bytes of memory length corresponding to a line of characters on the screen.
    // Otherwise you need to move the contents of the screen window down one line. The function
    // name ri (reverse index) refers to the control character RI or the escape sequence "ESC M".
299 static void ri(int currcons)
300 {
301     if (y>top) {
302         y--;

```

```

303         pos -= video_size_row;           // Minus the nr of bytes occupied by one line.
304         return;
305     }
306     scrdown(currcons);                   // Move the contents down one line.
307 }
308
// The cursor returns to column 0.
// Adjust the cursor corresponding to the memory location pos. The column number of the cursor
// *2 is the length of the memory byte occupied by the 0 column on the line to the column where
// the cursor is located. The function name cr (carriage return) indicates the control character
// being processed.
309 static void cr(int currcons)
310 {
311     pos -= x<<1;                         // bytes occupied by col 0 to col of cursor.
312     x=0;
313 }
314
// Erase the previous character of the cursor and move the cursor back one column.
// If the cursor is not in the 0 column, the cursor is backed by 2 bytes corresponding to the
// memory position pos (corresponding to one character on the screen), then the current cursor
// column is decremented by 1, and the character at the position of the cursor is erased.
315 static void del(int currcons)
316 {
317     if (x) {
318         pos -= 2;
319         x--;
320         *(unsigned short *)pos = video_erase_char;
321     }
322 }
323
//// Delete part of the content on the screen that is related to the cursor position.
// This function is used to handle ANSI control sequences. The delete character operation
// associated with the cursor position is performed according to the specified control sequence,
// some or all of the displayed characters are erased, and the cursor position is unchanged
// when the character or line is erased.
// The ANSI control sequence processed by this function is: 'ESC [ Ps J'. Among them, Ps = 0
// - means to delete the cursor to the bottom of the screen; 1 - delete the screen to the cursor;
// 2 - delete the entire screen.
// The function name csi_J (CSI - Control Sequence Introducer) indicates that the control
// sequence "CSI Ps J" is processed. The argument 'vpar' corresponds to the value of 'Ps' in
// the above control sequence. For an introduction to terminal control commands, see section
// 10.3.3.3 after the program listing. Commonly used escape sequences and control sequences
// can be found in Appendix 3 of the book.
324 static void csi_J(int currcons, int vpar)
325 {
326     long count __asm__("cx");             // set to use register variable.
327     long start __asm__("di");
328
// First, set the number of characters to be deleted and the display memory position at which
// deletion is started according to the above three cases.
329     switch (vpar) {
330         case 0: /* erase from cursor to end of display */
331             count = (scr_end-pos)>>1;

```

```

332         start = pos;
333         break;
334     case 1: /* erase from start to cursor */
335         count = (pos-origin)>>1;
336         start = origin;
337         break;
338     case 2: /* erase whole display */
339         count = video_num_columns * video_num_lines;
340         start = origin;
341         break;
342     default:
343         return;
344 }
// Then use the erase character to fill in the place where the character is deleted.
// %0 -ecx (nr of chars deleted); %1 -edi (delete start address); %2 -eax (filled erase char).
345     __asm__( "cld\n\t"
346             "rep\n\t"
347             "stosw\n\t"
348             ":: \"c\" (count),
349             "D\" (start), \"a\" (video_erase_char)
350             : \"cx\", \"di\" );
351 }
352
//// Delete part of the content related to the cursor position on one line.
// This function erases some or all of the characters in the line of the cursor according to
// the parameters. The erase operation removes characters from the screen without affecting
// other characters. The erased characters are discarded. The cursor position does not change
// when erasing characters or lines.
// ANSI escape sequence: 'ESC [ Ps K' (Ps = 0 to the end of the line; 1 to delete from the beginning;
// 2 to delete the entire line). The function parameter vpar corresponds to the Ps.
353 static void csi_K(int curcons, int vpar)
354 {
355     long count __asm__( "cx" );
356     long start __asm__( "di" );
357
// First, the number of characters to be deleted and the display memory position at which the
// deletion starts are set according to the three conditions in the control sequence.
358     switch (vpar) {
359     case 0: /* erase from cursor to end of line */
360         if (x>=video_num_columns)
361             return;
362         count = video_num_columns-x;
363         start = pos;
364         break;
365     case 1: /* erase from start of line to cursor */
366         start = pos - (x<<1);
367         count = (x<video_num_columns)?x:video_num_columns;
368         break;
369     case 2: /* erase whole line */
370         start = pos - (x<<1);
371         count = video_num_columns;
372         break;
373     default:

```

```

374         return;
375     }
    // Then use the erase character to fill in the place where the character is deleted.
    // %0 - ecx (delete count); %1 -edi (delete address start); %2 -eax (fill erase character).
376     __asm__( "cld\n\t"
377             "rep\n\t"
378             "stosw\n\t"
379             ":: \"c\" (count),
380             "D\" (start), \"a\" (video_erase_char)
381             : \"cx\", \"di\");
382 }
383
    //// Set the attributes of the display character.
    // The control sequence sets the character display attribute according to the parameter. All
    // characters sent to the terminal in the future will use the attributes specified here until
    // the control sequence is used again to reset the attributes of the character display.
    // ANSI escape sequence: 'ESC [ Ps; Ps m'. Where Ps = 0 - default attribute; 1 - bold and bright;
    // 4 - underline; 5 - flash; 7 - reverse; 22 - non-bold; 24 - no underline; 25 - no flicker;
    // 27 - normal ;30--38 - Set foreground color; 39 - Default foreground color (White);
    // 40--48 - Set background color; 49 - Default background color (Black).
384 void csi_m(int currcons )
385 {
386     int i;
387
    // A control sequence can have multiple different parameters. The parameters are stored in the
    // array par[]. The following code cyclically processes each parameter Ps according to the number
    // of parameters npar received.
    // If Ps = 0, the character attribute that is displayed next to the current virtual console
    // is set to the default attribute def_attr. At initialization, def_attr has been set to 0x07
    // (white on black).
    // If Ps = 1, the character attributes that will be displayed later are set to bold or highlighted.
    // If it is a color display, the character attribute or upper 0x08 is used to highlight the
    // character; if it is a monochrome display, the character is underlined.
    // If Ps = 4, the color and monochrome displays are treated differently. If the color display
    // mode is not available at this time, the characters are underlined. If it is a color display,
    // if the original vc_bold_attr is not equal to -1, the background color is reset; otherwise,
    // the foreground color is reversed. If the foreground color is the same as the background color,
    // the foreground color is increased by one and the other color is taken.
388     for (i=0;i<=npar;i++)
389         switch (par[i]) {
390             case 0: attr=def_attr;break; /* default */
391             case 1: attr=(iscolor?attr|0x08:attr|0x0f);break; /* bold */
392             /*case 4: attr=attr/0x01;break;*/ /* underline */
393             case 4: /* bold */
394                 if (!iscolor)
395                     attr |= 0x01; /* Mono is underlined.
396                 else
397                 { /* check if foreground == background */
398                     if (vc_cons[currcons].vc_bold_attr != -1)
399                         attr = (vc_cons[currcons].vc_bold_attr&0xf0) | (0xf0&(attr));
400                     else
401                     { short newattr = (attr&0xf0) | (0xf&(~attr));
402                         attr = ((newattr&0xf)==((attr>>4)&0xf)?

```

```

403             (attr&0xf0) | (((attr&0xf)+1)%0xf) :
404             newattr);
405         }
406     }
407     break;
// If Ps = 5, the character that is subsequently displayed in the current virtual console is
// set to blink, that is, the attribute byte bit 7 is set to 1.
// If Ps = 7, the characters displayed subsequently are set to reverse, that is, the foreground
// and background colors are swapped.
// If Ps = 22, the highlighting of subsequent characters is canceled (cancel bold display).
// If Ps = 24, the underline of the subsequent characters is canceled for monochrome display
// and green is cancelled for color display.
// If Ps = 25, the flashing of the subsequent characters is canceled.
// If Ps = 27, the reverse of the subsequent characters is canceled.
// If Ps = 39, the foreground color of the subsequent characters is reset to default (white).
// If Ps = 49, the background color of the subsequent characters is reset to default (black).
408     case 5: attr=attr|0x80;break; /* blinking */
409     case 7: attr=(attr<<4) | (attr>>4);break; /* negative */
410     case 22: attr=attr&0xf7;break; /* not bold */
411     case 24: attr=attr&0xfe;break; /* not underline */
412     case 25: attr=attr&0x7f;break; /* not blinking */
413     case 27: attr=def_attr;break; /* positive image */
414     case 39: attr=(attr & 0xf0) | (def_attr & 0x0f); break;
415     case 49: attr=(attr & 0x0f) | (def_attr & 0xf0); break;
// When Ps(par[i]) is another value, the specified foreground or background color is set.
// If Ps = 30..37, the foreground color is set; if Ps=40..47, the background color is set. See
// the instructions following the program list for color values.
416     default:
417         if (!can_do_colour)
418             break;
419         iscolor = 1;
420         if ((par[i]>=30) && (par[i]<=38)) // foreground color.
421             attr = (attr & 0xf0) | (par[i]-30);
422         else /* Background color */
423             if ((par[i]>=40) && (par[i]<=48)) // background color.
424                 attr = (attr & 0x0f) | ((par[i]-40)<<4);
425             else
426                 break;
427     }
428 }
429
//// Set the display cursor.
// The display position of the cursor in the controller is set according to the memory position
// pos of the cursor.
430 static inline void set_cursor(int currcons)
431 {
// Since we need to set the display cursor, it means there is a keyboard operation, so we need
// to restore the delay count value of the black screen operation. In addition, the console
// displaying the cursor must be the foreground console.
432     blankcount = blankinterval; // Resets the count of the black screen.
433     if (currcons != fg_console)
434         return;
// Then use the index register port to select the display control data register r14 (the current

```

```

// display position of the cursor high byte), and then write the current position of the cursor
// high byte (moving 9 bits to the right means the high byte is moved to the low byte and divided
// by 2). This is relative to the default display memory operation. Then use the index register
// to select r15 and write the low byte of the cursor's current position.
435     cli();
436     outb_p(14, video_port_reg);
437     outb_p(0xff & ((pos-video_mem_base)>>9), video_port_val);
438     outb_p(15, video_port_reg);
439     outb_p(0xff & ((pos-video_mem_base)>>1), video_port_val);
440     sti();
441 }
442
// Hide the cursor.
// Set the cursor to the end of the current virtual console window to hide the cursor.
443 static inline void hide_cursor(int currcons)
444 {
// First select the display control data register r14 (the current display position of the cursor
// high byte), and then write the high byte of the cursor position (moving 9 bits to the right
// means moving the high byte to the low byte and dividing by 2). Then select r15 and write
// the low byte of the cursor's current position.
445     outb_p(14, video_port_reg);
446     outb_p(0xff & ((scr_end-video_mem_base)>>9), video_port_val);
447     outb_p(15, video_port_reg);
448     outb_p(0xff & ((scr_end-video_mem_base)>>1), video_port_val);
449 }
450
//// Send a response sequence to VT100.
// That is, in response to the host requesting the terminal, the device attribute (DA) is sent
// to the host. The host requests the terminal to send back a device attribute (DA) control
// sequence by sending a DA control sequence ('ESC[0c' or 'ESC Z') with no parameters or parameter
// 0. The terminal then sends back the response sequence defined on line 147 (ie 'ESC [?1; 2c')
// in response to the host's sequence. This sequence tells the host that the terminal is a VT100
// compatible terminal with advanced video capabilities. The process is to put the response
// sequence into the read buffer queue and use the copy_to_cooked() function to process it and
// put it into the auxiliary (secondary) queue.
451 static void respond(int currcons, struct tty_struct * tty)
452 {
453     char * p = RESPONSE; // defined on line 147 ('ESC [?1; 2c').
454
455     cli();
456     while (*p) { // put response sequence into the read queue.
457         PUTCH(*p, tty->read_q); // put one by one. include/linux/tty.h, line 46.
458         p++;
459     }
460     sti();
461     copy_to_cooked(tty); // put into the auxiliary queue. tty_io.c, 120.
462 }
463
//// Insert a space character at the cursor.
// Move all the characters at the beginning of the cursor one space to the right and insert
// the erase character at the cursor.
464 static void insert_char(int currcons)
465 {

```

```

466     int i=x;
467     unsigned short tmp, old = video_erase_char;    // erase char (with attribute).
468     unsigned short * p = (unsigned short *) pos;    // memory position of the cursor.
469
470     while (i++<video_num_columns) {
471         tmp=*p;
472         *p=old;
473         old=tmp;
474         p++;
475     }
476 }
477
478 // Insert a line at the cursor.
479 // Scrolls the screen window down from the line where the cursor is located to the bottom of
480 // the window. The cursor will be on a new, empty line.
481 static void insert_line(int currcons)
482 {
483     int oldtop,oldbottom;
484
485     // First save the screen window scrolling start line top and last line bottom value, then scroll
486     // the screen content down one line from the line where the cursor is. Finally, restore the
487     // original value of the scroll start line 'top' and the last line 'bottom'.
488     oldtop=top;
489     oldbottom=bottom;
490     top=y;    // set start and end lines of the scroll screen.
491     bottom = video_num_lines;
492     scrdown(currcons);    // the screen content scrolls down one line
493     top=oldtop;
494     bottom=oldbottom;
495 }
496
497 // Delete a character.
498 // Delete one character at the cursor, and all characters to the right of the cursor are shifted
499 // to the left by one space.
500 static void delete_char(int currcons)
501 {
502     int i;
503     unsigned short * p = (unsigned short *) pos;
504
505     // Returns if the cursor's current column position x exceeds the rightmost column of the screen.
506     // Otherwise, all characters from the right character of the cursor to the end of the line are
507     // shifted to the left by one space. Then fill in the erase character at the last character.
508     if (x>=video_num_columns)
509         return;
510     i = x;
511     while (++i < video_num_columns) {    // shifted to the left by one space.
512         *p = *(p+1);
513         p++;
514     }
515     *p = video_erase_char;    // finally fill in the erase character.
516 }
517
518 // Delete one line where the cursor is located.

```



```

// Delete one line where the cursor is located, and then scrolls screen content up one line.
506 static void delete\_line(int currcons)
507 {
508     int oldtop, oldbottom;
509
// First save the screen scrolling start line 'top' and last line 'bottom', then scroll the
// screen content up one line from the line where the cursor is. Finally restore the original
// value of the screen scrolling start line 'top' and last line 'bottom'.
510     oldtop=top;
511     oldbottom=bottom;
512     top=y; // set start and end lines of the scroll screen.
513     bottom = video\_num\_lines;
514     scrup(currcons); // the screen content scrolls up one line.
515     top=oldtop;
516     bottom=oldbottom;
517 }
518
///// Insert nr characters at the cursor.
// Handling ANSI escape character sequences: 'ESC [ Pn @'. Insert one or more space characters
// at the current cursor. Pn is the number of characters inserted, and the default is 1. The
// cursor will still be at the first inserted space character. The character at the cursor and
// right border will shift to the right and characters beyond the right border will be lost.
// Parameter nr = Pn in the escape sequence.
519 static void csi\_at(int currcons, unsigned int nr)
520 {
// If the number of inserted characters is greater than one line of characters, it is truncated
// to one line of characters; if the number of inserted characters nr is 0, one character is
// inserted. Then cyclically insert nr space characters.
521     if (nr > video\_num\_columns)
522         nr = video\_num\_columns;
523     else if (!nr)
524         nr = 1;
525     while (nr--)
526         insert\_char(currcons);
527 }
528
///// Insert nr lines at the cursor position.
// Handling ANSI escape sequence: 'ESC [ Pn L'. The control sequence inserts one or more blank
// lines at the cursor. The cursor position does not change after the operation is completed.
// When a blank line is inserted, the line in the scroll area below the cursor moves down. The
// line scrolling out of the display page is lost. Parameter nr = Pn in the escape sequence.
529 static void csi\_L(int currcons, unsigned int nr)
530 {
// If the number of inserted lines is greater than the maximum number of screen lines, the number
// of lines is cut off to the screen lines; if the number of inserted lines nr is 0, one line
// is inserted.
Then cyclically inserts nr blank lines.
531     if (nr > video\_num\_lines)
532         nr = video\_num\_lines;
533     else if (!nr)
534         nr = 1;
535     while (nr--)
536         insert\_line(currcons);

```

```

537 }
538
539 // Delete nr characters at the cursor.
540 // Handling ANSI escape sequences: 'ESC [Pn P'. This control sequence deletes Pn characters
541 // from the cursor. When a character is deleted, all characters to the right of the cursor are
542 // shifted to the left, which produces a space character at the right border. Its properties
543 // should be the same as the last left-shift character, but here it is simplified, using only
544 // the default property of the character (black background white foreground space 0x0720) to
545 // set the space character. Parameter nr = Pn in the escape sequence.
546 static void csi\_P(int currcons, unsigned int nr)
547 {
548     // If the number of deleted characters is greater than one line of characters, it is truncated
549     // to one line of characters; if the number of deleted characters nr is 0, one character is
550     // deleted. Then iteratively deletes the specified number of characters nr at the cursor.
551     if (nr > video\_num\_columns)
552         nr = video\_num\_columns;
553     else if (!nr)
554         nr = 1;
555     while (nr--)
556         delete\_char(currcons);
557 }
558
559 // Delete the nr line at the beginning of the cursor.
560 // Process ANSI escape sequences: 'ESC [ Pn M'. The control sequence deletes one or more lines
561 // from the line where the cursor is located in the scroll area. When a line is deleted, the
562 // line below the deleted line in the scroll area moves up, and 1 blank line is added to the
563 // bottom line. If Pn is greater than the number of lines remaining on the display page, then
564 // this sequence only deletes these remaining lines and does not work outside the scroll area.
565 // Parameter nr = Pn in the escape sequence.
566 static void csi\_M(int currcons, unsigned int nr)
567 {
568     // If the number of deleted lines is greater than the maximum number of lines on the screen,
569     // the number of lines displayed is cut off. If the number of lines to be deleted nr is 0, 1
570     // line is deleted. Then iteratively deletes the specified nr lines.
571     if (nr > video\_num\_lines)
572         nr = video\_num\_lines;
573     else if (!nr)
574         nr=1;
575     while (nr--)
576         delete\_line(currcons);
577 }
578
579 // Save the current cursor position.
580 static void save\_cur(int currcons)
581 {
582     saved\_x=x;
583     saved\_y=y;
584 }
585
586 // Restore the saved cursor position.
587 static void restore\_cur(int currcons)
588 {
589     gotoxy(currcons, saved\_x, saved\_y);

```

```

568 }
569
570 // The following enumeration definitions are used in the following con_write() function to
// resolve escape sequences or control sequences. ESnormal is the initial entry state and is
// also the state when the escape or control sequence is processed.
// ESnormal - Indicates that it is in the initial normal state. At this time, if the normal
// display character is received, the character is directly displayed on the screen; if a
// control character (such as a carriage return) is received, the cursor position is set.
// When an escape or control sequence has just been processed, the program will return to
// this state.
// ESesc - indicates that the escape sequence leading character ESC (0x1b = 033 = 27) was
// received. If a '[' character is received in this state, it is an escape sequence guide
// code, so it jumps to ESSquare for processing, otherwise the received character is treated
// as an escape sequence. For selecting the character set escape sequences 'ESC (' and 'ESC )',
// we use a separate state ESsetgraph to handle. For the device control string sequence
// 'ESC P', we use a separate state ESsetterm to handle.
// ESSquare - Indicates that a control sequence preamble ('ESC [') has been received, indicating
// that a control sequence has been received, so this state performs a zero initialization
// on the parameter array par[]. If you receive a '[' character at this time, it means that
// you received the 'ESC [[' sequence, which is the sequence sent by the keyboard function
// key, so you can jump to ESfunkey to process it. Otherwise we need to prepare to receive
// the parameters of the control sequence, so set the state ESgetpars and directly enter
// the state to receive and save the sequence parameter characters.
// ESgetpars - This state indicates that we are going to receive the parameter values of the
// control sequence at this time. The arguments are represented in decimal numbers, and we
// convert the received numeric characters to numeric values and save them to the par[] array.
// If a semicolon ';' is received, it remains in this state and the received parameter value
// is saved in the next item of data par[]. If it is not a numeric character or a semicolon,
// indicating that all parameters have been obtained, then move to the state ESgotpars to
// process.
// ESgotpars - Indicates that we have received a complete control sequence. At this point we
// can process the corresponding control sequence according to the ending character received
// in this state. However, before processing, if we received '?' in the ESSquare state, this
// sequence is a private sequence of terminal devices. This kernel does not support the
// processing of this sequence, so we directly restore to the ESnormal state. Otherwise,
// the corresponding control sequence is executed, and the state is restored to ESnormal
// after the sequence is processed.
// ESfunkey - Indicates that we have received a sequence from the function keys on the keyboard
// that we don't need to display. So we return to the normal state ESnormal.
// ESsetterm - Indicates that it is in the device control string sequence (DCS) state. At this
// time, if the character 'S' is received, the initial display character attribute is
// restored. If the received character is 'L' or 'l', the folding line display mode is turned
// on or off.
// ESsetgraph - Represents the received character set escape sequence 'ESC (' or 'ESC )', which
// are used to specify the character set used by G0 and G1, respectively. At this time, if
// the character 'O' is received, the graphic character set is selected as G0 and G1, and
// if the received character is 'B', the ordinary ASCII character set is selected as the
// character set of G0 and G1.
571 enum { ESnormal, ESesc, ESSquare, ESgetpars, ESgotpars, ESfunkey,
572        ESsetterm, ESsetgraph };
573
574 // Console writes function.

```

```

// Characters are taken from the terminal's tty write buffer queue and analyzed for each
// character. If it is a control character or an escape or control sequence, control processing
// such as cursor positioning and character deletion is performed, and the normal character
// is directly displayed at the cursor.
// The parameter tty is the tty structure pointer used by the current console.
574 void con_write(struct tty_struct * tty)
575 {
576     int nr;
577     char c;
578     int currcons;
579
// The function first obtains the console number currcons according to the position of the tty
// in the tty table used by the current console, and then calculates (CHARS()) the number of
// characters nr contained in the current tty write queue, and loops out each character one
// by one for processing. However, if the current console is stopped due to a pause command
// issued by the keyboard or program (such as the button Ctrl-S), then the function stops
// processing the characters in the write queue and exits the function. In addition, if the
// control character CAN (Cancel, ASCII code 24, generated by Ctrl-X) or SUB (Substitute, 26,
// Ctrl-Z) is taken, then if the character is received during the escaping or control sequence,
// then the sequence will not execute and will terminate immediately, with the subsequent
// characters displayed. Note that the con_write() function only processes the characters
// currently in the write queue when fetching the number of characters in the queue. It is
// possible to read the number of characters during a sequence being placed in the write queue,
// so the 'state' may be in the other state of processing the escape or control sequence when
// the function exits last time.
580     currcons = tty - tty_table;
581     if ((currcons >= MAX_CONSOLES) || (currcons < 0))
582         panic("con_write: illegal tty");
583
584     nr = CHARS(tty->write_q);           // get nr of chars in the write queue.
585     while (nr-- > 0) {
586         if (tty->stopped)
587             break;
588         GETCH(tty->write_q, c);           // get one character.
589         if (c == 24 || c == 26)           // Control chars: Cancel or Substitute.
590             state = ESnormal;
591         switch(state) {
// If the character extracted is a normal display character, the corresponding display character
// is directly taken out from the current mapped character set, and placed at the display memory
// position where the current cursor is located, that is, the character is directly displayed.
// Then move the cursor position to the right by one character position. Specifically, if the
// character is not a control character or an extended character, ie (31 < c < 127), then if the
// current cursor is at the end of the line, or at a position other than the end, the cursor
// is moved to the first column of the next line. And adjust the memory pointer pos corresponding
// to the cursor position. The character is then written to the display memory pos, and the
// cursor is shifted to the right by 1 column, and pos is also moved 2 bytes correspondingly.
592             case ESnormal:
593                 if (c > 31 && c < 127) {           // normal display char.
594                     if (x >= video_num_columns) {           // change line?
595                         x -= video_num_columns;
596                         pos -= video_size_row;
597                         lf(currcons);
598                     }

```

```

599         __asm__( "movb %2, %%ah\n\t"           // write char.
600                 "movw %%ax, %1\n\t"
601                 ":: \"a\" (translate[c-32]),
602                 \"m\" (*(short *)pos),
603                 \"m\" (attr)
604                 : \"ax\");
605         pos += 2;
606         x++;
// If 'c' is the escape character ESC, the state is converted to ESesc (line 637).
// If c is LF(10), or VT(11), or FF(12), the cursor moves to the next line.
// If c is a carriage return CR (13), move the cursor to the head column (column 0).
// If c is DEL(127), the character to the left of the cursor is erased and the cursor is moved
// to the erased character position.
// If c is BS (backspace, 8), move the cursor to the left by 1 column and adjust the cursor
// corresponding to the memory pointer pos.
607     } else if (c==27)                // ESC - escape char.
608         state=ESesc;
609     else if (c==10 || c==11 || c==12)
610         lf(currcons);
611     else if (c==13)                // CR - carriage return.
612         cr(currcons);
613     else if (c==ERASE_CHAR(tty))
614         del(currcons);
615     else if (c==8) {                // BS - backspace.
616         if (x) {
617             x--;
618             pos -= 2;
619         }
// If c is horizontal tab HT(9), the cursor is moved to a multiple of 8. If the number of cursor
// columns exceeds the maximum columns on the screen, then move the cursor to the next line.
// If c is the bell BEL (7), the system beep function is called, and the speaker sounds.
// If c is the control character S0(14) or SI(15), the character set G1 or G0 is selected as
// the display character set accordingly.
620     } else if (c==9) {                // HT - Horizontal Tab.
621         c=8-(x&7);
622         x += c;
623         pos += c<<1;
624         if (x>video_num_columns) {
625             x -= video_num_columns;
626             pos -= video_size_row;
627             lf(currcons);
628         }
629         c=9;
630     } else if (c==7)                // BEL - Bell.
631         sysbeep();
632     else if (c == 14)                // S0 - Shift out, use G1.
633         translate = GRAF_TRANS;
634     else if (c == 15)                // SI - Shift in, use G0.
635         translate = NORM_TRANS;
636     break;
// If the escape character ESC (0x1b = 27) is received in the ESnormal state, it goes to this
// state processing. This state processes the control characters or escape characters in c.
// After processing, the default state will be ESnormal.

```

```

637         case ESesc:
638             state = ESnormal;
639             switch (c)
640             {
641                 case '[': // ESC [ - CSI sequence.
642                     state=ESsquare;
643                     break;
644                 case 'E': // ESC E - cursor next line & col 0.
645                     gotoxy(currcons, 0, y+1);
646                     break;
647                 case 'M': // ESC M - moves up one line.
648                     ri(currcons);
649                     break;
650                 case 'D': // ESC D - moves to next line.
651                     lf(currcons);
652                     break;
653                 case 'Z': // ESC Z - device property query.
654                     respond(currcons, tty);
655                     break;
656                 case '7': // ESC 7 - save cursor position.
657                     save_cur(currcons);
658                     break;
659                 case '8': // ESC 8 - restore cursor position.
660                     restore_cur(currcons);
661                     break;
662                 case '(' : case ')': // ESC (, ESC ) - select char set.
663                     state = ESsetgraph;
664                     break;
665                 case 'P': // ESC P - set terminal parameters.
666                     state = ESsetterm;
667                     break;
668                 case '#': // ESC # - modify line attributs.
669                     state = -1;
670                     break;
671                 case 'c': // ESC c - reset to default settings.
672                     tty->termios = DEF_TERMIOS;
673                     state = restate = ESnormal;
674                     checkin = 0;
675                     top = 0;
676                     bottom = video_num_lines;
677                     break;
678                 /* case '>': Numeric keypad */
679                 /* case '=': Appl. keypad */
680             }
681             break;

```

// If the character '[' is received in the state ESesc, it indicates that it is a CSI control sequence, so it goes to the state ESsquare to handle. First, the array par[] used to save the parameters of the ESC sequence is cleared, the index variable npar points to the first item, and we set the state to start the parameter ESgetpars. However, if the character received at this time is '[', it indicates that the sequence sent by the keyboard function key has been received, so the next state is set to ESfunkey. If the received character is not '?', go directly to the state ESgetpars to handle. If the received character is '?', it indicates that the sequence is a private sequence of terminal devices, followed by a function character.

```

// So go to the next character and go to the state ESgetpars to handle the code.
682         case ESsquare:
683             for(npar=0;npar<NPAR;npar++)        // Initialize para array.
684                 par[npar]=0;
685             npar=0;
686             state=ESgetpars;
687             if (c == '[')        /* Function key */    // 'ESC ['
688                 { state=ESfunckey;
689                   break;
690                 }
691             if (ques=(c=='?'))
692                 break;
// This state indicates that we are going to receive the parameter values of the control sequence
// at this time. The parameters are represented in decimal numbers, and we convert the received
// numeric characters to values and save them to the par[] array. If you receive a semicolon
// ';', it remains in this state and saves the received parameter value in the next item in
// the array par[]. If it is not a numeric character or a semicolon, indicating that all parameters
// have been obtained, then move to the state ESgotpars to process.
693         case ESgetpars:
694             if (c==';' && npar<NPAR-1) {
695                 npar++;
696                 break;
697             } else if (c>='0' && c<='9') {
698                 par[npar]=10*par[npar]+c-'0';
699                 break;
700             } else state=ESgotpars;
// The ESgotpars state indicates that we have received a complete control sequence. At this
// point we can process the control sequence based on the ending character received in this
// state. However, before processing, if we received '?' in the ESSquare state, this sequence
// is a private sequence of terminal devices. This kernel does not support the processing of
// this sequence, so we directly restore to the ESnormal state. Otherwise go to the corresponding
// control sequence. After the sequence is processed, the state is restored to ESnormal.
701         case ESgotpars:
702             state = ESnormal;
703             if (ques)        // received '?'
704                 { ques =0;
705                   break;
706                 }
707             switch(c) {
// If c is 'G' or '`', the first value in par[] represents the column number, and if it is not
// zero, the cursor is shifted to the left by one column.
// If c is 'A', the first value represents the number of lines moved up by the cursor. If the
// parameter is 0, it moves up one line.
// If c is 'B' or 'e', the first parameter represents the number of lines moved down by the
// cursor. If the parameter is 0, it will move down one line.
// If c is 'C' or 'a', the first parameter represents the number of columns to the right of
// the cursor. If the parameter is 0, it is shifted to the right by 1 column.
// If c is 'D', the first parameter represents the number of columns to the left of the cursor.
// If the parameter is 0, it will be shifted to the left by 1 column.
708                 case 'G': case '`': // CSI Pn G -move horizontally
709                     if (par[0]) par[0]--;
710                     gotoxy(currcons, par[0], y);
711                     break;

```

```

712         case 'A':           // CSI Pn A - move up.
713             if (!par[0]) par[0]++;
714             gotoxy(currcons, x, y-par[0]);
715             break;
716         case 'B': case 'e': // CSI Pn B - move down.
717             if (!par[0]) par[0]++;
718             gotoxy(currcons, x, y+par[0]);
719             break;
720         case 'C': case 'a': // CSI Pn C - move right.
721             if (!par[0]) par[0]++;
722             gotoxy(currcons, x+par[0], y);
723             break;
724         case 'D':           // CSI Pn D - move left.
725             if (!par[0]) par[0]++;
726             gotoxy(currcons, x-par[0], y);
727             break;
728
729         // If c is 'E', the first value represents the number of lines the cursor moves down and returns
730         // to column 0. If the value is 0, it will move down one line.
731         // If c is 'F', the first value represents the number of lines the cursor is moving up and returns
732         // to column 0. If the parameter is 0, it moves up one line.
733         // If c is 'd', the first value represents the line number (counted from 0) that the cursor
734         // is required to have.
735         // If c is 'H' or 'f', the first value represents the line number to which the cursor is moved,
736         // and the second parameter represents the column number to which the cursor is moved.
737         case 'E':           // CSI Pn E - move down, col 0.
738             if (!par[0]) par[0]++;
739             gotoxy(currcons, 0, y+par[0]);
740             break;
741         case 'F':           // CSI Pn F - move up, col 0.
742             if (!par[0]) par[0]++;
743             gotoxy(currcons, 0, y-par[0]);
744             break;
745         case 'd':           // CSI Pn d - set cursor line no.
746             if (par[0]) par[0]--;
747             gotoxy(currcons, x, par[0]);
748             break;
749         case 'H': case 'f': // CSI Pn H -set cursor position.
750             if (par[0]) par[0]--;
751             if (par[1]) par[1]--;
752             gotoxy(currcons, par[1], par[0]);
753             break;
754
755         // If the character c is 'J' (sequence 'ESC [ Ps J'), the first parameter represents the way
756         // the screen is cleared related to the cursor position.
757         // If c is 'K' ('ESC [ Ps K'), the first parameter represents the way in which the characters
758         // in the line are deleted related to the cursor position.
759         // If c is 'L' ('ESC [ Pn L'), it means that n lines are inserted at the cursor position.
760         // If c is 'M' ('ESC [ Pn M'), it means that n lines are deleted at the cursor position.
761         // If c is 'P' ('ESC [ Pn P'), it means that n chars are deleted at the cursor position.
762         // If c is '@' ('ESC [ Pn @'), it means that n chars are inserted at the cursor position.
763         case 'J':           // CSI Pn J - erase chars on screen.
764             csi_J(currcons, par[0]);
765             break;
766         case 'K':           // CSI Pn K - erase chars in a line.

```



```

749         csi\_K(currcons, par[0]);
750         break;
751     case 'L': // CSI Pn L - insert lines.
752         csi\_L(currcons, par[0]);
753         break;
754     case 'M': // CSI Pn M - delete lines.
755         csi\_M(currcons, par[0]);
756         break;
757     case 'P': // CSI Pn P - delete chars.
758         csi\_P(currcons, par[0]);
759         break;
760     case '@': // CSI Pn @ - insert chars.
761         csi\_at(currcons, par[0]);
762         break;
763     // If c is 'm' ('ESC [Pn m'), it means changing the display attributes of the characters at
764     // the cursor, such as bolding.
765     // If c is 'r' ('ESC [Pn;Pn r'), it means that the starting line number and the ending line
766     // number of the scrolling are set with two parameters.
767     // If c is 's' ('ESC [Pn s'), it means that the current cursor position is saved.
768     // If c is 'u' ('ESC [Pn u'), it means that the cursor is restored to the saved postion.
769     case 'm': // CSI Ps m - set char attributes.
770         csi\_m(currcons);
771         break;
772     case 'r': // CSI Pn;Pn r - set scroll range.
773         if (par[0] par[0]--);
774         if (!par[1]) par[1] = video\_num\_lines;
775         if (par[0] < par[1] &&
776             par[1] <= video\_num\_lines) {
777             top=par[0];
778             bottom=par[1];
779         }
780         break;
781     case 's': // CSI s - saved cursor postion.
782         save\_cur(currcons);
783         break;
784     case 'u': // CSI u - restore cursor postion.
785         restore\_cur(currcons);
786         break;
787     // If the character c is 'l' or 'b', it means setting the screen black screen interval time
788     // and setting the bold character display, respectively. At this time, par[1] and par[2] are
789     // feature values, which must be par[1]=par[0]+13;par[2]= par[0]+17 respectively. Under this
790     // condition, if c is 'l', then par[0] is the number of minutes to delay when starting a black
791     // screen; if c is 'b', the bold character attribute is set in par[0] value.
792     case 'l': /* blank interval */
793     case 'b': /* bold attribute */
794         if (!((npar >= 2) &&
795             ((par[1]-13) == par[0]) &&
796             ((par[2]-17) == par[0])))
797             break;
798         if ((c=='l')&&(par[0]>=0)&&(par[0]<=60))
799         {
800             blankinterval = HZ*60*par[0];
801             blankcount = blankinterval;

```

```

791     }
792     if (c=='b')
793         vc_cons[currcons].vc_bold_attr
794         = par[0];
795     }
796     break;
// The status ESfunkey indicates that we have received a sequence from the function keys on
// the keyboard. We don't need to display it, so we return to the normal state of ESnormal.
797     case ESfunkey:           // keyboard function key.
798         state = ESnormal;
799         break;
// The state ESsetterm indicates that it is in the device control string (DCS) sequence state.
// At this time, if the character 'S' is received, the initial display character attribute is
// restored. If the character is 'L' or 'l', the folding display mode is turned on or off.
800     case ESsetterm:  /* Setterm functions. */
801         state = ESnormal;
802         if (c == 'S') {
803             def_attr = attr;
804             video_erase_char = (video_erase_char&0xff) |
                                (def_attr<<8);
805         } else if (c == 'L')
806             ; /*linewrap on*/
807         else if (c == 'l')
808             ; /*linewrap off*/
809         break;
// The state ESsetgraph indicates that the set character set escape sequence 'ESC (' or 'ESC )'
// has been received. They are used to specify the character set used by G0 and G1, respectively.
// At this time, if the character 'O' is received, the graphic character set is selected as
// G0 and G1, and if the received character is 'B', the ordinary ASCII character set is selected
// as the character set of G0 and G1.
810     case ESsetgraph:        // 'CSI (O' or 'CSI (B' - select char set.
811         state = ESnormal;
812         if (c == 'O')
813             translate = GRAF_TRANS;
814         else if (c == 'B')
815             translate = NORM_TRANS;
816         break;
817     default:
818         state = ESnormal;
819 }
820 }
// Finally, set the cursor position in the display controller with the value set above.
821 set_cursor(currcons);
822 }
823
824 /*
825 * void con_init(void);
826 * 
827 * This routine initializes console interrupts, and does nothing
828 * else. If you want the screen to clear, call tty_write with
829 * the appropriate escape-seqece.
830 * 
831 * Reads the information preserved by setup.s to determine the current display

```

```

832 * type and sets everything accordingly.
833 */
834 void con_init(void)
835 {
836     register unsigned char a;
837     char *display_desc = "????";
838     char *display_ptr;
839     int currcons = 0;                // current console no.
840     long base, term;
841     long video_memory;
842
843     // First, according to the system hardware parameters obtained by the setup.s program, several
844     // static global variables specific to this function are initialized (see lines 60-68).
845     video_num_columns = ORIG_VIDEO_COLS;
846     video_size_row = video_num_columns * 2;    // bytes used by a display row.
847     video_num_lines = ORIG_VIDEO_LINES;        // lines of the display.
848     video_page = ORIG_VIDEO_PAGE;              // display page.
849     video_erase_char = 0x0720;                  // char: 0x20, attr: 0x07.
850     blankcount = blankinterval;                // unit: ticks.
851
852     // Then, according to whether the display mode is monochrome or color, the display memory start
853     // position and the display register index port number and the display register data port number
854     // are respectively set. If the obtained BIOS display mode is equal to 7, it means that it is
855     // a monochrome display card.
856     if (ORIG_VIDEO_MODE == 7)                    /* Is this a monochrome display? */
857     {
858         video_mem_base = 0xb0000;                // start addr of monochrome display.
859         video_port_reg = 0x3b4;                  // index register port.
860         video_port_val = 0x3b5;                  // data register port.
861
862         // Then we determine whether the display card is a monochrome card or a color card according
863         // to the display mode information obtained by the BIOS interrupt int 0x10 function 0x12. If
864         // the return value of the BX register obtained by the interrupt is not equal to 0x10, it means
865         // that it is an EGA card, so the initial display type is EGA monochrome. Although there is
866         // more display memory on the EGA card, it can only use up to 32KB of display memory with an
867         // address range between 0xb0000--0xb8000 in monochrome mode. The code then sets the display
868         // description string to 'EGAm' and it will be displayed in the upper right corner of the screen
869         // during system initialization.
870         // Note that the BX register is used to determine the type of card if it is changed before and
871         // after the interrupt int 0x10 is called. If the value of the BL is changed after the interrupt
872         // is called, it means that the display card supports the Ah=12h function call, which is a type
873         // of VGA or later VGA. If the return value of the interrupt call has not changed, indicating
874         // that the display card does not support this function, it indicates that it is a general
875         // monochrome display card.
876         if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
877         {
878             video_type = VIDEO_TYPE_EGAM;    // video type (EGA mono).
879             video_mem_term = 0xb8000;          // video memory end address.
880             display_desc = "EGAm";
881         }
882         // If the value of the BX register is equal to 0x10, it means a monochrome display card MDA
883         // and only 8KB of display memory.
884     }
885     else

```

```

862         {
863             video_type = VIDEO_TYPE_MDA;    // MDA mono.
864             video_mem_term = 0xb2000;      // memory end address.
865             display_desc = "MDA";
866         }
867     }
    // If the display mode is not 7, it indicates a color display card. At this time, the display
    // memory starting address used in the text mode is 0xb8000.
868     else                                     /* If not, it is color. */
869     {
870         can_do_colour = 1;                  // can use color flag.
871         video_mem_base = 0xb8000;          // memory start address.
872         video_port_reg = 0x3d4;           // index register port.
873         video_port_val = 0x3d5;           // data register port.
    // Then judge the display card category. If BX is not equal to 0x10, it means that it is an
    // EGA card. At this time, a total of 32KB of display memory is available (0xb8000-0xc0000).
    // Otherwise, it is a CGA card and can only use 8KB display memory (0xb8000-0xba000).
874     if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
875     {
876         video_type = VIDEO_TYPE_EGAC;    // EGA type.
877         video_mem_term = 0xc0000;        // mem end address.
878         display_desc = "EGAc";
879     }
880     else
881     {
882         video_type = VIDEO_TYPE_CGA;    // CGA type.
883         video_mem_term = 0xba000;        // meme end address.
884         display_desc = "CGA";
885     }
886 }

    // Now let's calculate the number of virtual consoles that can be opened on the current display
    // card. The number of virtual consoles allowed by the hardware is equal to the total amount
    // of memory video_memory divided by the number of bytes occupied by each virtual console. The
    // number of display memory occupied by each virtual console is equal to the number of lines
    // multiplied by the number of bytes per line of characters: video_num_lines * video_size_row.
    // If the number of virtual consoles allowed is greater than the maximum number of system-defined
    // MAX_CONSOLES, set the number of virtual consoles to MAX_CONSOLES. If the number of virtual
    // consoles thus calculated is 0, it is set to 1. Finally, the total number of video memory divided
    // by the number of virtual consoles is the number of bytes of memory occupied by each virtual
    // console (vc).
887     video_memory = video_mem_term - video_mem_base;
888     NR_CONSOLES = video_memory / (video_num_lines * video_size_row);
889     if (NR_CONSOLES > MAX_CONSOLES)      // MAX_CONSOLES = 8
890         NR_CONSOLES = MAX_CONSOLES;
891     if (!NR_CONSOLES)
892         NR_CONSOLES = 1;
893     video_memory /= NR_CONSOLES;          // memory of each vc.
894
895     /* Let the user known what kind of display driver we are using */
896
    // Then we display the description string in the upper right corner of the screen. The method
    // used is to write the string directly to the corresponding location in the display memory.

```

```

// First, the display pointer display_ptr points to the last 4 characters of the right end of
// the first line (each character requires 2 bytes, so minus 8), and then cyclically copies
// the character of the string, and reserve one attribute byte for each character.
897     display_ptr = ((char *)video mem base) + video size row - 8;
898     while (*display_desc)
899     {
900         *display_ptr++ = *display_desc++;
901         display_ptr++;
902     }
903
904     /* Initialize the variables used for scrolling (mostly EGA/VGA) */
905
// Note that the current virtual console number currcons has been initialized to 0 at this time.
// So the following is actually the initialization of all the fields in the structure vc_cons[0].
// For example, the symbol 'origin' here has been defined as 'vc_cons[0].vc_origin' on line
// 115 above. Below we first set the default scrolling start memory location video_mem_start
// and the default scrolling last line memory location (actually they are part of the display
// memory area occupied by console 0), and then set other properties of virtual console 0.
906     base = origin = video mem start = video mem base; // default scroll start address
907     term = video mem end = base + video_memory; // end mem of #0 vc.
908     scr end = video mem start + video num lines * video size row; // end mem of scroll.
909     top = 0; // top line number of scrolling.
910     bottom = video num lines; // bottom line number of scrolling.
911     attr = 0x07; // default attribute (white on black).
912     def attr = 0x07; // set default char attribute.
913     restate = state = ESnormal; // current and next state of escape sequence.
914     checkin = 0;
915     ques = 0; // got question mark flag.
916     iscolor = 0; // color flag.
917     translate = NORM TRANS; // char set used (ordinary ASCII table).
918     vc_cons[0].vc_bold_attr = -1; // bold char attribute flag (-1 not used).
919
// After setting the current cursor position of console 0 and the memory location pos
// corresponding to the cursor, we loop through the structure of the remaining virtual consoles.
// Except for the start and end positions of the display memory occupied by each, their remaining
// initial values are basically the same as console 0.
920     gotoxy(currcons, ORIG X, ORIG Y);
921     for (currcons = 1; currcons < NR CONSOLES; currcons++) {
922         vc_cons[currcons] = vc_cons[0]; // copy structure data of #0
923         origin = video mem start = (base += video_memory);
924         scr end = origin + video num lines * video size row;
925         video mem end = (term += video_memory);
926         gotoxy(currcons, 0, 0); // cursor is at top left.
927     }
// Finally, set the screen origin (upper left corner) position of the current foreground console
// and the cursor position in the display controller, and set the keyboard interrupt 0x21 trap
// gate descriptor. The masking of the keyboard interrupt is then disabled, allowing the IRQ1
// request signal from the keyboard to be responded to. Finally reset the keyboard controller
// to allow the keyboard to start working properly.
928     update screen(); // update foreground origin & cursor pos.
929     set trap gate(0x21, &keyboard interrupt); // refer to system.h, line 36.
930     outb_p(inb_p(0x21) & 0xfd, 0x21); // cancel masking of keyboard.
931     a = inb_p(0x61); // read keyboard port 0x61 (8255A PB).

```

```

932     outb_p(a|0x80, 0x61);                // disable keyboard (set bit 7).
933     outb_p(a, 0x61);                    // enable it again to reset keyboard.
934 }
935
// Update the current foreground console.
// Switch the foreground console to the virtual console specified by fg_console. fg_console
// is the foreground virtual console number set.
936 void update_screen(void)
937 {
938     set_origin(fg_console);                // set the scroll start memory address.
939     set_cursor(fg_console);                // set cursor position in the controller.
940 }
941
942 /* from bsd-net-2: */
943
////// Stop beeping.
// Reset bit 1 and bit 0 of the 8255A PB port. See the timer programming instructions after
// the kernel/sched.c program.
944 void sysbeepstop(void)
945 {
946     /* disable counter 2 */
947     outb(inb_p(0x61)&0xFC, 0x61);
948 }
949
950 int beepcount = 0;                        // beeping counts (ticks).
951
// Turn on the beep.
// Bit 1 of the 8255A chip PB port is used as the door open signal of the speaker; bit 0 is
// used as the door signal of the 8253 timer 2, and the output pulse of the timer is sent to
// the speaker as the frequency at which the speaker emits sound. Therefore, to make the speaker
// beep, two steps are required: first turn on the PB port (0x61) bit 1 and bit 0 (set), then
// set the timer 2 channel to send a certain timing frequency. See the 8259A interrupt controller
// chip programming method after the boot/setup.s program, and refer to the programming
// instructions for the timer after the kernel/sched.c program.
952 static void sysbeep(void)
953 {
954     /* enable counter 2 */
955     outb_p(inb_p(0x61)|3, 0x61);
956     /* set command for counter 2, 2 byte write */
957     outb_p(0xB6, 0x43);                    // timer control word register port.
958     /* send 0x637 for 750 HZ */
959     outb_p(0x37, 0x42);                    // send high & low count bytes to channel 2.
960     outb(0x06, 0x42);
961     /* 1/8 second */
962     beepcount = HZ/8;
963 }
964
////// Copy the screen.
// Copy the screen contents to the user buffer arg specified by the parameter.
// The parameter arg has two purposes, one is to pass the console number, and the other is to
// act as a user buffer pointer.
965 int do_screendump(int arg)
966 {

```

```

967     char *sptr, *buf = (char *)arg;
968     int currcons, l;
969
970     // The function first verifies the buffer capacity provided by the user, and if it is not enough,
971     // it expands appropriately. Then take the console number currcons from its beginning. After
972     // determining that the console number is valid, all the memory contents of the console screen
973     // are copied to the user buffer.
974     verify_area(buf, video_num_columns*video_num_lines);
975     currcons = get_fs_byte(buf);
976     if ((currcons<1) || (currcons>NR_CONSOLES))
977         return -EIO;
978     currcons--;
979     sptr = (char *) origin;
980     for (l=video_num_lines*video_num_columns; l>0 ; l--)
981         put_fs_byte(*sptr++, buf++);
982     return(0);
983 }
984
985 // Black screen processing.
986 // When the user does not press any button during the blankInterval interval, the screen is
987 // blacked out to protect the screen.
988 void blank_screen()
989 {
990     if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
991         return;
992     /* blank here. I can't find out how to do it, though */
993 }
994
995 // Restore the black screen.
996 // When the user presses any button, the screen in the black screen state is restored.
997 void unblank_screen()
998 {
999     if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
1000         return;
1001     /* unblank here */
1002 }
1003
1004 // The console print function.
1005 // This function is only used by the kernel display function printk() (kernel/printk.c) to print
1006 // kernel information on the current foreground console. The processing method is to loop out
1007 // the characters in the buffer and control the cursor movement or directly display on the screen
1008 // according to the characteristics of the characters. The argument b is a null-terminated string
1009 // buffer pointer.
1010 void console_print(const char * b)
1011 {
1012     int currcons = fg_console;
1013     char c;
1014
1015     // Cycle through the characters in the buffer. If the current character is a newline character,
1016     // a carriage return line feed operation is performed on the cursor; then the next character
1017     // is processed. If it is a carriage return, it will directly perform a carriage return. Then
1018     // go to the next character.
1019     while (c = *(b++)) {

```

```
1001         if (c == 10) {
1002             cr(currcons);
1003             lf(currcons);
1004             continue;
1005         }
1006         if (c == 13) {
1007             cr(currcons);
1008             continue;
1009         }
1010         // After reading a character that is not a carriage return or a newline, if you find that the
1011         // current cursor column position has reached the right end of the screen, let the cursor fold
1012         // back to the beginning of the next line. Then place the character at the memory location where
1013         // the cursor is located, which is displayed on the screen. Move the cursor right one column
1014         // to prepare for the next character.
1015         if (x >= video\_num\_columns) {
1016             x -= video\_num\_columns;
1017             pos -= video\_size\_row;
1018             lf(currcons);
1019         }
1020         // The register AL is the character to be displayed. Here the attribute byte is placed in AH,
1021         // and then the AX content is stored in the cursor memory location pos, that is, the character
1022         // is displayed at the cursor.
1023         __asm__("movb %2, %%ah\n\t"           // attribute byte -> ah.
1024               "movw %%ax, %1\n\t"           // put value of ax to pos.
1025               :: "a" (c),
1026               "m" (*(short *)pos),
1027               "m" (attr)
1028               : "ax");
1029         pos += 2;
1030         x++;
1031     }
1032     // Finally, set the cursor position in the display controller with the value set above.
1033     set\_cursor(currcons);
1034 }
1035
1036
```

---

## 10.3.3 Infomation

### 10.3.3.1 VGA adapter programming

Here we only give instructions on IBM VGA and its compatible graphics card ports, mainly describing the general programming ports of MDA, CGA, EGA and VGA display control cards. These ports are compatible with the MC6845 chip used by the CGA. The names and uses are shown in Table 10-5. The CGA/EGA/VGA port (0x3d0-0x3df) is taken as an example for description. The port range of the MDA is 0x3b0 - 0x3bf.

The basic steps for programming the display card are: first write the 0--17 value into the index register of the display card (port 0x3d4), select one of the display control internal registers (r0--r17), and the data register port (0x3d5) ) corresponds to the internal register. The parameters are then written to the data register port. That is, the data register port of the display card can only operate on one internal register in the display card at a time. The main internal registers of the display card are shown in Table 10-6.

Table 10-5 CGA port register name and functions



Port	R/W	Name and usage
0x3b4/0x3d4	W	6845 index register. Used to select which register (r0-r17) is to be accessed through port 0x3d5.
0x3b5/0x3d5	W	6845 data register. registersr 14-r17 may be read. The function description of each data register is shown in Table 10-6.
0x3b8/0x3d8	R/W	6845 mode control register bit 7-6 unused; bit 5=1 blinking on; bit 4=1 640*200 B/W graphics mode; bit 3=1 enable video signal; bit 2=1 B/W, 0= color; bit 1=1 320*200 graphics, =0 text; bit 0=1 80*25 text; =0 40*25 text.
0x3b9/0x3d9	R/W	CGA palette register. Choose the color you are using. bit 7-6 unused; bit 5=1 Activate the color set: cyan, magenta, white; =0 Activate color set: red, green, blue; bit 4=1 Enhance graphics, text display background color; bit 3=1 Enhanced display of 40*25 borders, 320*200 background, 640*200 foreground color bit 2=1 Display red: 40*25 border, 320*200 background, 640*200 foreground; bit 1=1 Display green: 40*25 border, 320*200 background, 640*200 foreground; bit 0=1 Display blue: 40*25 border, 320*200 background, 640*200 foreground;
0x3ba/0x3da	R	CGA status register. bit 7-4 unused; bit 3=1 virtical retrace, RAM access OK for next 1.25ms; bit 2=1 light pen off, =0 light pen on; bit 1=1 light pen trigger set; bit 0=1 access memory without disturbing the display; = 0 Do not use memory at this time.
0x3bb/0x3db	W	Clear the light pen latch (reset the light pen register).
0x3dc	R/W	Pre-set the light pen latch (forced light pen strobe is valid).

Table 10-6 MC6845 internal data register and initial values

Reg.	Register name	Unit	R/W	40*25mode	80*25 mode	Graphics mode
r0	Horizontal Total	Char	W	0x38	0x71	0x38
r1	Horizontal Displayed	Char	W	0x28	0x50	0x28
r2	Horizontal Sync Position	Char	W	0x2d	0x5a	0x2d
r3	Horizontal Sync Paulse Width	Char	W	0x0a	0x0a	0x0a
r4	Vertical Total	Char row	W	0x1f	0x1f	0x7f
r5	Vertical Total Adjust	Scan line	W	0x06	0x06	0x06
r6	Vertical Displayed	Char row	W	0x19	0x19	0x64
r7	Vertical Sync Position	Char row	W	0x1c	0x1c	0x70
r8	Interlace Mode		W	0x02	0x02	0x02
r9	Maximum Scan Line Address	Scan line	W	0x07	0x07	0x01

r10	Cursor Start	Scan line	W	0x06	0x06	0x06
r11	Cursor End	Scan line	W	0x07	0x07	0x07
r12	Start Mem Address (High)		W	0x00	0x00	0x00
r13	Start Mem Address (Low)		W	0x00	0x00	0x00
r14	Cursor position (High)		R/W	Vary		
r15	Cursor position (Low)		R/W			
r16	Light Pen (High)		R	Vary		
r17	Light Pen (Low)		R			

### 10.3.3.2 Principles of scrolling operation

Scrolling refers to moving a piece of text on the specified start and end lines on the screen up (scroll up) or down (scroll down). If you think of the screen as a window that displays the corresponding screen content on the memory, moving the screen contents up is to move the window down the display memory; moving the screen contents down is to move the window up. In the program, the starting position 'origin' of the video memory in the controller is re-set, and the corresponding variables in the adjustment program are adjusted. There are two cases for each of these two operations.

For the scroll up, when the corresponding display memory window of the screen is still within the display memory range after moving downward, that is, the memory block position corresponding to the current screen is always between the memory start position (video\_mem\_start) and the end position video\_mem\_end, then we only need to adjust the starting display memory location in the display controller. However, when the position of the memory block corresponding to the screen moves beyond the end of the actual memory (video\_mem\_end), it is necessary to move the data in the corresponding memory to ensure that all current screen data falls within the display memory range. In this second case, the program moves the memory data corresponding to the screen to the beginning of the actual display memory (video\_mem\_start).

The actual processing in the program is carried out in three steps. First adjust the screen display starting position origin; then determine whether the corresponding screen memory data exceeds the display memory lower bound (video\_mem\_end), if it is exceeded, move the screen corresponding memory data to the beginning of the actual display memory (video\_mem\_start); The new line that appears on the screen is filled with space characters. The operation diagram is shown in Figure 10-8. Figure (a) corresponds to the first simple case, and Figure (b) corresponds to the case when the memory data needs to be moved.

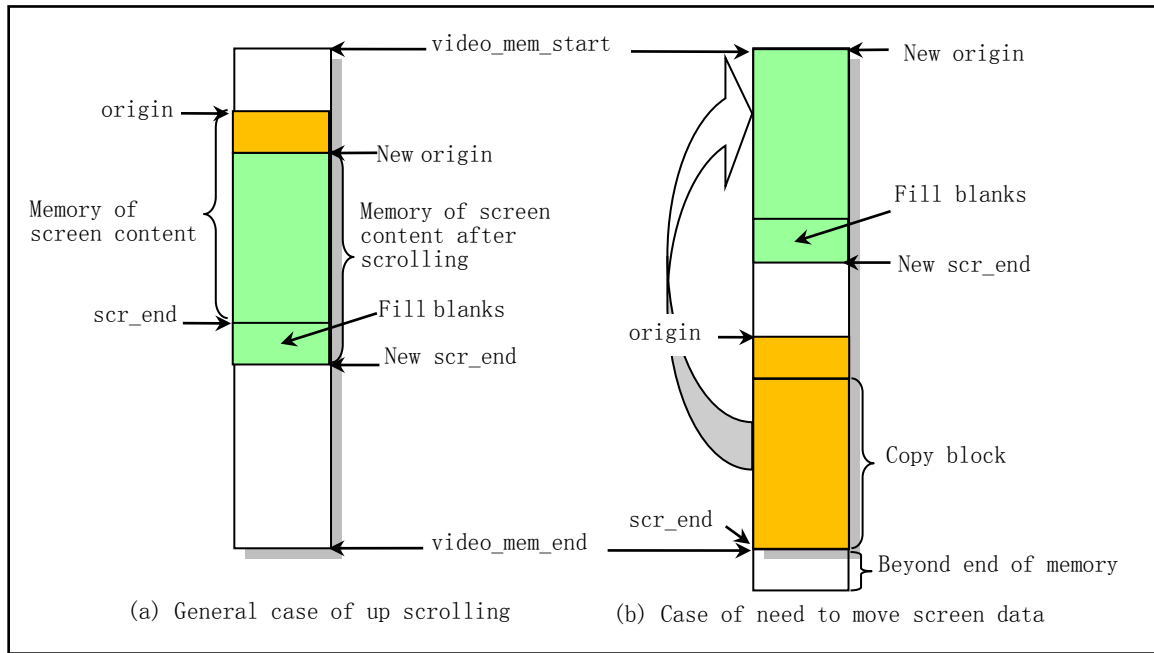


Figure 10-8 Schematic diagram of scroll up operation

Scrolling down the screen is similar to scrolling up and encounters two similar situations. Just because the screen window moves up, a blank line appears at the top of the screen, and when the memory corresponding to the screen content exceeds the display memory range, the screen data memory block needs to be moved down to the end position of the display memory.

### 10.3.3.3 Terminal Control Commands

Terminal devices usually have two functions, which serve as information input devices (keyboards) and output devices (displays) for the computer. The terminal can have a number of control commands that cause the terminal to perform certain operations rather than just displaying a character on the screen. In this way, the computer can order the terminal to perform operations such as moving the cursor, switching the display mode, and ringing. Terminal control commands can be further divided into two categories: control character commands and ANSI escape control sequences. As we discussed briefly, the console.c (including the keyboard.s above) program in the Linux kernel can actually be seen as a simulated terminal emulator. Therefore, in order to understand the processing of the console.c program, we outline how the program in the ROM in a terminal device handles the code data received from the host. We first briefly describe the structure of the ASCII code table, and then explain how the terminal device handles the received control characters and control sequence string code.

#### 1. Character encoding method

The conventional character terminal uses an 8-bit encoding scheme of ANSI (American National Standards Institute) and ISO (International Standards Organization) standards and a 7-bit code extension technique. ANSI and ISO specify character encoding standards in the computer and communications fields. The ANSI X3.4-1977 and ISO 646-1977 standards define the American Standard Code for Information Interchange, the ASCII code set. The ANSI X3.41-1974 and ISO 2022.2 standards describe code extension techniques for 7-bit and 8-bit code sets. ANSI X3.32, ANSI X3.64-1979 has developed a method for representing terminal control characters using text characters in ASCII code. Although the Linux 0.1x kernel only implements compatibility with VT100 and VT102 terminal devices of Digital Equipment Corporation DEC (now incorporated into Compaq and HP),

and these two de facto standard terminal devices only support 7-bit encoding schemes. However, for the sake of completeness and convenience of description, here we also introduce the 8-bit coding scheme.

## 2. ASCII code table

ASCII code has 7-bit and 8-bit code representation. The 7-bit code table has a total of 128 character codes, as shown in the left half of Table 10-7. Each of the rows represents a value of 4 bits lower in 7 bits, and each column is a high 3 bit value. For example, the binary value of the code 'A' of column 4 and row 1 is 0b0100, 0001 (0x41), and the decimal value is 65.

The characters in the table are divided into two types. One is a control character composed of the first and second columns, and the rest are graphic characters or display characters, text characters. The terminal will process the two types of characters separately. Graphic characters are characters that can be displayed on the screen, while control characters are usually not displayed on the screen. Control characters are used for special control during data communication and text processing. In addition, the DEL character (0x7F) is also a control character, and the space character (0x20) can be either a normal text character or a control character. Control characters and their functions have been standardized by ANSI, and the names are ANSI standard mnemonics. For example: CR (Carriage Return), FF (Form Feed) and CAN (Cancel). Usually the 7-bit encoding method is also applicable to 8-bit encoding. Tables 10-7 are 8-bit code tables (where the left half of the table is identical to the 7-bit code table), with the extension code for the right half not listed.

Table 10-7 8-bit ASCII code table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p			无					
1	SOH	DC1	!	1	A	Q	a	q								
2	STX	DC2	"	2	B	R	b	r								
3	ETX	DC3	#	3	C	S	c	s								
4	EOT	DC4	\$	4	D	T	d	t	IND							
5	ENQ	NAK	%	5	E	U	e	u	NEL							
6	ACK	SYN	&	6	F	V	f	v	SSA							
7	BEL	ETB	'	7	G	W	g	w	ESA							
8	BS	CAN	(	8	H	X	h	x	HTS							
9	HT	EM	)	9	I	Y	i	y	HTJ							
A	LF	SUB	*	:	J	Z	j	z	VTB							
B	VT	ESC	+	;	K	[	k	{	PLD	CSI						
C	FF	FS	,	<	L	\	l		PLU	ST						
D	CR	GS	-	=	M	]	m	}	RI	OSC						
E	SO	RS	.	>	N	^	n	~	SS2	PM						
F	SI	US	/	?	O	_	o	DEL	SS3	APC						NIL
	C0 codes		GL codes						C1 codes		GR codes					
	7-bit code table								The right half of 8-bit code table							

It has 8 more columns of code than the 7-bit code table, and contains a total of 256 codes. Similar to the 7-bit code table, each row represents the lower 4 bit value of the 8-bit code, and each column represents the high 4-bit value. The left half of the table (column 0 - column 7) is exactly the same as the 7-bit code table, and the 8th bit of their code is 0, so this bit can be ignored. The 8th bit of each code in the right half of the table (column 8 - column 15) is all 1, so these characters can only be used in an 8-bit environment. The 8-bit code

table has two control code sets: C0 and C1. There are also two graphic character sets: the left graphic character set GL (Graphic Left) and the right graphic character set GR (Graphic Right).

The functions of the control characters in C0 and C1 cannot be changed, but we can map different display characters to the GL and/or GR areas. Various text character sets that can be used (mapped) are usually stored in the terminal device. We must first do the mapping before using them. For DEC terminal devices that have become the de facto standard, the DEC multi-national character set (ASCII character set and DEC auxiliary character set), the DEC special character set, and the National Replacement Character set (NCR) are usually stored. When the terminal device is turned on, the DEC multi-national character set is used by default.

### 3. Control functions

In order to direct the terminal device how to process the received data, we need to use the control function of the terminal device. By transmitting a control code or a sequence of control codes, the host can control the display processing of the characters by the terminal device, which are only used to control the display, processing and transmission of text characters, and are not themselves displayed on the screen. Control functions have many uses, such as moving the cursor position on the display, deleting a line of text, changing characters, changing the character set, and setting the terminal operating mode. We can use all the control functions in text mode and use one byte or more bytes to represent the control functions.

It can be considered that all control characters or control character sequences that are not used for display on the screen are control functions. Not all control functions perform their control operations in each ANSI-compliant terminal device, but the device should be able to recognize all control functions and ignore control functions that do not work. So usually a terminal device only implements a subset of the ANSI control functions. Because different devices use different subsets of control functions, compatibility with ANSI standards does not mean that these devices are compatible with each other. Compatibility is only reflected in the fact that various devices use the same control functions.

The single-byte control functions are the control characters in C0 and C1 shown in Table 10-7. Limited control functions are available using the control characters in C0. The control characters in C1 can provide some additional control functions, but they can only be used in an 8-bit environment. Therefore, the VT100 type terminal emulated in the Linux kernel here can only use the control characters in C0. Multi-byte control code can provide a lot of control functions. These multibyte control codes are commonly referred to as Escape Sequences, Control Sequences, and Device Control Strings. Some of these control sequences are common sequences of ANSI standards in the industry, and others are proprietary control sequences designed by manufacturers for their own products. Like the ANSI standard sequence, proprietary control sequence characters also conform to the combined standard of ANSI character codes.

### 4. Escape Sequences

The host can send an escape sequence to control the display position and attributes of the text characters on the terminal screen. The escape sequence begins with the control character ESC (0x1b) in C0 and is followed by one or more ASCII display characters. The ANSI standard format for escape sequences is as follows:

ESC 0x1b	I.....I 0x20--0x2f	F 0x30--0x7e
Introducer	Intermediate chars (0 or multi chars)	Final char (one char)

The ESC is an escape sequence introducer defined in the ANSI standard. After receiving the introducer code ESC, the terminal needs to save (rather than display) all subsequent control characters in a certain order.

Intermediate characters are those received after ESC in the range 0x20 -- 0x2f (column 2 in the above ASCII table). Terminals need to save them as part of the control function.

The final character is a character that is received after ESC in the range 0x30 -- 0x7e (column 3 -- 7 in the ASCII table), and the final character indicates the end of a escape sequence. The intermediate and end characters together define the function of a sequence. At this point, the terminal can perform the functions specified by the escape sequence and continue to display the characters that are subsequently received. The final character of the ANSI standard escape sequence ranges from 0x40 to 0x7e (columns 4 - 7 in the ASCII table). The final characters of the proprietary escape sequences defined by each terminal device manufacturer range from 0x30 to 0x3f (column 3 in the ASCII table). For example, here is an escape sequence that specifies G0 as the ASCII character set:

ESC	(	B
0x1b	0x28	0x42

Since the escape sequence uses only 7-bit characters, we can use them in both 7-bit and 8-bit environments. Note that when using escape or control sequences, remember that they define a code sequence rather than a textual representation of the characters. One of the important uses of the escape sequence is to extend the functionality of the 7-bit control character. The ANSI standard allows us to use a 2-byte escape sequence as a 7-bit code extension to represent any control character in C1. This is a very useful feature in applications that require 7-bit compatibility. For example, the control characters CSI and IND in C1 can be represented using a 7-bit code extension form as follows:

C1 char	Escape sequences
CSI	ESC [
0x9b	0x1b 0x5b
IND	ESC D
0x84	0x1b 0x44

In general, we can use the above code extension technology in two ways. We can use a 2-character escape sequence to represent any control character in the 8-bit code table C1. The value of the second character is the value of the corresponding character in C1 minus 0x40 (64). Alternatively, we can convert any escape sequence with second character value between 0x40 and 0x5f to produce an 8-bit control character by deleting the control character ESC and adding 0x40 to the second character.

## 5. Control sequences

Control Sequences start with the control character CSI (0x9b) followed by one or more ASCII graphic characters. The ANSI standard format for the control sequence is as follows:

CSI	P.....P	I.....I	F
0x9b	0x30--0x3f	0x20--0x2f	0x40--0x7e
Introducer	Parameter chars (0 or more chars)	Intermediate chars (0 or multi chars)	Final char (one char)

The control sequence introducer is the CSI (0x9b) in the 8-bit control character C1. However, since the CSI can also be extended with the 7-bit code extension 'ESC [', all control sequences can be represented using

the escaped sequence in which the second character is the left bracket '['. After receiving the introducer CSI, the terminal needs to save (rather than display) all subsequent control characters in a certain order.

Parameter characters are characters received after CSI in the range 0x30 -- 0x3f (column 3 in the ASCII table), which are used to modify the role or meaning of the control sequence. When the parameter character begins with any '<=>?' (0x3c - 0x3f) character, the terminal will use this control sequence as a proprietary (private) control sequence. The terminal can use two types of parameter characters: numeric characters and select characters. The numeric character parameter represents a decimal number and is represented by Pn. The range is 0 -- 9. The select character parameter comes from a specified parameter list, denoted by Ps. If a control sequence contains more than one parameter, it is separated by a semicolon ';' (0x3b).

Intermediate characters are characters that are received after CSI in the range 0x20 -- 0x2f (column 2 in the ASCII table). Terminals need to save them as part of the control function. Note that the terminal device does not use intermediate characters.

The final character is a character received after CSI in the range 0x40 -- 0x7e (column 4 -- 7 in the ASCII table). The final character indicates the end of the control sequence. The intermediate and final characters together define the function of a sequence. At this point, the terminal can perform the specified function and continue to display the characters that are subsequently received. The final character of the ANSI standard escape sequence ranges from 0x40 to 0x6f (columns 4 -- 6 in the ASCII table). The final character of the proprietary escape sequences defined by each terminal device manufacturer range from 0x70 to 0x7e (column 7 in the ASCII table). For example, the following sequence defines a control sequence that moves the screen cursor to the specified position (row 5, column 9):

---

```

CSI  5  ;  9  H
0x9b 0x35 0x3b 0x41 0x48
or:
ESC  [  5  ;  9  H
0x1b 0x5b 0x35 0x3b 0x39 0x48

```

---

Figure 10-9 shows an example of a control sequence: cancel the attributes of all characters, then turn on the attributes of underline and reverse: ESC [ 0;4;7m

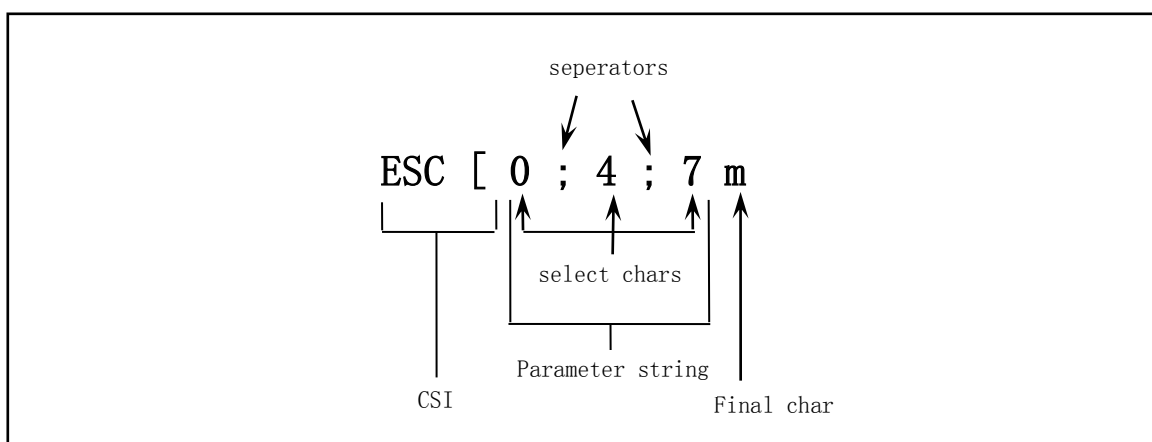


Figure 10-9 Example of control sequence

## 6. Terminal handling of received characters

Here is a brief description of how the terminal handles the received characters, that is, the response of the

terminal to the code sent from the application or host system. The characters received by the terminal can be divided into two categories: graphic (display or text) characters and control characters. Graphic characters are the characters that are received and displayed on the screen. The characters actually displayed on the screen depend on the selected character set. The character set can be selected through control functions.

All data received by the terminal consists of one or more character codes. These data include graphical characters, control characters, escape sequences, control sequences, and device control strings. Most of the data is made up of graphic characters that are only displayed on the screen and has no other effect. Control characters, escape sequences, control sequences, and device control strings are all "control functions," which we can use in our own programs or operating systems to indicate how the terminal processes, transmits, and displays characters. Each control function has a unique name and has a shorthand mnemonic. These names and mnemonics are standard. By default, the terminal's interpretation of a control or display character depends on the ASCII character set used. Note that for unsupported control code, the usual action taken by the terminal is to ignore it. Subsequent characters sent to the terminal that are not described here may have unpredictable consequences.

The appendix of this book gives a description of the commonly used control characters in the C0 and C1 tables, and outlines the actions that will be taken when the terminal receives it. For a particular terminal, it usually does not recognize all control characters in C0 and C1. In addition, the appendix also lists the escape sequences and control sequences used by the `console.c` program in the Linux 0.1x kernel. All sequences represent the sequence of control functions sent by the host, unless otherwise stated.

## 10.4 serial.c

### 10.4.1 Function

The program `serial.c` implements the system serial port initialization and is ready to use the serial terminal device. The default serial communication parameters are set in the `rs_init()` initialization function, and the interrupt trap gate (interrupt vector) of the serial port is also set. The `rs_write()` function is used to send the characters in the serial terminal device write buffer queue to the remote terminal device through the serial line.

The function `rs_write()` will be called when the character device file is used in the file system. When a program writes to a serial device `/dev/tty64` file, the system-call `sys_write()` is executed (in `fs/read_write.c`). When the system call determines that the file being read is a character device file, the `rw_char()` function (in `fs/char_dev.c`) is called. This function will call `rw_tty()` using the character device read/write function table (device switch table) according to the information such as the sub-device number of the device being read. This will eventually call the serial terminal write function `rs_write()` here.

The `rs_write()` function actually turns on the serial transmit hold register empty interrupt flag, allowing interrupt signals to be sent after the UART sends the data out. The specific send operation is done in the `rs_io.s` program.

### 10.4.2 Code annotation

Program 10-3 linux/kernel/chr\_drv/serial.c

---

```
1 /*  
2  * linux/kernel/serial.c  
3  *
```



```

4  * (C) 1991 Linus Torvalds
5  */
6
7  /*
8  *      serial.c
9  *
10 * This module implements the rs232 io functions
11 *      void rs_write(struct tty_struct * queue);
12 *      void rs_init(void);
13 * and all interrupts pertaining to serial IO.
14 */
15
16 // <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
17 //      communication.
18 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
19 //      data of the initial task 0, and some embedded assembly function macro statements
20 //      about the descriptor parameter settings and acquisition.
21 // <asm/system.h> System header file. An embedded assembly macro that defines or
22 //      modifies descriptors/interrupt gates, etc. is defined.
23 // <asm/io.h> Io header file. Defines the function that operates on the io port in the
24 //      form of a macro's embedded assembler.
25 #include <linux/tty.h>
26 #include <linux/sched.h>
27 #include <asm/system.h>
28 #include <asm/io.h>
29
30 // Begin sending when the write queue contains WAKEUP_CHARS characters.
31 #define WAKEUP_CHARS (TTY_BUF_SIZE/4)
32
33 extern void rs1_interrupt(void);      // serial 1 interrupt handler (rs_io.s, 34)
34 extern void rs2_interrupt(void);      // serial 2 interrupt handler (rs_io.s, 38)
35
36 // Initialize the serial port
37 // Sets the transmit baud rate (2400 bps) for the specified serial port and allows all interrupt
38 // sources except the write hold register empty. In addition, when outputting a 2-byte baud
39 // rate factor, the DLAB bit (bit 7) of the line control register must first be set.
40 // Parameters: port is the serial port base address, port 1 - 0x3F8; port 2 - 0x2F8.
41 static void init(int port)
42 {
43     outb_p(0x80, port+3);      /* set DLAB of line control reg */
44     outb_p(0x30, port);      /* LS of divisor (48 -> 2400 bps */
45     outb_p(0x00, port+1);      /* MS of divisor */
46     outb_p(0x03, port+3);      /* reset DLAB */
47     outb_p(0x0b, port+4);      /* set DTR, RTS, OUT_2 */
48     outb_p(0x0d, port+1);      /* enable all intrs but writes */
49     (void) inb(port);          /* read data port to reset things (?) */
50 }
51
52 // Initialize the serial interrupt routine and the serial interface.
53 // The gate descriptor setting macro set_intr_gate() in the interrupt descriptor table IDT is
54 // implemented in include/asm/system.h.
55 void rs_init(void)
56 {

```

```
// The following two sentences are used to set the interrupt gate descriptors of the two serial
// ports. rs1_interrupt is the interrupt handler pointer for serial port 1. The interrupt used
// by serial port 1 is int 0x24, and the port 2 is int 0x23. See Table 5-2 and system.h file.
39     set_intr_gate(0x24, rs1_interrupt); // set int gate descriptor of port 1 (IRQ4).
40     set_intr_gate(0x23, rs2_interrupt); // set int gate descriptor of port 2 (IRQ3).
41     init(tty_table[64].read_q->data); // Initialize port 1 (.data is base address).
42     init(tty_table[65].read_q->data); // Initialize port 2
43     outb(inb_p(0x21)&0xE7, 0x21); // enable 8259A to respond to IRQ3, IRQ4.
44 }
45
46 /*
47  * This routine gets called when tty_write has put something into
48  * the write_queue. It must check whether the queue is empty, and
49  * set the interrupt register accordingly
50  *
51  * void _rs_write(struct tty_struct * tty);
52  */
///// Serial data send write function.
// This function actually only turns on the transmit hold register empty interrupt flag.
// Thereafter, when the transmit holding register is empty, the UART generates an interrupt
// request. In the serial interrupt handler, the program fetches the character at the end of
// the write queue and outputs it to the transmit holding register. Once the UART sends the
// character out, the transmit holding register will be empty and this causes an interrupt request
// again. So as long as there are characters in the write queue, the system repeats the process
// and sends the characters one by one. When all the characters in the write queue are sent
// out, the write queue becomes empty, and the interrupt handler resets the transmit hold register
// interrupt enable flag in the interrupt enable register, thereby again disabling the transmit
// hold register empty to cause interrupt request. At this point, the "loop" send operation
// ends.
53 void rs_write(struct tty_struct * tty)
54 {
// If the write queue is not empty, first read the interrupt enable register contents from 0x3f9
// (or 0x2f9), add the transmit hold register interrupt enable flag (bit 1), and then write
// back to the register. Thus, the UART can initiate an interrupt when it is desired to obtain
// the character to be transmitted when the transmit holding register is empty. Just note that
// the write_q.data here is the serial port base address.
55     cli();
56     if (!EMPTY(tty->write_q))
57         outb(inb_p(tty->write_q->data+1) | 0x02, tty->write_q->data+1);
58     sti();
59 }
60
```

---

## 10.4.3 Information

### 10.4.3.1 Universal Asynchronous Serial Communication

The universal asynchronous serial communication transmission method is the most common traditional serial communication method in the industry, and is still widely used in various embedded systems. It uses a dedicated Universal Asynchronous Receive and Transmit (UART) controller chip to implement data transfer. The format of the communication frame used is shown in Figure 10-10. Transferring a character consists of a start bit, data bits, a parity bit, and 1 or 2 stop bits. The start bit plays a synchronous role and the value is always

zero. The data bits are the actual data transmitted, that is, the code of one character. Its length can be 5-8 bits. The parity bit is optional and can be set by the program. The stop bit is always 1, and can be set to 1, 1.5 or 2 bits by the program. Before the communication starts to send information, both parties must be set to the same format and use the same transmission rate. For example, it is necessary to have the same number of data bits and stop bits. 在 In the asynchronous communication specification, the transmission 1 is called a MARK and the transmission 0 is called a SPACE. Therefore we also use these two terms in the following description.

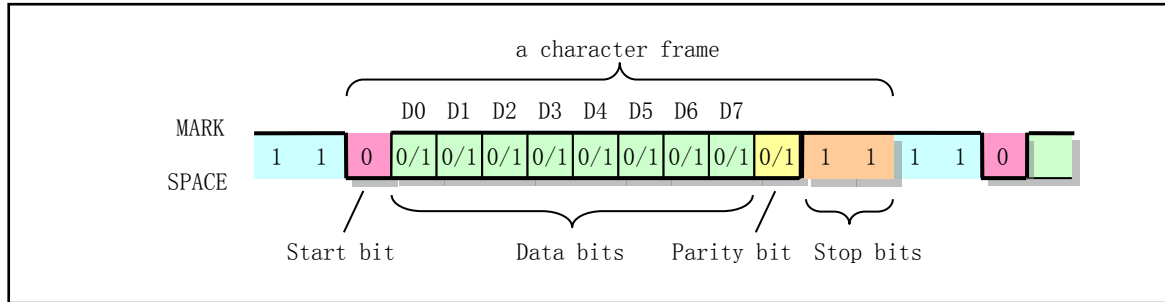


Figure 10-10 Asynchronous serial transmission frame format

When there is no data transmission, the sender is in the MARK state and continues to transmit 1. If data needs to be sent, the sender needs to first send a start bit SPACE of the bit interval. After receiving the SPACE bit, the receiver starts to synchronize with the sender and then receives the subsequent data. If the parity bit is set in the program, the parity bit needs to be received after the data is transmitted. Finally, the stop bit. After the character frame is sent, the next character frame can be sent immediately, or the MARK can be sent temporarily, and then the character frame is sent again.

When receiving a character frame, the receiver may detect one of three types of errors: (1) Parity error. At this point the program should ask the other party to resend the character; (2) Overspeed error. This error occurs because the program fetches characters slower than the receiving speed. At this point, you should modify the program to speed up the character frequency; (3) The frame format is incorrect. This error can occur when the format information requested to be received is incorrect. For example, a SPACE bit was received when a stop bit should be received. In general, in addition to line interference, it is likely that the frame format setting of the communication parties is different.

#### 1. Serial communication interface and UART structure

In order to realize serial communication, the PC usually has two serial interfaces conforming to the RS-232C standard, and uses a universal asynchronous receiver/transmitter (UART) control chip to process the serial data transmission and reception. The serial interface on the PC usually uses a 25-pin DB-25 or a 9-pin DB-9 connector, which is mainly used to connect the MODEM device for operation. Therefore, the RS-232C standard specifies many MODEM-specific interface pins. Please refer to other materials for a detailed description of the RS-232C standard and the working principle of the MODEM device. Here we mainly explain the structure of the UART control chip and prepare it for programming.

Previous PCs used National Semiconductor's NS8250 or NS16450 UART chips. Today's PCs use the 16650A or its compatible chips, but they are all compatible with the NS8250/16450 chip. The main difference between these chips is that the 16650A chip additionally supports FIFO transmission. In this way, the UART can trigger an interrupt after receiving or transmitting up to 16 characters, thus reducing the burden on the system and CPU. However, since the Linux 0.12 we discussed only uses the attributes of NS8250/16450, the FIFO mode is not further explained here.

The UART asynchronous serial port hardware logic used in the PC is shown in Figure 10-11. It can be divided into 3 parts. The first part mainly includes data bus buffer D7 -- D0, internal register select pin A0 -- A2, CPU read / write data select pin DISTR and DOSTR, chip reset pin MR, interrupt request output pin INTRPT and user Pin OUT2 defined to disable/allow interrupts. When OUT2 is 1, the UART can be disabled from issuing an interrupt request signal.

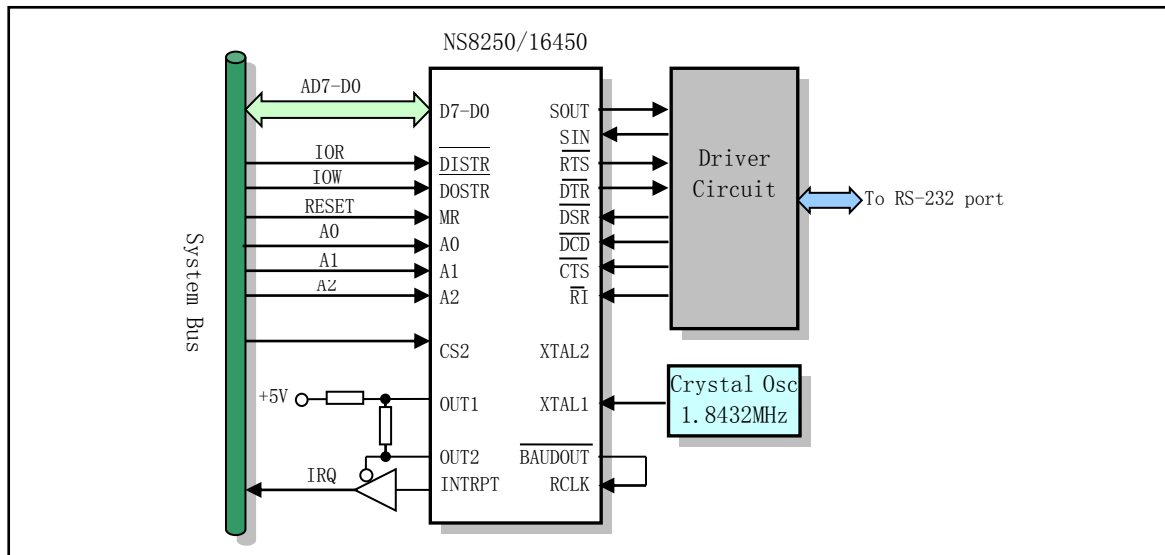


Figure 10-11 NS8250/16450 basic hardware structure diagram

The second part mainly includes the pin part between the UART and the RS-232 interface. These pins are primarily used to receive/transmit serial data and to generate or receive MODEM control signals. The serial output data (SOUT) pin sends a bit stream to the line; the input data (SIN) pin receives the bit stream from the line; the Data Device Ready (DSR) pin is used by the communication device (MODEM) to inform the UART that it has Ready to start receiving data; The request to send (RTS) pin is used to notify the MODEM, the computer requires switching to the sending mode; the clear sending (CTS) is the MODEM telling the computer that it has switched to the ready to receive mode; The Device Carrier Detect (DCD) pin is used to receive MODEM information, which tells the UART that the carrier signal has been received; the Ring Indicator (RI) pin is also used by the MODEM to tell the computer that the communication line is already connected.

The third part is the UART chip clock input circuit part. The UART's operating clock can be generated by connecting a crystal oscillator between pins XTAL1 and XTAL2, or directly from the outside via XTAL1. The PC uses the latter method to directly input the 1.8432MHz clock signal on the XTAL1 pin. The 16 times signal of the UART transmit baud rate is output by the pin BAUDOUT, and the pin RCLK is the baud rate of the received data. Since the two are connected together, the baud rate for sending and receiving data on the PC is the same.

Like the interrupt controller chip 8259A, the UART is also a programmable control chip. By setting its internal registers, we can set the operating parameters of the serial communication and how the UART works. The internal block diagram of the UART is shown in Figure 10-12.

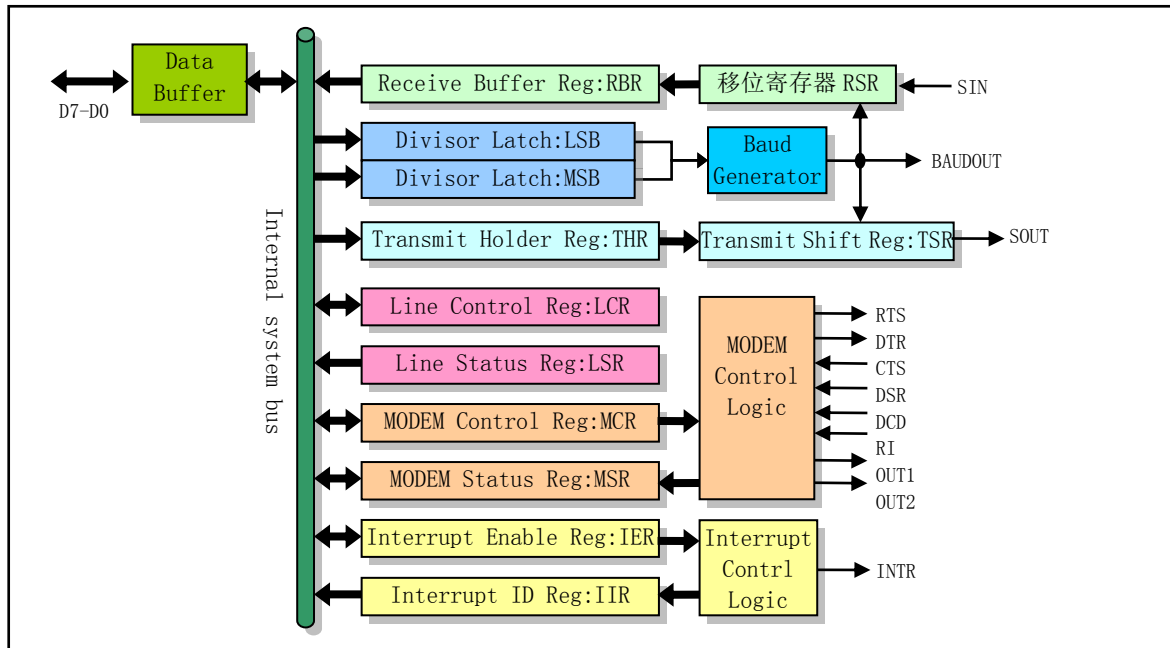


Figure 10-12 Basic block diagram of NS8250 UART internal components

For the NS8250 chip, there are 10 registers that the CPU can access, but the address line A2--A0 used to select these registers can only select up to 8 registers. Therefore, the NS8250 takes a bit (bit 7) in the line control register (LCR) to select the two divisor latch registers LSB and MSB. Bit 7 is referred to as the Divisor Latch Access Bit (DLAB). The purpose of these registers and the access port address are shown in Table 10-8.

Table 10-8 UART internal register ports and purposes

Port address	R/W	condition	Usage
0x3f8 (0x2f8)	W	DLAB=0	Writes transmit holding register, THR, contains char that will be sent.
	R	DLAB=0	Read the receive buffer register RBR. Contains character received.
	R/W	DLAB=1	Read/write baud rate factor low byte (LSB).
0x3f9 (0x2f9)	R/W	DLAB=1	Read/write baud rate factor high byte (MSB).
	R/W	DLAB=0	Read/write interrupt enable register IER. Bits 7-4 all 0, reserved; Bit 3=1 modem status interrupt allowed; Bit 2=1 Receiver line status interrupt enabled; Bit 1=1 Transmit Holding Register Empty Interrupt allowed; Bit 0=1 has received data interrupt enable.
0x3fa (0x2fa)	R		Read interrupt identification register IIR. The interrupt handler is used to determine which of the four types is interrupted. Bits 7-3 all 0 (not used); Bit 2-1 determines the priority of the interrupt; = 11 The receiving status has error and the priority is highest. Read the line status to reset it; = 10 Data has been received, priority 2. Read data can reset it; = 01 Transmit holding register empty, priority 3. Write THR to reset it; = 00 MODEM status changes, priority 4. Read MODEM state to reset it.

			Bit 0 = 0 pending interrupt; =1 no interrupt.
0x3fb (0x2fb)	W		<p>Write the line control register LCR.</p> <p>Bit 7 = 1 Divisor Latch Access Bit (DLAB).</p> <p>= 0 Receiver, transmit hold or interrupt enable register access;</p> <p>Bit 6=1 allows for a break;</p> <p>Bit 5=1 holds the parity bit;</p> <p>Bit 4=1 even parity; =0 odd parity;</p> <p>Bit 3=1 allows parity; =0 no parity;</p> <p>Bit 2=1 now depends on the data bit length. If the data bit length is 5 bits, the stop bit is 1.5 bits; if the data bit length is 6, 7 or 8 bits, the stop bit is 2 bits; = 0 stop bit is 1 bit;</p> <p>Bit 1-0 Data Bit Length:</p> <p>= 00 5 data bits;</p> <p>= 01 6-bit data bits;</p> <p>= 10 7-bit data bits;</p> <p>= 11 8-bit data bits.</p>
0x3fc (0x2fc)	W		<p>Write MODEM control register MCR.</p> <p>Bits 7-5 all 0, reserved;</p> <p>Bit 4=1 chip is in cyclic feedback diagnostic mode of operation;</p> <p>Bit 3=1, auxiliary user specifies output 2, enable INTRIPRT to system;</p> <p>Bit 2=1, auxiliary user specifies output 1, and the PC is not used;</p> <p>Bit 1=1 makes the request to send RTS valid;</p> <p>Bit 0 = 1 makes the data terminal ready DTR active.</p>
0x3fd (0x2fd)	R		<p>Read the line status register LSR.</p> <p>Bit 7=0 reserved;</p> <p>Bit 6=1 The transmit shift register is empty;</p> <p>Bit 5=1 Transmit holding register is empty and can get character to send;</p> <p>Bit 4=1 receives a sequence of bits that satisfy the break condition;</p> <p>Bit 3=1 frame format error;</p> <p>Bit 2=1 parity error;</p> <p>Bit 1=1 exceeds the overlay error;</p> <p>Bit 0=1 Receiver data is ready and the system is readable.</p>
0x3fe (0x2fe)	R		<p>Read the MODEM status register MSR. <math>\delta</math> indicates a change in the signal or condition.</p> <p>Bit 7=1 carrier detect (CD) is valid;</p> <p>Bit 6=1 ring indication (RI) is valid;</p> <p>Bit 5=1 Data Device Ready (DSR) is valid;</p> <p>Bit 4=1 Clear Transmit (CTS) is valid;</p> <p>Bit 3=1 detects the <math>\delta</math> carrier;</p> <p>Bit 2=1 detects the edge of the ring signal;</p> <p>Bit 1 = 1 <math>\delta</math> Data Device Ready (DSR);</p> <p>Bit 0 = 1 <math>\delta</math> Clear Transmit (CTS).</p>

## 2. UART initialization programming method

When the PC is powered on, the system RESET signal resets the UART internal registers and control logic through the MR pin of the NS8250 chip. After that, if you want to use the UART, you need to initialize it to set the working baud rate, number of data bits and working mode of the UART. Below we take the serial port 1 on the PC as an example to illustrate the steps to initialize it. The port base address of the serial port is port = 0x3f8, and the UART chip interrupt pin INTRPT is connected to the interrupt control chip pin IRQ4. Of course, the interrupt descriptor entry of the serial interrupt handler should be set first in the IDT table before initialization.

a) Set the transmission baud rate.

Setting the communication transmission baud rate is to set the values of the two divisor latch registers LSB and MSB, that is, the 16-bit baud rate factor. As can be seen from Table 10-8, to access the two divisor latch registers, we must first set bit 7 DLAB=1 of the line control register LCR, that is, write 0x80 to port+3 (0x3fb). Then we perform output operations on port (0x3f8) and port+1 (0x3f9), and we can write the baud rate factor into LSB and MSB respectively. For a specified baud rate (eg 2400 bps), the baud rate factor is calculated as:

$$\text{Baudrate factor} = \frac{\text{UART Freq.}}{\text{Baudrate} \times 16} = \frac{1.8432 \text{ MHz}}{2400 \times 16} = \frac{1843200}{2400 \times 16} = 48$$

So to set the baud rate to 2400pbs, we need to write 48 (0x30) in the LSB and 0 in the MSB. After the baud rate is set, we also need to reset the DLAB bit of the line control register.

b) Set the transfer format.

The serial communication transmission format is defined by the bits in the line control register LCR. The meaning of each of them is shown in Table 10-8. If we need to set the transmission format to no parity, 8 data bits and 1 stop bit, then we need to output the value 0x03 to the LCR. The lowest 2 bits of the LCR indicate the data bit length, and when it is 0b11, the data length is 8 bits.

c) Set the MODEM control register.

Writing to this register sets the mode of operation of the UART and controls the MODEM. There are two modes of UART operation: interrupt mode and query mode. There is also a loop feedback method, but this method is only used to diagnose and test the quality of the UART chip, and cannot be used as an actual communication method. The query method is used in the PC ROM BIOS, but the Linux system discussed in this book uses an efficient interrupt method. Therefore, we will only introduce the operation programming method of the UART in the interrupt mode.

Setting bit 4 of the MCR allows the UART to be in the loop feedback diagnostic mode of operation. In this mode, the UART chip automatically "snap" the input (SIN) and output (SOUT) pins. Therefore, if the data sequence sent at this time is equal to the received sequence, then the UART chip is working properly.

The interrupt mode means that the UART is allowed to send an interrupt request signal to the CPU through the INTRPT pin when the MODEM status changes, or when an error occurs, or when the transmit holding register is empty, or when a character is received. As for allowing interrupt requests to be issued under those conditions, it is determined by the interrupt enable register IER. However, if the UART interrupt request signal can be sent to the 8259A interrupt controller, bit 3 (OUT2) of the MODEM control register MCR needs to be set. Because in the PC, this bit controls the INTRPT pin to the 8259A circuit, see Figure 10-11.

The query mode means that the program receives/transmits the serial data by cyclically polling the contents of the UART register under the condition that the MODEM control register MCR bit 3 (OUT2) is

reset. When the MCR bit 3 = 0, the UART can still generate an interrupt request signal on the INTRPT pin under the condition that the MODEM state changes, and the interrupt flag register IIR can be set according to the condition that the interrupt is generated, but the interrupt request signal cannot be sent to 8259A. Therefore, the program can only judge the current working state of the UART by querying the contents of the line status register LSR and the interrupt identification register IIR, and perform data receiving and transmitting operations.

Bits 1 and 0 of MCR are used to control the MODEM. When these two bits are set, the data terminal ready DTR pin of the UART and the request to send RTS pin output are valid. To set the UART to interrupt mode and make DTR and RTS valid, then we need to write 0x0b, the binary number 01011, to the MODEM control register.

d) Initialize the interrupt enable register IER

The interrupt enable register IER is used to set the condition that can generate an interrupt, that is, the interrupt source type. There are four types of interrupt sources to choose from, as shown in Table 10-8. If the corresponding bit is 1, it means that the condition is allowed to generate an interrupt, otherwise it is disabled. When an interrupt source type generates an interrupt, the interrupt generated by which interrupt source is specified by bit 2 - bit 1 in the interrupt flag register IIR, and the contents of the specific register can be read and written to reset the UART interrupt. Bit 0 of the IER is used to determine if there is an interrupt currently, and bit 0 = 0 indicates that there is an interrupt to be processed.

In the serial port initialization function of Linux 0.12, the setting allows three types of interrupt sources to generate interrupts (write 0x0d), that is, when the MODEM status changes, when the receiving error occurs, the interrupt is allowed to be generated when the receiver receives the character. However, it is not allowed the transmit hold register empty to generate an interrupt because we have no data to send at this time. When the write queue of the corresponding serial terminal has data to be sent out, the `tty_write()` function calls the `rs_write()` function to set the transmit hold register empty enable interrupt flag, thereby during the serial interrupt processing initiated by the interrupt source, the kernel program can start to fetch the character in the write queue and send the output to the transmit holding register for the UART to send out. Once the UART sends the character out, the transmit holding register will empty and cause an interrupt request again. So as long as there are characters in the write queue, the system repeats the process and sends the characters one by one. When all the characters in the write queue are sent out, the write queue becomes empty, and the interrupt handler resets the transmit hold register interrupt enable flag in the interrupt enable register, thereby again disabling the transmit hold register empty to cause an interrupt request. This "loop" send operation also ends.

### 3. UART interrupt handler programming method

In the Linux kernel, serial terminals use read/write queues to receive and transmit terminal data. The data received from the serial port is placed in the read queue header for the `tty_io.c` program to read; and the data that needs to be sent to the serial terminal is placed at the write queue pointer. Therefore, the main task of the serial interrupt handler is to put the character in the receive buffer register RBR received by the UART to the pointer at the end of the read queue; the character taken from the pointer at the end of the write queue is placed in the transmit hold register THR of the UART and sent out. At the same time, the serial interrupt handler also needs to handle some other error conditions.

As can be seen from the above description, the UART can generate interrupts with four different interrupt source types. Therefore, when the serial interrupt handler first starts executing, it is only known that an interrupt has occurred, but it is not known which case caused the interrupt. So the first task of the serial interrupt handler is to determine the specific conditions under which the interrupt will be generated. This requires the use of the



interrupt flag register IIR to determine the source type from which the current interrupt was generated. Therefore, the serial interrupt handler can be processed separately according to the source type that generates the interrupt using the subroutine address jump table `jmp_table[]`. The block diagram is shown in Figure 10-13. The structure of the `rs_io.s` program is basically the same as this block diagram.

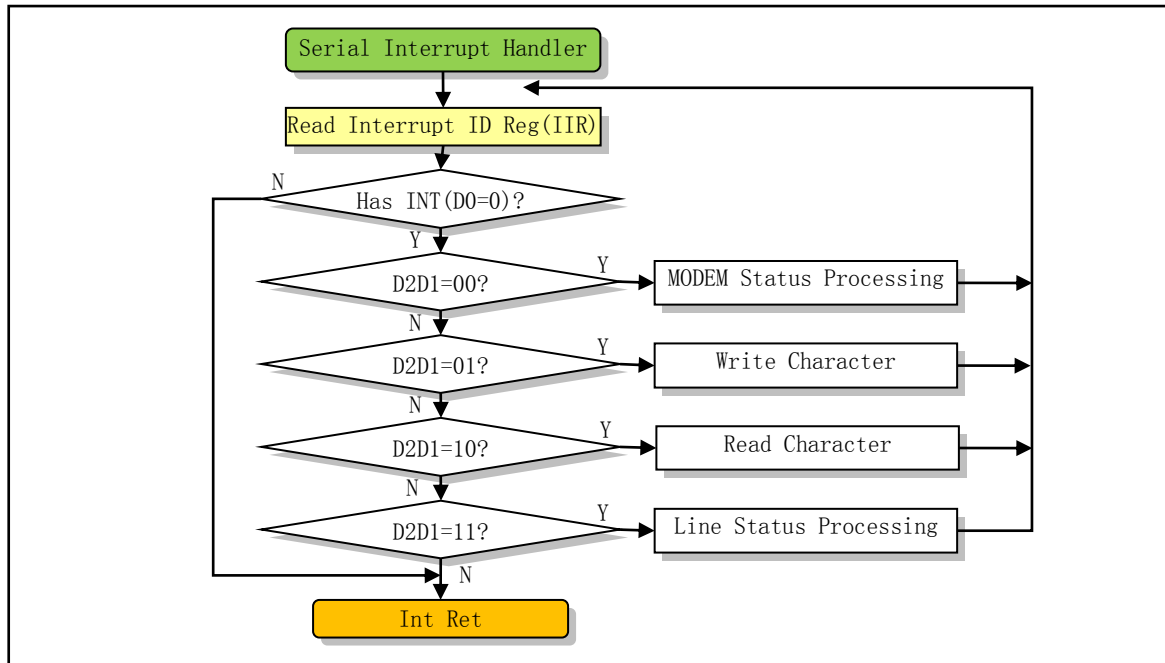


Figure 10-13 Serial communication interrupt processing block diagram

After the content of the IIR is taken out, it is necessary to first judge whether there is an interrupt to be processed according to bit 0. If bit 0 = 0, there is an interrupt that needs to be processed. Then, according to bit 2 and bit 1, the corresponding interrupt source type processing subroutine is called by using the pointer jump table. In each subroutine, the corresponding interrupt source of the UART is reset after processing. After the subroutine returns, this code loops to determine if there are other interrupt sources (bit 0 = 0?). If there are other interrupt sources for this interrupt, bit 0 of IIR is still 0, so the interrupt handler will call the corresponding interrupt source subroutine to continue processing. All interrupt sources that caused this interrupt are processed and reset. At this point, the UART will automatically set the IIR bit 0 = 1, indicating that there is no pending interrupt, so the interrupt handler can exit.

## 10.5 rs\_io.s

### 10.5.1 Function

The `rs_io.s` assembly file implements the rs232 serial communication interrupt processing. During the transmission and storage of characters, the interrupt process mainly operates on the read and write buffer queues of the terminal. It stores the characters received from the serial line into the read buffer queue `read_q` of the serial terminal, or sends out the characters in the write buffer queue `write_q` to the remote serial terminal device through the serial line.

There are four types of situations that cause the system to generate a serial interrupt: a. because the

MODEM status has changed; b. because the line status has changed; c. because the character was received; d. because the transmit and hold register empty interrupt enable flag is set, there are characters to send. The first two cases of causing an interrupt are reset by reading the corresponding status register value. For the case of receiving a character, the program first puts the character into the read buffer queue read\_q, and then calls the copy\_to\_cooked() function to convert it into the canonical mode character in the character line unit into the auxiliary queue secondary. For the case where a character needs to be sent, the program first takes out a character from the write buffer queue write\_q and sends it out, and then judges whether the write buffer queue is empty, and if there are still characters, it performs a transmission operation cyclically.

Therefore, before reading this program, it is best to look at the include/linux/tty.h header file. It gives the data structure tty\_queue of the character buffer queue, the data structure tty\_struct of the terminal and the values of some control characters. There are also macro definitions that operate on the buffer queue. The buffer queue and its operation diagram are shown in Figure 10-14.

## 10.5.2 Code annotation

Program 10-4 linux/kernel/chr\_drv/rs\_io.s

```

1 /*
2  *  linux/kernel/rs_io.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 /*
8  *      rs_io.s
9  *
10 * This module implements the rs232 io interrupts.
11 */
12
13 .text
14 .globl _rs1_interrupt, _rs2_interrupt
15
16 // size is the length in bytes of the read and write queue buff
17 size = 1024                                /* must be power of two !
18                                           and must match the value
19                                           in tty_io.c!!! */
20
21 /* these are the offsets into the read/write buffer structures */
22 // Corresponds to the offset of each field in the tty_queue structure in the include/linux/tty.h
23 // file, where rs_addr corresponds to the data field of the tty_queue structure. But for serial
24 // terminal buffer queues, this field holds the serial port base address (0x3f8 or 0x2f8).
25 rs_addr = 0                                // field offset of the serial ports (0x3f8 or 0x2f8).
26 head = 4                                    // the head pointer field offset in the buffer.
27 tail = 8                                    // the tail pointer field offset in the buffer.
28 proc_list = 12                             // wait for the buffered process field offset.
29 buf = 16                                    // buffer field offset.
30
31 // When a write buffer queue is full, the kernel puts the process (which fills the queue) into
32 // a wait state. When there are no more than 256 characters remaining in the write buffer queue,
33 // the interrupt handler can wake up the waiting process and continue to put characters into
34 // the write queue.

```

```

27 startup = 256                                /* chars left in write queue when we restart it */
28
29 /*
30  * These are the actual interrupt routines. They look where
31  * the interrupt is coming from, and take appropriate action.
32  */
33 // Serial port 1 interrupt handler entry point.
34 // At initialization, the rsl_interrupt address is placed in the interrupt descriptor 0x24,
35 // which corresponds to the 8259A interrupt request IRQ4 pin. First, the address of the serial
36 // terminal 1 (serial port 1) read/write buffer queue pointer in the tty table is first pushed
37 // onto the stack (tty_io.c, 81), and then jumped to rs_int to continue processing. This will
38 // allow the processing codes of serial port 1 and serial port 2 to be shared. The character
39 // buffer queue structure tty_queue can be found in include/linux/tty.h, line 22.
40 .align 2
41 _rsl_interrupt:
42     pushl $_table_list+8    // serial port 1 r/w queues pointer is pushed onto stack.
43     jmp rs_int
44 .align 2
45 // Serial port 2 interrupt handler entry point.
46 _rs2_interrupt:
47     pushl $_table_list+16   // serial port 2 r/w queues pointer is pushed onto stack.
48
49 // This code first makes the segment registers ds, es point to the kernel data segment, and
50 // then takes the serial port base address from the data field of the corresponding read/write
51 // buffer queue. This address plus 2 is the port address of the interrupt identification register
52 // IIR. If its bit 0 = 0, it indicates that there is an interrupt to be processed. Then, according
53 // to bit 2 and bit 1, the corresponding interrupt source type processing subroutine is called
54 // by using the pointer jump table. Each subroutine will resets the corresponding interrupt
55 // source of the UART after processing.
56 // After the subroutine returns, this code loops to determine if there are other interrupt sources
57 // (bit 0 = 0?). If there are other interrupt sources for this interrupt, bit 0 of the IIR is
58 // still 0. The interrupt handler then calls the corresponding interrupt source subroutine again
59 // to continue processing. Until all interrupt sources causing this interrupt are processed
60 // and reset, the UART will automatically set the IIR bit 0 =1, indicating that there is no
61 // pending interrupt, so the interrupt handler can exit.
62 rs_int:
63     pushl %edx
64     pushl %ecx
65     pushl %ebx
66     pushl %eax
67     push %es
68     push %ds                /* as this is an interrupt, we cannot */
69     pushl $0x10             /* know that bs is ok. Load it */
70     pop %ds                 // Let ds, es point to the kernel data segment.
71     pushl $0x10
72     pop %es
73     movl 24(%esp),%edx       // get serial buffer queue address.
74     movl (%edx),%edx         // get read queue struct address -> edx.
75     movl rs_addr(%edx),%edx  // get serial port 1 (or port 2) base address -> edx.
76     addl $2,%edx            /* interrupt ident. reg */
77                             // The IIR port address is 0x3fa (0x2fa).
78
79 // Get the content of IIR (interrupt identification byte) to determine the source of the
80 // interrupt. There are 4 types of interrupt conditions determined by bit 2 and bit1: MODEM

```

```

// status changes; characters to be written (sent); characters to be read (received); line status
// changes. The code first determine if there is any interrupt to be processed, then processes
// the corresponding interrupt accordingly.
55 rep_int:
56     xorl %eax,%eax
57     inb %dx,%al          // get interrupt identification byte.
58     testb $1,%al        // bit0 = 0 ? (any interrupts ?)
59     jne end              // no, jump to end.
60     cmpb $6,%al         /* this shouldn't happen, but ... */
61     ja end               // if al > 6, jump to end.
62     movl 24(%esp),%ecx   // get serial buffer queue address -> ecx.
63     pushl %edx           // save the IIR port address temperoly.
64     subl $2,%edx        // edx restore to port base address 0x3f8 (0x2f8).
65     call jmp_table(,%eax,2) /* NOTE! not *4, bit0 is 0 already */
// The above statement means that when there is an interrupt to be processed, bit 0=0 in the
// AL, bit 2, bit 1 is the interrupt type, so it is equivalent to multiplying the interrupt
// type by 2. And because the address has 4 bytes, here multiply by 2, you can get the jump
// table (line 79) corresponding to each interrupt type address, and then jump there to do the
// corresponding processing.
// Allowing the transmission of character interrupts is accomplished by setting the transmit
// holding register flag. In the serial.c program, when there is data in the write queue, the
// rs_write() function modifies the contents of the interrupt enable register and adds the
// transmit hold register interrupt enable flag, causing a serial interrupt to occur when the
// system needs to send a character.
66     popl %edx           // restore IIR port address 0x3fa (or 0x2fa).
67     jmp rep_int         // jump to loop processing any pending interrupts.

68 end:    movb $0x20,%al   // send EOI instruction to interrupt controller.
69     outb %al,$0x20       /* EOI */
70     pop %ds
71     pop %es
72     popl %eax
73     popl %ebx
74     popl %ecx
75     popl %edx
76     addl $4,%esp        # jump over _table_list entry    # discard the queue address.
77     iret
78
// The address jump table of each interrupt type processing subroutine. There are 4 types of
// interrupt conditions: MODEM status changes; characters to be written (sent); characters to
// be read (received); line status changes.
79 jmp_table:
80     .long modem_status,write_char,read_char,line_status
81
// This interrupt is caused by a change in the MODEM state. A reset operation is performed on
// the MODEM status register MSR by reading it.
82 .align 2
83 modem_status:
84     addl $6,%edx         /* clear intr by reading modem status reg */
85     inb %dx,%al          // MODEM status register port: 0x3fe
86     ret
87
// This serial interrupt is caused by a change in the line state. A reset operation is performed

```

```

// on the line status register LSR by reading it.
88 .align 2
89 line_status:
90     addl $5,%edx          /* clear intr by reading line status reg. */
91     inb %dx,%al          // LSR port: 0x3fd
92     ret
93
// A processing subroutine for interrupts caused by the UART receiving a character. A read from
// the receive buffer register can reset this interrupt source. This subroutine puts the received
// character into the head of the read buffer queue read_q and moves the pointer forward by
// one character position. If the head pointer has reached the end of the buffer, let it fold
// back to the beginning of the buffer. Finally, the C function do_tty_interrupt() (that is,
// copy_to_cooked()) is called, and the read characters are processed into a canonical mode
// buffer queue (secondary buffer).
// To obtain the current serial port no, let the current serial port queue address minus the
// queue table address table_list, and then divided by 8. the result 1 is for serial 1, 2 for
// serial 2.
94 .align 2
95 read_char:
96     inb %dx,%al          // read char in receive buffer register RBR into AL.
97     movl %ecx,%edx       // serial buffer queue address -> edx
98     subl $_table_list,%edx // current serial no = (queue address - table_list) / 8
99     shr $3,%edx
100    movl (%ecx),%ecx      # read-queue
101    movl head(%ecx),%ebx  // get the head pointer in the read queue -> ebx.
102    movb %al,buf(%ecx,%ebx) // put the char at the head position.
103    incl %ebx            // and move the head pointer forward by one.
104    andl $size-1,%ebx    // modulate the head pointer by the buffer length.
105    cmpl tail(%ecx),%ebx  // compare the head with the tail pointer.
106    je 1f               // queue is full if equal, then jump forward.
107    movl %ebx,head(%ecx)  // otherwise save the new head pointer.
108 1:    addl $63,%edx      // convert serial no to tty no (63 or 64) & push into stack.
109    pushl %edx
110    call _do_tty_interrupt // Call tty int handler C function (tty_io.c, line 397).
111    addl $4,%esp         // discard paras and ret.
112    ret
113
// The subroutine of sending characters. This interrupt is caused by setting the transmit holding
// register enable interrupt flag. Indicates that there are characters in the write queue
// corresponding to the serial terminal that need to be sent. Then the number of characters
// currently contained in the write queue is calculated, and if it is less than 256, the process
// waiting for the write operation is woken up. Then take a character from the end of the write
// buffer queue and adjust and save the tail pointer. If the write buffer queue is empty, jump
// to label write_buffer_empty to handle the case.
114 .align 2
115 write_char:
116     movl 4(%ecx),%ecx     # write-queue // address -> ecx
117     movl head(%ecx),%ebx  // get head pointer of write queue to ebx
118     subl tail(%ecx),%ebx  // number of chars = head - tail.
119     andl $size-1,%ebx    # nr chars in queue
120     je write_buffer_empty // if head == tail, the queue is empty, then jump.
121     cmpl $startup,%ebx    // check the number of chars in write queue.
122     ja 1f               // jump if more than 256 chars.

```

```

123     movl proc_list(%ecx),%ebx    # wake up sleeping process // get wait proc list.
124     testl %ebx,%ebx             # is there any?    # any process waiting to write ?
125     je 1f                      // none, then jump forward to label 1.
126     movl $0, (%ebx)            // otherwise wake process (change state to runnable).
127 1:     movl tail(%ecx),%ebx      // get a char from the tail position to AL.
128     movb buf(%ecx,%ebx),%al
129     outb %al,%dx               // output to THR register (port 0x3f8 or 0x2f8).
130     incl %ebx                  // move the tail forward by one.
131     andl $size-1,%ebx          // modulate and save the tail pointer.
132     movl %ebx,tail(%ecx)
133     cmpl head(%ecx),%ebx        // tail == head ?
134     je write_buffer_empty       // if true, it means queue is empty, then jump.
135     ret

// The following code handles the case where the write buffer queue write_q is empty. If there
// is a process waiting to write to the serial terminal, it wakes up, and then masks the transmit
// holding register empty interrupt, and disable the UART to initiate transmit holding register
// empty interrupt. If the write buffer queue write_q is empty at this time, it means that no
// characters need to be sent at present. So we should do the following two things. First look
// at whether there is a process waiting for the write queue to be vacant, and if so, wakes
// it up. In addition, because the system has no characters to send now, we need to temporarily
// disable the interrupts generated by transmission hold register THR empty. When a character
// is placed in the write queue again, the rs_write() function in serial.c will again allow
// the interrupt to be generated when the transmit holding register is empty, so the UART will
// "automatically" fetch the characters in the write queue, and send it out.
136 .align 2
137 write_buffer_empty:
138     movl proc_list(%ecx),%ebx    # wake up sleeping process
139     testl %ebx,%ebx             # is there any?
140     je 1f                      // no process waiting, jump forward to label 1.
141     movl $0, (%ebx)            // otherwise wake up the process.
142 1:     incl %edx               // read int enable register IER (0x3f9 or 0x2f9).
143     inb %dx,%al
144     jmp 1f                     // delay for a while.
145 1:     jmp 1f                 // masks transmit hold register empty int (bit 1).
146 1:     andb $0xd,%al          /* disable transmit interrupt */
147     outb %al,%dx
148     ret

```

## 10.6 tty\_io.c

### 10.6.1 Function

Each tty device has three buffer queues, read queue (read\_q), write queue (write\_q), and secondary queue (secondary), which are defined in the tty\_struct structure (include/linux/tty.h). For each buffer queue, the read operation takes the character from the left end of the buffer queue and moves the buffer tail pointer to the right, while the write operation adds characters to the right end of the buffer queue and also moves the head pointer to the right. If either of these two pointers moves beyond the end of the buffer queue, it will revert to the left and start again, as shown in Figure 10-14.

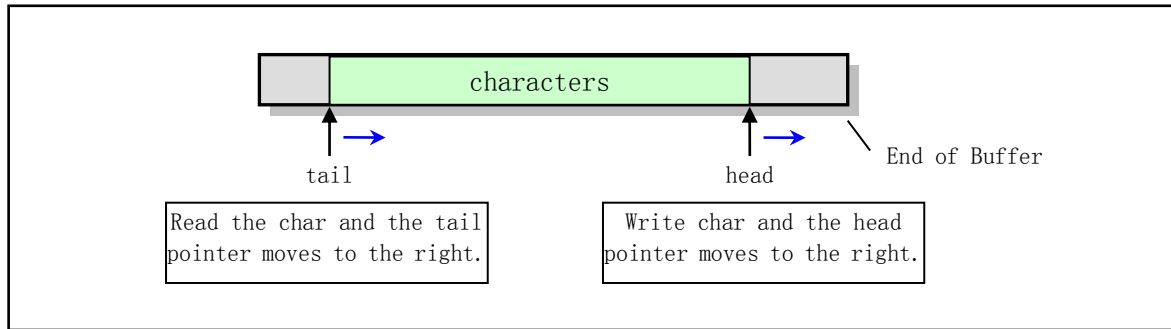


Figure 10-14 tty character buffer queue operation

The program `tty_io.c` includes the upper interface functions of the character device. It mainly contains the terminal read/write functions `tty_read()` and `tty_write()`, and the canonical mode function `copy_to_cooked()` of the read operation is also implemented here.

`tty_read()` and `tty_write()` will be called when the character device file is used in the file system. For example, when a program reads the `/dev/tty` file, it will execute the system-call `sys_read()` (in `fs/read_write.c`), and the system-call will call `rw_char()` function (in `fs/char_dev.c`) when it determines that the file being read is a character device file, the function will call `rw_tty()` from the character device read/write function table (device switch table) according to the sub-device number of the read device, etc., and finally call the terminal read operation function `tty_read()`.

The `copy_to_cooked()` function is invoked by the keyboard interrupt handler (via `do_tty_interrupt()`) to process the characters in the `read_q` queue according to the character input/output flags (such as `INLCR`, `OUCLC`) set in the terminal `termios` structure, and convert the characters into a sequence of canonical mode lines of characters and stored in the auxiliary queue (canonical mode queue) for reading by `tty_read()` above. During the conversion process, if the terminal's echo flag `L_ECHO` is set, the character is also placed in the write queue `write_q`, and the terminal write function is called to display the character on the screen. If it is a serial terminal, then the write function will be `rs_write()` (in `serial.c`, line 53). `rs_write()` will send the characters in the serial terminal write queue to the serial terminal over the serial line and display it on the screen of the remote serial terminal. The `copy_to_cooked()` function will also wake up the processes waiting for the auxiliary queue. The steps to implement the function are as follows:

1. If the read queue is empty or the auxiliary queue is full, skip to the last step (step 10), otherwise perform the following operations;
2. Get a character from the tail pointer of the read queue `read_q`, and move the tail pointer forward by one character position;
3. If it is a carriage return (CR) or line feed (NL) character, the character is converted according to the state of the input flag (`ICRNL`, `INLCR`, `INOCR`) in the terminal `termios` structure. For example, if you are reading a carriage return character and the `ICRNL` flag is set, replace it with a newline character;
4. If the uppercase to lowercase flag `IUCLC` is set, replace the character with the corresponding lowercase character;
5. If the canonical mode flag `ICANON` is set, the character is processed in canonical mode:
  - a. If it is a delete character (^U), delete a line of characters in the auxiliary queue secondary (the queue head pointer is backed up until a carriage return or line feed is encountered or the queue is empty);
  - b. If it is an erase character (^H), delete a character at the head pointer in the secondary, and the head pointer moves back by one character position;

- c. If it is a stop character (^S), set the stop flag of the terminal stopped=1;
- d. If it is the start character (^Q), reset the stop flag of the terminal.
6. If the receive keyboard signal flag ISIG is set, generate a signal corresponding to the typed control character for the process;
7. If it is a line ending character (such as NL or ^D), the line count statistics data of the secondary queue is increased by 1;
8. If the local echo flag is set, the character is also placed in the write queue write\_q, and the terminal write function is called to display the character on the screen;
9. Put the character in the auxiliary queue secondary, return to the above step 1 to continue to loop through the other characters in the read queue;
10. Finally wake up the processes that sleep on the secondary queue.

You can check the include/linux/tty.h header file when reading the following program. The header file defines the data structure of the tty character buffer queue and some macro operation definitions. It also defines the ASCII code value of the control character.

## 10.6.2 Code annotation

Program 10-5 linux/kernel/chr\_drv/tty\_io.c

```

1  /*
2   *  linux/kernel/tty_io.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  'tty_io.c' gives an orthogonal feeling to tty's, be they consoles
9   *  or rs-channels. It also implements echoing, cooked mode etc.
10  *
11  *  Kill-line thanks to John T Kohl, who also corrected VMIN = VTIME = 0.
12  */
13
14  // <ctype.h> The character type file. Defines some macros for character type conversion.
15  // <errno.h> Error number header file. Contains various error numbers in the system.
16  // <signal.h> Signal header file. Define signal symbol constants, signal structures, and
17  //   signal manipulation function prototypes.
18  // <unistd.h> Linux standard header file. Various symbol constants and types are defined
19  //   and various functions are declared. If __LIBRARY__ is defined, it also includes the
20  //   system call number and the inline assembly _syscall0().
21  #include <ctype.h>
22  #include <errno.h>
23  #include <signal.h>
24  #include <unistd.h>
25
26  // Give the corresponding bit mask of the alarm signal in the signal bitmap.
27  #define ALRMMASK (1<<(SIGALRM-1))
28
29  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
30  //   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.
// <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 operatio
21 #include <linux/sched.h>
22 #include <linux/tty.h>
23 #include <asm/segment.h>
24 #include <asm/system.h>
25
// Kill the process group function (send a signal to the process group). The parameter pgrp
// specifies the process group number; sig specifies the signal; priv is the priority.
// The main purpose of this function is to send the specified signal sig to each process in
// the specified 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 specified process group number pgrp,
// the error number -ESRCH is returned. If the process with the process group number is pgrp
// is found, but the send signal operation fails, the error code for fail sending is returned.
26 int kill_pg(int pgrp, int sig, int priv); // kernel/exit.c, line 171.
// Determine if a process group is an orphan process. Returns 0 if not; else returns 1.
27 int is_orphaned_pgrp(int pgrp); // kernel/exit.c, line 232.
28
// Get one of the three mode flag sets in the termios structure, or use to determine if a flag
// set contains a set flag.
29 #define L_FLAG(tty, f) ((tty)->termios.c_lflag & f) // local mode flags.
30 #define I_FLAG(tty, f) ((tty)->termios.c_iflag & f) // input mode flags.
31 #define O_FLAG(tty, f) ((tty)->termios.c_oflag & f) // output mode flags.
32
// Take a flag in the special (local) mode flag set of the termios structure.
33 #define L_CANON(tty) L_FLAG((tty), ICANON) // canonical mode flag.
34 #define L_ISIG(tty) L_FLAG((tty), ISIG) // signal (INTR, QUIT etc.) flag.
35 #define L_ECHO(tty) L_FLAG((tty), ECHO) // echo char flag.
36 #define L_ECHOE(tty) L_FLAG((tty), ECHOE) // echo erase flag in canon mode.
37 #define L_ECHOK(tty) L_FLAG((tty), ECHOK) // KILL erase line flag in canon mode.
38 #define L_ECHCTL(tty) L_FLAG((tty), ECHCTL) // echo control char flag.
39 #define L_ECHOKL(tty) L_FLAG((tty), ECHOKL) // KILL erase line & echo flags in canon mode
40 #define L_TOSTOP(tty) L_FLAG((tty), TOSTOP) // send SIGTTOU signal for backgnd output.
41
// Get a flag from the termios structure input mode flag set.
42 #define I_UCLC(tty) I_FLAG((tty), IUCLC) // get uppercase to lowercase flag.
43 #define I_NLCR(tty) I_FLAG((tty), INLCR) // Map Line feed NL to CR flag on input.
44 #define I_CRNL(tty) I_FLAG((tty), ICRNL) // Carriage return CR to NL flag.
45 #define I_NOCR(tty) I_FLAG((tty), IGNCR) // Ignore CR flag.
46 #define I_IXON(tty) I_FLAG((tty), IXON) // Input control flow flag XON.
47
// Get a flag from the termios structure output mode flag set.
48 #define O_POST(tty) O_FLAG((tty), OPOST) // Post-process output.
49 #define O_NLCR(tty) O_FLAG((tty), ONLCR) // Map NL to CR-NL flag.
50 #define O_CRNL(tty) O_FLAG((tty), OCRNL) // Map CR to NL flag.
51 #define O_NLRET(tty) O_FLAG((tty), ONLRET) // NL performs CR function flag.
52 #define O_LCUC(tty) O_FLAG((tty), OLCUC) // Lowercase to uppercase flag.
53

```

```

// Get the baud rate in the termios structure control flag. CBAUD is the baud rate mask (0000017).
54 #define C_SPEED(tty) ((tty)->termios.c_cflag & CBAUD)
// Determine whether the tty terminal has been hanged up, that is, whether its transmission
// baud rate is B0 (0).
55 #define C_HUP(tty) (C_SPEED((tty)) == B0)
56
57 #ifndef MIN
58 #define MIN(a,b) ((a) < (b) ? (a) : (b))
59 #endif
60
// The following defines the buffer queue structure array tty_queues used by the tty terminal
// and the tty terminal table structure array tty_table. QUEUES is the maximum number of buffer
// queues used by tty terminals. The pseudo terminal is divided into two types (master and slave).
// Each tty terminal uses three tty buffer queues, which are the read queue for the buffered
// keyboard or serial input read_queue, the write queue for the buffered screen or serial output,
// write_queue, and the auxiliary queue secondary for saving the canonical mode characters.
61 #define QUEUES (3*(MAX_CONSOLES+NR_SERIALS+2*NR_PTYS)) // total 54 queues.
62 static struct tty_queue tty_queues[QUEUES];
63 struct tty_struct tty_table[256];
64
// The following sets the starting position of the buffer queue structure used by various types
// of tty terminals in the tty_queues[] array. 8 virtual console terminals occupy the first
// 24 items of the tty_queues[] array (3 X MAX_CONSOLES) (0 -- 23);
// The two serial terminals occupy the next six items (3 X NR_SERIALS) (24 -- 29).
// The four main pseudo terminals occupy the next 12 items (3 X NR_PTYS) (30 -- 41).
// The four slave pseudo terminals occupy the next 12 items (3 X NR_PTYS) (42 -- 53).
65 #define con_queues tty_queues
66 #define rs_queues ((3*MAX_CONSOLES) + tty_queues)
67 #define mpty_queues ((3*(MAX_CONSOLES+NR_SERIALS)) + tty_queues)
68 #define spty_queues ((3*(MAX_CONSOLES+NR_SERIALS+NR_PTYS)) + tty_queues)
69
// The following sets the starting position of the tty structure used by various types of tty
// terminals in the tty_table[] array. 8 virtual console terminals can use 64 items (0 -- 63)
// at the beginning of the tty_table[] array;
// The two serial terminals use the next two items (64 -- 65).
// The four main pseudo terminals use items starting from 128, up to 64 items (128 -- 191).
// The four slave pseudo-terminals use items starting from 192, up to 64 items (192 -- 255).
70 #define con_table tty_table // Define the console terminal tty table symbol.
71 #define rs_table (64+tty_table) // Serial terminal tty table.
72 #define mpty_table (128+tty_table) // The main pseudo terminal tty table.
73 #define spty_table (192+tty_table) // The slave pseudo terminal tty table.
74
75 int fg_console = 0; // Current foreground console number (range 0--7).
76
77 /*
78  * these are the tables used by the machine code handlers.
79  * you can implement virtual consoles.
80  */
// tty read and write buffer queue structure address table. Used by the rs_io.s program to get
// the address of the read-write buffer queue structure.
81 struct tty_queue * table_list[]={
82     con_queues + 0, con_queues + 1, // foreground console read/write queue address.
83     rs_queues + 0, rs_queues + 1, // serial terminal 1 read/write queue address.

```

```

84     rs\_queues + 3, rs\_queues + 4        // serial terminal 2 read/write queue address.
85     };
86
87     // Change the foreground console function.
88     // Set the foreground console to the specified virtual console.
89     // Parameters: new_console - the new console number specified.
90     void change\_console(unsigned int new_console)
91     {
92         // Exit if the console specified by the parameter is already in the foreground or the parameter
93         // is invalid. Otherwise, the current foreground console number is set, and the foreground
94         // console read and write queue structure addresses in table_list[] are updated. Finally update
95         // the current front console screen.
96         if (new_console == fg\_console || new_console >= NR\_CONSOLES)
97             return;
98         fg\_console = new_console;
99         table\_list[0] = con\_queues + 0 + fg\_console*3;
100        table\_list[1] = con\_queues + 1 + fg\_console*3;
101        update\_screen(); // kernel/chr_drv/console.c, line 936.
102    }
103
104    // If the queue buffer is empty, the process enters an interruptible sleep state.
105    // Parameters: queue - a pointer to the specified queue. The process needs to call this function
106    // to verify before fetching the characters in the queue buffer. If the current process has
107    // no signal to process and the specified queue buffer is empty, let the process enter an
108    // interruptible sleep state and let the queue's process wait pointer points to the process.
109    static void sleep\_if\_empty(struct tty\_queue * queue)
110    {
111        cli();
112        while (!(current->signal & ~current->blocked) && EMPTY(queue))
113            interruptible\_sleep\_on(&queue->proc_list);
114        sti();
115    }
116
117    // If the queue buffer is full, the process enters an interruptible sleep state.
118    // Parameters: queue - a pointer to the specified queue. This function needs to be called to
119    // determine the queue condition before the process writes characters to the queue buffer.
120    // Returns if the queue buffer is not full. Otherwise, if the process has no signal to process,
121    // and the free area remaining in the queue buffer is < 128, the process is put into an
122    // interruptible sleep state, and the queue's process waits pointer points to the process.
123    static void sleep\_if\_full(struct tty\_queue * queue)
124    {
125        if (!FULL(queue))
126            return;
127        cli();
128        while (!(current->signal & ~current->blocked) && LEFT(queue)<128)
129            interruptible\_sleep\_on(&queue->proc_list);
130        sti();
131    }
132
133    // Wait for a key to be pressed.
134    // If foreground console read queue is empty, the process enters interruptible sleep state.
135    void wait\_for\_keypress(void)
136    {

```

```

117     sleep\_if\_empty(tty\_table[fg\_console].secondary);
118 }
119
120 // Copy and convert to character sequences of canonical mode.
121 // The characters in the specified terminal read queue are copied and converted into the canonical
122 // mode (cooked mode) characters and stored in the auxiliary queue according to various flags
123 // set in the terminal termios structure.
124 void copy\_to\_cooked(struct tty\_struct * tty)
125 {
126     signed char c;
127
128     // First check whether the buffer queue pointers in the current terminal tty structure are valid.
129     // If all three queue pointers are NULL, there is a problem with tty initialization function.
130     if (!(tty->read_q || tty->write_q || tty->secondary)) {
131         printk("copy_to_cooked: missing queues\n|r");
132         return;
133     }
134
135     // Otherwise, we will properly process each character fetched from the tty read queue buffer
136     // according to the input and local flags in the terminal termios structure, and then put it
137     // into the auxiliary queue secondary.
138     // In the loop body below, if the read queue is already empty or the auxiliary queue is full
139     // of characters, the loop body is exited. Otherwise, the program takes a character from the
140     // pointer at the end of the read queue and moves the tail pointer forward by one position.
141     // It is then processed according to the character code. In addition, if _POSIX_VDISABLE(\0)
142     // is defined, if the character code value is equal to the value of _POSIX_VDISABLE, the function
143     // of the corresponding special control character is prohibited.
144     while (1) {
145         if (EMPTY(tty->read_q))
146             break;
147         if (FULL(tty->secondary))
148             break;
149         GETCH(tty->read_q, c); // get a char and the tail moves foreward.
150
151         // If the character is a carriage return CR (13), then if the carriage return to newline (line
152         // feed) flag CRNL is set, the character is converted to a newline NL (10). Otherwise, if the
153         // Ignore carriage return flag NOCR is set, the character is ignored and continues to process
154         // other characters. If the character is a newline character NL(10) and the line feed to carriage
155         // return flag NLCR is set, it is converted to a CR (13). If the uppercase to lowercase input
156         // flag UCLC is set, the character is converted to lowercase.
157         if (c==13) {
158             if (I\_CRNL(tty))
159                 c=10;
160             else if (I\_NOCR(tty))
161                 continue;
162         } else if (c==10 && I\_NLCR(tty))
163             c=13;
164         if (I\_UCLC(tty))
165             c=tolower(c);
166
167         // If the canonical mode flag CANON in the local mode flag set is set, the characters read are
168         // processed in the following manner.
169         // First, if the character is the keyboard termination control character KILL (^U), the deletion
170         // processing is performed on the current line that has been input. The process of deleting
171         // a line of characters is: If the tty auxiliary queue is not empty, and the last character

```

```

// fetched in the auxiliary queue is not a newline character NL(10), and the character is not
// the end-of-file character (^D) or the end-of-file character is not equal to _POSIX_VDISABLE,
// then we loop through the other characters on the line perform deletion processing. During
// the deletion process, if the local echo flag ECHO is set, then the erase control character
// ERASE (^H) needs to be placed in the write queue. If the character is a control character
// (value < 32) and is represented by a two-byte representation (eg ^V), an additional erase
// character ERASE must be placed. The tty write function is then called to output all the
// characters in the write queue to the terminal device. Finally, the tty auxiliary queue head
// pointer is backed by 1 byte.
143         if (L_CANON(tty)) {
144             if ((KILL_CHAR(tty) != _POSIX_VDISABLE) &&
145                 (c==KILL_CHAR(tty))) {
146                 /* deal with killing the input line */
147                 while(!EMPTY(tty->secondary) ||
148                     (c=LAST(tty->secondary))==10 ||
149                     ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
150                     (c==EOF_CHAR(tty))))) {
151                     if (L_ECHO(tty)) {
152                         if (c<32)                // control char ?
153                             PUTCH(127,tty->write_q);
154                             PUTCH(127,tty->write_q);
155                             tty->write(tty);
156                     }
157                     DEC(tty->secondary->head);
158                 }
159                 continue;                // processing other chars in the line.
160             }

// If the character is the delete control character ERASE(^H) and the delete character is not
// equal to _POSIX_VDISABLE, then: if the tty's auxiliary queue is empty, or its last character
// is a newline NL(10), or the end of file character but not equal to _POSIX_VDISABLE, continue
// to process other characters. During the deletion process, if the local echo flag ECHO is
// set, then the erase control character ERASE (^H) needs to be placed in the write queue. If
// the character is a control character (value < 32) and is represented by a two-byte
// representation (eg ^V), an additional erase character ERASE must be placed. The tty write
// function is then called to output all the characters in the write queue to the terminal device.
// Finally, the tty auxiliary queue head pointer is backed by 1 byte, and the code continue
// to process other characters.

161         if ((ERASE_CHAR(tty) != _POSIX_VDISABLE) &&
162             (c==ERASE_CHAR(tty))) {
163             if (EMPTY(tty->secondary) ||
164                 (c=LAST(tty->secondary))==10 ||
165                 ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
166                 (c==EOF_CHAR(tty)))))
167                 continue;
168             if (L_ECHO(tty)) {
169                 if (c<32)
170                     PUTCH(127,tty->write_q);
171                     PUTCH(127,tty->write_q);
172                     tty->write(tty);
173             }
174             DEC(tty->secondary->head);
175             continue;

```

```

176         }
177     }
    // If the IXON flag is set, the terminal stop/start output control character is activated. If
    // this flag is not set, the stop and start characters will be read as normal characters for
    // the process. In this code, if the character read is the stop character STOP(^S), set the
    // tty stop flag, let the tty pause the output, discard the special control character (not placed
    // in the auxiliary queue), and continue to process the other character. If the character is
    // the start character START(^Q), the tty stop flag is reset, the tty output is restored, the
    // control character is discarded, and other characters continue to be processed.
    // For the console, tty->write() is the con_write() function in console.c. So the console will
    // immediately pause displaying new characters on the screen (chr_drv/console.c, line 586) due
    // to the discovery of stopped=1. For the pseudo terminal, the write operation is suspended
    // (chr_drv/pty.c, line 24) because the terminal stopped flag is set. For the serial terminal,
    // the transmission should be suspended according to the terminal stopped flag during the sending
    // terminal, but this version is not implemented.
178     if (I_IXON(tty)) {
179         if ((STOP_CHAR(tty) != _POSIX_VDISABLE) &&
180             (c==STOP_CHAR(tty))) {
181             tty->stopped=1;
182             tty->write(tty);
183             continue;
184         }
185         if ((START_CHAR(tty) != _POSIX_VDISABLE) &&
186             (c==START_CHAR(tty))) {
187             tty->stopped=0;
188             tty->write(tty);
189             continue;
190         }
191     }
    // If the ISIG flag is set in the input mode flag set, indicating that the terminal keyboard
    // can generate a signal, and when the control characters INTR, QUIT, SUSP or DSUSP are received,
    // a corresponding signal needs to be generated for the process. If the character is a keyboard
    // interrupt (^C), the keyboard interrupt signal SIGINT is sent to all processes in the process
    // group of the current process, and the next character continues to be processed. If the
    // character is an quit character (^_), the keyboard quit signal SIGQUIT is sent to all processes
    // in the process group of the current process, and the next character continues to be processed.
    // If the character is a suspend character (^Z), a stop signal SIGTSTP is sent to the current
    // process. Similarly, if _POSIX_VDISABLE(^0) is defined, if the character code value is equal
    // to the value of _POSIX_VDISABLE during character processing, the function of the corresponding
    // special control character is prohibited.
192     if (L_ISIG(tty)) {
193         if ((INTR_CHAR(tty) != _POSIX_VDISABLE) &&
194             (c==INTR_CHAR(tty))) {
195             kill_pg(tty->pgrp, SIGINT, 1);
196             continue;
197         }
198         if ((QUIT_CHAR(tty) != _POSIX_VDISABLE) &&
199             (c==QUIT_CHAR(tty))) {
200             kill_pg(tty->pgrp, SIGQUIT, 1);
201             continue;
202         }
203         if ((SUSPEND_CHAR(tty) != _POSIX_VDISABLE) &&
204             (c==SUSPEND_CHAR(tty))) {

```

```

205         if (!is_orphaned_pgrp(tty->pgrp))
206             kill_pg(tty->pgrp, SIGTSTP, 1);
207         continue;
208     }
209 }
// If the character is a newline character NL(10) or a end of file character EOF(4, ^D), indicating
// that a line of characters has been processed, the line number 'secondary.data' currently
// contained in the auxiliary queue is incremented by one. If you take a line of characters
// from the auxiliary queue in the function tty_read(), the line number is decremented by one,
// see line 315.
210         if (c==10 || (EOF_CHAR(tty) != POSIX_VDISABLE &&
211             c==EOF_CHAR(tty)))
212             tty->secondary->data++;
// If the echo flag ECHO in the local mode flag set is set, then if the character is a newline
// NL(10), the newline NL(10) and the CR(13) must also be placed in the tty write queue, and
// if the character is a control character (value <32) and the echo control character flag ECHOCTL
// is set, the character '^' and the character c+64 are placed in the tty write queue (^C, ^H,
// etc. will be displayed); otherwise, the character will be placed directly into the tty write
// queue. Finally, the tty write operation function is called.
213         if (L_ECHO(tty)) {
214             if (c==10) {
215                 PUTCH(10, tty->write_q);
216                 PUTCH(13, tty->write_q);
217             } else if (c<32) {
218                 if (L_ECHOCTL(tty)) {
219                     PUTCH('^', tty->write_q);
220                     PUTCH(c+64, tty->write_q);
221                 }
222             } else
223                 PUTCH(c, tty->write_q);
224         tty->write(tty);
225     }
// The processed characters are placed in the auxiliary queue at the end of each loop.
// Finally, wake up any processes waiting for the auxiliary queue after exiting the loop body.
226         PUTCH(c, tty->secondary);
227     }
228     wake_up(&tty->secondary->proc_list);
229 }
230
231 /*
232  * Called when we need to send a SIGTTIN or SIGTTOU to our process
233  * group
234  *
235  * We only request that a system call be restarted if there was if the
236  * default signal handler is being used. The reason for this is that if
237  * a job is catching SIGTTIN or SIGTTOU, the signal handler may not want
238  * the system call to be restarted blindly. If there is no way to reset the
239  * terminal pgrp back to the current pgrp (perhaps because the controlling
240  * tty has been released on logout), we don't want to be in an infinite loop
241  * while restarting the system call, and have it always generate a SIGTTIN
242  * or SIGTTOU. The default signal handler will cause the process to stop
243  * thus avoiding the infinite loop problem. Presumably the job-control
244  * cognizant parent will fix things up before continuing its child process.

```

```

245  */
    /// Signals to all processes in the process group that use the terminal.
    // This function is used to send a SIGTTIN or SIGTTOU signal to all processes in the background
    // process group when a process in the group accesses the control terminal. Regardless of whether
    // the process in the background process group has blocked or ignored these two signals, the
    // current process will immediately exit the read and write operations and return.
246 int tty_signal(int sig, struct tty_struct *tty)
247 {
    // We do not want to stop processes in an orphan process group (see the description on line
    // 232 in the file kernel/exit.c). So if the current process group is an orphan process group,
    // an error is returned. Otherwise, the specified signal sig is sent to all processes of the
    // current process group.
248     if (is_orphaned_pgrp(current->pgrp))
249         return -EIO;          /* don't stop an orphaned pgrp */
250     (void) kill_pg(current->pgrp, sig, 1);    // send signal sig.
    // If this signal is blocked (masked) by the current process, or ignored, an error is returned;
    // Otherwise, if the current process has set a new handler for the signal sig, then we can return
    // the information that we can be interrupted; Otherwise, it returns information that can be
    // executed after the system-call is restarted.
251     if ((current->blocked & (1<<(sig-1))) ||
252         ((int) current->sigaction[sig-1].sa_handler == 1))
253         return -EIO;          /* Our signal will be ignored */
254     else if (current->sigaction[sig-1].sa_handler)
255         return -EINTR;        /* We _will_ be interrupted :-) */
256     else
257         return -ERESTARTSYS;   /* We _will_ be interrupted :-) */
258                                 /* (but restart after we continue) */
259 }
260
    /// tty read function.
    // Reads the specified number of characters from the terminal auxiliary queue and places them
    // in the user-specified buffer. Parameters: channel - the sub-device number; buf - the user
    // buffer pointer; nr - the number of bytes to read. Returns the number of bytes read.
261 int tty_read(unsigned channel, char * buf, int nr)
262 {
263     struct tty_struct * tty;
264     struct tty_struct * other_tty = NULL;
265     char c, * b=buf;
266     int minimum, time;
267
    // First determine the validity of the function parameters and take the tty structure pointer
    // of the terminal. If the three queue pointers of the tty terminal are both NULL, an EIO error
    // message is returned. If the tty terminal is a pseudo terminal, another tty structure other_tty
    // corresponding to the pseudo terminal needs to be obtained.
268     if (channel > 255)
269         return -EIO;
270     tty = TTY_TABLE(channel);
271     if (!(tty->write_q || tty->read_q || tty->secondary))
272         return -EIO;
    // If the current process uses the tty terminal being processed here, but the process group
    // number of the terminal is different from the current process group number, it indicates that
    // the current process is a process in the background process group, that is, the process is
    // not in the foreground. So we need to stop all processes in the current process group. Therefore,

```



```

// it is necessary to send the SIGTTIN signal to the current process group, and return, waiting
// to become the foreground process group and then perform the read operation.
// In addition, if the current terminal is a pseudo terminal, the corresponding other pseudo
// terminal is other_tty. If the tty here is the primary pseudo terminal, then other_tty is
// the corresponding slave pseudo terminal, and vice versa.
273     if ((current->tty == channel) && (tty->pgrp != current->pgrp))
274         return(tty\_signal(SIGTTIN, tty));
275     if (channel & 0x80)
276         other_tty = tty\_table + (channel ^ 0x40);

// The code then sets the read character operation timeout timing value and the minimum number
// of characters to be read based on the control character array values corresponding to VTIME
// and VMIN. In non-canonical mode, these two are timeout timing values. VMIN represents the
// minimum number of characters that need to be read in order to satisfy the read operation.
// VTIME is a 1/10 second count timing value.
277     time = 10L*tty->termios.c_cc[VTIME];           // set read timeout timing value.
278     minimum = tty->termios.c_cc[VMIN];           // the minimum nr of chars to read.
// If the tty terminal is in the canonical mode, set the minimum number of characters to be
// read to be equal to the number of characters to be read, nr, and set the timeout value of
// the process to read nr characters to a maximum value (no timeout). Otherwise, the terminal
// is in non-canonical mode. If the minimum number of read characters is set at this time, the
// temporary read-timeout timing value is set to infinity temporarily, so that the process reads
// the existing characters in the auxiliary queue first. If the number of characters read is
// less than minimum, the following code sets the read timeout value of the process according
// to the specified timeout value time, and waits for the rest of the characters to be read,
// see line 328.
// If the minimum number of read characters minimum is not set at this time, it is set to the
// number of characters to be read nr, and if the timeout timing value is set, the process read
// timeout timing value timeout is set to the current system time + the specified timeout time,
// and then the time is reset. In addition, if the minimum number of read characters set above
// is greater than the number of characters nr to be read by the process, let minimum=nr. That
// is, for the read operation in the canonical mode, it is not subject to the constraints and
// control of the corresponding control character values of VTIME and VMIN, and they only function
// in the non-canonical mode (raw mode) operation.
279     if (L\_CANON(tty)) {
280         minimum = nr;
281         current->timeout = 0xffffffff;
282         time = 0;
283     } else if (minimum)
284         current->timeout = 0xffffffff;
285     else {
286         minimum = nr;
287         if (time)
288             current->timeout = time + jiffies;
289         time = 0;
290     }
291     if (minimum>nr)
292         minimum = nr;           // reads up to the required number of chars.

// Now we start to loop out the characters from the auxiliary queue and put them in the user
// buffer buf. When the number of bytes to be read is greater than 0, the following loop operation
// is performed. During the loop, if the current terminal is a pseudo terminal, then we execute
// the write operation function of its corresponding other pseudo terminal, allowing another

```

```

// pseudo terminal to write the character into the current pseudo terminal auxiliary queue
// buffer. That is, the other terminal copies the characters in the write queue buffer to the
// current pseudo terminal read queue buffer, and converts it into the current pseudo terminal
// auxiliary queue after being converted by the canonical mode function.
293     while (nr>0) {
294         if (other_tty)
295             other_tty->write(other_tty);

// If the tty auxiliary queue is empty, or the canonical mode flag is set and the tty read queue
// is not full, and the number of character lines in the auxiliary queue is 0, then if the process
// read character timeout value (0) is not set, or the current process receives the signal,
// then we exit the loop body first. Otherwise, if the terminal is a slave pseudo terminal and
// its corresponding master pseudo terminal has been hung up, then we also exit the loop body.
// If it is not one of the above two situations, we will let the current process enter the
// interruptible sleep state and continue processing after returning. Since the kernel provides
// data to the user in character line units in the canonical mode, there must be at least one
// line of characters in the auxiliary queue in this mode, that is, the minimum of secondary.data
// is 1.
296     cli();
297     if (EMPTY(tty->secondary) || (L_CANON(tty) &&
298         !FULL(tty->read_q) && !tty->secondary->data)) {
299         if (!current->timeout ||
300             (current->signal & ~current->blocked)) {
301             sti();
302             break;
303         }
304         if (IS_A_PTY_SLAVE(channel) && C_HUP(other_tty))
305             break;
306         interruptible_sleep_on(&tty->secondary->proc_list);
307         sti();
308         continue;
309     }
310     sti();

// The following begins the formal execution of the character fetching operation. The number
// of characters to be read nr is successively decremented until nr=0 or the auxiliary buffer
// queue is empty.
// In this loop, the auxiliary queue character c is first taken, and the queue tail pointer
// tail is shifted to the right by one character position. If the character got is a file
// terminator (^D) or a newline character NL(10), the number of character lines contained in
// the auxiliary queue is decremented by one. If the character is a file end character (^D)
// and the canonical mode flag is set, the loop is interrupted, otherwise the file end character
// has not been encountered or is in the raw (non-canonical) mode. In this mode, the user uses
// the character stream as the read object and does not recognize the control characters (such
// as the file terminator). The code then puts the character directly into the user data buffer
// buf and decrements the number of characters to be read by one. At this time, if the number
// of characters to be read is already 0, the loop is interrupted. In addition, if the terminal
// is in canonical mode and the character read is a newline NL(10), the loop is also exited.
// In addition, as long as the nr characters have not been taken, and the auxiliary queue is
// not empty, the characters in the queue are continuously read.
311     do {
312         GETCH(tty->secondary, c);
313         if ((EOF_CHAR(tty) != POSIX_VDISABLE &&
314             c==EOF_CHAR(tty)) || c==10)

```

```

315         tty->secondary->data--;
316         if ((EOF\_CHAR(tty) != POSIX\_VDISABLE &&
317             c==EOF\_CHAR(tty)) && L\_CANON(tty))
318             break;
319         else {
320             put\_fs\_byte(c, b++);
321             if (!--nr)
322                 break;
323         }
324         if (c==10 && L\_CANON(tty))
325             break;
326     } while (nr>0 && !EMPTY(tty->secondary));

// At this point, if the tty terminal is in canonical mode, we may have read a newline or
// encountered a end of file character. If it is in non-canonical mode, then we have read nr
// characters, or the auxiliary queue has been taken out. So we first wake up the process waiting
// for the read queue, and then see if the timeout timing value is set. If the timeout timing
// value is not 0, we will wait for a certain amount of time for other processes to write characters
// to the read queue. So we set the process read timeout timing value to the system current
// time jiffies + read timeout time. Of course, if the terminal is in canonical mode, or if
// nr characters have been read, we can exit this big loop directly.
327     wake\_up(&tty->read_q->proc_list);
328     if (time)
329         current->timeout = time+jiffies;
330     if (L\_CANON(tty) || b-buf >= minimum)
331         break;
332 }

// At this point, the tty character loop operation ends, so we reset the process's read timeout
// timing value timeout. If the current process has received a signal and no characters have
// been read yet, it is returned by restarting the system-call number, otherwise it returns
// the number of characters read (b-buf).
333     current->timeout = 0;
334     if ((current->signal & ~current->blocked) && !(b-buf))
335         return -ERESTARTSYS;
336     return (b-buf);
337 }
338

//// tty write function.
// Put the characters in the user buffer into the tty write queue buffer.
// Parameters: channel - the sub-device number; buf - the buffer pointer; nr - the number of
// bytes written. Returns the number of bytes written.
339 int tty\_write(unsigned channel, char * buf, int nr)
340 {
341     static cr_flag=0;
342     struct tty\_struct * tty;
343     char c, *b=buf;
344
// First determine the validity of the parameters and take the tty structure pointer of the
// terminal. If the three queue pointers of the tty terminal are both NULL, an EIO error message
// is returned.
345     if (channel > 255)
346         return -EIO;
347     tty = TTY\_TABLE(channel);

```

```

348         if (!(tty->write_q || tty->read_q || tty->secondary))
349             return -EIO;
// If TOSTOP is set in the terminal local mode flag set, it means that the background process
// output needs to send a signal SIGTTOU. At this time, if the current process uses the tty
// terminal being processed here, but the process group number of the terminal is different
// from the current process group number, it means that the current process is a process in
// the background process group, that is, the process is not in the foreground. So we need to
// stop all processes in the current process group. Therefore, it is necessary to send the SIGTTOU
// signal to the current process group, and return, waiting to become the foreground process
// group and then perform the write operation.
350         if (L_TOSTOP(tty) &&
351             (current->tty == channel) && (tty->pgrp != current->pgrp))
352             return(tty_signal(SIGTTOU, tty));

// Now we start looping out the characters from the user buffer buf and putting them into the
// write queue buffer. When the number of bytes to be written is greater than 0, the following
// loop operation is performed. During the loop, if the tty write queue is full at this time,
// the current process enters an interruptible sleep state. If the current process has a signal
// to process, exit the loop body.
353         while (nr>0) {
354             sleep_if_full(tty->write_q);
355             if (current->signal & ~current->blocked)
356                 break;
// When the number of characters to be written nr is still greater than 0 and the tty write
// queue buffer is not full, the following operations are performed cyclically. First take 1
// byte from the user buffer. If the execution output processing flag OPOST in the terminal
// output mode flag set is set, the post-processing operation on the character is performed.
357             while (nr>0 && !FULL(tty->write_q)) {
358                 c=get_fs_byte(b);
359                 if (O_POST(tty)) {
// If the character is a carriage return '\r' (CR, 13) and the carriage return conversion line
// character OCRNL is set, the character is replaced with a newline character '\n' (NL, 10);
// Otherwise, if the character is a newline character '\n' and the line feed return function
// flag ONLRET is set, the character is replaced with a carriage return character '\r'.
360                     if (c=='\r' && O_CRNL(tty))
361                         c='\n';
362                     else if (c=='\n' && O_NLRET(tty))
363                         c='\r';
// If the character is a newline character '\n' and the carriage return flag cr_flag is not
// set, but the line feed to the car-new line flag ONLCR is set, the cr_flag flag is set and
// a CR is placed in the write queue. Then we continue to process the next character.
// If the lowercase to uppercase flag OLCUC is set, the character is converted to uppercase
// character.
364                     if (c=='\n' && !cr_flag && O_NLCR(tty)) {
365                         cr_flag = 1;
366                         PUTCH(13, tty->write_q);
367                         continue;
368                     }
369                     if (O_LCUC(tty))
370                         c=toupper(c);
371                 }
// Next, the user data buffer pointer b is forwarded by 1 byte; the number of bytes to be written
// is reduced by one byte; the cr_flag flag is reset, and the byte is placed in the tty write

```

```

// queue.
372         b++; nr--;
373         cr_flag = 0;
374         PUTCH(c, tty->write_q);
375     }
// If the required characters are all written, or the write queue is full, the program exits
// the loop. At this point, the corresponding tty write function is called to display the
// characters in the write queue on the console screen or send them out through the serial port.
// If the currently processed tty is a console terminal, then tty->write() is con_write(); if
// the tty is a serial terminal, tty->write() is rs_write() function. If there are still bytes
// to write, then we need to wait for the characters in the write queue to be taken. So call
// the scheduler here and go to other tasks first.
376         tty->write(tty);
377         if (nr>0)
378             schedule();
379     }
380     return (b-buf); // finally returns the number of bytes written.
381 }
382
383 /*
384  * Jeh, sometimes I really like the 386.
385  * This routine is called from an interrupt,
386  * and there should be absolutely no problem
387  * with sleeping even in an interrupt (I hope).
388  * Of course, if somebody proves me wrong, I'll
389  * hate intel for all time :-). We'll have to
390  * be careful and see to reinstating the interrupt
391  * chips before calling this, though.
392  *
393  * I don't think we sleep here under normal circumstances
394  * anyway, which is good, as the task sleeping might be
395  * totally innocent.
396  */
//// Function called in interrupt handler - character canonical mode processing.
// Parameters: tty - the specified tty terminal number.
// Copy or convert the characters in the specified tty terminal queue into canonical (cooked)
// mode characters and store them in the auxiliary queue. This function is called in the serial
// port read character interrupt (rs_io.s, 110) and the keyboard interrupt (kerboard.S, 76).
397 void do tty interrupt(int tty)
398 {
399     copy\_to\_cooked(TTY\_TABLE(tty));
400 }
401
//// Character device initialization function. Empty, ready for future expansion.
402 void chr\_dev\_init(void)
403 {
404 }
405
//// tty terminal initialization function.
// Initialize all terminal buffer queues, initialize serial terminal and console terminal.
406 void tty\_init(void)
407 {
408     int i;

```

```

409 // First initialize the buffer queue structure of all terminals and set the initial value. For
// serial terminal read/write buffer queues, set their data field to the serial port base address.
// Serial port 1 is 0x3f8, and serial port 2 is 0x2f8. Then initially set the tty structure
// of all terminals. The initial value of the special character array c_cc[] is defined in the
// include/linux/tty.h file.
410     for (i=0 ; i < QUEUES ; i++)
411         tty_queues[i] = (struct tty_queue) {0,0,0,0, ""};
412     rs_queues[0] = (struct tty_queue) {0x3f8,0,0,0, ""}; // read queue.
413     rs_queues[1] = (struct tty_queue) {0x3f8,0,0,0, ""}; // write queue.
414     rs_queues[3] = (struct tty_queue) {0x2f8,0,0,0, ""};
415     rs_queues[4] = (struct tty_queue) {0x2f8,0,0,0, ""};
416     for (i=0 ; i<256 ; i++) {
417         tty_table[i] = (struct tty_struct) {
418             {0, 0, 0, 0, 0, INIT_C_CC},
419             0, 0, 0, NULL, NULL, NULL, NULL
420         };
421     }
// Then initialize the console terminal (console.c, line 834). We put con_init() here because
// we need to determine the number of virtual consoles in the system, NR_CONSOLES, based on
// the type of display card and the amount of video memory. This value is used in the subsequent
// console tty structure initialization loop. For the tty structure of the console, the 425--430
// lines are the termios structure fields contained in the tty structure. The input mode flag
// set is initialized to an ICRNL flag; the output mode flag is initialized to include a
// post-processing flag OPOST and a flag ONLCR that converts NL to CRNL; the local mode flag
// set is initialized to include IXON, ICANON, ECHO, ECHOCTL, and ECHOKE flags; The control
// character array c_cc[] is set to contain the initial value INIT_C_CC. The read buffer, write
// buffer, and auxiliary buffer queue structures in the tty structure of the console terminal
// are initialized on line 435, which respectively point to the corresponding structure items
// in the tty queue structure array tty_table[], see the description on lines 61--73.
422     con_init();
423     for (i = 0 ; i<NR_CONSOLES ; i++) {
424         con_table[i] = (struct tty_struct) {
425             {ICRNL, /* change incoming CR to NL */
426             OPOST|ONLCR, /* change outgoing NL to CRNL */
427             0, // control mode flag set.
428             IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, // local flag set.
429             0, /* console termio */ // 0 -- TTY.
430             INIT_C_CC}, // control char array c_cc[]
431             0, /* initial pgrp */
432             0, /* initial session */
433             0, /* initial stopped */
434             con_write, // console write function.
435             con_queues+0+i*3, con_queues+1+i*3, con_queues+2+i*3
436         };
437     }
// Then initialize the fields in the tty structure of the serial terminal. Line 450 initializes
// the read/write and auxiliary buffer queue structures in the serial terminal tty structure,
// which point to the corresponding structure items in the tty buffer queue structure array
// tty_table[]. See the description on line 61--73.
438     for (i = 0 ; i<NR_SERIALS ; i++) {
439         rs_table[i] = (struct tty_struct) {
440             {0, /* no translation */ // input mode flag set

```

---

```

441         0, /* no translation */ // output mode flag set.
442         B2400 | CS8, // control mode flag set. 2400bps, 8bits.
443         0, // local mode flag set.
444         0, // line procedure, 0 -- TTY.
445         INIT_C_CC}, // control character array.
446         0, // initial process group.
447         0, // init session.
448         0, // initial stopped flag.
449         rs_write, // serial port write function.
450         rs_queues+0+i*3, rs_queues+1+i*3, rs_queues+2+i*3 // three queues.
451     };
452 }

// Then we reinitialize the tty structure used by the pseudo terminal. The pseudo terminal is
// paired, that is, a master pseudo terminal is equipped with a slave pseudo terminal. Therefore,
// they must be initialized. In the following loop, we first initialize the tty structure of
// each master pseudo terminal, and then initialize its corresponding tty structure of each
// slave pseudo terminal.
453 for (i = 0 ; i < NR_PTYS ; i++) {
454     mpty_table[i] = (struct tty_struct) {
455         {0, /* no translation */ // input mode flag set.
456         0, /* no translation */ // output mode flag set.
457         B9600 | CS8, // control mode flag set. 9600bps, 8bits.
458         0, // local mode flag set.
459         0, // line procedure, 0--TTY.
460         INIT_C_CC}, // control character array.
461         0, // initial process group.
462         0, // initial session.
463         0, // initial stopped flag.
464         mpty_write, // master pseudo write function.
465         mpty_queues+0+i*3, mpty_queues+1+i*3, mpty_queues+2+i*3
466     };
467     spty_table[i] = (struct tty_struct) {
468         {0, /* no translation */
469         0, /* no translation */
470         B9600 | CS8,
471         IXON | ISIG | ICANON, // local mode flag set.
472         0,
473         INIT_C_CC},
474         0,
475         0,
476         0,
477         spty_write, // slave pseudo write function.
478         spty_queues+0+i*3, spty_queues+1+i*3, spty_queues+2+i*3
479     };
480 }

// Finally, the serial interrupt handler and serial interfaces 1 and 2 (serial.c, line 37) are
// initialized, and the number of virtual consoles NR_CONSOLES and the number of pseudo terminals
// NR_PTYS contained in the system are displayed.
481 rs_init();
482 printk("%d virtual consoles\n\r", NR_CONSOLES);
483 printk("%d pty's\n\r", NR_PTYS);
484 }
485

```

---

## 10.6.3 Information

### 10.6.3.1 Control characters VTIME, VMIN

In non-canonical mode, these two values are the timeout timing value and the minimum number of characters read. VMIN indicates the minimum number of characters that need to be read in order to satisfy the read operation. VTIME is a one-tenth second count timeout value. When both are set, the read operation will wait until at least one character is read. If VMIN characters are received before the timeout, the read operation is satisfied. If the timeout has expired before the VMIN characters are received, the characters that have been received at this time are returned to the user. If only VMIN is set, the read operation will not return until VMIN characters are read. If only VTIME is set, the read will return immediately after reading at least one character or timing out. If neither is set, the read will return immediately, giving only the number of bytes currently read. See the `termios.h` file for details.

## 10.7 tty\_ioctl.c

### 10.7.1 Function

This program is used for the control operation of character devices and implements the function `tty_ioctl()`. By using this function, the user program can modify information such as setting flags in the `termios` structure of the specified terminal. The `tty_ioctl()` function will be called by the input and output control system in `sys_ioctl()` in `fs/ioctl.c` to implement a file system-based unified device access interface.

Instead of using the `sys_ioctl()` system-call directly, the general user program uses the associated functions implemented in the library file. For example, for the terminal IO control command `TIOCGPGRP` that obtains the terminal process group number (ie, the foreground process group number), the library file `libc` uses the command to call the `sys_ioctl()` system-call to implement the function `tcgetpgrp()`. Therefore, ordinary users only need to use `tcgetpgrp()` to achieve the same purpose. Of course, we can also use the library function `ioctl()` to achieve the same functionality.

### 10.7.2 Code annotation

Program 10-6 linux/kernel/chr\_drv/tty\_ioctl.c

---

```
1 /*
2  * linux/kernel/chr_drv/tty_ioctl.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 // <errno.h> Error number header file. Contains various error numbers in the system.
8 // <termios.h> Terminal input and output function header file. It mainly defines the terminal
9 // interface that controls the asynchronous communication port.
10 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
11 // of the initial task 0, and some embedded assembly function macro statements about the
12 // descriptor parameter settings and acquisition.
13 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
14 // used functions of the kernel.
```



```

// <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
//      communication.
// <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.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//      descriptors/interrupt gates, etc. is defined.
7 #include <errno.h>
8 #include <termios.h>
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h>
12 #include <linux/tty.h>
13
14 #include <asm/io.h>
15 #include <asm/segment.h>
16 #include <asm/system.h>
17
// The following lines gives two external function prototypes and an array of baud rate factors
// for the serial ports. The first function is used to obtain the session number to which the
// process group belongs according to the process group number pgrp (defined in kernel/exit.c,
// line 161). The second function tty_signal() is used to signal to all processes in the process
// group that use the specified tty terminal, which is defined in chr_drv/tty_io.c, line 246.
// The baud rate factor array (or array of divisors) gives the correspondence between the baud
// rate and the baud rate factor. For example, when the baud rate is 2400 bps, the corresponding
// factor is 48 (0x30); the factor of 9600 bps is 12 (0x1c). See the instructions after the
// program list.
18 extern int session\_of\_pgrp(int pgrp);
19 extern int tty\_signal(int sig, struct tty\_struct *tty);
20
21 static unsigned short quotient[] = {
22     0, 2304, 1536, 1047, 857,
23     768, 576, 384, 192, 96,
24     64, 48, 24, 12, 6, 3
25 };
26
///// Modify the transmission baud rate.
// Parameters: tty - the tty data structure corresponding to the terminal. When the divisor
// latch flag DLAB is set, the baud rate factor low byte and the high byte are respectively
// written to the UART through the ports 0x3f8 and 0x3f9 of the serial port 1, and the DLAB
// bit is reset after the writing. For serial port 2, the two ports are 0x2f8 and 0x2f9.
27 static void change\_speed(struct tty\_struct * tty)
28 {
29     unsigned short port,quot;
30
// The function first checks if the terminal specified by the parameter tty is a serial terminal,
// and if not, exits. For the tty structure of the serial terminal, the data field of the read
// queue stores the base address of the serial port (0x3f8 or 0x2f8), and the read_q.data field
// value of the general console terminal is 0. Then we obtain the baud rate index number that
// has been set from the control mode flag set of the terminal termios structure, and obtain
// the corresponding baud rate factor value from the baud rate factor array quotient[]. CBAUD
// is the baud rate bits mask in the control mode flag set.

```

```

31         if (!(port = tty->read_q->data))
32             return;
33         quot = quotient[tty->termios.c_cflag & CBAUD];
// Then, the baud rate factor is written into the baud rate factor latch of the UART chip
// corresponding to the serial port. Before writing, we must first set the divisor latch access
// bit DLAB (bit 7) of the line control register LCR, and then write the 16-bit baud rate factor
// low and high byte to port 0xf8, 0xf9 respectively. Finally reset the DLAB flag bit of the
// LCR.
34         cli();
35         outb\_p(0x80, port+3);          /* set DLAB */
36         outb\_p(quot & 0xff, port);    /* LS of divisor */
37         outb\_p(quot >> 8, port+1);    /* MS of divisor */
38         outb(0x03, port+3);          /* reset DLAB */
39         sti();
40     }
41
// Clear the tty buffer queue.
// Parameter: queue is the specified buffer queue pointer.
// Let the buffer head pointer be equal to the tail pointer, so as to clear the buffer.
42 static void flush(struct tty\_queue * queue)
43 {
44     cli();
45     queue->head = queue->tail;
46     sti();
47 }
48
49 static void wait\_until\_sent(struct tty\_struct * tty)
50 {
51     /* do nothing - not implemented */
52 }
53
54 static void send\_break(struct tty\_struct * tty)
55 {
56     /* do nothing - not implemented */
57 }
58
// Obtain terminal termios structure information.
// Parameters: tty - specifies the terminal's tty structure pointer; termios - the user buffer
// that used to hold the termios structure.
59 static int get\_termios(struct tty\_struct * tty, struct termios * termios)
60 {
61     int i;
62
// First, verify that the memory area indicated by the user buffer pointer is sufficient. If
// not, allocate memory. Then copy the termios structure information of the specified terminal
// to the user buffer. Finally returns 0.
63     verify\_area(termios, sizeof (*termios));          // kernel/fork.c, line 24.
64     for (i=0 ; i< (sizeof (*termios)) ; i++)
65         put\_fs\_byte( ((char *)&tty->termios)[i] , i+(char *)termios );
66     return 0;
67 }
68
// Set the information in the termios structure of the terminal.

```

```

// Parameters: tty - specifies the terminal's tty structure pointer; termios - user data area
// termios structure pointer.
69 static int set\_termios(struct tty\_struct * tty, struct termios * termios,
70                        int channel)
71 {
72     int i, retsig;
73
74     /* If we try to set the state of terminal and we're not in the
75 foreground, send a SIGTTOU. If the signal is blocked or
76 ignored, go ahead and perform the operation. POSIX 7.2) */
// If the process group number of the tty terminal of the current process is different from
// the process group number of the process, that is, the current process terminal is not in
// the foreground, indicating that the current process attempts to modify the termios structure
// of the uncontrolled terminal. Therefore, according to the requirements of the POSIX standard,
// the SIGTTOU signal needs to be sent here to allow the process using this terminal to temporarily
// stop execution, so that we can modify the termios structure first. However, if the send
// function tty_signal() returns a value of ERESTARTSYS or EINTR, then the operation will be
// performed again later.
77     if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
78         retsig = tty\_signal(SIGTTOU, tty); // chr_drv/tty_io.c, line 246.
79         if (retsig == -ERESTARTSYS || retsig == -EINTR)
80             return retsig;
81     }
// The termios structure information in the user data area is then copied to the termios structure
// of the specified terminal tty structure. Because the user may have modified the terminal
// serial port transmission baud rate, the baud rate in the serial UART chip is modified again
// according to the baud rate information in the control mode flag c_cflag in the termios
// structure, and finally returns 0. .
82     for (i=0 ; i< (sizeof (*termios)) ; i++)
83         ((char *)&tty->termios)[i]=get\_fs\_byte(i+(char *)termios);
84     change\_speed(tty);
85     return 0;
86 }
87
//// Get the information in the termio structure.
// Parameters: tty - specifies the terminal's tty structure pointer; termio - the user buffer
// that holds the termio structure information.
88 static int get\_termio(struct tty\_struct * tty, struct termio * termio)
89 {
90     int i;
91     struct termio tmp_termio;
92
// First verify that the user's buffer capacity is sufficient, if not enough, then allocate
// memory. The information for the termios structure is then copied into the temporary termio
// structure. These two structures are basically the same, but the data types of the input,
// output, control, and local flagsets are different. The former's data type is long, while
// the latter is short. Therefore, the purpose of copying to the temporary termio structure
// is for data type conversion. Finally, the information in the temporary termio structure is
// copied byte by byte into the user buffer and returns 0.
93     verify\_area(termio, sizeof (*termio));
94     tmp_termio.c_iflag = tty->termios.c_iflag;
95     tmp_termio.c_oflag = tty->termios.c_oflag;
96     tmp_termio.c_cflag = tty->termios.c_cflag;

```

```

97     tmp_termio.c_lflag = tty->termios.c_lflag;
98     tmp_termio.c_line = tty->termios.c_line;
99     for(i=0 ; i < NCC ; i++)
100         tmp_termio.c_cc[i] = tty->termios.c_cc[i];
101     for (i=0 ; i< (sizeof (*termio)) ; i++)
102         put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio );
103     return 0;
104 }
105
106 /*
107  * This only works as the 386 is low-byt-first
108  */
109
110     /// Set the termio structure information of the terminal.
111     // Parameters: tty - specifies the terminal's tty structure pointer; termio - the termio
112     // structure in the user data area. This function is used to copy the information of the user
113     // buffer termio into the terminal's termios structure and return 0.
114 static int set_termio(struct tty_struct * tty, struct termio * termio,
115                     int channel)
116 {
117     int i, retsig;
118     struct termio tmp_termio;
119
120     // As with set_termios(), if the process group number of the terminal used by the process is
121     // different from the process group number of the process, that is, the current process terminal
122     // is not in the foreground, indicating that the current process attempts to modify the termios
123     // structure of the uncontrolled terminal. Therefore, according to the requirements of the POSIX
124     // standard, the SIGTTOU signal needs to be sent here to allow the process using this terminal
125     // to temporarily stop execution, so that we can modify the termios structure first. However,
126     // if the send function tty_signal() returns a value of ERESTARTSYS or EINTR, then the operation
127     // will be performed again later.
128     if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
129         retsig = tty_signal(SIGTTOU, tty);
130         if (retsig == -ERESTARTSYS || retsig == -EINTR)
131             return retsig;
132     }
133
134     // Then copy the termio structure information in the user data area to the temporary termio
135     // structure, and then copy the information into the tty's termios structure. The purpose of
136     // this is to convert the type of the mode flag set, that is, from the short integer type of
137     // termio to the long integer type of termios. But the c_line and c_cc[] fields of the two
138     // structures are identical.
139     for (i=0 ; i< (sizeof (*termio)) ; i++)
140         ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
141     *(unsigned short *)&tty->termios.c_iflag = tmp_termio.c_iflag;
142     *(unsigned short *)&tty->termios.c_oflag = tmp_termio.c_oflag;
143     *(unsigned short *)&tty->termios.c_cflag = tmp_termio.c_cflag;
144     *(unsigned short *)&tty->termios.c_lflag = tmp_termio.c_lflag;
145     tty->termios.c_line = tmp_termio.c_line;
146     for(i=0 ; i < NCC ; i++)
147         tty->termios.c_cc[i] = tmp_termio.c_cc[i];
148
149     // Finally, because the user may have modified the terminal serial port speed, the baud rate
150     // in the serial UART chip is then modified according to the baud rate information in the control
151     // mode flag c_cflag in the termios structure, and returns 0.
152     change_speed(tty);

```

```

130         return 0;
131     }
132
133     //tty terminal device input and output control function.
134     // Parameters: dev - device number; cmd - ioctl command; arg - operation parameter pointer.
135     // The function first finds the tty structure of the corresponding terminal according to the
136     // device number, and then processes it according to the control command cmd.
137 int tty\_ioctl(int dev, int cmd, int arg)
138 {
139     struct tty\_struct * tty;
140     int pgrp;
141
142     // First, the tty sub-device number is obtained according to the device number, thereby obtaining
143     // the tty structure of the terminal. If the major device number is 5 (control terminal), the
144     // tty field of the process is the tty sub-device number. At this time, if the process's tty
145     // sub-device number is a negative number, it indicates that the process does not have a control
146     // terminal, that is, the ioctl call cannot be issued, and an error message is displayed and
147     // the machine is stopped. If the major device number is not 5 but 4, we can get the sub-device
148     // number from the device number. The sub-device number can be 0 (console terminal), 1 (serial
149     // 1 terminal), or 2 (serial 2 terminal).
150     if (MAJOR(dev) == 5) {
151         dev=current->tty;
152         if (dev<0)
153             panic("tty_ioctl: dev<0");
154     } else
155         dev=MINOR(dev);
156
157     // Then according to the sub-device number and the tty table, we can get the tty structure of
158     // the corresponding terminal. So let tty point to the tty structure corresponding to the
159     // sub-device number, and then separately process according to the ioctl command cmd provided
160     // by the parameter. The second half of line 144 is used to select the corresponding tty structure
161     // in the tty_table[] table based on the sub-device number dev. If dev = 0, it means that the
162     // foreground terminal is being used, so we can directly use the terminal number fg_console
163     // as the tty_table[] entry index to get the tty structure. If dev is greater than 0, then it
164     // should be considered in two cases: (1) dev is the virtual terminal number; (2) dev is the
165     // serial terminal number or pseudo terminal number. For virtual terminals, the tty structure
166     // in tty_table[] is indexed by dev-1(0 -- 63), and for other types of terminals, their tty
167     // structure index entry is dev. For example, if dev = 64, indicating a serial terminal 1, its
168     // tty structure is tty_table[dev]. If dev = 1, the tty structure of the corresponding terminal
169     // is tty_table[0]. See lines 70-73 of the tty_io.c program.
170     tty = tty\_table + (dev ? ((dev < 64)? dev-1:dev) : fg\_console);
171
172     // TCGETS: Get the termios structure information of the corresponding terminal. At this point
173     // the parameter arg is the user buffer pointer.
174     // TCSETSF: Before setting the termios, we need to wait for all data in the output queue to
175     // be processed and flush the input queue. Then perform the operation of setting the termios.
176     // TCSETSW: Before setting the termios, we need to wait for all the data in output queue to
177     // be handled. This form is required for situations where modifying params affects the output.
178     // TCSETS: Set the corresponding terminal termios structure information. At this point the
179     // parameter arg is the user buffer pointer that holds the termios structure.
180     // TCGETA: Get the information in the termio structure of the corresponding terminal. At this
181     // point the parameter arg is the user buffer pointer.
182     // TCSETAF: Before setting termio, we need to wait for all data in the output queue to be processed
183     // and the input queue is flushed. Then perform the setting terminal termio operation.

```

```

// TCSETAW: Before setting the termios, we need to wait for all data in the output queue to
// be processed. This form is required for situations where modifying params affects the output.
// TCSETA: Set the termio structure information of the corresponding terminal. At this point
// the parameter arg is the user buffer pointer that holds the termio structure.
// TCSBRK: If the value of the parameter arg is 0, wait for the output queue to be processed
// and then send a break.
145     switch (cmd) {
146         case TCGETS:
147             return get_termios(tty, (struct termios *) arg);
148         case TCSETSF:
149             flush(tty->read_q);    /* fallthrough */
150         case TCSETSW:
151             wait_until_sent(tty);  /* fallthrough */
152         case TCSETS:
153             return set_termios(tty, (struct termios *) arg, dev);
154         case TCGETA:
155             return get_termio(tty, (struct termio *) arg);
156         case TCSETAF:
157             flush(tty->read_q);    /* fallthrough */
158         case TCSETAW:
159             wait_until_sent(tty);  /* fallthrough */
160         case TCSETA:
161             return set_termio(tty, (struct termio *) arg, dev);
162         case TCSBRK:
163             if (!arg) {
164                 wait_until_sent(tty);
165                 send_break(tty);
166             }
167             return 0;
// Start/stop flow control. If the parameter arg is TCOOFF (Terminal Control Output OFF), the
// output is suspended; if it is TCOON, the pending output is restored. At the same time as
// suspending or restoring the output, the characters in the write queue need to be output to
// speed up the user interaction response. If arg is TCIOFF (Terminal Control Input ON), the
// input is suspended; if it is TCION, the pending input is re-opened.
168         case TCXONC:
169             switch (arg) {
170                 case TCOOFF:
171                     tty->stopped = 1;    // stop the terminal output.
172                     tty->write(tty);    // write queue output.
173                     return 0;
174                 case TCOON:
175                     tty->stopped = 0;    // restore the terminal output.
176                     tty->write(tty);
177                     return 0;
// If the parameter arg is TCIOFF, it means that the terminal is required to stop input, so
// we put the STOP character into the terminal write queue, and the input will be suspended
// when the terminal receives the character. If the parameter is TCION, it means that a START
// character is sent to let the terminal resume the transmission.
// STOP_CHAR(tty) is defined as ((tty->termios.c_cc[VSTOP]), which is used to obtain the
// corresponding value of the control character array of the terminal termios structure. If
// the kernel defines _POSIX_VDISABLE(\0), then when an item value equals the value of
// _POSIX_VDISABLE, it means that the corresponding special character is prohibited. So here
// directly check if the value is 0 or not to determine whether to put the stop control character

```

```

// into the terminal write queue. The same is true of the following.
178         case TCIOFF:
179             if (STOP\_CHAR(tty))
180                 PUTCH(STOP\_CHAR(tty), tty->write_q);
181             return 0;
182         case TCION:
183             if (START\_CHAR(tty))
184                 PUTCH(START\_CHAR(tty), tty->write_q);
185             return 0;
186     }
187     return -EINVAL; /* not implemented */
// Flushes data that has been written but not yet sent, or received but not yet read. If the
// parameter arg is 0, the input queue is flushed ; if it is 1, the output queue is flushed;
// if it is 2, the input and output queues are both flushed.
188     case TCFLSH:
189         if (arg==0)
190             flush(tty->read_q);
191         else if (arg==1)
192             flush(tty->write_q);
193         else if (arg==2) {
194             flush(tty->read_q);
195             flush(tty->write_q);
196         } else
197             return -EINVAL;
198     return 0;
// TIOCEXCL: Sets the terminal serial line dedicated mode.
// TIOCNXCL: Resets the terminal serial line dedicated mode.
// TIOCSCTTY: Set tty as the control terminal. (TIOCNOTTY - no control terminal).
// TIOCGPRGP: Read terminal process group number (that is, read the foreground process group
// number). First verify the user buffer length, then copy the pgrp field of the terminal tty
// to the user buffer. At this point the parameter arg is the user buffer pointer.
199     case TIOCEXCL:
200         return -EINVAL; /* not implemented */
201     case TIOCNXCL:
202         return -EINVAL; /* not implemented */
203     case TIOCSCTTY:
204         return -EINVAL; /* set controlling term NI */
205     case TIOCGPRGP:
206         verify\_area((void *) arg, 4);
207         put\_fs\_long(tty->pgrp, (unsigned long *) arg);
208     return 0;

// Set the terminal process group number pgrp (that is, set the foreground process group number).
// The parameter arg is now a pointer to the process group number pgrp in the user buffer. The
// prerequisite for executing this command is that the process must have a control terminal.
// If the current process does not have a control terminal, or dev is not its control terminal,
// or the control terminal is now the terminal dev being processed, but the session number of
// the process is different from the session number of the terminal dev, then a terminalless
// error message is returned.
// Then we get the process group number from the user buffer and verify the validity of the
// group number. If the group number pgrp is less than 0, an invalid group number error message
// is returned; if the session number of pgrp is different from the current process, an permission
// error message is returned. Otherwise we can set the process group number of the terminal

```

```

// to prgp. At this point prgp becomes the foreground process group.
209         case TIOCSPPGRP: // implement function tcsetpgrp().
210             if ((current->tty < 0) ||
211                 (current->tty != dev) ||
212                 (tty->session != current->session))
213                 return -ENOTTY;
214             pgrp=get_fs_long((unsigned long *) arg); // get from user buffer.
215             if (pgrp < 0)
216                 return -EINVAL;
217             if (session_of_pgrp(pgrp) != current->session)
218                 return -EPERM;
219             tty->pgrp = pgrp;
220             return 0;
// TIOCOUTQ: Returns the number of characters that remains in the output queue. First verify
// the user buffer length, then copy the number of characters in the queue to the user. At this
// point the parameter arg is the user buffer pointer.
// TIOCINQ: Returns the number of characters not yet read in the input auxiliary queue. The
// user buffer length is first verified, and then the number of characters in the auxiliary
// queue is copied to the user. At this point the parameter arg is the user buffer pointer.
221         case TIOCOUTQ:
222             verify_area((void *) arg, 4);
223             put_fs_long(CHARS(tty->write_q), (unsigned long *) arg);
224             return 0;
225         case TIOCINQ:
226             verify_area((void *) arg, 4);
227             put_fs_long(CHARS(tty->secondary),
228                         (unsigned long *) arg);
229             return 0;
// TIOCSSTI: Simulate terminal input operations. The command takes a pointer to a character as
// a parameter and assumes that the character was typed on the terminal. The user must have
// superuser privileges or read permission on the control terminal.
// TIOCGWINSZ: Read terminal device window size (see the winsize structure in termios.h).
// TIOCSWINSZ: Sets the terminal device window size information (see winsize structure).
230         case TIOCSSTI:
231             return -EINVAL; /* not implemented */
232         case TIOCGWINSZ:
233             return -EINVAL; /* not implemented */
234         case TIOCSWINSZ:
235             return -EINVAL; /* not implemented */
// TIOCMGET: Returns the current status bit flag set for the MODEM status control pin.
// (see lines 185 -- 196 in termios.h).
// TIOCMBIS: Sets the state of a single MODEM state control pin (true or false).
// TIOCMBIC: Resets the state of a single MODEM state control pin.
// TIOCMSET: Sets the state of the MODEM status pin. If a bit is set, the corresponding status
// pin will be asserted.
236         case TIOCMGET:
237             return -EINVAL; /* not implemented */
238         case TIOCMBIS:
239             return -EINVAL; /* not implemented */
240         case TIOCMBIC:
241             return -EINVAL; /* not implemented */
242         case TIOCMSET:
243             return -EINVAL; /* not implemented */

```



```

// TIOCGSOFTCAR: Read the software carrier detect flag (1 - on; 0 - off).
// TIOCSSOFTCAR: Set the software carrier detect flag (1 - on; 0 - off).
244     case TIOCGSOFTCAR:
245         return -EINVAL; /* not implemented */
246     case TIOCSSOFTCAR:
247         return -EINVAL; /* not implemented */
248     default:
249         return -EINVAL;
250 }
251 }
252

```

### 10.7.3 Information

#### 10.7.3.1 Baud Rate and Baud Rate Factor

The baud rate is calculated as: baud rate = 1.8432MHz / (16 X baud rate factor).

The corresponding relationship between the common baud rate and the baud rate factor is shown in Table 10-9.

Table 10-9 Baud rate and baud rate factor table

Baud rate	Baud rate factor		Baud rate	Baud rate factor	
	MSB,LSB	Value		MSB,LSB	Value
50	0x09,0x00	2304	1200	0x00,0x60	96
75	0x06,0x00	1536	1800	0x00,0x40	64
110	0x04,0x17	1047	2400	0x00,0x30	48
134.5	0x03,0x59	857	4800	0x00,0x18	24
150	0x03,0x00	768	9600	0x00,0x1c	12
200	0x02,0x40	576	19200	0x00,0x06	6
300	0x01,0x80	384	38400	0x00,0x03	3
600	0x00,0xc0	192			

## 10.8 Summary

This chapter describes the implementation of the character device in detail, and introduces the working principle and implementation code of the local terminal device and the dumb terminal based on serial communication. We first give the hierarchical relationship of modules accessing character devices in the UNIX operating system, as well as several common types of terminals. Then combined with the termios data structure and three buffer queue structures used by the terminal, the definitions of the canonical mode (cooked mode) and the non-canonical mode (raw mode) and its program implementation are described in detail, and the console terminal and serial terminal are explained. We then introduced and annotated each program in detail in the order of the files.

In the next chapter, we will detail the basic functions of the math coprocessor and the basic data types used, and specify the implementation code for the software emulation of the math coprocessor in the Linux kernel.













# 11 Math Coprocessor (math)

The kernel directory `kernel/math` directory contains the math coprocessor simulation programs, which contains a total of nine C programs, as shown in Listing 11-1. The content of this chapter is closely related to the specific hardware structure, so the reader needs to have a deep knowledge of Intel CPU and coprocessor instruction code structure. Fortunately, this content has little to do with the kernel implementation, so skipping this chapter does not prevent the reader from a complete understanding of the kernel implementation. However, if you understand the contents of this chapter, it will be very helpful to implement system-level applications (such as assembly and disassembly) and to develop coprocessor floating-point handlers.

List 11-1 `linux/kernel/math`

---

	Name	Size	Last modified time (GMT)	Desc
	<a href="#">Makefile</a>	3377 bytes	1991-12-31 12:26:48	
	<a href="#">add.c</a>	1999 bytes	1992-01-01 16:42:02	
	<a href="#">compare.c</a>	904 bytes	1992-01-01 17:15:34	
	<a href="#">convert.c</a>	4348 bytes	1992-01-01 19:07:43	
	<a href="#">div.c</a>	2099 bytes	1992-01-01 01:41:43	
	<a href="#">ea.c</a>	1807 bytes	1991-12-31 11:57:05	
	<a href="#">error.c</a>	234 bytes	1991-12-28 12:42:09	
	<a href="#">get_put.c</a>	5145 bytes	1992-01-01 01:38:13	
	<a href="#">math_emulate.c</a>	11540 bytes	1992-01-07 21:12:05	
	<a href="#">mul.c</a>	1517 bytes	1992-01-01 01:42:33	

---

## 11.1 Function Description

Performing computationally intensive operations on a computer can usually be done in three ways. One is to perform calculations directly using CPU normal instructions. Since general CPU instructions are a class of general-purpose instructions, the use of these instructions for complex and large-scale computation requires complex computational subroutines, and generally only those skilled in mathematics and computers can program these subroutines. Another method is to configure a dedicated math coprocessor chip for the CPU. Using a coprocessor chip can greatly simplify the difficulty of mathematical programming, and the speed and efficiency of the operation will be doubled, but additional hardware investment is required. Another method is to use an emulator at the kernel level of the system to simulate the computational functions of the coprocessor. This method may be the one with the lowest speed and efficiency, but it is as easy to use as a coprocessor to program the calculation program, and can run the program without any changes to the program on a machine with a coprocessor and therefore has good code compatibility.

In the early days of Linux 0.1x and even Linux 0.9x kernel development, the math coprocessor chip 80387 (or its compatible chip) was expensive and has always been a luxury in ordinary PCs. Therefore, unless there is

a large amount of scientific calculations or where it is particularly needed, the 80387 chip will not be installed in a general PC. Although the math coprocessor feature is built into the current Intel processor, the coprocessor emulator code is no longer required in the current operating system. But because the 80387 emulation program is completely based on the analog 80387 chip processing structure and analysis of instruction code structure. So after learning the contents of this chapter, we can not only fully understand the programming method of the 80387 coprocessor, but also help to write system-level programs like assembler and disassembler.

If the 80386 PC does not include the 80387 math coprocessor chip, then when the CPU executes a coprocessor instruction, it will cause a "device nonexistent" exception interrupt. The processing code for this exception procedure starts at line 158 of `sys_call.s`. If the operating system has already set the EM bit of the CPU control register CR0 at initialization, then the `math_emulate()` function in the `math_emulate.c` program is called to "interpret" each instruction of the coprocessor with the software.

The math coprocessor emulator `math_emulate.c` in the Linux 0.12 kernel fully emulates the way the 80387 chip executes coprocessor instructions. Before processing a coprocessor instruction, the program first creates a "soft" 80387 environment in memory using data structures and other types, including emulating all 80387 internal stack accumulator groups ST[], control word registers CWD, status word register SWD, and feature word TWD (TAG word) registers. Then, the current coprocessor instruction opcode causing the exception is analyzed, and the corresponding mathematical simulation operation is performed according to the specific opcode. Therefore, before describing the processing of the `math_emulate.c` program, it is necessary to introduce the internal structure and basic working principle of the 80387.

### 11.1.1 Floating point data type

This section focuses on the types of floating point data used by the coprocessor. First, let's briefly review several representations of integers, and then explain several standard representations of floating-point numbers and temporary real-number representations used in operations in 80387.

#### 1. Integer data type

For Intel 32-bit CPUs, there are three basic unsigned data types: byte, word, and double word, with 8, 16, and 32 bits, respectively. The representation of an unsigned number is simple. Each bit in a byte represents a binary number and has different weights depending on where it is located. For example, an 8-bit unsigned binary number 0b10001011 can be expressed as:

$$U = 0b10001011 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 139$$

It corresponds to the decimal number 139. The one with the smallest weight ( $2^0$ ) is usually called the Least Significant Bit (LSB), and the bit with the largest weight ( $2^7$ ) is called the Most Significant Bit (MSB).

Moreover, there are usually three types of integer data representations with negative values in the computer: Two's complement, biased number, and Sign magnitude representation. Table 11-1 gives some of the values represented by these three forms.

Table 11-1 Several representations of integers

Decimal	2's complement	Biased (127)	Sign magnitude
128	Not Available (NA)	0b11111111	NA
127	0b01111111	0b11111110	0b01111111
126	0b01111110	0b11111101	0b01111110

2	0b00000010	0b10000001	0b00000010
1	0b00000001	0b10000000	0b00000001
0	0b00000000	0b01111111	0b00000000
-0	NA	NA	0b10000000
-1	0b11111111	0b01111110	0b10000001
-2	0b11111110	0b01111101	0b10000010
-126	0b10000010	0b00000001	0b11111110
-127	0b10000001	0b00000000	0b11111111
-128	0b10000000	NA	NA

The 2's complement notation is the integer representation used by most computer CPUs today, because the simple addition of unsigned numbers of the CPU is also applicable to data operations in this format. Using this notation, a negative number is the number plus one after each bit is inverted. The MSB bit is the sign bit of the number. MSB = 0 for a positive number; MSB = 1 for a negative number. The 80386 CPU has 8-bit (1 byte), 16-bit (1 word) and 32-bit (double word) 2's complement data types. The data range that can be represented by each is: -128 -- 127, -32768 -- 32767, -2147483648 -- 2147483647.

The biased representation of numbers is typically used to represent index field values in the floating point format. Adding a number to the specified biased value is the biased number representation of the number. As can be seen from Table 11-1, the numerical values of this representation have an order of magnitude of unsigned numbers. Therefore, this representation is easy to compare numerical values, that is, the large value of the biased representation value is always a large number of unsigned values, while the other two representations are not.

The signed number representation has a bit dedicated to the representation of sign (0 for positive number and 1 for negative number), while other bits are identical to the values represented by unsigned integers. The valid number (mantissa) portion of a floating point number uses the representation method, and the sign bit represents the sign of the entire floating point number.

In addition, a format we call a temporary integer type is used in the 80387 emulator, as shown in Figure 11-1. It is 10 bytes long and can represent a 64-bit integer data type. The lower 8 bytes can represent a maximum of 63 unsigned numbers, while the highest 2 bytes use only the most significant bits to indicate the positive or negative value. For 32-bit integer values, the lower 4 bytes are used, and the 16-bit integer value is represented by the lower 2 bytes.

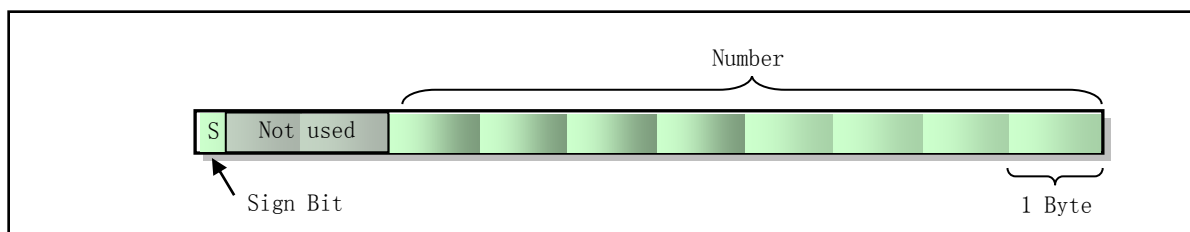


Figure 11-1 Temporary integer format supported by the emulator

## 2. BCD data type

The BCD (Binary Coded Decimal) code value is a binary coded decimal value. For compressed BCD code, each byte can represent a two-digit decimal number, where each 4 bits represents a digit of 0-9. For example, the compressed BCD code representation of the decimal number 59 is 0x01011001. For uncompressed BCD

codes, each byte uses only the lower 4 bits to represent a 1-digit decimal number.

The 80387 coprocessor supports the representation and operation of a 10-byte compressed BCD code, which can represent an 18-bit decimal number, as shown in Figure 11-2. Similar to the temporary integer format, where the highest byte uses only the sign bit (most significant bit) to indicate the positive or negative of the value, and the remaining bits are not used. If the BCD code data is negative, the highest valid bit of at the highest byte is used to indicate a negative value. Otherwise all bits of the highest byte are 0.

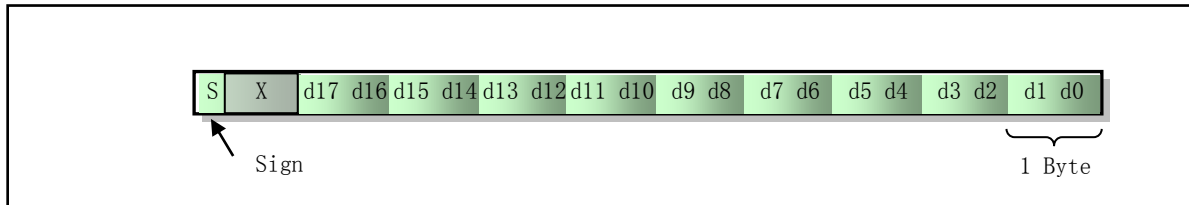


Figure 11-2 BCD code data type supported by 80387

### 3. Floating point data type

A number with an integer part and a fractional (mantissa) part is called a real number or a floating point number. In fact, an integer is a real number with a fraction part of 0 and is a subset of the real set. Since computers use fixed-length bits to represent a number, computers are not able to accurately represent all real numbers. Since the number of bits allocated to the fractional part is not fixed in order to represent the most accurate real value in a fixed length bit when the computer represents a real number, that is, the decimal point can be "floating", so the real data type represented by the computer also known as floating point numbers. In order to facilitate the porting of the program, the floating point number representation specified by IEEE Standard 754 (or 854) is currently used in the computer to represent the real number.

The general format of this real number representation is shown in Figure 11-3. It consists of a Significand part, an Exponent part, and a Sign bit. Significand is composed of an integer 1 and a fraction part. The 80387 coprocessor supports three real types, and the number of bits used in each part is shown in Figure 11-4. With the exception of the 80-bit temporary real (or called extended-real) format, all of these data types exist in memory only. When they are loaded into 80387 coprocessor data registers, they are converted into temporary-real format and operated on in that format.

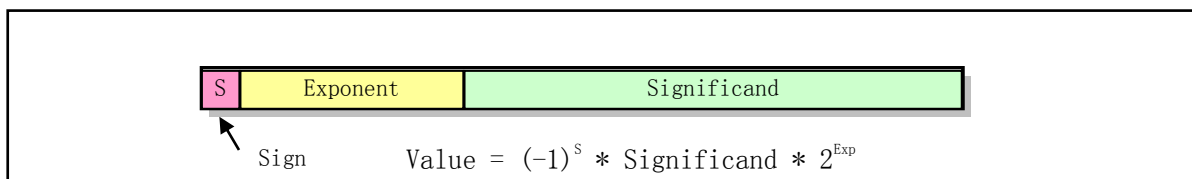


Figure 11-3 Floating-point general format

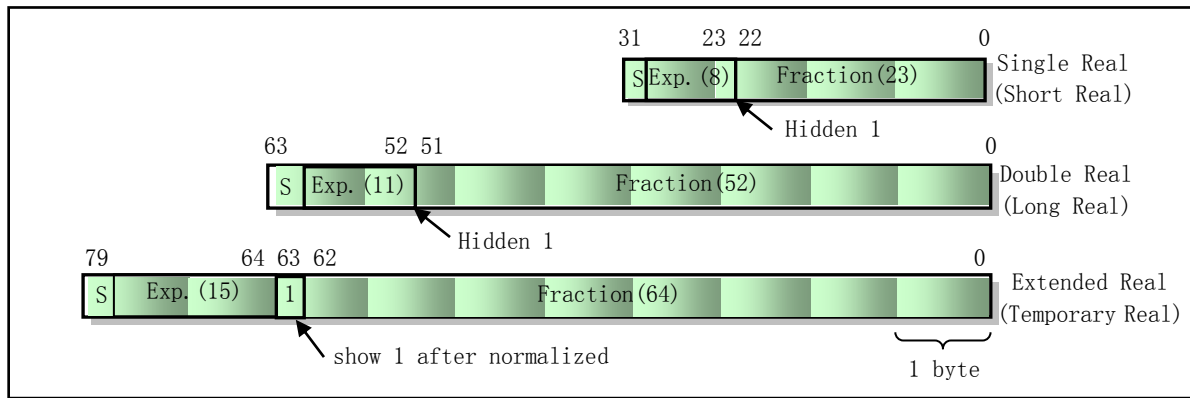


Figure 11-4 Real number format used by the 80387 coprocessor

In Figure 11-4, S is a sign bit of one bit. S=1 indicates a negative real number; S=0 indicates a positive real number. The Significand gives the significant digits or mantissa of the real number. When using the exponential form, a real number can be expressed in a variety of forms. For example, the decimal real number 10.34 can be expressed as  $1034.0 \times 10^{-2}$ , or  $10.34 \times 10^0$ , or  $1.034 \times 10^1$  or  $0.1034 \times 10^2$ , and the like. In order for the calculation to get the maximum precision value, we always normalize the real number, that is, adjust the exponent value of the real number so that the binary most significant value is always 1, and the decimal point is on the right side. Therefore, the correct normalization result of the above example is  $1.034 \times 10^1$ . For binary numbers it is  $1.XXXXX \times 2^N$  (where X is 1 or 0). If we always use this form to represent a real number, then the left of the decimal point must be 1. So in the short real (single precision) and long real (double precision) formats of 80387, this '1' does not need to be explicitly expressed. Therefore, in a binary real number of a short real number or a long real number, 0x0111...010 is actually 0x1.0111...010.

The exponent field in the format contains the power of 2 required to represent a number as a normalized form. As mentioned earlier, in order to facilitate comparison of the size of the numbers, the 80387 uses the biased number form to store the exponent value. The biased bases of the short real number, long real number, and the extended real number (temporary real number) are 127, 1023, and 16383, respectively, and the corresponding hexadecimals are 0x7F, 0x3FF, and 0x3FFF, respectively. Therefore, a short real exponent value of 0b10000000 actually represents the  $2^1$  ( $0b01111111 + 0b00000001$ ).

In addition, the temporary real number is the format of the number represented by the 80387 internal operation. Its most significant number 1 is explicitly placed at bit 63, and regardless of the data type you are giving (for example, integer, short real or BCD code, etc.), 80387 will convert it to this temporary real number format. The purpose of the 80387 is to maximize accuracy and minimize overflow exceptions during the operation. Explicitly indicating 1 is because 80387 does need this bit during the operation (used to represent very small values). When a short or long real number entered into 80387 is converted to a temporary real number format, a 1 is explicitly placed at bit 63.

During the processing, the program defines a data structure for the temporary real number in the linux/math\_emu.h file:

```
62 typedef struct {
63     long a,b;          // a is lower 32 bits, b is upper 32 bits (including 1 fixed bit).
64     short exponent;
65 } temp_real;
```

Among them, the 64-bit mantissa is represented by two long variables. The variable *a* is the lower 32 bits, and *b* is the upper 32 bits (including the 1 fixed bit). In addition, in order to solve the problem of byte alignment of the gcc compiler in the data structure at the time, another structure with the same function is defined.

---

```

67 typedef struct {
68     short m0, m1, m2, m3;
69     short exponent;
70 } temp_real_unaligned;

```

---

#### 4. Special real numbers

Similar to the case where some values in the above table cannot be represented, some values expressed in real numbers also have special meaning. For temporary real numbers in the 80-bit length format, 80387 does not use all of the range values it can represent. Table 11-2 is all possible values that can be represented by the temporary real number in use of 80387, where the 1 bit of the left side of the dashed line of the significant number column indicates the bit 63 of the temporary real number, that is, the bit of the value 1 is explicitly indicated. Short real numbers and long real numbers do not have this bit, so there are no pseudo denormalized categories in the table. Let's take a look at some of the special values: zero, infinity, denormalized, pseudo-normalized, and the signalling NaN (Not a Number) and quiet NaN.

Table 11-2 The type and range of values that temporary real numbers represent.

Sign	Biased Exponent	Significand		Types
0/1	11...11	1	11...11	NaNs - QNaNs
0/1	11...11	1	...	Quiet NaNs
0/1	11...11	1	10...00	Indefinite
0/1	11...11	1	01...11	Signalling NaNs - SNaNs
0/1	11...11	1	...	
0/1	11...11	1	00...01	
0/1	11...11	1	00...00	Infinite
0/1	11...10	1	11...11	Normals
0/1	...	1	...	
0/1	00...01	1	00...00	
0/1	00...00	1	11...11	Pseudo-Denormals
0/1	00...00	1	...	
0/1	00...00	1	00...00	
0/1	00...00	0	11...11	Denormals
0/1	00...00	0	...	
0/1	00...00	0	00...01	
0/1	00...00	0	00...00	Zero

Zero is the value where the exponent and the significand are both 0, and the remaining exponents with a value of 0 are reserved, that is, the value of the exponent of 0 cannot represent a normal real value. Infinite are values where the exponent value is all 1, the significand value is all zeros, and all remaining values with



exponent values of 0x11...11 are also reserved.

The denormals number is a special class value used to represent very small values. It can indicate a progressive underflow or a loss of progressive accuracy. The value is usually required to be represented as a normalized number (left shift until the most significant bit of the significant number is bit 1), whereas the most significant digit of the non-normalized number is not 1. At this time, the biased exponent 0x00...00 is a special representation of the short real number, the long real number, and the temporary real number exponent value of  $2^{-126}$ ,  $2^{-1022}$ ,  $2^{-16382}$ , respectively. This representation is special because the biased exponent value 0x00...01 also represents the same exponent values  $2^{-126}$ ,  $2^{-1022}$ ,  $2^{-16382}$  for the three real types, respectively.

The pseudo-denormals class value is the value of the most significant bit of the significand is 1, and the bit of the denormals class value is 0. Pseudo-denormal numbers are rare, they can be represented by normalized class numbers but not. Since it has been explained above that the special biased exponent 0x00...00 has the same value as the exponent 0x00...01 of the normalized number, the pseudo denormalized class number can be expressed as a normalized class value.

Another special case is NaN (Not a Number). NaN comes in two forms: it produces signals (Signaling NaN) and does not produce signals or is called Quiet NaN. An invalid operation exception is thrown when a signalling NaN (SNaN) is used for operation, while a quiet NaN (QNaN) does not. SNaN can be used to determine that all variables have been initialized before use. The method is that the program can initialize the variables to SNaN values, so that if an uninitialized value is used during the operation, an exception is thrown. Of course, NaN class values can also be used to store other information.

80387 itself does not produce a value of the SNaN class, but it will produce a value of the QNaN class. When a invalid operation exception occurs, 80387 will generate a QNaN class value, and the result of the operation will be an indefinite type value. The indefinite value is a special QNaN class value. Each data type has a number that represents an indefinite value. For integers, the largest negative number is used to represent its indefinite.

There are also some temporary real values that are not supported by 80387, that is, those that are not listed in the above table. If 80387 encounters these unsupported values, an invalid operation exception will be thrown.

### 11.1.2 Math Coprocessor Function and Structure

Although the 80386 is a general-purpose microprocessor, its instructions are not very suitable for mathematical calculations. Therefore, if you use 80386 to perform mathematical calculations, you need to compile very complex programs, and the execution efficiency is relatively low. As an auxiliary processing chip of the 80386, the 80387 greatly expands the programmer's programming range. What the programmers were not likely to do before, using the coprocessor, can be done easily and quickly and accurately.

The 80387 has a special set of registers that allow the 80387 to operate directly on orders of magnitude larger or smaller than the 80386 can handle. The 80386 uses a binary complement to represent a number. This method is not suitable for representing decimals. The 80387 does not use the 2's complement method to represent the value. It uses the 80-bit (10-byte) format specified by IEEE Standard 754. This format not only has broad compatibility, but also the ability to represent large (or small) values in binary. For example, it can represent values as large as  $1.21 \times 10^{4932}$  and can handle numbers as small as  $3.3 \times 10^{-4932}$ . The 80387 does not maintain a fixed decimal point position. If the value is small, it uses more decimal places; if the value is large, it uses a few decimal places. Therefore the position of the decimal point can be "floating". This is also the origin of the term "floating point".

To support floating point operations, the 80387 contains three sets of registers, as shown in Figure 11-5. (1) Eight 80-bit data registers (accumulators) that can be used to temporarily store eight floating-point operands, and that these accumulators can perform stack operations; (2) Three 16-bit status and control registers: a status word register SWD, a control word register CWD, and a feature (TAG) register; (3) Four 32-bit error pointer registers (FIP, FCS, FOO, and FOS) are used to determine the instruction and memory operands that caused the 80387 internal exception.

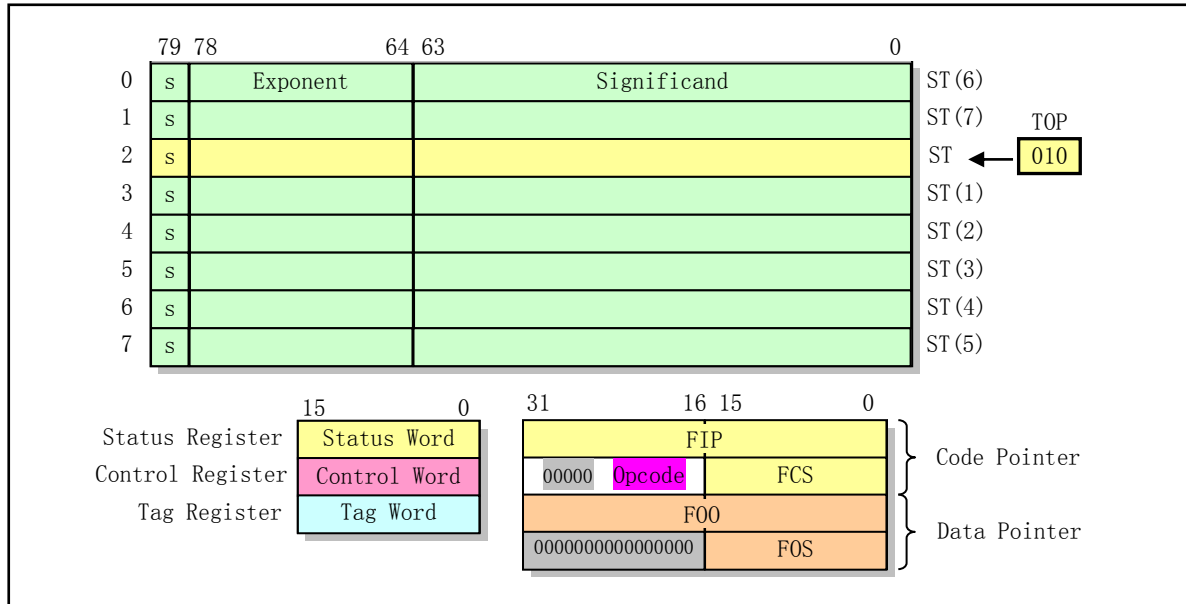


Figure 11-5 80387 registers

#### 1. Stack floating point accumulator

During the execution of floating-point instructions, eight 80-bit physical register sets are used as stack accumulators. Although each 80-bit register has a fixed physical order position (ie 0--7 on the left), the current top of the stack is indicated by ST (ie ST(0)). The remaining accumulators under ST are indicated by the name ST(i) (i = 1 -- 7). As for which 80-bit physical register is the current stack top ST, it is specified by the specific operation process. The 3-bit field named TOP in the status word register (shown in Figure 11-6) contains the absolute position of the 80-bit physical register corresponding to the current top ST. A push or load operation will decrement the TOP field value by one and store the new value in the new ST. After the push operation, the original ST becomes ST(1), and the original ST(7) becomes the current ST. That is, the names of all accumulators have changed from the original ST(i) to ST((i+1)&0x7). A pop or store operation will read the value from the current TOP register ST and store it in memory, and increment the TOP field value by 1. Therefore, after the pop operation, the original ST (ie ST(0)) becomes ST(7), and the original ST(1) becomes the new ST. That is, the names of all accumulators are changed from the original ST(i) to ST((i-1)& 0x7).

ST acts like an accumulator because it is used as an implicit operand for all floating point instructions. If there is another operand, then the second operand can be one of any remaining accumulators ST(i), or a memory operand. Each accumulator in the stack provides an 80-bit space stored in a temporary real format for a real number, with the most significant bit being the sign bit, bits 78--64 being the 15-bit exponent field, and bits 63--0 being 64-bit valid number field.

Floating-point instructions are designed to take full advantage of this accumulator stack mode. A floating-point load instruction (FLD, etc.) reads an operand from memory and pushes it onto the stack, while a

floating-point store instruction takes a value from the current top of the stack and writes it to memory. If the value in the stack is no longer needed, the pop operation can also be performed at the same time. Operations such as summation and multiplication will take the contents of the current ST register as one operand and the other from other registers or memory, and save the result in ST after the calculation. There is also a type of "operate and pop-up" operation for computing between ST and ST(1). This form of operation performs a pop-up and then places the result in a new ST.

## 2. Status and Control Register

Three 16-bit registers (TAG words, control words, and status words) control the operation of floating point instructions and provide status information for them. Their specific format is shown in Figure 11-6, which will be explained one by one below.

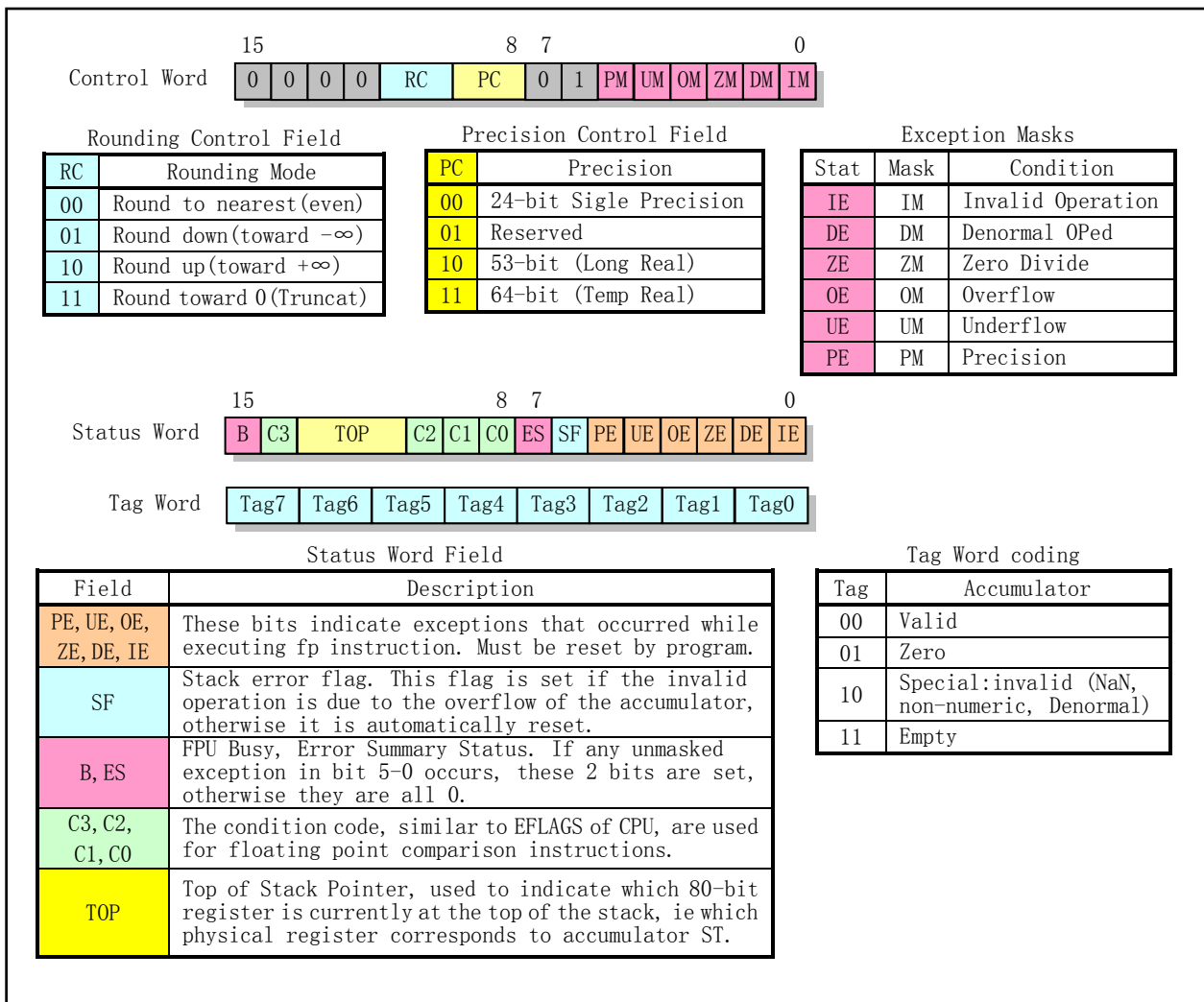


Figure 11-6 Control, Status, and Tag Register Format

### A. Control Word

The 16-bit Control Word can be used to program various processing options to control the operation of the 80387. It can be divided into three parts: (1) RC (Rounding Control) of bits 11--10 is a rounding control field for rounding the calculation result; (2) The PC (Precision Control) of bits 9-8 is a precision control field for adjusting the accuracy of the calculation result before saving to the specified memory

location. All other operations use temporary real format precision, or use the precision specified by the instruction; (3) Bit 5--0 is the exception mask bit used to control coprocessor exception handling. These 6 bits correspond to the six anomalies that may occur in 80387, each of which can be masked separately. If a particular exception occurs and its corresponding mask bit is not set, then 80387 will notify the CPU of the exception and will cause the CPU to generate an exception int 16. However, if the corresponding mask bit is set, then 80387 will handle and correct the abnormal problem itself without notifying the CPU. This register can be read and written at any time. The specific meaning of each bit is shown in Figure 11-6.

#### B. Status Word

During operation, the 80387 sets the bits in the Status Word for the program to detect specific conditions. When an exception occurs, it lets the CPU determine the cause of the exception. Because all six exceptions of the coprocessor will cause the CPU to generate the same exception int16.

#### C. Tag Word

The Tag Word register contains eight 2-bit Tag fields for eight physical floating point data registers. These tag fields indicate that the corresponding physical register contains a valid, zero, or special floating point value, respectively, or is empty. Special values are those that are infinite, non-numeric, denormalized, or unsupported. The tag field can be used to detect the overflow of the accumulator stack. If the push operation decrements TOP and points to a non-empty register, an overflow on the stack occurs. If a pop operation attempts to read or pop an empty register, it will cause an underflow of the stack. Both the overflow and underflow of the stack will throw an invalid operation exception.

### 3. Error-Pointer Register

The error-pointer register is a four 32-bit 80387 register containing 80387 pointers to the last executed instruction and the data used, as shown in Figure 11-6. The first two registers FIP (FPU Instruction Pointer) and FCS (FPU Code Selector) are pointers to the two opcodes in the last executed instruction (ignoring the prefix code). FCS is the segment selector and opcode, and FIP is the intra-segment offset. The last two registers, FOO (FPU Operand Offset) and FOS (FPU Operand Selector), are the last pointers to the instruction memory operands. In FOS is the segment selector, and FOO is the intra-segment offset value. If the last executed coprocessor instruction does not contain a memory operand, the last two register values are useless. The instructions FLDENV, FSTENV, FNSTENV, FRSTOR, FSAVE, and FNSAVE are used to load and store the contents of these four registers. The first three instructions load or store a total of 28 bytes of content: control word, status word and feature word, and four error pointer registers. The control word, status word, and feature word are all operated in 32 bits, and the upper 16 bits are 0. The last three instructions are used to load or store all 108 bytes of register contents of the coprocessor.

### 4. Floating-point instruction format

The simulation of the coprocessor is to analyze the specific floating-point instruction opcode and operands, and use the 80386 ordinary instructions to perform the corresponding emulation operations according to the structure of each instruction. There are more than 70 instructions in the math coprocessor 80387, which are divided into 5 categories, as shown in Table 11-3. The operation code of each instruction has 2 bytes, and the upper 5 bits of the first byte are binary 11011. The 5-bit value (0x1b or decimal 27) is exactly the ASCII code value of the character ESC (escape), so all math coprocessor instructions are visually referred to as ESC escape instructions. The same ESC bit can be ignored when simulating floating-point instructions, as long as the value of the lower 11 bits is determined.

Table 11-3 Floating-point instruction type

	1st Byte	2nd Byte	Option Fields
--	----------	----------	---------------

1	1 1 0 1 1	OPA		1	MOD		1	OPB		R/M			SIB	DISP
2	1 1 0 1 1	MF		OPA	MOD		OPB			R/M			SIB	DISP
3	1 1 0 1 1	d	P	OPA	1	1	OPB			ST(i)				
4	1 1 0 1 1	0	0	1	1	1	1	OP						
5	1 1 0 1 1	0	1	1	1	1	1	OP						
	15 -- 11	10	9	8	7	6	5	4	3	2	1	0		

The meanings of the fields in the table are as follows (refer to the 80X86 processor manual for the specific meaning and detailed description of these fields):

- OP (Operation opcode) is the instruction opcode. In some instructions, it is divided into two parts: OPA and OPB.
- MF (Memory Format) is a memory format: 00-32-bit real number; 01-32-bit integer; 10-64-bit real number; 11-64-bit integer.
- P(Pop) indicates whether to perform a popup process after the operation: 0 - no need; 1 - pop up the stack after the operation.
- d (destination) indicates the accumulator that saves the result of the operation: 0 - ST(0); 1 - ST(i).
- MOD (Mode) and R/M (Register/Memory) are the operation mode field and the operand position field.
- SIB (Scale Index Base) and DISP (Displacement) are optional follow-up fields with MOD and R/M field instructions.

In addition, all assembly language mnemonics for floating point instructions start with the letter F, for example: FADD, FLD, and so on. There are also some standard representations as follows:

- FI All instructions for operating integer data start with FI, such as FIADD, FILD, etc.
- FB All instructions for operating BCD type data start with FB, such as FBLD, FBST, etc.
- FxxP All instructions that perform a pop operation are terminated with the letter P, such as FSTP, FADDP, etc.
- FxxPP All instructions that perform two pop operations are terminated with the letter PP, such as FCOMPP, FUCOMPP, etc.;
- e) FNxx Except for instructions starting with FN, all instructions will detect unmasked operation exceptions before execution. Instructions that start with FN do not detect arithmetic anomalies, such as FNINIT, FNSAVE, and so on.

## 11.2 math-emulation.c

### 11.2.1 Function

All functions in the math\_emulate.c program can be divided into three categories: the first class is "device does not exist" exception handler interface function math\_emulate(), this class has only one such function; The second type is the floating-point instruction emulation processing main function do\_emu(), and this class only has this one function; In addition, all functions are simulation operation auxiliary functions, including functions in the other C programs.

In a PC that does not include a 80387 coprocessor chip, if the emulation flag EM = 1 is set in CR0 during kernel initialization, then the CPU will generate an exception int7 when the CPU encounters a floating-point

instruction, and the `math_emulate(long __false)` function at line 476 in this program is called during the exception processing.

In the `math_emulate()` function, if it is judged that the current process has not used the co-processing operation of the simulation, the 80387 control word, the status word and the feature word (Tag Word) of the simulation are initialized, and all 6 coprocessor exception mask bits in the control word are set, and the status word and the tag word are reset, and then call the simulation processing main function `do_emu()`. The parameters used when calling are the return address pointers that call the `math_emulate()` function during interrupt processing as follows. The info structure is actually a structure of some data in the stack that has been gradually pushed onto the stack since the CPU generated the interrupt `int7`, so it is basically the same as the distribution of data in the kernel stack when the system-call is executed. See line 11 of the `include/linux/math_emu.h` file and the beginning of `kernel/sys_call.s`.

---

```
11 struct info {
12     long __math_ret;           // caller (int7) return address.
13     long __orig_eip;          // a place where the original EIP is temporarily saved.
14     long __edi;               // registers pushed on stack during int7 processing.
15     long __esi;
16     long __ebp;
17     long __sys_call_ret;      // process system-call's return code.
18     long __eax;               // below are the same as stacks of the system-call.
19     long __ebx;
20     long __ecx;
21     long __edx;
22     long __orig_eax;          // its -1 if not a system-call.
23     long __fs;
24     long __es;
25     long __ds;
26     long __eip;               // pushed by CPU automatically.
27     long __cs;
28     long __eflags;
29     long __esp;
30     long __ss;
31 };
```

---

The `do_emu()` function (line 52) first determines whether there is a coprocessor internal exception for the simulation based on the status word. If there is one, set the busy bit B (bit 15) of the status word, otherwise reset the busy bit B. Then, the two-byte floating-point instruction code that generates the coprocessor exception is obtained from the EIP field in the above info structure, and is used for performing software simulation operation processing after masking the ESC code (binary 11011) bit portion therein. For ease of processing, the function uses five switch statements to process five types of floating point instruction codes. For example, the first switch statement (line 75) is used to handle floating-point instructions that do not involve addressing memory operands, while the last two switch statements (lines 419, 432) are dedicated to handling instructions with operands related to memory. For the latter type of instruction, the basic flow of the process is to first obtain the effective address of the memory operand according to the addressing mode byte in the instruction code, and then read the corresponding data from the effective address (integer, Real number or BCD code value). The read value is then converted to a temporary real number format used by the 80387 internal processing. After the calculation is completed, the value of the temporary real number format is converted into the original data type, and finally saved to the user data area.

In addition, when simulating a specific floating point instruction, if the floating point instruction is found to be invalid, the program will immediately call the abandon execution function `__math_abort()`. This function will send the specified signal to the current process, and modify the stack pointer `esp` to point to the return address (`__math_ret`) of the `math_emulate()` function in the interrupt process, and immediately return to the interrupt handler.

## 11.2.2 Code annotation

Program 11-1 linux/kernel/math/math\_emulate.c

---

```

1  /*
2  * linux/kernel/math/math_emulate.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7  /*
8  * Limited emulation 27.12.91 - mostly loads/stores, which gcc wants
9  * even for soft-float, unless you use bruce evans' patches. The patches
10 * are great, but they have to be re-applied for every version, and the
11 * library is different for soft-float and 80387. So emulation is more
12 * practical, even though it's slower.
13 *
14 * 28.12.91 - loads/stores work, even BCD. I'll have to start thinking
15 * about add/sub/mul/div. Urgel. I should find some good source, but I'll
16 * just fake up something.
17 *
18 * 30.12.91 - add/sub/mul/div/com seem to work mostly. I should really
19 * test every possible combination.
20 */
21
22 /*
23 * This file is full of ugly macros etc: one problem was that gcc simply
24 * didn't want to make the structures as they should be: it has to try to
25 * align them. Sickening code, but at least I've hidden the ugly things
26 * in this one file: the other files don't need to know about these things.
27 *
28 * The other files also don't care about ST(x) etc - they just get addresses
29 * to 80-bit temporary reals, and do with them as they please. I wanted to
30 * hide most of the 387-specific things here.
31 */
32
33 // <signal.h> Signal header file. Define signal symbol constants, signal structures, and
34 //     signal manipulation function prototypes.
35 // <linux/math_emu.h> Coprocessor emulation header file. A coprocessor data structure and
36 //     a floating point representation structure are defined.
37 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
38 //     commonly used functions of the kernel.
39 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined
40 //     for segment register operations.
41 #include <signal.h>

```

```

34
35 #define ALIGNED_TEMP_REAL 1
36 #include <linux/math_emu.h>
37 #include <linux/kernel.h>
38 #include <asm/segment.h>
39
40 #define bswapw(x) __asm__("xchgb %%al, %%ah": "=a" (x): "" ((short)x)) // exchange 2 bytes.
41 #define ST(x) (*__st((x))) // get simulated ST(x) value.
42 #define PST(x) ((const temp_real *) __st((x))) // get pointer to the simulated ST(x).
43
44 /*
45  * We don't want these inlined - it gets too messy in the machine-code.
46  */
47 // The following are simulation functions for floating-point instructions of the same name.
48 static void fpop(void);
49 static void fpush(void);
50 static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b);
51 static temp_real_unaligned * __st(int i);
52
53 // Perform floating-point instruction simulation.
54 // The function first checks if there is an unmasked exception flag set in the simulated I387
55 // structure status word register, and if so, sets the busy flag B in the status word. Then
56 // save the instruction pointer, and take out the 2-byte floating-point instruction code code
57 // at the pointer EIP, and then analyze the code and process it according to its meaning. For
58 // different code types, Linus uses several different switch blocks for simulation processing.
59 // The argument is a pointer to the info structure.
60 static void do_emu(struct info * info)
61 {
62     unsigned short code;
63     temp_real tmp;
64     char * address;
65
66     // The function first checks if the exception flag is set in the status word register, and if
67     // so, sets the busy flag B (bit 15) in the status word, otherwise resets the B flag. Then we
68     // save the original instruction pointer. Then, let's see if the code that executes this function
69     // is user code. If not (ie the caller's code segment selector is not equal to 0x0f), then there
70     // is code in the kernel that uses floating point instructions, but this is not allowed. The
71     // kernel then shuts down after displaying the CS, EIP, and "Needs Mathematical Simulation in
72     // the Kernel" information for the floating-point instructions.
73     if (I387.cwd & I387.swd & 0x3f)
74         I387.swd |= 0x8000; // sets the busy flag B.
75     else
76         I387.swd &= 0x7fff; // clear the flag B.
77     ORIG_EIP = EIP; // save the original EIP.
78     /* 0x0007 means user code space */
79     if (CS != 0x000F) { // if not in user code space, STOP.
80         printk("math_emulate: %04x:%08x\n|r", CS, EIP);
81         panic("Math emulation needed in kernel");
82     }
83
84     // Then we get the 2-byte floating-point instruction code at the pointer EIP. Since the Intel
85     // CPU stores data in the "Little endien" style, the code fetched at this time is exactly reversed
86     // from the first and second bytes of the instruction. So we need to swap the order of the two
87     // bytes in 'code'. Then mask out the ESC bit in the first code byte (binary 11011).

```



```

// Next, the floating point instruction pointer EIP is saved to the fip field in the TSS segment
// i387 structure, and the CS is saved to the fcs field, and the slightly processed floating
// point instruction code is placed in the upper 16 bits of the fcs field. These values are
// saved so that the program can be processed like a real coprocessor in the event of a simulated
// processor exception. Finally, let the EIP point to the subsequent floating point instruction
// or operand.
68     code = get\_fs\_word((unsigned short *) EIP); // get 2 bytes floating-point code.
69     bswapw(code); // exchange bytes.
70     code &= 0x7ff; // Mask the ESC part in the code.
71     I387.fip = EIP; // save EIP, code selector and code.
72     *(unsigned short *) &I387.fcs = CS;
73     *(1+(unsigned short *) &I387.fcs) = code;
74     EIP += 2; // point to next instruction code.

// Then we analyze the code and process it according to its meaning. For different code type
// values, Linus uses several different switch blocks for processing.
// (1) First, if the instruction opcode has a fixed code value (independent of the register
// or the like), it is processed below.
// The macro math_abort() is used to terminate the coprocessor emulation operation, defined
// in the file linux/math_emu.h, line 52. The actual implementation code is at line 488 of the
// program. See the description before line 50 of the linux/math_emu.h file.
75     switch (code) {
76         case 0x1d0: /* fnop */ // like nop.
77             return;
78         case 0x1d1: case 0x1d2: case 0x1d3: // invalid code, send signal and exit.
79         case 0x1d4: case 0x1d5: case 0x1d6: case 0x1d7:
80             math\_abort(info, 1<<(SIGILL-1));
81         case 0x1e0: // FCHS - Change the ST sign bit: ST = -ST.
82             ST(0).exponent ^= 0x8000;
83             return;
84         case 0x1e1: // FABS - get absolute value. ST = |ST|.
85             ST(0).exponent &= 0x7fff;
86             return;
87         case 0x1e2: case 0x1e3: // invalid code. send signal and exit.
88             math\_abort(info, 1<<(SIGILL-1));
89         case 0x1e4: // FTST - Test TS and set Cn in status word.
90             ftst(PST(0));
91             return;
92         case 0x1e5: // FXAM - Check TS and modify Cn in status word.
93             printk("fxam not implemented\n\r");
94             math\_abort(info, 1<<(SIGILL-1));
95         case 0x1e6: case 0x1e7: // invalid code. send signal and exit.
96             math\_abort(info, 1<<(SIGILL-1));
97         case 0x1e8: // FLD1 - Load constant 1.0 to accumulator ST.
98             fpush();
99             ST(0) = CONST1;
100            return;
101         case 0x1e9: // FLDL2T - Load constant Log2(10) to ST.
102             fpush();
103             ST(0) = CONSTL2T;
104             return;
105         case 0x1ea: // FLDL2E - Load constant Log2(e) to ST.
106             fpush();

```

```

107         ST(0) = CONSTL2E;
108         return;
109     case 0x1eb: // FLDPI - Load constant Pi to ST.
110         fpush();
111         ST(0) = CONSTPI;
112         return;
113     case 0x1ec: // FLDLG2 - Load constant Log10(2) to ST.
114         fpush();
115         ST(0) = CONSTLG2;
116         return;
117     case 0x1ed: // FLDLN2 - Load constant Loge(2) to ST.
118         fpush();
119         ST(0) = CONSTLN2;
120         return;
121     case 0x1ee: // FLDZ - Load constant 0.0 to ST.
122         fpush();
123         ST(0) = CONSTZ;
124         return;
125     case 0x1ef: // invalid code. send signal and exit.
126         math\_abort(info, 1 << (SIGILL-1));
127     case 0x1f0: case 0x1f1: case 0x1f2: case 0x1f3:
128     case 0x1f4: case 0x1f5: case 0x1f6: case 0x1f7:
129     case 0x1f8: case 0x1f9: case 0x1fa: case 0x1fb:
130     case 0x1fc: case 0x1fd: case 0x1fe: case 0x1ff:
131         printf("%04x fxxx not implemented\n\r", code + 0xc800);
132         math\_abort(info, 1 << (SIGILL-1));
133     case 0x2e9: // FUCOMPP - no order comparison.
134         fcom(PST(1), PST(0));
135         fpop(); fpop();
136         return;
137     case 0x3d0: case 0x3d1: // FNOP - on 387. !! should be 0x3e0, 0x3e1.
138         return;
139     case 0x3e2: // FCLEX - Clears exception flag in status word.
140         I387.swd &= 0x7f00;
141         return;
142     case 0x3e3: // FINIT - Initializes the coprocessor.
143         I387.cwd = 0x037f;
144         I387.swd = 0x0000;
145         I387.twd = 0x0000;
146         return;
147     case 0x3e4: // FNOP - on 80387.
148         return;
149     case 0x6d9: // FCOMPP - compares ST(1) with ST, pops twice.
150         fcom(PST(1), PST(0));
151         fpop(); fpop();
152         return;
153     case 0x7e0: // FSTSW AX - Saves status word to AX register.
154         *(short *) EAX = I387.swd;
155         return;
156 }

```

// (2) Next, we process the instruction that the last 3 bits of the 2nd byte are REG. That is,  
 // the code in the form of "11011,XXXXXXXX,REG", and the prefix "11011" already be cleared.

// The macro `real_to_real(a, b)` is used for assignment between two temporary reals, defined  
 // in file `linux/math_emu.h`, line 72.

```

157     switch (code >> 3) {
158         case 0x18:                // FADD ST, ST(i)
159             fadd(PST(0), PST(code & 7), &tmp);
160             real\_to\_real(&tmp, &ST(0));
161             return;
162         case 0x19:                // FMUL ST, ST(i)
163             fmul(PST(0), PST(code & 7), &tmp);
164             real\_to\_real(&tmp, &ST(0));
165             return;
166         case 0x1a:                // FCOM ST(i)
167             fcom(PST(code & 7), &tmp);
168             real\_to\_real(&tmp, &ST(0));
169             return;
170         case 0x1b:                // FCOMP ST(i)
171             fcom(PST(code & 7), &tmp);
172             real\_to\_real(&tmp, &ST(0));
173             fpop();
174             return;
175         case 0x1c:                // FSUB ST, ST(i)
176             real\_to\_real(&ST(code & 7), &tmp);
177             tmp.exponent ^= 0x8000;
178             fadd(PST(0), &tmp, &tmp);
179             real\_to\_real(&tmp, &ST(0));
180             return;
181         case 0x1d:                // FSUBR ST, ST(i)
182             ST(0).exponent ^= 0x8000;
183             fadd(PST(0), PST(code & 7), &tmp);
184             real\_to\_real(&tmp, &ST(0));
185             return;
186         case 0x1e:                // FDIV ST, ST(i)
187             fdiv(PST(0), PST(code & 7), &tmp);
188             real\_to\_real(&tmp, &ST(0));
189             return;
190         case 0x1f:                // FDIVR ST, ST(i)
191             fdiv(PST(code & 7), PST(0), &tmp);
192             real\_to\_real(&tmp, &ST(0));
193             return;
194         case 0x38:                // FLD ST(i)
195             fpush();
196             ST(0) = ST((code & 7)+1);
197             return;
198         case 0x39:                // FXCH ST(i)
199             fxchg(&ST(0), &ST(code & 7));
200             return;
201         case 0x3b:                // FSTP ST(i)
202             ST(code & 7) = ST(0);
203             fpop();
204             return;
205         case 0x98:                // FADD ST(i), ST
206             fadd(PST(0), PST(code & 7), &tmp);
207             real\_to\_real(&tmp, &ST(code & 7));

```

```

208         return;
209     case 0x99:                // FMUL ST(i), ST
210         fmul(PST(0), PST(code & 7), &tmp);
211         real\_to\_real(&tmp, &ST(code & 7));
212         return;
213     case 0x9a:                // FCOM ST(i)
214         fcom(PST(code & 7), PST(0));
215         return;
216     case 0x9b:                // FCOMP ST(i)
217         fcom(PST(code & 7), PST(0));
218         fpop();
219         return;
220     case 0x9c:                // FSUBR ST(i), ST
221         ST(code & 7).exponent ^= 0x8000;
222         fadd(PST(0), PST(code & 7), &tmp);
223         real\_to\_real(&tmp, &ST(code & 7));
224         return;
225     case 0x9d:                // FSUB ST(i), ST
226         real\_to\_real(&ST(0), &tmp);
227         tmp.exponent ^= 0x8000;
228         fadd(PST(code & 7), &tmp, &tmp);
229         real\_to\_real(&tmp, &ST(code & 7));
230         return;
231     case 0x9e:                // FDIVR ST(i), ST
232         fdiv(PST(0), PST(code & 7), &tmp);
233         real\_to\_real(&tmp, &ST(code & 7));
234         return;
235     case 0x9f:                // FDIV ST(i), ST
236         fdiv(PST(code & 7), PST(0), &tmp);
237         real\_to\_real(&tmp, &ST(code & 7));
238         return;
239     case 0xb8:                // FFREE ST(i)
240         printk("ffree not implemented\n|r");
241         math\_abort(info, 1 << (SIGILL-1));
242     case 0xb9:                // FXCH ST(i)
243         fxchg(&ST(0), &ST(code & 7));
244         return;
245     case 0xba:                // PST ST(i)
246         ST(code & 7) = ST(0);
247         return;
248     case 0xbb:                // FSTP ST(i)
249         ST(code & 7) = ST(0);
250         fpop();
251         return;
252     case 0xbc:                // FUCOM ST(i)
253         fucom(PST(code & 7), PST(0));
254         return;
255     case 0xbd:                // FUCOMP ST(i)
256         fucom(PST(code & 7), PST(0));
257         fpop();
258         return;
259     case 0xd8:                // FADDP ST(i), ST
260         fadd(PST(code & 7), PST(0), &tmp);

```

```

261         real\_to\_real(&tmp,&ST(code & 7));
262         fpop();
263         return;
264     case 0xd9:                // FMULP ST(i), ST
265         fmul(PST(code & 7),PST(0),&tmp);
266         real\_to\_real(&tmp,&ST(code & 7));
267         fpop();
268         return;
269     case 0xda:                // FCOMP ST(i)
270         fcom(PST(code & 7),PST(0));
271         fpop();
272         return;
273     case 0xdc:                // FSUBRP ST(i), ST
274         ST(code & 7).exponent ^= 0x8000;
275         fadd(PST(0),PST(code & 7),&tmp);
276         real\_to\_real(&tmp,&ST(code & 7));
277         fpop();
278         return;
279     case 0xdd:                // FSUBP ST(i), ST
280         real\_to\_real(&ST(0),&tmp);
281         tmp.exponent ^= 0x8000;
282         fadd(PST(code & 7),&tmp,&tmp);
283         real\_to\_real(&tmp,&ST(code & 7));
284         fpop();
285         return;
286     case 0xde:                // FDIVRP ST(i), ST
287         fdiv(PST(0),PST(code & 7),&tmp);
288         real\_to\_real(&tmp,&ST(code & 7));
289         fpop();
290         return;
291     case 0xdf:                // FDIVP ST(i), ST
292         fdiv(PST(code & 7),PST(0),&tmp);
293         real\_to\_real(&tmp,&ST(code & 7));
294         fpop();
295         return;
296     case 0xf8:                // FFREE ST(i)
297         printk("ffree not implemented\n\r");
298         math\_abort(info,1<<(SIGILL-1));
299         fpop();
300         return;
301     case 0xf9:                // FXCH ST(i)
302         fxchg(&ST(0),&ST(code & 7));
303         return;
304     case 0xfa:                // FSTP ST(i)
305     case 0xfb:                // FSTP ST(i)
306         ST(code & 7) = ST(0);
307         fpop();
308         return;
309 }

```

// (3) Next, we process the code in the form of the second byte 7--6 is MOD, the bit 2--0 is R/M, that is, "11011, XXX, MOD, XXX, R/M". The MOD will be processed in each subroutine, // so first let the code AND 0xe7 (ie 0b11100111) to mask out the MOD.

```

310     switch ((code>>3) & 0xe7) {
311         case 0x22:                // FST - Saves single precision real (short real)
312             put\_short\_real(PST(0), info, code);
313             return;
314         case 0x23:                // FSTP - Saves single precision real (short real)
315             put\_short\_real(PST(0), info, code);
316             fpop();
317             return;
318         case 0x24:                // FLDENV - Load status and control registers, etc.
319             address = ea(info, code); // get efficient address.
320             for (code = 0 ; code < 7 ; code++) {
321                 ((long *) & I387)[code] =
322                     get\_fs\_long((unsigned long *) address);
323                 address += 4;
324             }
325             return;
326         case 0x25:                // FLDCW - Load control word.
327             address = ea(info, code);
328             *(unsigned short *) &I387.cwd =
329                 get\_fs\_word((unsigned short *) address);
330             return;
331         case 0x26:                // FSTENV - Store status and control registers, etc.
332             address = ea(info, code);
333             verify\_area(address, 28);
334             for (code = 0 ; code < 7 ; code++) {
335                 put\_fs\_long((long *) & I387)[code],
336                     (unsigned long *) address);
337                 address += 4;
338             }
339             return;
340         case 0x27:                // FSTCW - Store control word.
341             address = ea(info, code);
342             verify\_area(address, 2);
343             put\_fs\_word(I387.cwd, (short *) address);
344             return;
345         case 0x62:                // FIST - Stores short integer.
346             put\_long\_int(PST(0), info, code);
347             return;
348         case 0x63:                // FISTP - Stores short integer.
349             put\_long\_int(PST(0), info, code);
350             fpop();
351             return;
352         case 0x65:                // FLD - Load an extended (temporary) real number.
353             fpush();
354             get\_temp\_real(&tmp, info, code);
355             real\_to\_real(&tmp, &ST(0));
356             return;
357         case 0x67:                // FSTP - Store the temporary real.
358             put\_temp\_real(PST(0), info, code);
359             fpop();
360             return;
361         case 0xa2:                // FST - Stores double precision (long) real.
362             put\_long\_real(PST(0), info, code);

```

```

363         return;
364     case 0xa3:          // FSTP - Stores double precision (long) real.
365         put\_long\_real(PST(0), info, code);
366         fpop();
367         return;
368     case 0xa4:          // FRSTOR - Restores all 108 bytes register contents.
369         address = ea(info, code);
370         for (code = 0 ; code < 27 ; code++) {
371             ((long *) & I387)[code] =
372                 get\_fs\_long((unsigned long *) address);
373             address += 4;
374         }
375         return;
376     case 0xa6:          // FSAVE - Saves all 108 bytes register contents.
377         address = ea(info, code);
378         verify\_area(address, 108);
379         for (code = 0 ; code < 27 ; code++) {
380             put\_fs\_long((long *) & I387[code],
381                 (unsigned long *) address);
382             address += 4;
383         }
384         I387.cwd = 0x037f;
385         I387.swd = 0x0000;
386         I387.twd = 0x0000;
387         return;
388     case 0xa7:          // FSTSW - Store status word.
389         address = ea(info, code);
390         verify\_area(address, 2);
391         put\_fs\_word(I387.swd, (short *) address);
392         return;
393     case 0xe2:          // FIST - Stores short integer.
394         put\_short\_int(PST(0), info, code);
395         return;
396     case 0xe3:          // FISTP - Stores short integer.
397         put\_short\_int(PST(0), info, code);
398         fpop();
399         return;
400     case 0xe4:          // FBLD - Loads the number of BCD type.
401         fpush();
402         get\_BCD(&tmp, info, code);
403         real\_to\_real(&tmp, &ST(0));
404         return;
405     case 0xe5:          // FILD - Loads a long integer.
406         fpush();
407         get\_longlong\_int(&tmp, info, code);
408         real\_to\_real(&tmp, &ST(0));
409         return;
410     case 0xe6:          // FBSTP - Stores number of BCD type.
411         put\_BCD(PST(0), info, code);
412         fpop();
413         return;
414     case 0xe7:          // BISTP - Stores a long integer.
415         put\_longlong\_int(PST(0), info, code);

```

```

416         fpop();
417         return;
418     }
    // (4) The second type of floating point instructions are processed below. First, the number
    // of the specified type is taken according to the MF of the bit 10--9 of the instruction code,
    // and then separately processed according to the combined value of the OPA and the OPB. That
    // is, the instruction code in the form of "11011, MF, 000, XXX, R/M" is processed.
419     switch (code >> 9) {
420         case 0:                // MF = 00, short real (32-bit real).
421             get\_short\_real(&tmp, info, code);
422             break;
423         case 1:                // MF = 01, short integer (32-bit integer).
424             get\_long\_int(&tmp, info, code);
425             break;
426         case 2:                // MF = 10, long real (64-bit real).
427             get\_long\_real(&tmp, info, code);
428             break;
429         case 4:                // MF = 11, a long integer (64-bit). should be 'case 3' !
430             get\_short\_int(&tmp, info, code);
431     }
    // (5) Process the OPB code in the second byte of the floating point instruction.
432     switch ((code>>3) & 0x27) {
433         case 0:                // FADD
434             fadd(&tmp, PST(0), &tmp);
435             real\_to\_real(&tmp, &ST(0));
436             return;
437         case 1:                // FMUL
438             fmul(&tmp, PST(0), &tmp);
439             real\_to\_real(&tmp, &ST(0));
440             return;
441         case 2:                // FCOM
442             fcom(&tmp, PST(0));
443             return;
444         case 3:                // FCOMP
445             fcom(&tmp, PST(0));
446             fpop();
447             return;
448         case 4:                // FSUB
449             tmp.exponent ^= 0x8000;
450             fadd(&tmp, PST(0), &tmp);
451             real\_to\_real(&tmp, &ST(0));
452             return;
453         case 5:                // FSUBR
454             ST(0).exponent ^= 0x8000;
455             fadd(&tmp, PST(0), &tmp);
456             real\_to\_real(&tmp, &ST(0));
457             return;
458         case 6:                // FDIV
459             fdiv(PST(0), &tmp, &tmp);
460             real\_to\_real(&tmp, &ST(0));
461             return;
462         case 7:                // FDIVR
463             fdiv(&tmp, PST(0), &tmp);

```



```

464         real\_to\_real(&tmp, &ST(0));
465         return;
466     }
    // Process the instruction code of the form "11011, XX, 1, XX, 000, R/M".
467     if ((code & 0x138) == 0x100) {           // FLD, FILD
468         fpush();
469         real\_to\_real(&tmp, &ST(0));
470         return;
471     }
    // The rest are invalid instructions.
472     printk("Unknown math-insns: %04x:%08x %04x\n\r", CS, EIP, code);
473     math\_abort(info, 1<<(SIGFPE-1));
474 }
475
    // The 80387 emulation interface function called in exception Interrupts int7.
    // If the current process has not used the coprocessor, set the use of the coprocessor flag
    // used_math, and then initialize the 80387 control word, status word and tag word. Finally,
    // use the return address when int 7 invokes this function as a parameter to call the floating
    // point instruction emulation function do_emu(). The parameter ___false is _orig_eip.
476 void math\_emulate(long ___false)
477 {
478     if (!current->used_math) {
479         current->used_math = 1;
480         I387.cwd = 0x037f;
481         I387.swd = 0x0000;
482         I387.twd = 0x0000;
483     }
484     /* &___false points to info->__orig_eip, so subtract 1 to get info */
485     do\_emu((struct info *) (&___false) - 1));
486 }
487
    // Terminate the simulation.
    // When an invalid instruction code or an unimplemented instruction is processed, the function
    // first restores the original EIP of the program and then sends a specified signal to the current
    // process. Finally, the stack pointer is pointed to the return address when int7 invokes this
    // function, and directly returns to the interrupt. This function will be used in the macro
    // math_abort(). See the linux/math_emu.h file at line 50 for more instructions.
488 void math\_abort(struct info * info, unsigned int signal)
489 {
490     EIP = ORIG\_EIP;
491     current->signal |= signal;
492     __asm__("movl %0, %%esp ; ret"::"g" ((long) info));
493 }
494
    // Accumulator stack pop-up operation.
    // Increase the TOP field of the status word by 1 and modulo 7 .
495 static void fpop(void)
496 {
497     unsigned long tmp;
498
499     tmp = I387.swd & 0xffffc7ff;
500     I387.swd += 0x00000800;
501     I387.swd &= 0x00003800;

```

```
502         I387.swd |= tmp;
503     }
504
505     // Accumulator stack Push operation.
506     // The TOP field of status word is decremented by 1 (ie, 7 is added) and modulo 7.
507 static void fpush(void)
508 {
509     unsigned long tmp;
510
511     tmp = I387.swd & 0xffffc7ff;
512     I387.swd += 0x00003800;
513     I387.swd &= 0x00003800;
514     I387.swd |= tmp;
515 }
516
517 // Swap the values of the two accumulator registers.
518 static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b)
519 {
520     temp_real_unaligned c;
521
522     c = *a;
523     *a = *b;
524     *b = c;
525 }
526
527 // Get the ST(i) memory pointer in the I387 structure.
528 // Take the TOP field value in the status word, add the specified physical data register number
529 // and modulo 7, and finally return the pointer corresponding to ST(i).
530 static temp_real_unaligned * __st(int i)
531 {
532     i += I387.swd >> 11;           // Get the TOP field in the status word.
533     i &= 7;
534     return (temp_real_unaligned *) (i*10 + (char *) (I387.st_space));
535 }
```

---

## 11.3 error.c

### 11.3.1 Function

When the coprocessor detects that it has an error, it will notify the CPU via the 80387 chip ERROR pin. The error.c program is used to process the error signal sent by the coprocessor, mainly to execute the math\_error() function.

### 11.3.2 Code annotation

Program 11-2 linux/kernel/math/error.c

---

```
1 /*
2  * linux/kernel/math/error.c
```

```

3  *
4  * (C) 1991 Linus Torvalds
5  */
6
// <signal.h> Signal header file. Define signal symbol constants, signal structures, and
// signal manipulation function prototypes.
// <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.
7 #include <signal.h>
8
9 #include <linux/sched.h>
10
// Coprocessor error handling function called in int16.
// The following code is used to handle the error sent by the coprocessor. It causes 80387 to
// clear all exception flags and busy bits in the status word. If the last task used coprocessor,
// then set the coprocessor error signal flag. After returning, the function will jump to the
// system-call interrupt return place of ret_from_sys_call to continue execution.
11 void math_error(void)
12 {
13     __asm__("fnclex"); // clear all exception flags and busy bit in status word.
14     if (last_task_used_math) // if used coprocessor, set signal flag.
15         last_task_used_math->signal |= 1<<(SIGFPE-1);
16 }
17

```

## 11.4 ea.c

### 11.4.1 Function

The ea.c program is used to calculate the effective address used by the operand when simulating floating-point instructions. In order to analyze the effective address information in an instruction, we must have an understanding of the instruction encoding method. The general encoding format for Intel processor instructions is shown in Figure 11-7.

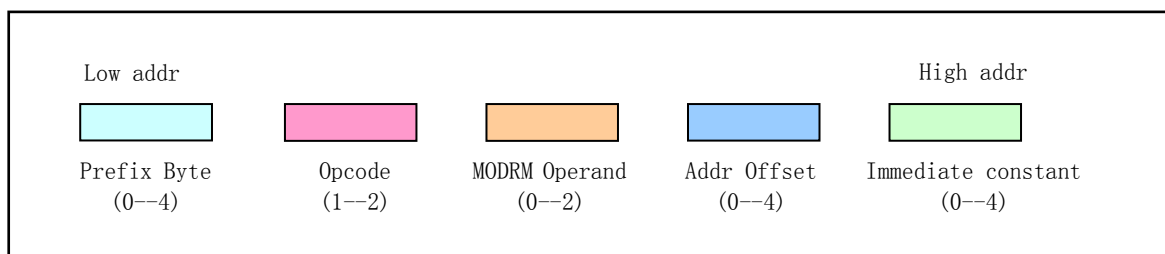


Figure 11-7 General instruction encoding format

As can be seen from the figure, each instruction can have up to 5 fields. The prefix field can consist of 0 to 4 bytes and is used to modify the next instruction. The opcode field is the main field that indicates the operation

of the instruction. Each instruction must have at least 1 byte of operation code. If necessary, the instruction opcode field indicates whether or not to follow a MODRM operand indicator, which is used to explicitly indicate the type and number of operands. For memory operands, the address displacement field is used to give the offset of the operand. The MOD subfield of the MODRM field indicates whether the instruction contains an address offset field and its length. The immediate constant field gives the operand required by the instruction opcode, which is the simplest operand given in the instruction. See the Intel manual for a detailed description of the immediate operands, register operands, and memory operand encoding.

The encoding format of all instructions is shown schematically in Figure 11-8. The figure shows the instruction formats for 10 different encoding methods. Wherein, the symbol OPCode or OPC is an operation code; REG is a register field; the R/M field is used to indicate a register as an operand, or is combined with a MOD field to specify an addressing mode. Some MODR/M encodings require a second addressing byte (called SIB byte) to indicate the addressing mode. This byte has three subfield contents: (1) Scale - scale (S) field specifies the scale factor; (2) index - index (The Idx) field specifies the index register; (3) the Base - Base field specifies the base address register.

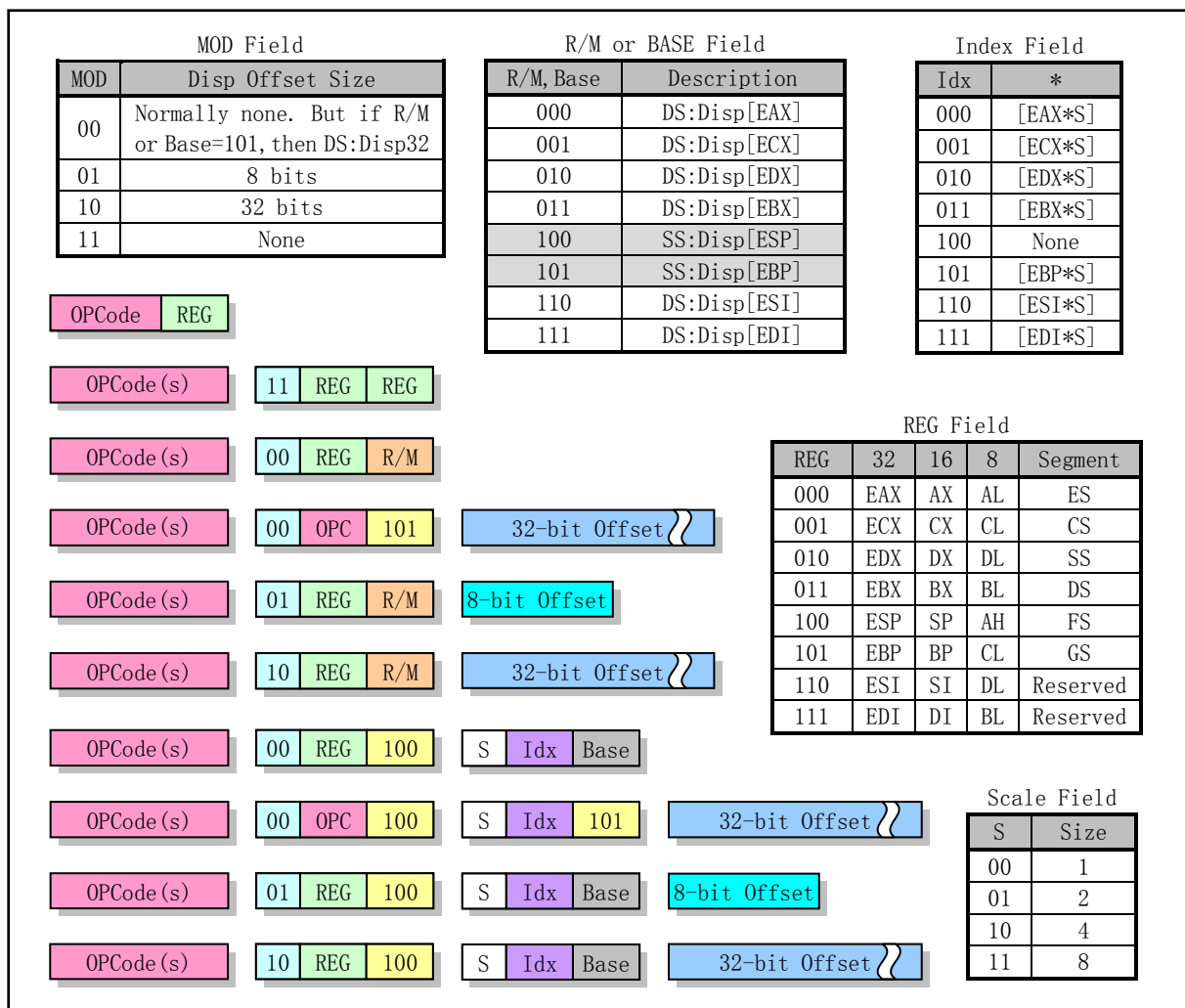


Figure 11-8 Instruction encoding and addressing format summary

The `ea()` function in this program is used to calculate the effective address based on the addressing mode byte in the instruction. It first takes the MOD field and the R/M field value in the instruction code. If

MOD=0b11, it means a single-byte instruction with no offset field. If the R/M field = 0b100 and MOD is not 0b11, it means 2-byte address mode addressing. At this time, the function `sib()` which processes the second operand instruction byte SIB (Scale, Index, Base) is called to obtain the offset value and return. If the R/M field is 0b101 and MOD is 0, it indicates a single-byte address mode encoding followed by a 32-byte offset value. For the rest of the case, it is processed according to the MOD.

## 11.4.2 Code annotation

Program 11-3 linux/kernel/math/ea.c

```

1  /*
2   * linux/kernel/math/ea.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * Calculate the effective address.
9   */
10
11 // <stddef.h> The standard definition header file. NULL, offsetof(TYPE, MEMBER) are defined.
12 // <linux/math_emu.h> Coprocessor emulation header file. A coprocessor data structure and
13 // a floating point representation structure are defined.
14 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined
15 // for segment register operations.
16 #include <stddef.h>
17
18 #include <linux/math_emu.h>
19 #include <asm/segment.h>
20
21 // The offset position of each register in the info structure. offsetof() is used to find the
22 // offset position of the specified field in the structure. See the include/stddef.h file.
23 static int __regoffset[] = {
24     offsetof(struct info, __eax),
25     offsetof(struct info, __ecx),
26     offsetof(struct info, __edx),
27     offsetof(struct info, __ebx),
28     offsetof(struct info, __esp),
29     offsetof(struct info, __ebp),
30     offsetof(struct info, __esi),
31     offsetof(struct info, __edi)
32 };
33
34 // Get the contents of the register at the specified position in the info structure.
35 #define REG(x) (*(long *) (__regoffset[(x)]+(char *) info))
36
37 // Get the value of the second operand indication byte SIB (Scale, Index, Base).
38 static char * sib(struct info * info, int mod)
39 {
40     unsigned char ss, index, base;
41     long offset = 0;
42
43     // The SIB byte is first taken from the user code segment, then the field bit values are taken.

```

```

34     base = get\_fs\_byte((char *) EIP);
35     EIP++;
36     ss = base >> 6;                // scale size.
37     index = (base >> 3) & 7;
38     base &= 7;
    // If the index code is 0b100, it means there is no index offset value. Otherwise index offset
    // value offset = register content * scale factor.
39     if (index == 4)
40         offset = 0;
41     else
42         offset = REG(index);
43     offset <=< ss;
    // If the MOD in the previous MODRM byte is not zero, or if Base is not equal to 0b101, it means
    // that there is an offset value in the register specified by base. Therefore, the offset needs
    // to be added to the contents of the base corresponding register. If MOD=1, the offset value
    // is 1 byte (8-bit). Otherwise, if MOD=2, or base=0b101, the offset value is 4 bytes.
44     if (mod || base != 5)
45         offset += REG(base);
46     if (mod == 1) {
47         offset += (signed char) get\_fs\_byte((char *) EIP);
48         EIP++;
49     } else if (mod == 2 || base == 5) {
50         offset += (signed) get\_fs\_long((unsigned long *) EIP);
51         EIP += 4;
52     }
    // Finally save and return the offset value.
53     I387.foo = offset;
54     I387.fos = 0x17;
55     return (char *) offset;
56 }
57
    // Calculating the effective address base on the addressing mode byte in the instruction code.
58 char * ea(struct info * info, unsigned short code)
59 {
60     unsigned char mod,rm;
61     long * tmp = &EAX;
62     int offset = 0;
63
    // First take the MOD field and the R/M field value in the instruction code. If MOD=0b11, it
    // means a single-byte instruction with no offset field. If the R/M field = 0b100 and MOD is
    // not 0b11, it means 2-byte address mode addressing, so call sib() to find the offset value
    // and return.
64     mod = (code >> 6) & 3;          // MOD field.
65     rm = code & 7;                 // R/M field.
66     if (rm == 4 && mod != 3)
67         return sib(info, mod);
    // If the R/M field is 0b101 and MOD is 0, it indicates a single-byte address mode encoding
    // followed by a 32-byte offset value. Then take the 4-byte offset value in the user code, save
    // it and return it.
68     if (rm == 5 && !mod) {
69         offset = get\_fs\_long((unsigned long *) EIP);
70         EIP += 4;
71         I387.foo = offset;

```

```

72         I387.fos = 0x17;
73         return (char *) offset;
74     }
    // For the rest of the case, it is processed according to the MOD. First, the value of the contents
    // of the corresponding register of the R/M code is taken out as the pointer tmp. For MOD=0,
    // there is no offset value. For MOD=1, the code is followed by a 1-byte offset value. For MOD=2,
    // there is a 4-byte offset after the code. Finally save and return the valid address value.
75     tmp = & REG(rm);
76     switch (mod) {
77         case 0: offset = 0; break;
78         case 1:
79             offset = (signed char) get\_fs\_byte((char *) EIP);
80             EIP++;
81             break;
82         case 2:
83             offset = (signed) get\_fs\_long((unsigned long *) EIP);
84             EIP += 4;
85             break;
86         case 3:
87             math\_abort(info, 1<<(SIGILL-1));
88     }
89     I387.foo = offset;
90     I387.fos = 0x17;
91     return offset + (char *) *tmp;
92 }
93

```

## 11.5 convert.c

### 11.5.1 Function

The convert.c program contains data type conversion functions during the 80387 emulation operation. Before performing the simulation calculation, we need to convert the integer or real type provided by the user into the temporary real number format used in the simulation, and then convert back to the original format after the simulation is completed. For example, Figure 11-9 shows a schematic diagram of a conversion of a short real into a temporary real number format.

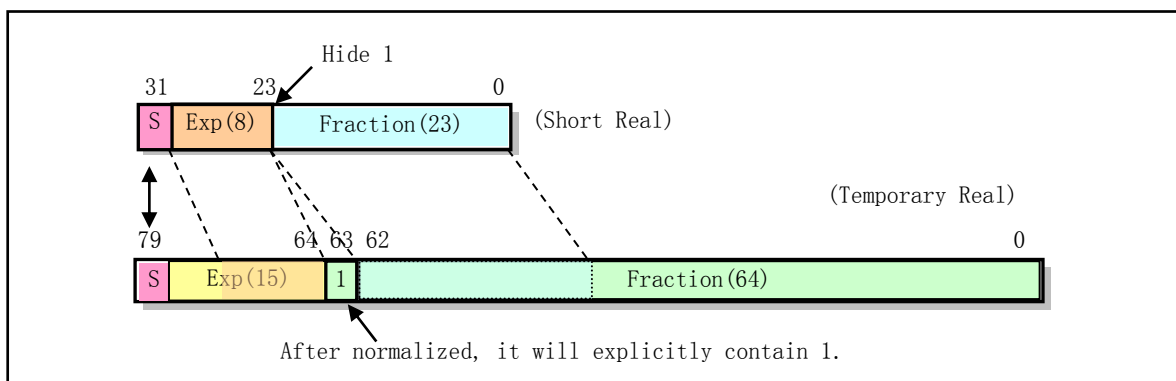


Figure 11-9 Conversion diagram of short real to temporary real format

## 11.5.2 Code annotation

Program 11-4 linux/kernel/math/convert.c

---

```

1  /*
2   * linux/kernel/math/convert.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  #include <linux/math_emu.h>
8
9  /*
10 * NOTE!!! There is some "non-obvious" optimisations in the temp_to_long
11 * and temp_to_short conversion routines: don't touch them if you don't
12 * know what's going on. They are the adding of one in the rounding: the
13 * overflow bit is also used for adding one into the exponent. Thus it
14 * looks like the overflow would be incorrectly handled, but due to the
15 * way the IEEE numbers work, things are correct.
16 *
17 * There is no checking for total overflow in the conversions, though (ie
18 * if the temp-real number simply won't fit in a short- or long-real.)
19 */
20
21 // Convert short real to temporary real number.
22 // The short real number is 32 bits long, its significant number (mantissa) is 23 bits long,
23 // the exponent is 8 bits, and there is 1 sign bit.
24 void short_to_temp(const short_real * a, temp_real * b)
25 {
26     // First we handle the case where the short real number is 0. If it is 0, the significand
27     // the temporary real number b is set to 0. The sign bit of the temporary real number is then
28     // set according to the short real sign bit, ie the most significant bit of exponent.
29     if (!(*a & 0x7fffffff)) {
30         b->a = b->b = 0;           // set significand of the temp real = 0.
31         if (*a)
32             b->exponent = 0x8000; // set sign bit.
33         else
34             b->exponent = 0;
35         return;
36     }
37
38     // For a general short real, first determine the exponent value corresponding to the temporary
39     // real number. Here we need to use the concept of the integer number biased representation
40     // method, see section 11.1. The biase number of the short real exponent is 127, while the biase
41     // of the temporary real exponent is 16383. Therefore, after extracting the exponent value in
42     // the short real, it is necessary to change the biase value to 16383. This forms the exponent
43     // value in the temporary real format. Also, if the short real number is negative, you need
44     // to set the sign bit of the temporary real (bit 79). Next set the mantissa. The method is
45     // to shift the short real number to the left by 8 bits, and let the 23 most significant digit
46     // of the mantissa be at the bit 62 of the temporary real number. Bit 63 of the temporary real
47     // mantissa needs to be set to 1, which requires an OR 0x80000000 operation. Finally, the low
48     // 32-bit significant number of the temporary real is cleared.

```



```

31     b->exponent = ((*a>>23) & 0xff)-127+16383;    // change biased value to 16383.
32     if (*a<0)
33         b->exponent |= 0x8000;                    // add negative sign if need.
34     b->b = (*a<<8) | 0x80000000;                  // put mantissa, set 63th bit.
35     b->a = 0;
36 }
37
// Convert long real to temporary real.
// The method is exactly the same as short_to_temp().However, the long real is 64 bits long,
// its significant number (mantissa) is 52 bits long, the exponent is 11 bits, and there is
// 1 sign bit. In addition, the long real exponent biased value is 1023.
38 void long_to_temp(const long_real * a, temp_real * b)
39 {
40     if (!a->a && !(a->b & 0x7fffffff)) {
41         b->a = b->b = 0;
42         if (a->b)
43             b->exponent = 0x8000;                // set sign bit.
44         else
45             b->exponent = 0;
46         return;
47     }
48     b->exponent = ((a->b >> 20) & 0x7ff)-1023+16383; // change biased value to 16383.
49     if (a->b<0)
50         b->exponent |= 0x8000;
51     b->b = 0x80000000 | (a->b<<11) | (((unsigned long)a->a)>>21); // place 1.
52     b->a = a->a<<11;
53 }
54
// Convert temporary real to short real.
// The procedure is the opposite of short_to_temp() but we requires handling precision and
// rounding issues.
55 void temp_to_short(const temp_real * a, short_real * b)
56 {
57     // If the exponent part is 0, the short real number is set to -0 or 0 depending on whether or
58     // not there is a sign bit. refer to Table 11-2.
59     if (!(a->exponent & 0x7fff)) {
60         *b = (a->exponent)?0x80000000:0;
61         return;
62     }
63     // First, the exponent portion is processed, that is, the amount of the exponent bias (16383)
64     // is replaced by the biased amount 127 of the short real number, and the sign bit is set if
65     // it is a negative number.
66     *b = (((long) a->exponent)-16383+127) << 23) & 0x7f800000;
67     if (a->exponent < 0) // set sign if negative.
68         *b |= 0x80000000;
69     // Then get the highest 23 bits of the signifcand from the temporary real number and perform
70     // the rounding operation according to the rounding setting in the control word.
71     *b |= (a->b >> 8) & 0x007fffff; // get higher 23bits of the temp real.
72     switch (ROUNDING) {
73     case ROUND_NEAREST:
74         if ((a->b & 0xff) > 0x80)
75             ++*b;
76         break;

```

```

70         case ROUND_DOWN:
71             if ((a->exponent & 0x8000) && (a->b & 0xff))
72                 ++*b;
73             break;
74         case ROUND_UP:
75             if (!(a->exponent & 0x8000) && (a->b & 0xff))
76                 ++*b;
77             break;
78     }
79 }
80
81 // Convert temporary real to long real number.
82 // The long real is 64 bits long, its significant number (mantissa) is 52 bits long, the exponent
83 // is 11 bits, and there is 1 sign bit, and the exponent biased value is 1023.
84 void temp_to_long(const temp_real * a, long_real * b)
85 {
86     if (!(a->exponent & 0x7fff)) {
87         b->a = 0;
88         b->b = (a->exponent)?0x80000000:0;
89         return;
90     }
91     b->b = (((0x7fff & (long) a->exponent)-16383+1023) << 20) & 0x7ff00000;
92     if (a->exponent < 0)
93         b->b |= 0x80000000;
94     b->b |= (a->b >> 11) & 0x000fffff;
95     b->a = a->b << 21;
96     b->a |= (a->a >> 11) & 0x001fffff;
97     switch (ROUNDING) {
98     case ROUND_NEAREST:
99         if ((a->a & 0x7ff) > 0x400)
100             __asm__ ("addl $1,%0 ; adc1 $0,%1"
101                     : "=r" (b->a), "=r" (b->b)
102                     : "" (b->a), "1" (b->b));
103         break;
104     case ROUND_DOWN:
105         if ((a->exponent & 0x8000) && (a->b & 0xff))
106             __asm__ ("addl $1,%0 ; adc1 $0,%1"
107                     : "=r" (b->a), "=r" (b->b)
108                     : "" (b->a), "1" (b->b));
109         break;
110     case ROUND_UP:
111         if (!(a->exponent & 0x8000) && (a->b & 0xff))
112             __asm__ ("addl $1,%0 ; adc1 $0,%1"
113                     : "=r" (b->a), "=r" (b->b)
114                     : "" (b->a), "1" (b->b));
115         break;
116     }
117 }
118
119 // Convert temporary real to a temporary integer format.
120 // Temporary integers are also represented by 10 bytes. The lower 8 bytes are unsigned integer
121 // values, and the upper 2 bytes represent exponent value and sign bit. If the highest significant
122 // byte of the upper 2 bytes is 1, it indicates a negative number; if the bit is 0, it indicates

```

```

// a positive number.
116 void real_to_int(const temp_real * a, temp_int * b)
117 {
118     int shift = 16383 + 63 - (a->exponent & 0x7fff);
119     unsigned long underflow;
120
121     b->a = b->b = underflow = 0;
122     b->sign = (a->exponent < 0);
123     if (shift < 0) {
124         set_OE();
125         return;
126     }
127     if (shift < 32) {
128         b->b = a->b; b->a = a->a;
129     } else if (shift < 64) {
130         b->a = a->b; underflow = a->a;
131         shift -= 32;
132     } else if (shift < 96) {
133         underflow = a->b;
134         shift -= 64;
135     } else
136         return;
137     __asm__ ("shrdl %2, %1, %0" // 32-bit shift right.
138             : "=r" (underflow), "=r" (b->a)
139             : "c" ((char) shift), "" (underflow), "I" (b->a));
140     __asm__ ("shrdl %2, %1, %0"
141             : "=r" (b->a), "=r" (b->b)
142             : "c" ((char) shift), "" (b->a), "I" (b->b));
143     __asm__ ("shrl %1, %0"
144             : "=r" (b->b)
145             : "c" ((char) shift), "" (b->b));
146     switch (ROUNDING) {
147     case ROUND_NEAREST:
148         __asm__ ("addl %4, %5 ; adcl $0, %0 ; adcl $0, %1"
149                 : "=r" (b->a), "=r" (b->b)
150                 : "" (b->a), "I" (b->b)
151                 , "r" (0x7fffffff + (b->a & 1))
152                 , "m" (*&underflow));
153         break;
154     case ROUND_UP:
155         if (!b->sign && underflow)
156             __asm__ ("addl $1, %0 ; adcl $0, %1"
157                     : "=r" (b->a), "=r" (b->b)
158                     : "" (b->a), "I" (b->b));
159         break;
160     case ROUND_DOWN:
161         if (b->sign && underflow)
162             __asm__ ("addl $1, %0 ; adcl $0, %1"
163                     : "=r" (b->a), "=r" (b->b)
164                     : "" (b->a), "I" (b->b));
165         break;
166     }
167 }

```

```

168 // Convert a temporary integer to a temporary real format.
169 void int_to_real(const temp_int * a, temp_real * b)
170 {
171     // Since the original value is an integer, when converting to a temporary real, the exponent
172     // adds 63 in addition to the biased amount of 16383. This means that the significand needs
173     // to be multiplied by 2^63, which means that the significand are also integer values.
174     b->a = a->a;
175     b->b = a->b;
176     if (b->a || b->b)
177         b->exponent = 16383 + 63 + (a->sign? 0x8000:0);
178     else {
179         b->exponent = 0;
180         return;
181     }
182     // The normal real number after the conversion format is normalized, that is, the high significant
183     // bit of the significand is not zero.
184     while (b->b >= 0) {
185         b->exponent--;
186         __asm__ ("addl %0,%0 ; adcl %1,%1"
187                 : "=r" (b->a), "=r" (b->b)
188                 : "" (b->a), "I" (b->b));
189     }
190 }

```

## 11.6 add.c

### 11.6.1 Function

The add.c program is used to handle the addition during the simulation. In order to calculate the mantissa of the floating-point number, we need to first symbolize the mantissa, and then perform non-symbolization after the calculation, and then resume using the temporary real format to represent the floating-point number. A schematic diagram of the symbolization and non-symbolic format conversion of floating-point mantissas is shown in Figure 11-10.

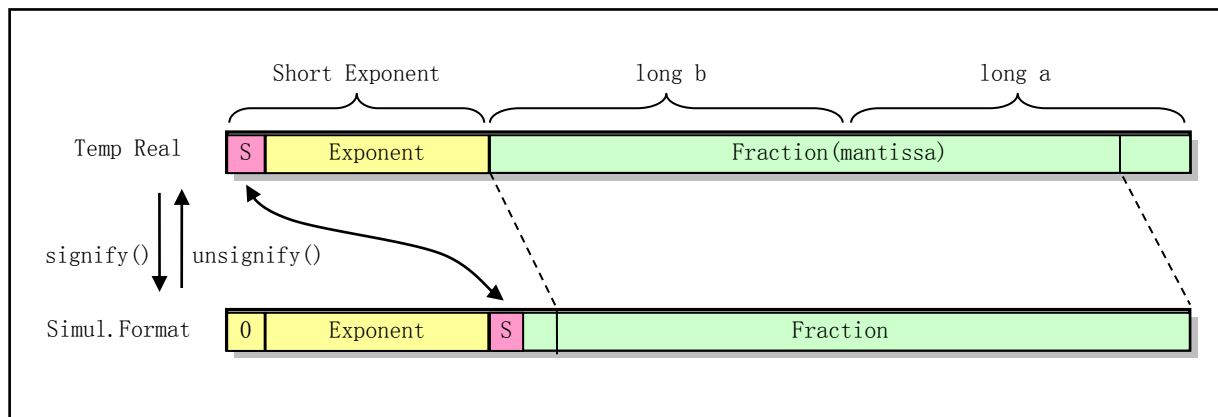


Figure 11-10 Transformation between temporary real and simulation formats

## 11.6.2 Code annotation

Program 11-5 linux/kernel/math/add.c

---

```

1  /*
2   * linux/kernel/math/add.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * temporary real addition routine.
9   *
10  * NOTE! These aren't exact: they are only 62 bits wide, and don't do
11  * correct rounding. Fast hack. The reason is that we shift right the
12  * values by two, in order not to have overflow (1 bit), and to be able
13  * to move the sign into the mantissa (1 bit). Much simpler algorithms,
14  * and 62 bits (61 really - no rounding) accuracy is usually enough. The
15  * only time you should notice anything weird is when adding 64-bit
16  * integers together. When using doubles (52 bits accuracy), the
17  * 61-bit accuracy never shows at all.
18  */
19
20 #include <linux/math_emu.h>
21
22 // Get a negative number (two's complement) representation of a number.
23 // The operation is to invert the mantissa (significand) of the temporary real number and add
24 // 1 to it. The parameter 'a' is a pointer to a temporary real structure. The combination of
25 // its fields a and b is the significand of the temporary real.
26
27 #define NEGINT(a) \
28 __asm__( "notl %0 ; notl %1 ; addl $1, %0 ; adcl $0, %1" \
29         : "=r" (a->a), "=r" (a->b) \
30         : "" (a->a), "1" (a->b))
31
32 // Signified the mantissa.
33 // That is, transform the temporary real number into an exponential and integer representation
34 // to facilitate the simulation operation. So we call it the simulation format here.
35 // The operation is to move the 64-bit binary mantissa right by 2 bits (so the exponent needs
36 // to add 2). Since the highest bit of the exponent field is the sign bit, if the exponent value
37 // is less than zero, the number is a negative. So we represent the mantissa in complements
38 // (take negative) and then take the exponent to a positive value. At this time, the mantissa
39 // contains not only the significand shifted by 2 bits but also the sign bit of the value.
40
41 static void signify(temp_real * a)
42 {
43
44     // On line 30: %0 is a->a; %1 is a->b. The assembly instruction "shrdl $2, %1, %0" performs
45     // a double-precision (64-bit) right shift, which shifts the combined mantissa <b, a> to the
46     // right by 2 bits. Since this move does not change the value in %1 (a->b), it also needs to
47     // be shifted to the right by 2 bits.
48
49     a->exponent += 2; // increases exponent by 2.
50     __asm__( "shrdl $2, %1, %0 ; shr1 $2, %1" // mantissa is shifted to right by 2 bits.
51             : "=r" (a->a), "=r" (a->b)

```

```

32         : "" (a->a), "1" (a->b));
33     if (a->exponent < 0)
34         NEGINT(a);                // be negative.
35     a->exponent &= 0x7fff;        // remove the sign bit (if any).
36 }
37
    // Unsignedified the mantissa.
    // Convert the emulation format to a temporary real format. That is, the real number represented
    // by the exponent and the integer is converted into a temporary real format.
38 static void unsignify(temp_real * a)
39 {
    // For the number with a value of 0, do not process, just return. Otherwise, we first reset
    // the sign bit of the temporary rea, and then determine whether the high 32-bit field a->b
    // of the mantissa has a sign bit. If so, add a sign bit to the exponent field and also represent
    // (complement) the mantissa as an unsigned number. Finally, the mantissa is normalized, and
    // the exponent value is decremented accordingly. That is, we perform a left shift operation
    // so that the most significant bit of the mantissa is not 0 (the last a->b value appears to
    // be like a negative value).
40     if (!(a->a || a->b)) {        // ret if zero.
41         a->exponent = 0;
42         return;
43     }
44     a->exponent &= 0x7fff;        // reset the sign bit.
45     if (a->b < 0) {              // if negative, let mantissa be a positive.
46         NEGINT(a);
47         a->exponent |= 0x8000;   // add a sign bit.
48     }
49     while (a->b >= 0) {
50         a->exponent--;           // the mantissa is normalized.
51         __asm__ ("addl %0,%0 ; adcl %1,%1"
52                 : "=r" (a->a), "=r" (a->b)
53                 : "" (a->a), "1" (a->b));
54     }
55 }
56
    // Simulate floating-point addition instruction.
    // Temporary real number parameter: src1 + src2 -> result.
57 void fadd(const temp_real * src1, const temp_real * src2, temp_real * result)
58 {
59     temp_real a,b;
60     int x1,x2, shift;
61
    // First take the exponential values x1, x2 of the two numbers (with the sign bit removed),
    // then let the variable a be equal to the maximum value, and let the variable 'shift' equal
    // the exponential difference (ie, the multiple of 2 of the difference).
62     x1 = src1->exponent & 0x7fff;
63     x2 = src2->exponent & 0x7fff;
64     if (x1 > x2) {
65         a = *src1;
66         b = *src2;
67         shift = x1-x2;
68     } else {
69         a = *src2;

```

```
70         b = *src1;
71         shift = x2-x1;
72     }
    // If the difference between the two is too large, greater than or equal to 2^64, we can ignore
    // the small number (ie b value). So you can return value a directly. Otherwise, if the difference
    // is greater than or equal to 2^32, then we can ignore the lower 32-bit value of the small
    // value b. So we shift b's high long field value b.b to the right by 32 bits, that is, put
    // it in b.a. Then increase the exponent of b by 32 times. That is, the exponent difference
    // is subtracted by 32. After this adjustment, the mantissas of the two added numbers fall
    // substantially in the same range.
73     if (shift >= 64) {
74         *result = a;
75         return;
76     }
77     if (shift >= 32) {
78         b.a = b.b;
79         b.b = 0;
80         shift -= 32;
81     }
    // Then we make a fine adjustment to adjust the exponent of the two to the same value. The
    // adjustment method is to shift the mantissa of the small value b to the right by 'shift' bits.
    // Thus the exponents of the two are the same, in the same order of magnitude. So we can add
    // the two mantissas. Before adding, we need to convert them into a simulation format and convert
    // it back to the temporary real format after the addition operation.
82     __asm__ ("shrdl %4,%1,%0 ; shr1 %4,%1" // Double precision (64 bit) right shift.
83             : "=r" (b.a), "=r" (b.b)
84             : "" (b.a), "1" (b.b), "c" ((char) shift));
85     signify(&a); // change format.
86     signify(&b);
87     __asm__ ("addl %4,%0 ; adcl %5,%1" // adding by using normal instructions.
88             : "=r" (a.a), "=r" (a.b)
89             : "" (a.a), "1" (a.b), "g" (b.a), "g" (b.b));
90     unsignify(&a); // change back to temp real format.
91     *result = a;
92 }
93
```

---

## 11.7 compare.c

### 11.7.1 Function

The compare.c program is used to compare the size of two temporary real numbers in the accumulator during the simulation.

### 11.7.2 Code annotation

Program 11-6 linux/kernel/math/compare.c

---

```
1 /*
2  * linux/kernel/math/compare.c
```

```

3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7  /*
8  * temporary real comparison routines
9  */
10
11 #include <linux/math_emu.h>
12
13 // Reset the C3, C2, C1, and C0 condition bits in the status word.
14 #define clear_Cx() (I387.swd &= ~0x4500)
15
16 // The temporary real is normalized, that is, expressed as an exponent and a significand.
17 // For example: 102.345 is expressed as 1.02345 X 102. 0.0001234 is expressed as 1.234 X 10-4.
18 // Of course, these numbers are represented in binary in the function.
19 static void normalize(temp_real * a)
20 {
21     int i = a->exponent & 0x7fff;          // get the exponent (ignore sign bit).
22     int sign = a->exponent & 0x8000;       // get sign.
23
24     // If the 64-bit significant number (mantissa) of the temporary real a is 0, then a is equal
25     // to 0. So clear the exponent of a and return.
26     if (!(a->a || a->b)) {
27         a->exponent = 0;
28         return;
29     }
30     // If the mantissa of a has a zero-valued bit at the far left, shift the mantissa to the left
31     // and adjust the exponent value (decrement). Until the MSB (most significant bit) of the b
32     // field is 1 (when b appears as a negative value). Finally add the sign bit.
33     while (i && a->b >= 0) {
34         i--;
35         __asm__( "addl %0,%0 ; adcl %1,%1"
36                 : "=r" (a->a), "=r" (a->b)
37                 : "" (a->a), "I" (a->b));
38     }
39     a->exponent = i | sign;
40 }
41
42 // Simulate floating-point instructions FTST (Floating-point Test).
43 // That is, the top stack accumulator ST(0) is compared with 0, and the condition bits in the
44 // status word are set according to the comparison result. If ST > 0.0, C3, C2, and C0 are
45 // respectively 000; if ST < 0.0, the condition bit is 001; if ST == 0.0, the condition bit
46 // is 100; if not, the condition bit is 111.
47 void ftst(const temp_real * a)
48 {
49     temp_real b;
50
51     // First, the condition flag in the status word is cleared, and the comparison value b (ST)
52     // is normalized. If b is not equal to zero and the sign bit is set (is a negative number),
53     // condition bit C0 is set, otherwise condition bit C3 is set.
54     clear_Cx();
55     b = *a;

```



```
39     normalize(&b);
40     if (b.a || b.b || b.exponent) {
41         if (b.exponent < 0)
42             set_C0();
43     } else
44         set_C3();
45 }
46
// Simulate floating-point instruction FCOM (Floating-point Compare).
// Compare the two parameters src1, src2 and set the condition bits according to the comparison
// result. If src1 > src2, C3, C2, and C0 are respectively 000; if src1 < src2, the condition
// bit is 001; if the two are equal, the condition bit is 100.
47 void fcom(const temp_real * src1, const temp_real * src2)
48 {
49     temp_real a;
50
51     a = *src1;
52     a.exponent ^= 0x8000;           // invert the sign bit.
53     fadd(&a, src2, &a);           // add the two (ie subtract).
54     ftst(&a);                     // test results, set the condition.
55 }
56
// Simulate floating-point instruction FUCOM (no order comparison).
// This function is used for comparison operations where one of the operands is a NaN.
57 void fucom(const temp_real * src1, const temp_real * src2)
58 {
59     fcom(src1, src2);
60 }
61
```

---

## 11.8 get\_put.c

### 11.8.1 Function

The get\_put.c program handles all access to user memory: fetch and store instructions/real values/BCD values. This is the only part that involves other data formats. All other operations in the simulation process use the temporary real number format.

### 11.8.2 Code annotation

Program 11-7 linux/kernel/math/get\_put.c

---

```
1  /*
2   * linux/kernel/math/get_put.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * This file handles all accesses to user memory: getting and putting
9   * ints/reals/BCD etc. This is the only part that concerns itself with
```

```

10 * other than temporary real format. All other cals are strictly temp_real.
11 */
    // <signal.h> Signal header file. Define signal symbol constants, signal structures, and
    //     signal manipulation function prototypes.
    // <linux/math_emu.h> Coprocessor emulation header file. A coprocessor data structure and
    //     a floating point representation structure are defined.
    // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
    //     commonly used functions of the kernel.
    // <asm/segment.h> Segment operation header file. An embedded assembly function is defined
    //     for segment register operations.
12 #include <signal.h>
13
14 #include <linux/math_emu.h>
15 #include <linux/kernel.h>
16 #include <asm/segment.h>
17
    // Get a short real (single precision real number) in user memory.
    // According to the content in the addressing mode byte in the floating-point instruction code
    // and the content in the current register in the info structure, the effective address
    // (see file math/ea.c) where the short real number is located is obtained, and then the
    // corresponding real value is read from the user data area. Finally, the user's short real
    // number is converted into a temporary real number (math/convert.c) and returned.
    // Parameters: tmp - a pointer to a temporary real; info - info structure pointer;
    // code - instruction code.
18 void get_short_real(temp_real * tmp,
19     struct info * info, unsigned short code)
20 {
21     char * addr;
22     short_real sr;
23
24     addr = ea(info, code);                // calculate effective address.
25     sr = get_fs_long((unsigned long *) addr); // get from user data area.
26     short_to_temp(&sr, tmp);                // convert to temp real format.
27 }
28
    // Get a long real (double precision real number) in user memory.
    // This function is handled in the same way as get_short_real().
29 void get_long_real(temp_real * tmp,
30     struct info * info, unsigned short code)
31 {
32     char * addr;
33     long_real lr;
34
35     addr = ea(info, code);                // get effective address.
36     lr.a = get_fs_long((unsigned long *) addr); // get long real.
37     lr.b = get_fs_long(1 + (unsigned long *) addr);
38     long_to_temp(&lr, tmp);                // convert to temp real format.
39 }
40
    // Take the temporary real number in the user's memory.
    // First, according to the contents of the addressing mode byte in the floating-point instruction
    // code and the contents of the current register in the info structure, obtain the effective
    // address (math/ea.c) where the temporary real number is located, and then read the

```

```

// corresponding temporary real value from the user data area.
// Parameters: tmp - a pointer to a temporary real number; info - info structure pointer;
// code - instruction code.
41 void get_temp_real(temp_real * tmp,
42     struct info * info, unsigned short code)
43 {
44     char * addr;
45
46     addr = ea(info, code);           // get effective address.
47     tmp->a = get_fs_long((unsigned long *) addr);
48     tmp->b = get_fs_long(1 + (unsigned long *) addr);
49     tmp->exponent = get_fs_word(4 + (unsigned short *) addr);
50 }
51
// Get a short integer in the user's memory.
// The function first obtains the effective address of the short integer according to the content
// in the addressing mode byte in the floating-point instruction code and the contents of the
// current register in the info structure, and then reads the corresponding integer value from
// the user data area, and save as a temporary integer format. Finally, convert the temporary
// integer value to a temporary real number.
// Temporary integers are also represented by 10 bytes. The lower 8 bytes are unsigned integer
// values, and the upper 2 bytes represent exponent value and sign bits. If the highest
// significant byte of the upper 2 bytes is 1, it means a negative number; if the most significant
// bit is 0, it means a positive number.
// Parameters: tmp - a pointer to a temporary real number; info - info structure pointer;
// code - instruction code.
52 void get_short_int(temp_real * tmp,
53     struct info * info, unsigned short code)
54 {
55     char * addr;
56     temp_int ti;
57
58     addr = ea(info, code);           // get effective in the instruction code.
59     ti.a = (signed short) get_fs_word((unsigned short *) addr);
60     ti.b = 0;
61     if (ti.sign = (ti.a < 0))           // set sign bit if it is a negative.
62         ti.a = - ti.a;                 // and unsignify the mantissa part.
63     int_to_real(&ti, tmp);             // change to temp real.
64 }
65
// Get a long integer in the user's memory and convert it to a temporary real format.
// This function is handled in the same way as get_short_int() above.
66 void get_long_int(temp_real * tmp,
67     struct info * info, unsigned short code)
68 {
69     char * addr;
70     temp_int ti;
71
72     addr = ea(info, code);
73     ti.a = get_fs_long((unsigned long *) addr);
74     ti.b = 0;
75     if (ti.sign = (ti.a < 0))
76         ti.a = - ti.a;

```

```

77     int_to_real(&ti, tmp);
78 }
79
// Get a 64-bit long integer (extended long integer) in user memory.
// First, according to the contents of the addressing mode byte in the floating-point instruction
// code and the contents of the current register in the info structure, the effective address
// of the 64-bit long integer is obtained, and then the corresponding integer value is read
// from the user data area and saved as a temporary integer. Finally, convert the temporary
// integer value to a temporary real number.
// Parameters: tmp - temporary real pointer; info - info pointer; code - instruction code.
80 void get_longlong_int(temp_real * tmp,
81     struct info * info, unsigned short code)
82 {
83     char * addr;
84     temp_int ti;
85
86     addr = ea(info, code);                // get effective address.
87     ti.a = get_fs_long((unsigned long *) addr); // get 64-bit longlong integer.
88     ti.b = get_fs_long(1 + (unsigned long *) addr);
89     if (ti.sign = (ti.b < 0))                // set sign bit if it is a negative,
90         __asm__("notl %0 ; notl %1\n\t"      // and complement, carry adjustment.
91             "addl $1, %0 ; adcl $0, %1"
92             : "=r" (ti.a), "=r" (ti.b)
93             : "" (ti.a), "1" (ti.b));
94     int_to_real(&ti, tmp);                // change to temporary real.
95 }
96
// Multiply a 64-bit integer (such as N) by 10.
// This macro is used in the conversion of the following BCD code values into a temporary real
// number format. The method is: N<<1 + N<<3.
97 #define MUL10(low, high) \
98     __asm__("addl %0, %0 ; adcl %1, %1\n\t" \
99         "movl %0, %%ecx ; movl %1, %%ebx\n\t" \
100         "addl %0, %0 ; adcl %1, %1\n\t" \
101         "addl %0, %0 ; adcl %1, %1\n\t" \
102         "addl %%ecx, %0 ; adcl %%ebx, %1" \
103         : "=a" (low), "=d" (high) \
104         : "" (low), "1" (high): "cx", "bx")
105
// 64-bit addition. Add the 32-bit unsigned number val to the 64-bit <high, low>.
106 #define ADD64(val, low, high) \
107     __asm__("addl %4, %0 ; adcl $0, %1": "=r" (low), "=r" (high) \
108         : "" (low), "1" (high), "r" ((unsigned long) (val)))
109
// Take the BCD code value in the user's memory and convert it to temporary real format.
// The function first obtains the effective address of the BCD code according to the content
// in the addressing mode byte in the floating-point instruction code and the content in the
// registers in the info structure, and then reads the corresponding BCD code of 10 bytes from
// the user data area (where 1 byte is used for sign) and is converted to a temporary integer
// form. Finally, the temporary integer value is converted to a temporary real number.
// Parameters: tmp - a pointer to a temporary real number; info - info structure pointer;
// code - instruction code.
110 void get_BCD(temp_real * tmp, struct info * info, unsigned short code)

```

```

111 {
112     int k;
113     char * addr;
114     temp_int i;
115     unsigned char c;
116
117     // Get the memory effective address of the BCD code value, and then start processing from the
118     // last BCD code byte (most significant bit). First obtain the sign bit of the BCD code value
119     // and set the sign bit of the temporary integer. The 9-byte BCD code value is then converted
120     // to a temporary integer format, and finally the temporary integer value is converted to a
121     // temporary real number.
122     addr = ea(info, code); // get effective address.
123     addr += 9; // point to the last (10th) byte.
124     i.sign = 0x80 & get_fs_byte(addr--); // get sign bit.
125     i.a = i.b = 0;
126     for (k = 0; k < 9; k++) { // change to temporary integer.
127         c = get_fs_byte(addr--);
128         MUL10(i.a, i.b);
129         ADD64((c>>4), i.a, i.b);
130         MUL10(i.a, i.b);
131         ADD64((c&0xf), i.a, i.b);
132     }
133     int_to_real(&i, tmp); // change to temporary real.
134 }
135
136 // Save the result in the short (single precision) real format to the user data area.
137 // The function first obtains the effective address addr in the instruction code, and then
138 // converts the result of the temporary real into a short real format and stores it at the location
139 // of addr.
140 // Parameters: tmp - temporary real format result value; info - info structure pointer;
141 // code - instruction code.
142 void put_short_real(const temp_real * tmp,
143     struct info * info, unsigned short code)
144 {
145     char * addr;
146     short_real sr;
147
148     addr = ea(info, code); // get effective address.
149     verify_area(addr, 4); // verify the data area.
150     temp_to_short(tmp, &sr); // convert to short real.
151     put_fs_long(sr, (unsigned long *) addr); // store at location addr.
152 }
153
154 // Save the result in the long (double precision) real format to the user data area.
155 // This function is handled in the same way as put_real_real() above.
156 void put_long_real(const temp_real * tmp,
157     struct info * info, unsigned short code)
158 {
159     char * addr;
160     long_real lr;
161
162     addr = ea(info, code);
163     verify_area(addr, 8);

```

```

151     temp_to_long(tmp,&lr);
152     put_fs_long(lr.a, (unsigned long *) addr);
153     put_fs_long(lr.b, 1 + (unsigned long *) addr);
154 }
155
    // Save the result in the temporary real format to the user data area.
    // The function first obtains the address addr for saving the result, and after verifying that
    // there is enough (10 bytes) of user memory at the address, then stores the temporary real
    // number to the addr.
    // Parameters: tmp - temporary real format result value; info - info structure pointer;
    // code - instruction code.
156 void put_temp_real(const temp_real * tmp,
157     struct info * info, unsigned short code)
158 {
159     char * addr;
160
161     addr = ea(info,code);                // get effective address.
162     verify_area(addr,10);                // verify there is enough uesr memory.
163     put_fs_long(tmp->a, (unsigned long *) addr); // store temp real to user area.
164     put_fs_long(tmp->b, 1 + (unsigned long *) addr);
165     put_fs_word(tmp->exponent, 4 + (short *) addr);
166 }
167
    // Save the result in the short integer format to the user data area.
    // The function first obtains the address addr for saving the result, and then converts the
    // result of the temporary real format into a temporary integer format. If it is a negative
    // number, set the integer sign bit. Finally, the integer is saved to the user memory.
    // Parameters: tmp - temporary real format result value; info - info structure pointer;
    // code - instruction code.
168 void put_short_int(const temp_real * tmp,
169     struct info * info, unsigned short code)
170 {
171     char * addr;
172     temp_int ti;
173
174     addr = ea(info,code);                // get effective address.
175     real_to_int(tmp,&ti);                // change to temp integer.
176     verify_area(addr,2);                // verify user area (need 2 bytes)
177     if (ti.sign)
178         ti.a = -ti.a;                    // negative the result.
179     put_fs_word(ti.a, (short *) addr);    // store to user data area.
180 }
181
    // Save the result in a long integer format to the user data area.
    // This function is handled in the same way as put_short_int() above.
182 void put_long_int(const temp_real * tmp,
183     struct info * info, unsigned short code)
184 {
185     char * addr;
186     temp_int ti;
187
188     addr = ea(info,code);                // get effective address.
189     real_to_int(tmp,&ti);                // change to temp integer.

```

```

190     verify\_area(addr, 4);                // verify user area (4 bytes).
191     if (ti.sign)                        // if signed, negative the result.
192         ti.a = -ti.a;
193     put\_fs\_long(ti.a, (unsigned long *) addr);    // store to the user data area.
194 }
195
// Save the result in the 64-bit integer format to the user data area.
// The function first obtains the address addr for saving the result, and then converts the
// result of the temporary real format into a temporary integer format. If it is a negative
// number, set the integer sign bit. Finally, the integer is saved to the user memory.
// Parameters: tmp - temporary real format result value; info - info structure pointer;
// code - instruction code.
196 void put\_longlong\_int(const temp\_real * tmp,
197     struct info * info, unsigned short code)
198 {
199     char * addr;
200     temp\_int ti;
201
202     addr = ea(info, code);                // get effective address.
203     real\_to\_int(tmp, &ti);                // change to temporary integer.
204     verify\_area(addr, 8);                // verify that 8 bytes are available.
205     if (ti.sign)                        // if signed, change to negative value.
206         __asm__ ("notl %0 ; notl %l\n\t"    // invert and add one.
207             "addl $1,%0 ; adcl $0,%l"
208             : "=r" (ti.a), "=r" (ti.b)
209             : "" (ti.a), "l" (ti.b));
210     put\_fs\_long(ti.a, (unsigned long *) addr);    // store to the user data area.
211     put\_fs\_long(ti.b, 1 + (unsigned long *) addr);
212 }
213
// The unsigned number <high, low> is divided by 10 and the remainder is placed in rem.
214 #define DIV10(low, high, rem) \
215     __asm__ ("divl %6 ; xchgl %1,%2 ; divl %6" \
216         : "=d" (rem), "=a" (low), "=b" (high) \
217         : "" (0), "l" (high), "2" (low), "c" (10))
218
// Save the result in the BCD code format to the user data area.
// This function first obtains the address addr for saving the result and verifies the user
// space for storing the 10-byte BCD code. The result of the temporary real number format is
// then converted to data in BCD code format and saved to the user memory. If it is negative,
// the most significant bit of the highest memory byte is set.
// Parameters: tmp - temporary real format result value; info - info structure pointer;
// code - instruction code.
219 void put\_BCD(const temp\_real * tmp, struct info * info, unsigned short code)
220 {
221     int k, rem;
222     char * addr;
223     temp\_int i;
224     unsigned char c;
225
226     addr = ea(info, code);                // get the address for storing result.
227     verify\_area(addr, 10);                // verify memory space.
228     real\_to\_int(tmp, &i);                // change to temporary integer.

```

---

```

229         if (i.sign)                                // if signed, set MSB bit of high byte.
230             put_fs_byte(0x80, addr+9);
231         else                                         // otherwise reset the MSB bit.
232             put_fs_byte(0, addr+9);
233         for (k = 0; k < 9; k++) {                   // change to BCD code and stored.
234             DIV10(i.a, i.b, rem);
235             c = rem;
236             DIV10(i.a, i.b, rem);
237             c += rem<<4;
238             put_fs_byte(c, addr++);
239         }
240     }
241

```

---

## 11.9 mul.c

### 11.9.1 Function

The functions in the mul.c program use the CPU's normal arithmetic instructions to simulate the 80387 multiplication.

### 11.9.2 Code annotation

Program 11-8 linux/kernel/math/mul.c

---

```

1  /*
2   * linux/kernel/math/mul.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * temporary real multiplication routine.
9   */
10
11 #include <linux/math_emu.h>
12
13 // Shift the 16-byte value at the parameter c pointer to the left by 1 bit (multipl by 2).
14 static void shift(int * c)
15 {
16     __asm__ ("movl (%0), %%eax ; addl %%eax, (%0) |n|t"
17             "movl 4(%0), %%eax ; adcl %%eax, 4(%0) |n|t"
18             "movl 8(%0), %%eax ; adcl %%eax, 8(%0) |n|t"
19             "movl 12(%0), %%eax ; adcl %%eax, 12(%0)"
20             ":: "r" ((long) c): "ax");
21 }
22
23 // Two temporary reals are multiplied, the result is placed at the c pointer (16 bytes).
24 static void mul64(const temp_real * a, const temp_real * b, int * c)
25 {

```



```

24     __asm__( "movl (%0), %%eax|n|t"
25              "mull (%1)|n|t"
26              "movl %%eax, (%2)|n|t"
27              "movl %%edx, 4(%2)|n|t"
28              "movl 4(%0), %%eax|n|t"
29              "mull 4(%1)|n|t"
30              "movl %%eax, 8(%2)|n|t"
31              "movl %%edx, 12(%2)|n|t"
32              "movl (%0), %%eax|n|t"
33              "mull 4(%1)|n|t"
34              "addl %%eax, 4(%2)|n|t"
35              "adcl %%edx, 8(%2)|n|t"
36              "adcl $0, 12(%2)|n|t"
37              "movl 4(%0), %%eax|n|t"
38              "mull (%1)|n|t"
39              "addl %%eax, 4(%2)|n|t"
40              "adcl %%edx, 8(%2)|n|t"
41              "adcl $0, 12(%2)"
42              ":: "b" ((long) a), "c" ((long) b), "D" ((long) c)
43              : "ax", "dx");
44 }
45
// Simulation floating-point calculation instruction FMUL (Floating-point Multiply).
// Temporary reals src1 * src2 -> result.
46 void fmul(const temp_real * src1, const temp_real * src2, temp_real * result)
47 {
48     int i, sign;
49     int tmp[4] = {0, 0, 0, 0};
50
// First determine the sign of the multiplication of the two numbers. The sign is equal to the
// sign bit XOR of both. Then calculate the multiplied exponent value. The exponent needs to
// be added when multiplying. However, since the exponent is stored in a biased number format,
// the biased amount is added twice when the exponents of the two numbers are added, so it is
// necessary to subtract a biased number ( The biased number of the temporary real is 16383).
51     sign = (src1->exponent ^ src2->exponent) & 0x8000;
52     i = (src1->exponent & 0x7fff) + (src2->exponent & 0x7fff) - 16383 + 1;

// If the resulting exponent becomes negative, it means that the two numbers are multiplied
// to produce an underflow. So directly return the signed zero value.
// If the result exponent is greater than 0x7fff, it indicates that an overflow occurred, so
// the status word overflow exception flag is set and returned.
53     if (i < 0) {
54         result->exponent = sign;
55         result->a = result->b = 0;
56         return;
57     }
58     if (i > 0x7fff) {
59         set_OE();
60         return;
61     }

// If the mantissas of two numbers are multiplied and the result is not 0, the resulting mantissa
// is normalized. That is, the resulting mantissa value is shifted to the left so that the most
// significant bit is 1, and the exponent is adjusted accordingly. If the mantissa of the 16-byte

```

---

```

// result obtained by multiplying the mantissa of the two numbers is 0, the exponent is also
// set to 0. Finally, the multiplication result is saved in the temporary real 'result'.
62     mul64(src1, src2, tmp);
63     if (tmp[0] || tmp[1] || tmp[2] || tmp[3])
64         while (i && tmp[3] >= 0) {
65             i--;
66             shift(tmp);
67         }
68     else
69         i = 0;
70     result->exponent = i | sign;
71     result->a = tmp[2];
72     result->b = tmp[3];
73 }
74

```

---

## 11.10 div.c

### 11.10.1 Function

The div.c program uses the CPU normal calculation instructions to simulate the division of the 80387 coprocessor.

### 11.10.2 Code annotation

Program 11-9 linux/kernel/math/div.c

---

```

1  /*
2   * linux/kernel/math/div.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * temporary real division routine.
9   */
10
11 #include <linux/math_emu.h>
12
13 // Shift the contents of the 4 bytes to the left by 1 bit (multiply by 2).
14 static void shift\_left(int * c)
15 {
16     __asm__ __volatile__(
17         "movl 4(%0), %%eax ; adcl %%eax, 4(%0) |n|t"
18         "movl 8(%0), %%eax ; adcl %%eax, 8(%0) |n|t"
19         "movl 12(%0), %%eax ; adcl %%eax, 12(%0)"
20         "::"r" ((long) c):"ax");
21
22 // Shift the contents of the 4 bytes pointed to by pointer c to the right by 1 bit.

```

```

22 static void shift\_right(int * c)
23 {
24     __asm__ ("shrl $1, 12(%0) ; rcr1 $1, 8(%0) ; rcr1 $1, 4(%0) ; rcr1 $1, (%0)"
25             :: "r" ((long) c));
26 }
27
// 16-byte subtraction function.
// 16-byte subtraction, (a - b) -> a. Finally, ok is set according to whether there is a borrow
// flag (CF=1). If there is no borrow (CF=0) then ok = 1, otherwise ok = 0.
28 static int try\_sub(int * a, int * b)
29 {
30     char ok;
31
32     __asm__ __volatile__ ("movl (%1), %%eax ; subl %%eax, (%2) |n|t"
33                          "movl 4(%1), %%eax ; sbb1 %%eax, 4(%2) |n|t"
34                          "movl 8(%1), %%eax ; sbb1 %%eax, 8(%2) |n|t"
35                          "movl 12(%1), %%eax ; sbb1 %%eax, 12(%2) |n|t"
36                          "setae %%al": "=a" (ok): "c" ((long) a), "d" ((long) b));
37     return ok;
38 }
39
// 16-byte division function.
// Parameter a / b -> c. The method is to simulate multi-byte division using subtraction.
40 static void div64(int * a, int * b, int * c)
41 {
42     int tmp[4];
43     int i;
44     unsigned int mask = 0;
45
46     c += 4;
47     for (i = 0 ; i < 64 ; i++) {
48         if (!(mask >>= 1)) {
49             c--;
50             mask = 0x80000000;
51         }
52         tmp[0] = a[0]; tmp[1] = a[1];
53         tmp[2] = a[2]; tmp[3] = a[3];
54         if (try\_sub(b, tmp)) {
55             *c |= mask;
56             a[0] = tmp[0]; a[1] = tmp[1];
57             a[2] = tmp[2]; a[3] = tmp[3];
58         }
59         shift\_right(b);
60     }
61 }
62
// Simulate floating point instruction FDIV.
// Temporary real division: src1 / src 2 -> result.
63 void fdiv(const temp\_real * src1, const temp\_real * src2, temp\_real * result)
64 {
65     int i, sign;
66     int a[4], b[4], tmp[4] = {0, 0, 0, 0};
67

```

```
68     sign = (src1->exponent ^ src2->exponent) & 0x8000;
69     if (!(src2->a || src2->b)) {
70         set\_ZE\(\);
71         return;
72     }
73     i = (src1->exponent & 0x7fff) - (src2->exponent & 0x7fff) + 16383;
74     if (i < 0) {
75         set\_UE\(\);
76         result->exponent = sign;
77         result->a = result->b = 0;
78         return;
79     }
80     a[0] = a[1] = 0;
81     a[2] = src1->a;
82     a[3] = src1->b;
83     b[0] = b[1] = 0;
84     b[2] = src2->a;
85     b[3] = src2->b;
86     while (b[3] >= 0) {
87         i++;
88         shift\_left(b);
89     }
90     div64(a, b, tmp);
91     if (tmp[0] || tmp[1] || tmp[2] || tmp[3]) {
92         while (i && tmp[3] >= 0) {
93             i--;
94             shift\_left(tmp);
95         }
96         if (tmp[3] >= 0)
97             set\_DE\(\);
98     } else
99         i = 0;
100     if (i > 0x7fff) {
101         set\_OE\(\);
102         return;
103     }
104     if (tmp[0] || tmp[1])
105         set\_PE\(\);
106     result->exponent = i | sign;
107     result->a = tmp[2];
108     result->b = tmp[3];
109 }
110
```

---

## 11.11 Summary

This chapter describes the method and code implementation of the Linux kernel for emulating the 80387 math coprocessor chip. First, we introduced several commonly used integer and floating point types, described the 80387 runtime and the temporary real types used in software simulation, and gave their specific representation format. Then we introduce the composition and working method of the math coprocessor, and

explain the working implementation of each emulation code program with the `math_emulate.c` program as the main line.

In the next chapter we provide a comprehensive introduction to the MINIX file system used by kernel 0.12. After a brief description of the MINIX file system structure and various file types, we will detail how the cache used to access the file system works. It also briefly describes the role of each underlying file and function and the main functions for accessing data in various files, including file and directory management functions provided by system calls. Later, before starting to comment on the code in detail, an example of a simple file system on the block device was specifically presented and described.




## 12 File System (fs)

This chapter covers the implementation code for the file system in the Linux kernel and the cache manager for block devices. When developing the Linux 0.12 kernel file system, Mr. Linus mainly referred to the MINIX operating system at the time, and used the 1.0 version of the MINIX file system. Therefore, when reading the contents of this chapter, you can first read about the MINIX file system. For the introduction of the working principle and implementation method of the cache, you can first browse the book "Design of UNIX Operating System" by Mr. M.J.Bach.

There are 18 source files associated with the file system implementation, as shown in Listing 12-1. The cross-references between these files and the functions in them can be found in Section 5.10.

List 12-1 linux/fs

Filename	Size	Last modified time (GMT)	Desc.
 <a href="#">Makefile</a>	7176 bytes	1992-01-12 19:49:06	
 <a href="#">bitmap.c</a>	4007 bytes	1992-01-11 19:57:29	
 <a href="#">block_dev.c</a>	1763 bytes	1991-12-09 21:11:23	
 <a href="#">buffer.c</a>	9072 bytes	1991-12-06 20:21:00	
 <a href="#">char_dev.c</a>	2103 bytes	1991-11-19 09:10:22	
 <a href="#">exec.c</a>	9908 bytes	1992-01-13 23:36:33	
 <a href="#">fcntl.c</a>	1455 bytes	1991-10-02 14:16:29	
 <a href="#">file_dev.c</a>	1852 bytes	1991-12-01 19:02:43	
 <a href="#">file_table.c</a>	122 bytes	1991-10-02 14:16:29	
 <a href="#">inode.c</a>	7166 bytes	1992-01-10 22:27:26	
 <a href="#">ioctl.c</a>	1136 bytes	1991-12-21 01:58:35	
 <a href="#">namei.c</a>	18958 bytes	1992-01-12 04:09:58	
 <a href="#">open.c</a>	4862 bytes	1992-01-08 20:01:36	
 <a href="#">pipe.c</a>	2834 bytes	1992-01-10 22:18:11	
 <a href="#">read_write.c</a>	2802 bytes	1991-11-25 15:47:20	
 <a href="#">select.c</a>	6381 bytes	1992-01-13 22:25:23	
 <a href="#">stat.c</a>	1875 bytes	1992-01-11 20:39:19	
 <a href="#">super.c</a>	5603 bytes	1991-12-09 21:11:34	
 <a href="#">truncate.c</a>	1692 bytes	1992-01-11 19:47:28	

### 12.1 Main Functions

The file system is an important part of the operating system and is the place where the operating system stores a large amount of programs and data for a long time. When the system loads the executable program, it needs to be quickly read from the file system to the memory to run. Some temporary files generated during the

system operation also need to be dynamically saved in the file system. Therefore, the file system needs to use high-speed devices to store programs and data, so the operating system usually uses a block device that can store a large amount of information as a device of the file system. In addition, UNIX-like operating systems usually access devices through device files, so the composition and implementation of the file system is very complicated.

The procedures noted in this chapter are large, but through the analysis of the Linux source code directory structure in Section 5.10 (see Figure 5-29), we can divide these files into four parts for discussion. The first part is about the high-speed buffer (cache) management program, which mainly implements the function of high-speed data access to block devices such as hard disks. This part of the content is concentrated in the `buffer.c` program; The second part of the code describes the low-level generic functions of the file system. The management of file index nodes, the allocation and release of disk data blocks, and the conversion algorithm of file names and i nodes are described; The third part of the program is about reading and writing data in the file, including access to data in character devices, pipes, and block read and write files; The fourth part of the program mainly involves the implementation of the system-call interface of the file, mainly related to the system-calls of file opening, closing, creation and related file directory operations.

Let's first introduce the basic structure of the MINIX file system, and then explain the four parts separately.

### 12.1.1 MINIX file system

At present, the version of the MINIX operating system is 2.0, and the file system used is version 2.0. It is different from the version before the 1.5 version of the system, and its capacity has been expanded. However, since the Linux kernel annotated in this book uses the MINIX file system version 1.0, only the version 1.0 file system is briefly introduced here.

The MINIX file system is basically the same as the standard UNIX file system. It consists of six parts: (1) Boot block, (2) Super block, (3) i node bitmap, (4) Logic block bitmap, (5) i nodes, (6) Data Blocks. For a common disk block device, the distribution of its parts is shown in Figure 12-1.

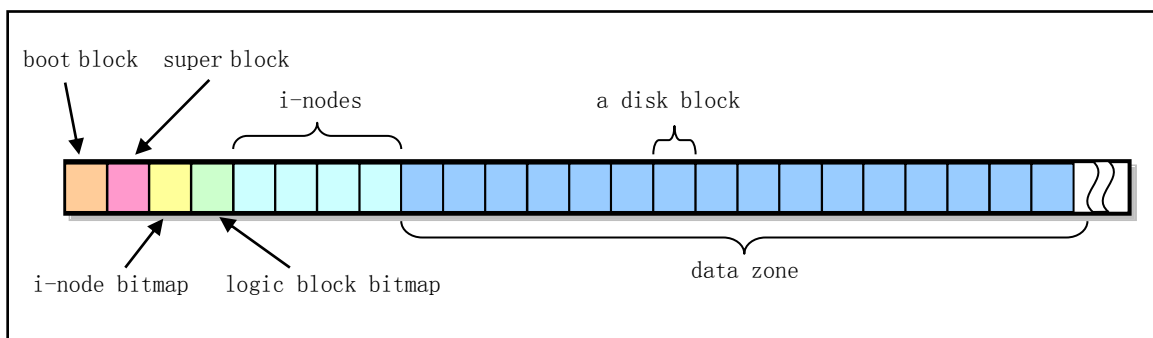


Figure 12-1 Schematic diagram of the layout of each part of the MINIX file system

In the figure, the entire block device is divided into disk blocks in units of 1 KB, so for a 360 KB floppy disk device, there are 360 disk blocks in the above figure, and each square represents one disk block. As we will see in the following description, in the MINIX 1.0 file system, the disk block size is exactly the same as the logical block size, which is also 1 KB. Therefore 360KB discs also contain 360 logical blocks. We will sometimes mix these two names in the discussion that follows.

The boot block is an execution code and data disk block that can be automatically read by the ROM BIOS when the computer is powered on. However, not all disk devices in a system are used as boot devices, so for disks that are not used for booting, this disk block may contain no code. However, any disk block device must



have boot block space to maintain the uniformity of the MINIX file system format. That is, the file system simply leaves a space for storing the boot block on the block device. If you put the kernel image file in the file system, you can store the actual bootloader in the first block of the device where the filesystem resides (ie the boot blockspace), and let it get and load the kernel image file in the filesystem.

For a large-capacity hard disk block device, several partitions can usually be divided on it, and a different complete file system can be stored in each partition, as shown in Figure 12-2. The figure shows that there are 4 partitions, which store the FAT32 file system, NTFS file system, MINIX file system and EXT2 file system. The first sector of the hard disk is the master boot sector, which stores the hard disk boot code and partition table information. The information in the partition table indicates the type of each partition on the hard disk, the starting position parameter and the ending position parameter, and the total number of sectors occupied in the hard disk. See the hard disk partition table structure after the kernel/blk\_drv/hd.c file.

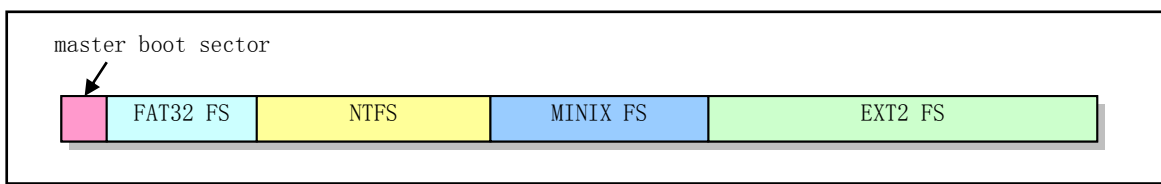


Figure 12-2 Partitions and file systems on hard disk devices

The super block is used to store the structure information of the file system on the disk device, and describes the size of each part. The structure is shown in Figure 12-3. Where `s_ninodes` represents the total number of i-nodes on the device; `s_nzones` represents the total number of logical blocks in the logical block on the device; `s_imap_blocks` and `s_zmap_blocks` represent the number of disk blocks occupied by the i-node bitmap and the logical block bitmap, respectively; `s_firstdatazone` represents the first logical block number occupied by the beginning of the data area on the device; `S_log_zone_size` is the number of disk blocks contained in each logical block represented by a base 2 logarithm. For the MINIX 1.0 file system, this value is 0, so the size of its logical block is equal to the disk block size, which is 1 KB. `s_max_size` is the maximum file length in bytes, which does not exceed 4GB. Of course, this length value will be limited by the disk capacity. `S_magic` is the file system magic number used to indicate the type of file system. For the MINIX 1.0 file system, its magic number is 0x137f.

In the Linux 0.12 system, the loaded file system superblock is stored in the superblock table (array) `super_block[]`. There are 8 entries in this table, so the Linux 0.12 system can load up to 8 file systems at the same time. The superblock table will be initialized in the `mount_root()` function of the `super.c` program. In the `read_super()` function, a superblock will be set in the table for the newly loaded filesystem, and the superblock will be released in the `put_super()` function.

	Field name	Data type	Description
available in disk and memory	s_ninodes	short	Number of i-nodes
	s_nzones	short	Number of zones.
	s_imap_blocks	short	Number of blocks for i-node bitmap
	s_zmap_blocks	short	Number of blocks for logic bitmap
	s_firstdatazone	short	Fisrt logic block no. in data zone.
	s_log_zone_size	short	$\text{Log}_2(\text{DiskBlocks}/\text{LogicBlocks})$
	s_max_size	long	Maximum file size
	s_magic	short	FS magic number (0x137f)
available only in memory.	s_imap[8]	buffer_head *	i-node bitmap block array in cache.
	s_zmap[8]	buffer_head *	Logic block bitmap block array in cache.
	s_dev	short	Device no of the super block.
	s_isup	m_inode *	i-node of the installed fs.
	s_imount	m_inode *	i-node the fs installed to.
	s_time	long	Modified time.
	s_wait	task_struct *	Tasks waiting for this superblock.
	s_lock	char	Locked flag.
	s_rd_only	char	Readonly flag.
	s_dirt	char	Modified flag (dirty flag)

Figure 12-3 Super block structure of MINIX file system

The logic block bitmap is used to describe the usage of each data block on the disk. In addition to the first bit (bit 0), each bit in the logical block bitmap represents, in turn, a logical block in the data area on the disk. Therefore, bit 1 of the logic block bitmap represents the first data disk block in the data area on the disk, not the first disk block (boot block) on the disk. When a data disk block is occupied, the corresponding bit in the logic block bitmap is set. Since the function to find the free disk block returns a value of 0 when all disk data blocks are occupied, the lowest bit (bit 0) of the logic block bitmap is idle and is set to 1 when the file system is created.

From the structure of the super block, we can also see that the logic block bitmap uses up to 8 buffer blocks (s\_zmap[8]), and each block size is 1024 bytes, and each bit represents the occupation status of one disk block. Therefore a buffer block can represent 8192 disk blocks, and 8 buffer blocks can represent a total of 65,536 disk blocks, so the maximum block device capacity (length) that MINIX file system 1.0 can support is 64 MB.

The i-node is used to store index information for each file and directory name on the disk device. The i-node bitmap is used to indicate whether the i-node is used, and each bit represents an i-node. For a size of 1K block, a disk block can represent the usage of 8192 i-nodes. Similar to the case of a logical block bitmap, since the function that looks for an idle i-node returns a value of 0 when all i-nodes are used, the lowest bit (bit 0) of the first byte of the i-node bitmap and the corresponding i-node 0 is left idle, and the bit position in the corresponding bit map of the i-node 0 is set to 1 in advance when the file system is created. Therefore, only the state of 8191 i-nodes can be represented in the first i-node bitmap block.

The i-node part of the disk holds the index node of the file or directory name in the file system, and each file or directory name has an i-node. Each i-node structure stores information about the corresponding file or directory, such as the file host's id (uid), the file's group id (gid), the file length, the access modification time, and the location of the file data block on the disk. A total of 32 bytes are used for the entire i-node structure, as shown in Figure 12-4.

	Field name	Data type	Description
fields in disk and memory total 32 bytes	i_mode	short	File types & attributes (rwx bits)
	i_uid	short	File owner's user id.
	i_size	long	File size (bytes)
	i_mtime	long	Modified time (from 1970.1.1:0, sec)
	i_gid	char	File group id
	i_nlinks	char	Number of links (how many directory entries point to this i-node).
	i_zone[9]	short	Logic block no. occupied by file: zone[0]-zone[6] direct block no. zone[7] indirect block no. zone[8] 2 <sup>nd</sup> indirect block no. Note: for device file, zone[0] is device no. of the specified device file.
fields available only in memory	i_wait	task_struct *	tasks waiting for the i-node.
	i_atime	long	Last access time.
	i_ctime	long	i-node modified time.
	i_dev	short	Device no the i-node belongs.
	i_num	short	i-node no.
	i_count	short	Reference count of the i-node. 0 - idle.
	i_lock	char	i-node is locked.
	i_dirt	char	i-node has been modified (dirty flag).
	i_pipe	char	i-node is used for pipe.
	i_mount	char	i-node was installed with other fs.
	i_seek	char	Search flag (for lseek).
	i_update	char	i-node updated flag.

Figure 12-4 i-node structure of MINIX file system version 1.0

The `i_mode` field is used to save the file type and access rights properties. Its bits 15-12 are used to save the file type, bits 11-9 hold the information set when the file is executed, and bits 8-0 indicate the access rights of the file, as shown in Figure 12-5. See the file `include/sys/stat.h` on lines 20 -- 50 and file `include/fcntl.h` for details.

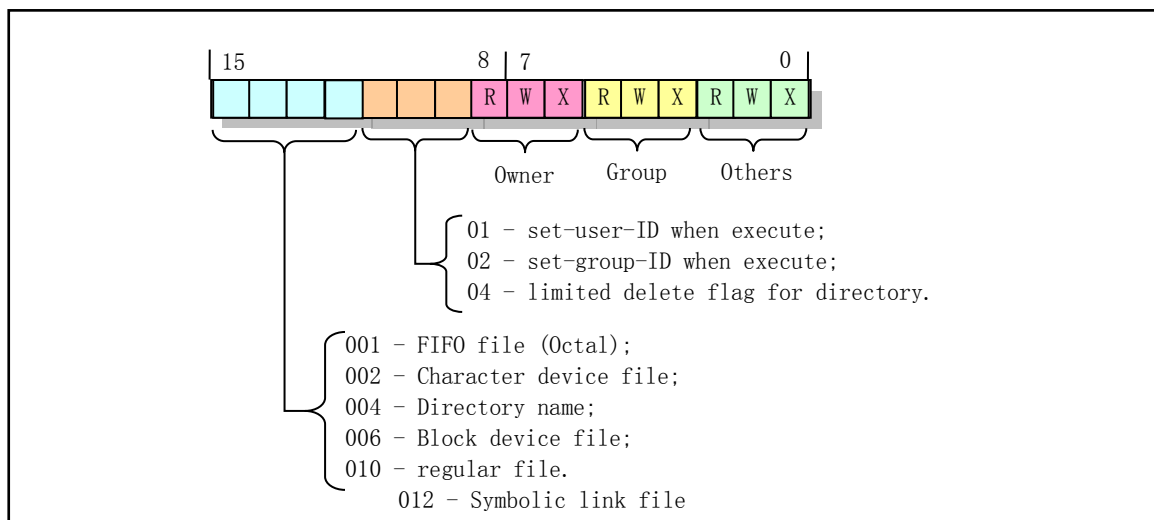


Figure 12-5 i-node mode field content

The data in the file is stored in the data area of the disk block, and a file name is associated with the data disk block through the corresponding i-node. The number of these disk blocks is stored in the logical block array `i_zone[]` of the i-node. The `i_zone[]` array is used to store the disk block number of the file corresponding to the i-node. `i_zone[0]` to `i_zone[6]` are used to store the 7 disk block numbers at the beginning of the file, called direct blocks. If the file length is less than or equal to 7K bytes, the disk block it uses can be quickly found according to its i-node. If the file is larger, you need to use an indirect block (`i_zone[7]`), which holds the additional block number. For the MINIX file system it can store 512 disk block numbers, so 512 disk blocks can be addressed. If the file is larger, you need to use a secondary indirect disk block (`i_zone[8]`). The primary disk block of the secondary indirect block functions similarly to the one-time indirect disk block, so 512\*512 disk blocks can be addressed using the secondary indirect disk block, as shown in Figure 12-6. So for MINIX file system version 1.0, the maximum length of a file is  $(7 + 512 + 512 * 512) = 262,663$  KB.

In addition, for device files in the `/dev/` directory, they do not occupy the data disk blocks in the disk data area, that is, the length of their files is zero. The i-node of the device file name is only used to save the attributes and device numbers of the device it defines. The device number is stored in `zone[0]` of the device file i node.

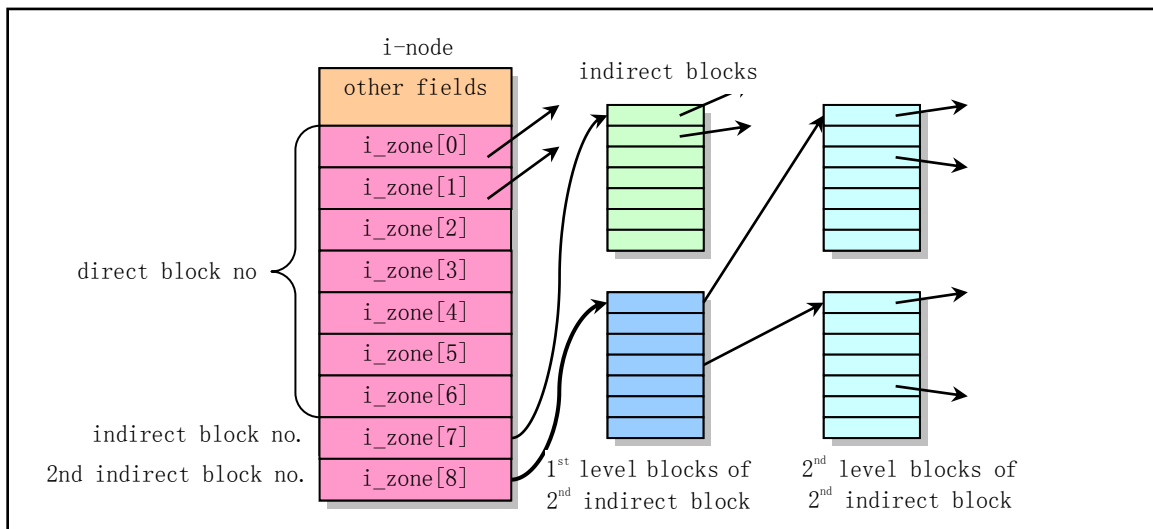


Figure 12-6 The function of the logical block array of the i-node

Here again, when all i-nodes are used, the function that looks for the idle i-node will return a value of 0. Therefore, the lowest bit of the i-node bitmap and the i-node 0 are not used. The structure of i-node 0 is initialized to all zeros, and the bit position of 0 in the i-node bitmap is set when the file system is created.

For a PC, the length of one sector (512 bytes) is generally used as the data block length of the block device. The MINIX file system treats two consecutive sectors of data (1024 bytes) as one block of data, called a disk block, and its length is the same as the buffer block length in the cache. The number is counted from the first disk block on the disk, that is, the boot block is the 0th disk block.

The above logical block or block is a power multiple of 2 of the disk block. A logical block length can be equal to 1, 2, 4 or 8 disk block lengths. For the Linux kernel discussed in this book, the length of the logic block is equal to the length of the disk block, so the two terms have the same meaning in the code comments. However, the term data logic block (or data disk block) refers to the disk block numbered from the first data disk block in the data portion of the disk device.

## 12.1.2 File Types, Attributes, and Directory Items

### 12.1.2.1 File types and attributes

Files in UNIX-like operating systems can be generally divided into six categories: (1) regular file; (2) directory name; (3) symbolic link file; (4) named pipe file; (5) character device file; (6) block device files. If you execute the "ls -l" command at the shell command line prompt, we can know the type of the file from the listed file status information, as shown in Figure 12-7.

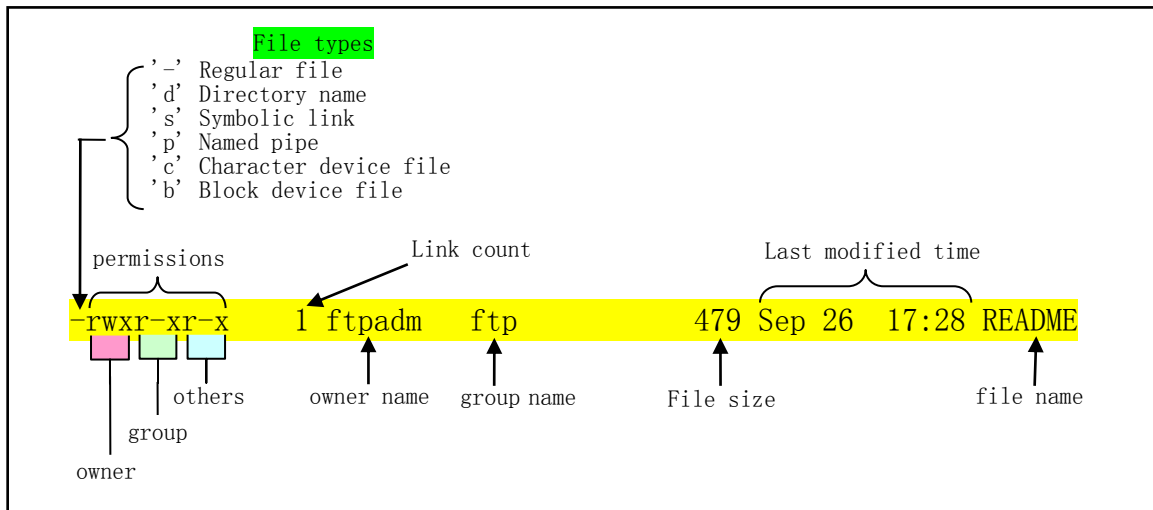


Figure 12-7 File information displayed by the command 'ls -l'

In the figure, the first character of the file information displayed indicates the type of the listed file. For example, the '-' in the figure indicates that the file is a regular (normal) file.

A regular file ('-') is a file that the file system does not interpret and can contain byte streams of any length. For example, source files, binary executables, documents, and script files.

The directory ('d') also appears as a file in the UNIX-like operating system, but file system management interprets its contents so that people can see which files are contained in a directory and how they are organized together to form a hierarchical file system.

A symbolic link ('s') is used to reference another file with a different file name. A symbolic link can connect to a file in another file system. Deleting a symbolic link does not affect the file being connected. There is also a connection method called "hard link". It has the same status as the linked file in the symbolic link described here, and is treated as a general file, but cannot be linked across file systems (or devices) and increments the link count value of the file. See the description of the link count below.

A named pipe ('p') file is a file created when the system creates a named pipe and can be used for communication between unrelated processes.

Character device ('c') files are used to access character devices, such as tty terminals, memory devices, and network devices, by operating files.

A block device ('b') file is used to access devices such as hard disks, floppy disks, and the like. In UNIX-like operating systems, block device files and character device files are generally stored in the /dev directory of the system.

In the Linux kernel, the file type information is stored in the `i_mode` field of the corresponding i-node and is represented by the upper 4 bits. When working with the file system, the system uses some macros that

determine file types, such as `S_ISBLK`, `S_ISDIR`, etc. These macros are defined in the `include/sys/stat.h` file.

In the figure, the file type character is followed by three sets of file permission attributes composed of one set of three characters, which are used to indicate the access rights of the file host, the same group of users, and other users to the file. 'rwx' indicates permission to read, write, and execute the file, respectively. For directory name files, executable means that you can enter the directory. When operating on the permissions of a file, they are generally represented in octal. For example, '755' (0b111, 101, 101) indicates that the file host can read/write/execute the file, and the same group of users and others can read and execute the file. In the Linux 0.12 source code, the file permission information is also stored in the `i_mode` field of the i-node, and the lower 9 bits of the field are used to represent the three sets of permissions, and are often represented by the variable `mode`. Macros for file permissions are defined in the file `include/fcntl.h`.

The 'Link Count' field in the figure indicates the number of times the file was referenced by a hard link. When the count is reduced to zero, the file is deleted. 'Username' indicates the name of the file host, and 'Group name' is the name of the group to which the user belongs.

### 12.1.2.2 File System Directory Item Structure

The Linux 0.12 system uses the MINIX file system version 1.0. Its directory structure and directory entry structure is the same as that of the traditional UNIX file system, and is defined in the `include/linux/fs.h` file. In a directory of the file system, directory entries for all file names in the directory are stored in data blocks of the directory file. For example, the directory entry for all file names under the directory name `root/` is stored in the data block of the `root/` directory name file. All file name information in the root directory of the file system is stored in the data block of the specified i-node (ie, the i-node of number 1). The file name directory entry structure is as follows:

---

```
// Defined in the include/linux/fs.h file.
#define NAME_LEN 14                // maximum name length.
#define ROOT_INO 1                // root i-node.

// File directory entry structure.
struct dir_entry {
    unsigned short inode;           // i-node number.
    char name[NAME_LEN];           // filename.
};
```

---

Each file directory entry includes only a file name string of 14 bytes in length and a 2-byte i-node number corresponding to the file name. Therefore, a logical disk block can store  $1024/16=64$  directory entries. Other information about the file is stored in the i-node structure specified by the i-node number. The structure mainly includes information such as file access attribute, host, length, access save time, and disk block. The i-node of each inode number is located at a fixed location on the disk.

When a file is opened, the file system will find its i-node number according to the given file name, and find the disk block location where the file is located by its corresponding i-node information, as shown in Figure 12-8. For example, to find the i-node number of the file name `/usr/bin/vi`, the file system first starts from the root directory with the fixed i-node number (1), that is, the name is found from the data block of the i-node number 1. The directory entry of `usr`, thus getting the i node number of the file `/usr`. According to the i-node number, the file system can smoothly obtain the directory `/usr`, and the directory entry of the file name `bin` can be found therein. In this way, we also know the i-node number of `/usr/bin`, so we can know the location of the directory `/usr/bin` directory, and find the directory entry of the `vi` file in the directory. Finally, we can get the i-node number of the file path name `/usr/bin/vi`, so that the i-node structure information can be obtained from

the disk.

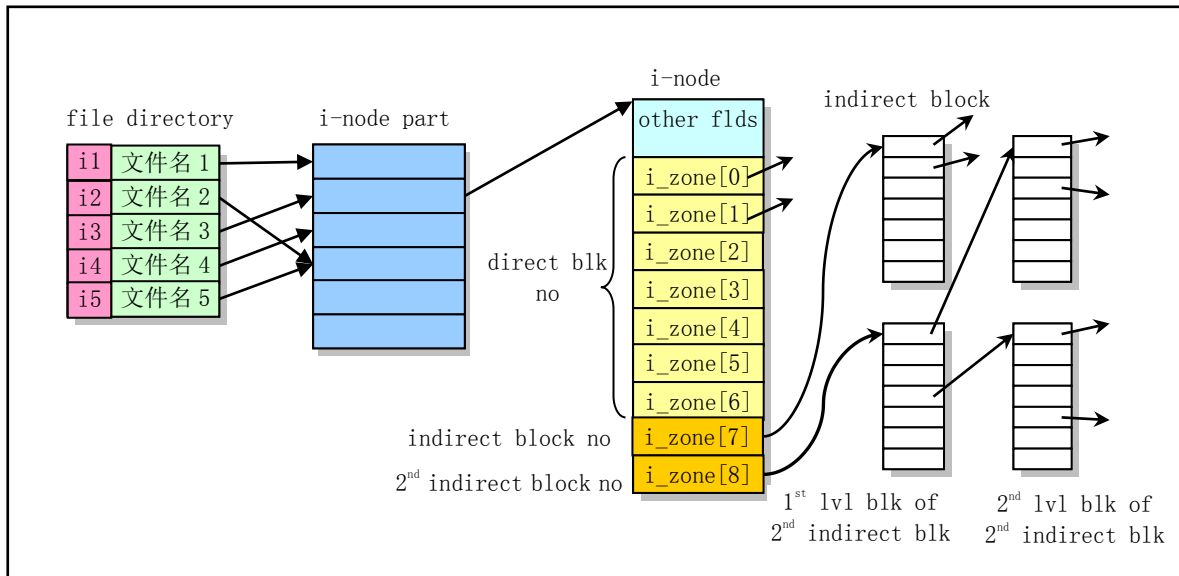


Figure 12-8 Find the file disk block location by file name

If you look at the distribution of a file on disk, the process of searching for a file block information can be represented by Figure 12-9 (where boot blocks, super blocks, i-nodes, and logical block bitmaps are not shown).

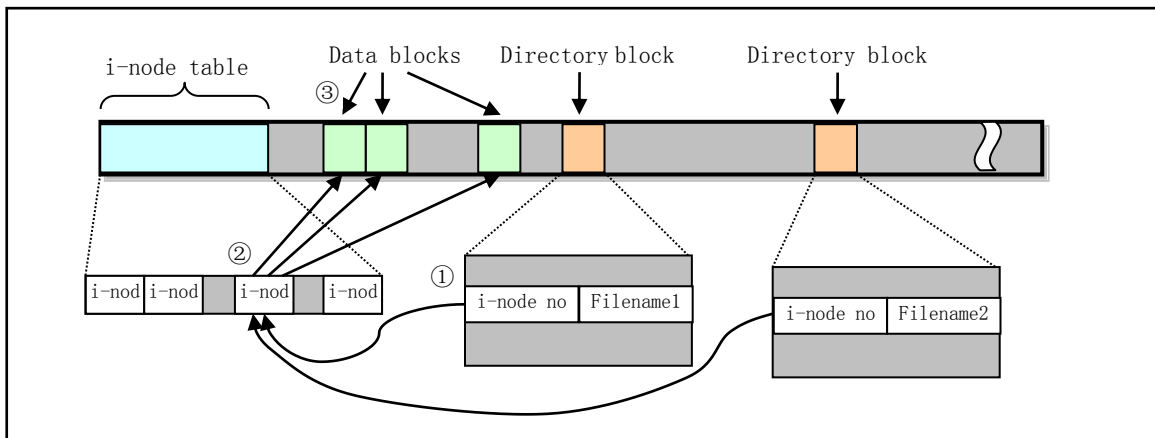


Figure 12-9 Get its data block from the file name

We can find the corresponding directory entry by the file name specified by the user program. According to the i-node number in the directory entry, the corresponding i-node structure in the i-node table can be found. The i-node structure contains the block number information of the file data, so that the data information corresponding to the file name can be finally obtained. In the figure, there are two directory entries pointing to the same i-node, so the same data on the disk can be obtained based on the two file names. There is a link count field `i_nlinks` in each i-node structure that records the number of directory entries pointing to the i-node, that is, the hard link count value of the file. In this case, the value of this field is 2. When performing a delete file operation, the kernel will actually delete the data of the file on the disk only when the i-node link count value is equal to zero. In addition, since the i-node number in the directory entry can only be used in the current file system, the directory entry of one file system cannot be used to point to the i-node in another file system, that is,

the hard link cannot cross the file system.

Unlike hard links, file name directory entries for symbolic link types do not directly point to the corresponding i-node. The symbolic link directory entry stores the pathname string of a file in the data block of the corresponding file. When accessing a symbolic link directory entry, the kernel reads the contents of the file and then accesses the specified file based on the pathname string. Therefore, symbolic links may not be limited to a file system. We can create a symbolic link in a file system that points to a file name in another file system.

There are also two special file directory entries in each directory, whose names are fixed to '.' and '..' respectively. The '.' directory entry gives the i-node number of the current directory, and the '..' directory entry gives the i-node number of the parent directory. Therefore, when a relative path name is given, the file system can use these two special directory entries for lookup operations. For example, to find './kernel/Makefile', you can first get the i-node number of the parent directory according to the '..' directory entry of the current directory, and then perform the lookup operation according to the process described above.

For a directory entry for each directory file, the link count field in its i-node also indicates the number of directory entries connected to that directory. Therefore, the link count value for each directory file is at least 2. One of them is a link to a directory entry that contains a directory file, and the other is a link to a '..' directory entry in the directory. For example, if we create a subdirectory named 'mydir' in the current directory, the link between the current directory and the subdirectory is shown in Figure 12-10.

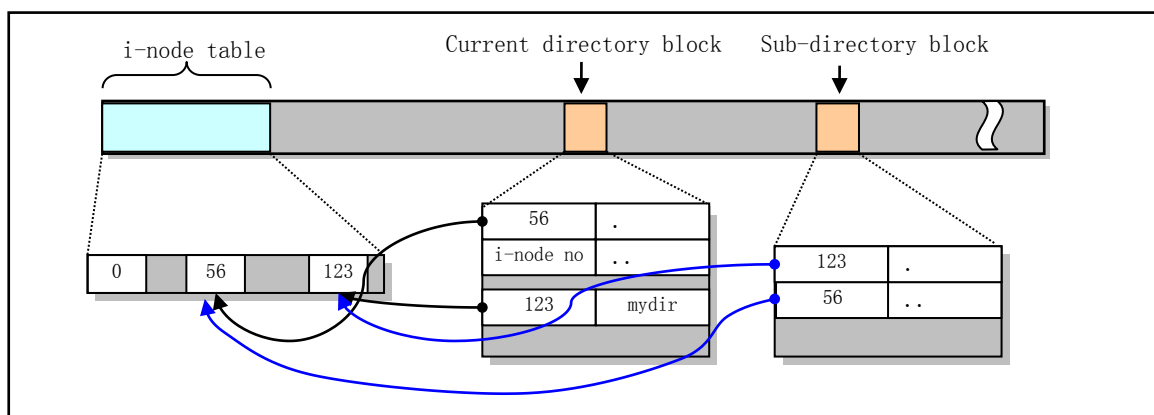


Figure 12-10 Directory file entries and subdirectory links

The figure shows that we have created a 'mydir' subdirectory in the directory with i node number 56, and the i node number of this subdirectory is 123. The '.' directory entry in the 'mydir' subdirectory points to its own i-node 123, while its '..' directory entry points to the i-node 56 of its parent directory. It can be seen that since the directory entry of a directory always has two links, if there are subdirectories, the number of i-node links of the parent directory is equal to the number of 2+ subdirectories.

### 12.1.2.3 Directory Structure Example

Take the Linux 0.12 system as an example, let's look at its root directory structure. After running the Linux 0.12 system in the bochs simulation system, we first list its file system root directory entries, including the implied '.' and '..' directory entries. Then we use the hexdump command to view the contents of the data block of the '.' or '..' file, you can see the contents of each directory item contained in the root directory.

```
[/usr/root]# cd /
[/]# ls -la
total 10
```



---

```

drwxr-xr-x 10 root    root      176 Mar 21  2004 .
drwxr-xr-x 10 root    4096      176 Mar 21  2004 ..
drwxr-xr-x  2 root    4096      912 Mar 21  2004 bin
drwxr-xr-x  2 root    root      336 Mar 21  2004 dev
drwxr-xr-x  2 root    root      224 Mar 21  2004 etc
drwxr-xr-x  8 root    root      128 Mar 21  2004 image
drwxr-xr-x  2 root    root       32 Mar 21  2004 mnt
drwxr-xr-x  2 root    root       64 Mar 21  2004 tmp
drwxr-xr-x 10 root    root      192 Mar 29  2004 usr
drwxr-xr-x  2 root    root       32 Mar 21  2004 var

[/]# hexdump .
00000000 0001 002e 0000 0000 0000 0000 0000 0000 // .
00000010 0001 2e2e 0000 0000 0000 0000 0000 0000 // ..
00000020 0002 6962 006e 0000 0000 0000 0000 0000 // bin
00000030 0003 6564 0076 0000 0000 0000 0000 0000 // dev
00000040 0004 7465 0063 0000 0000 0000 0000 0000 // etc
00000050 0005 7375 0072 0000 0000 0000 0000 0000 // usr
00000060 0115 6e6d 0074 0000 0000 0000 0000 0000 // mnt
00000070 0036 6d74 0070 0000 0000 0000 0000 0000 // tmp
00000080 0000 6962 2e6e 656e 0077 0000 0000 0000 // idle, not used.
00000090 0052 6d69 6761 0065 0000 0000 0000 0000 // image
000000a0 007b 6176 0072 0000 0000 0000 0000 0000 // var
000000b0
[/]#

```

---

After executing the 'hexdump.' command, all directory entries contained in the i-node data block are listed. Each row corresponds to a directory entry. The first two bytes of each line are the i-node number, and the next 14 bytes are the file name or directory name string. If the i-node number in a directory entry is 0, it means that the directory entry is not used, or the corresponding file has been deleted or removed. The i-node numbers of the first two directory entries ('.' and '..') are all number 1. This is a special feature of the file system root directory structure, which is different from the rest of the subdirectory structure.

Now let's look at the etc/ subdirectory entry. Also using the hexdump command on the etc/ directory, we can display the directory entries contained in the etc/ subdirectory, as shown below.

---

```

[/]# ls etc -la
total 32
drwxr-xr-x  2 root    root      224 Mar 21  2004 .
drwxr-xr-x 10 root    root      176 Mar 21  2004 ..
-rw-r--r--  1 root    root      137 Mar  4  2004 group
-rw-r--r--  1 root    root    11801 Mar  4  2004 magic
-rw-r--r--  1 root    root       11 Jan 22 18:12 mtab
-rw-r--r--  1 root    root      142 Mar  5  2004 mtools
-rw-r--r--  1 root    root      266 Mar  4  2004 passwd
-rw-r--r--  1 root    root      147 Mar  4  2004 profile
-rw-r--r--  1 root    root       57 Mar  4  2004 rc
-rw-r--r--  1 root    root     1034 Mar  4  2004 termcap
-rwx--x--x  1 root    root    10137 Jan 15 1992 update

[/]# hexdump etc
00000000 0004 002e 0000 0000 0000 0000 0000 0000 // .

```

```
0000010 0001 2e2e 0000 0000 0000 0000 0000 0000 // ..
0000020 0007 6372 0000 0000 0000 0000 0000 0000 // rc
0000030 000b 7075 6164 6574 0000 0000 0000 0000 // update
0000040 0113 6574 6d72 6163 0070 0000 0000 0000 // termcap
0000050 00ee 746d 6261 0000 0000 0000 0000 0000 // mtab
0000060 0000 746d 6261 007e 0000 0000 0000 0000 // not used.
0000070 007c 616d 6967 0063 0000 0000 0000 0000 // magic
0000080 0016 7270 666f 6c69 0065 0000 0000 0000 // profile
0000090 007e 6170 7373 6477 0000 0000 0000 0000 // passwd
00000a0 0081 7267 756f 0070 0000 0000 0000 0000 // group
00000b0 01ee 746d 6f6f 736c 0000 0000 0000 0000 // mtools
00000c0
[/]#
```

---

At this point, we can see that the data block corresponding to the `etc/` directory name i-node contains the directory entry information of all the files in the subdirectory. The i-node of the directory entry `'.'` is the i-node number 4 of the `etc/` directory entry, and the i-node of `'..'` is the i-node number 1 of the `etc/parent` directory.

### 12.1.3 High speed buffer (Buffer cache)

A high-speed buffer is a must for the file system to access data in a block device. In order to access data in the file system on the block device, the kernel can access the block device each time for a read or write operation. But the time of each I/O operation is very slow compared to the processing speed of the memory and CPU. In order to improve the performance of the system, the kernel opens up a high-speed data buffer (buffer cache) in memory and divides it into buffer blocks equal in size to the disk data block for use and management, in order to reduce the number of times of accessing block devices. In the Linux kernel, the buffer cache is located between the kernel code area and the main memory area, as shown in Figure 5-5. The data block in each block device that has been used recently is stored in the buffer cache. When you need to read data from a block device, the buffer manager first looks in the buffer cache. If the corresponding data is already in the buffer, there is no need to read it from the block device again. If the data is not in the buffer cache, a command to read the block device is issued to read the data into the cache. When data needs to be written to the block device, the system will request a free buffer block in the cache to temporarily store the data. As for when to actually write the data to the device, it is achieved through the device data synchronization function.

The program that implements the buffer cache in the Linux kernel is `buffer.c`. Other programs in the file system call its block read and write functions by specifying the device number and data logic block number that need to be accessed. These interface functions are: block read function `bread()`, block advance read-ahead function `breada()` and page block read function `bread_page()`. The page block read function reads the number of buffer blocks (4 blocks) that can be accommodated in one page of memory at a time.

### 12.1.4 File System Lower Level Functions

The underlying processing functions of the file system are contained in the following five files:

- ♦ The `bitmap.c` program includes a release and occupancy handler for the i-node bitmap and the logic block bitmap. The functions that operate on the i-node bitmap are `free_inode()` and `new_inode()`, and the functions that operate on the logic block bitmap are `free_block()` and `new_block()`.
- ♦ The `truncate.c` program includes a function `truncate()` that truncates the length of the data file to zero. It cuts the file length on the device specified by the i-node to 0 and releases the device logic block occupied by the file data.

- ♦ The `inode.c` program includes a function `input()` that allocates the i-node function and a function `iput()` that puts back the memory i-node, and a function `bmap()` that takes the logical block number of the file data block on the device.
- ♦ The `namei.c` program mainly includes the function `namei()`. This function maps the given file pathname to its i-node using `iget()`, `input()`, and `bmap()`.
- ♦ The `super.c` program is designed to handle file system superblocks, including the functions `get_super()`, `put_super()`, and `free_super()`. Also included are several file system load/unload handlers and system-calls such as `sys_mount()`.

The hierarchical relationship between functions in these files is shown in Figure 12-11.

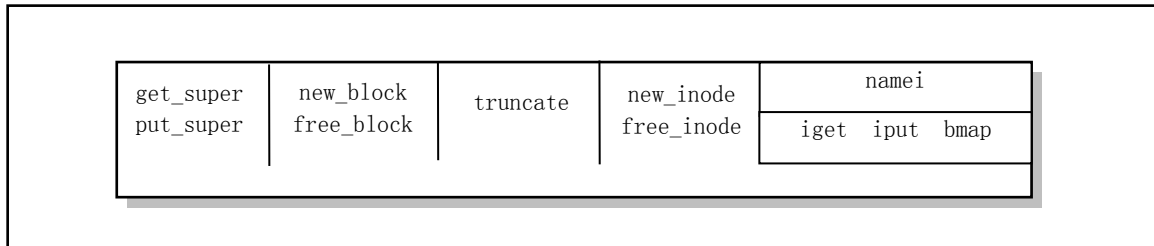


Figure 12-11 File system low-level function hierarchy

### 12.1.5 File Data Access Operations

The access operation code for the data in the file mainly involves five files: `block_dev.c`, `file_dev.c`, `char_dev.c`, `pipe.c`, and `read_write.c`. The read and write functions involved are shown in Figure 12-12. The first four files can be thought of as block devices, character devices, pipeline devices, and interface programs for ordinary files and file read/write system-calls. They collectively implement the `read()` and `write()` system-calls in `read_write.c`. By judging the attributes of the file being manipulated, the two system-calls will call the relevant processing functions in these files to operate.

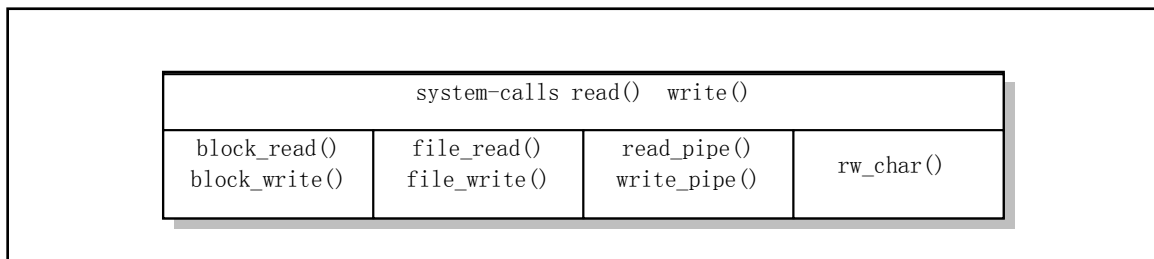


Figure 12-12 File data access functions

The functions `block_read()` and `block_write()` in the `block_dev.c` program are used to read and write data from a block device special file. The parameters used specify the device number to be accessed, the starting position and length of the read and write.

The `file_read()` and `file_write()` functions in the `file_dev.c` program are used to access general regular files. By giving the i-node and file structure of the file, we can know the device number where the file is located and the current read-write pointer of the file.

The pipe read and write functions `read_pipe()` and `write_pipe()` are implemented in the `pipe.c` file. It also

implements the system-call `pipe()` that creates an anonymous pipe. Pipes are mainly used to transfer data between processes in a first-in, first-out manner, or they can be used to synchronize processes. There are two types of pipes: named pipes and unnamed pipes. Named pipes are created using the `open` call of the file system, while unnamed pipes are created using the system-call `pipe()`. When using pipes, the regular file `read()`, `write()`, and `close()` functions are used. Only the descendants of the `pipe()` call can share access to the unnamed pipe, and all processes can access the named pipe as long as the permission is granted.

For pipe read and write, it can be seen that one process writes data to one end of the pipe, while another process reads data from the other end of the pipe. The way the kernel accesses the data in the pipeline is exactly the same as the way it accesses the data in a normal formal file. The difference between allocating storage for pipes and allocating space for regular files is that pipes use only direct blocks of i-nodes. The kernel manages the direct block of the i-node as a circular queue of the pipeline, and ensures the order of first-in first-out by modifying the read-write pointer.

For character device files, the system-calls `read()` and `write()` call the `rw_char()` function in `char_dev.c`. Character devices include console terminals (`tty`), serial terminals (`ttyx`), and memory character devices.

In addition, the kernel uses the file structure, the file table `file_table[]`, and the in-memory i-node table `inode_table[]` to manage operational access to the file. The definition of these data structures and tables can be found in the header file `include/linux/fs.h`. The file structure is defined as follows.

---

```
struct file {
    unsigned short f_mode;           // file operating mode (RW bits)
    unsigned short f_flags;          // file open and control flags.
    unsigned short f_count;          // file handler count.
    struct m_inode * f_inode;         // point to i-node in memory (the v-node).
    off_t f_pos;                     // current read/write position.
};
struct file file_table[NR_FILE]     // file table array, total 64 items.
```

---

It is used to establish a relationship between a file handle and an i-node entry in a memory i-node table. The file type and access attribute field `f_mode` have the same meaning as the `i_mode` field in the file i-node structure, as described above; The `f_flags` field is a combination of some open operation control flags given by the parameter flag in the open file function `open()` and control function `fcntl()`, which are defined in file `include/fcntl.h`. There are some of the following flags:

---

```
// file access mode used by open() and fcntl(). Only one of three can be used at the same time.
8 #define O_RDONLY      00           // open file in read only mode.
9 #define O_WRONLY      01           // open file in write only mode.
10 #define O_RDWR       02           // open file with read/write mode.
    // File creation flags for open(). Can be used with the above access mode in a 'bit or' mode.
11 #define O_CREAT       00100        // create if not exist (not used by fcntl).
12 #define O_EXCL        00200        // exclude using file flag.
13 #define O_NOCTTY      00400        // no control tty.
14 #define O_TRUNC       01000        // truncate to 0 if file exist and in write mode.
15 #define O_APPEND      02000        // open in append mode, point to eof.
16 #define O_NONBLOCK    04000        // open in nonblack mode.
17 #define O_NDELAY      O_NONBLOCK
```

---

The file reference count field `f_count` in the file structure indicates the number of times the file is

referenced by the file handle; The memory i-node structure field `f_inode` points to the memory i-node structure item in the corresponding i-node table of this file. The file table is an array of file structure items in the kernel. In the Linux 0.12 kernel, the file table can have up to 64 items, so the entire system can open up to 64 files at the same time. In the process data structure (that is, the process control block or process descriptor), the file structure pointer array `filp[NR_OPEN]` field of the open file is specifically defined. Where `NR_OPEN = 20`, so each process can open up to 20 files at the same time. The sequence number of the pointer array item corresponds to the file descriptor (file handle), and the pointer of the item points to the file item opened in the file table. For example, `filp[0]` is the file structure pointer corresponding to the currently open file descriptor 0 (handle 0) of the process.

The i-node table `inode_table[NR_INODE]` in the kernel is an array of memory i-node structures, where `NR_INODE = 32`, so at the moment the kernel can only store 32 memory i-node information at the same time. The relationship between the file opened by a process and the kernel file table and the corresponding memory i-node can be represented by Figure 12-13. One file in the figure is opened as standard input for the process (file handle 0) and the other is opened as standard output (file handle 1).

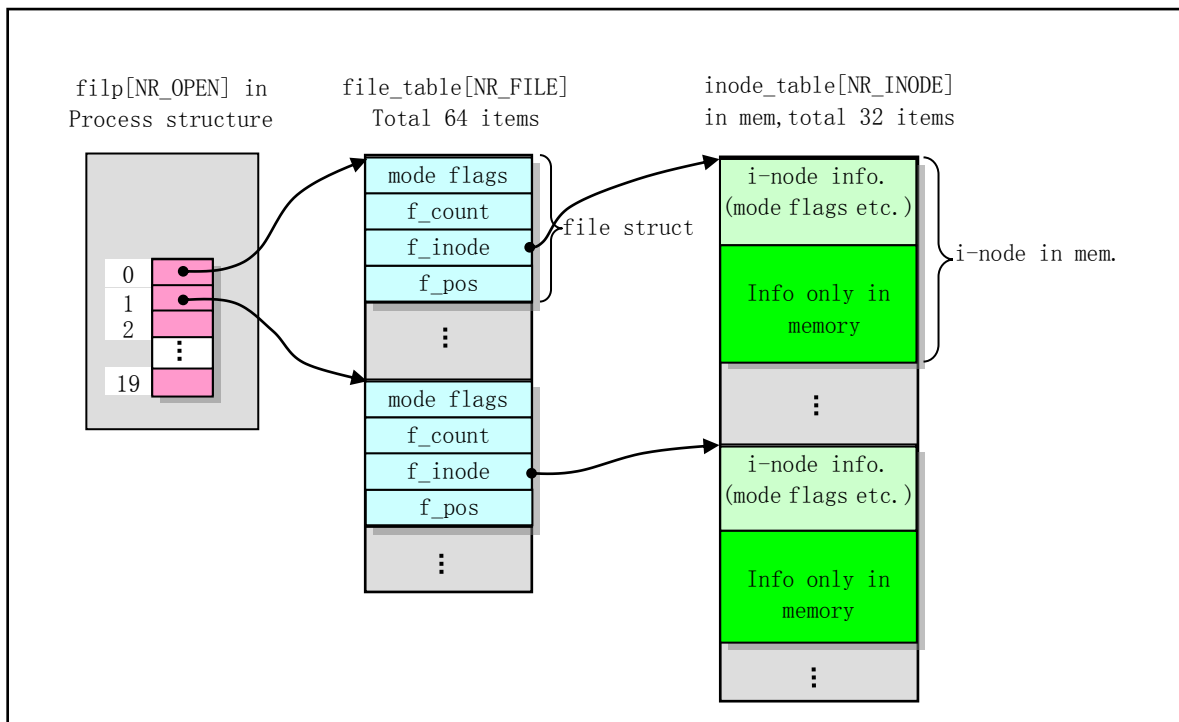


Figure 12-13 The structures used by the process to open the file

### 12.1.6 File and Directory Management System-Calls

User access to the file system are achieved through the system-calls provided by the kernel. The upper layer implementation of the system-calls for the file basically consists of the five files listed in Figure 12-14.

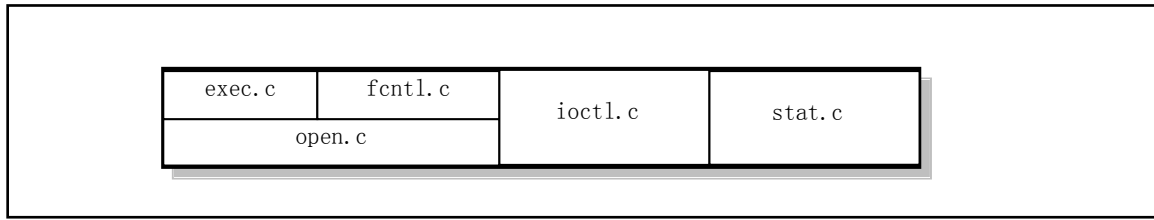


Figure 12-14 File system upper layer operating programs

The `open.c` file is used to implement system-calls related to file operations. There are mainly file creation, opening and closing, file host and attribute modification, file access permission modification, file time modification, and system file root change.

The `exec.c` program implements the loading and execution of binary executables and shell script files. The main function is `do_execve()`, which is the C handler for the system-call interrupt (int 0x80) function number `__NR_execve()`, and is the main implementation function of the `exec()` function cluster.

`Fcntl.c` implements the file control system-call `fcntl()`, and two file handle duplication system-calls `dup()` and `dup2()`. `Dup2()` needs to specify the value of the new handle, while `dup()` returns the unused handle with the smallest current value. The handle duplication operation is mainly used for standard input/output redirection and pipeline operations of files.

The `ioctl.c` file implements the input/output control system-call `ioctl()`, which mainly calls the `tty_ioctl()` function to control the I/O of the terminal.

The `stat.c` file is used to implement the file status information system-calls `stat()` and `fstat()`. `Stat()` uses the file name to get the information, and `fstat()` uses the file handle (descriptor) to get the information.

### 12.1.7 Analysis of a File System in 360KB Floppy Disk

In order to deepen the understanding of the file system structure shown in Figure 12-1, we used the Linux 0.12 system to create a MINIX 1.0 file system in the 360KB floppy image, which only stores a file named `hello.c`.

We first run the following command in the Linux 0.12 system in the bochs environment to create a file system.

```

[/usr/root]# mkfs /dev/fd1 360 // Create a 360KB file system in the 2nd fd.
120 inodes // 120 i-nodes and 360 disk (logic) blocks.
360 blocks
Firstdatazone=8 (8) // Starting block number of the data zone is 8.
Zonesize=1024 // Block size is 1024 bytes.
Maxsize=268966912 // Maximum file size (obviously incorrect).

[/usr/root]# mount /dev/fd1 /mnt // Mount to the /mnt directory and copy a file.
[/usr/root]# cp hello.c /mnt
[/usr/root]# ll -a /mnt // There are 3 directory entries.
total 3
drwxr-xr-x 2 root root 48 Feb 23 17:48 .
drwxr-xr-x 10 root root 176 Mar 21 2004 ..
-rw----- 1 root root 74 Feb 23 17:48 hello.c

[/usr/root]# umount /dev/fd1 // Unmount the file system.

```

---

```
[/usr/root]#
```

---

In the above series of operations, after executing the `mkfs` command on the floppy disk (image file) in the second floppy drive, a MINIX file system is created on the disk. It can be seen from the content displayed after the execution of the command that the file system contains a total of 120 i-nodes and 360 disk blocks. The first block number in the data zone of the disk is 8, and the size of the logical block is 1024 bytes, which is the same with disk block, and the file maximum size that can be stored is 268,966,912 bytes (obviously wrong).

Then we use the `mount` command to mount the device to the directory `/mnt`, and then copy the file `hello.c` to it, and then unmount the file system. Now we have created a MINIX file system with only one file, which is stored in the disk image file (`diskb.img`) corresponding to the second floppy drive of `bochs`.

Now let's look at the specific content of this file system. For convenience, here we use the `hexdump` command in the Linux 0.12 system to observe the contents. You can also exit the `bochs` system and view it using an editing program such as `notepad++` that can modify the binary. After executing the `hexdump` command on the device `/dev/fd1`, the following is displayed (slightly sorted out).

---

```
[/usr/root]# hexdump /dev/fd1 | more
0000000 44eb 4d90 6f74 6c6f 2073 0020 0102 0001 // 0x0000 - 0x03ff (1KB) is boot block.
0000010 e002 4000 f00b 0009 0012 0002 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
* // the data in this range is 0.
0000400 0078 0168 0001 0001 0008 0000 1c00 1008 // 0x0400 - 0x07ff (1KB) is super block.
0000410 137f 0000 0000 0000 0000 0000 0000 0000
0000420 0000 0000 0000 0000 0000 0000 0000 0000
*
0000800 0007 0000 0000 0000 0000 0000 0000 ff00 // 0x0800 - 0x0bff (1KB) is i-node bitmap.
0000810 ffff ffff ffff ffff ffff ffff ffff ffff
* // the data in this range is 1.
0000c00 0007 0000 0000 0000 0000 0000 0000 0000 // 0x0c00 - 0x0fff (1KB), logical blk bitmap.
0000c10 0000 0000 0000 0000 0000 0000 0000 0000
0000c20 0000 0000 0000 0000 0000 0000 fffe ffff
0000c30 ffff ffff ffff ffff ffff ffff ffff ffff
*
0001000 41ed 0000 0030 0000 c200 421c 0200 0008 // 0x1000 - 0x1fff (4KB) is 120 i-nodes.
0001010 0000 0000 0000 0000 0000 0000 0000 0000
0001020 8180 0000 004a 0000 c200 421c 0100 0009
0001030 0000 0000 0000 0000 0000 0000 0000 0000
*
0002000 0001 002e 0000 0000 0000 0000 0000 0000 // The following is data block contents.
0002010 0001 2e2e 0000 0000 0000 0000 0000 0000 // 0x2000 - 0x23ff (1KB), root i-node data
0002020 0002 6568 6c6c 2e6f 0063 0000 0000 0000
0002030 0000 0000 0000 0000 0000 0000 0000 0000
*
0002400 6923 636e 756c 6564 3c20 7473 6964 2e6f // 0x2400 - 0x27ff (1KB), the hello.c file.
0002410 3e68 0a0a 6e69 2074 616d 6e69 2928 7b0a
0002420 090a 7270 6e69 6674 2228 6548 6c6c 2c6f
0002430 7720 726f 646c 5c21 226e 3b29 090a 6572
0002440 7574 6e72 3020 0a3b 0a7d 0000 0000 0000
0002450 0000 0000 0000 0000 0000 0000 0000 0000
--More--
```

---

Now let's analyze the contents given above one by one. According to Figure 12-1, the first disk block of the MINIX 1.0 file system is a boot block, so disk block 0 (0x0000 - 0x03ff, 1KB) is the boot block content. Regardless of whether your disk is used to boot the system, each newly created file system will retain a boot disk block. For newly created disk image files, the boot disk block contents should all be zero. The content in the boot disk block in the above display data is the data left in the original image file, that is, the mkfs command does not modify the contents of the boot disk block when the file system is created.

Disk block 1 (0x0400 - 0x07ff, 1 KB) is the content of the super block. According to the super block data structure of the MINIX file system (see Figure 12-3), we can know the file system super block information listed in Table 12-1, which has a total of 18 bytes. Since the logarithm of the number of disk blocks per logical block to 2 is 0, for the MINIX file system, the disk block size is equal to the logical block (block) size.

Table 12-1 MINIX file system super block in 360KB disk

Field Name	Description	Content or Value
s_ninodes	Number of i-nodes	0x0078 (=120 in decimal)
s_nzones	Zone block (logical block) number	0x0168 (=360 in decimal)
s_imap_blocks	Number of blocks occupied by i-node bitmap	0x0001
s_zmap_blocks	Number of blocks in zone bitmap	0x0001
s_firstdatazone	First data block number	0x0008
s_log_zone_size	Log2(blocks/zone)	0x0000
s_max_size	Maximum file size	0x10081c00 = 268966912 Bytes
s_magic	FS magic number	0x137f

Disk block 2 (0x0800 - 0x0bff, 1 KB) contains i-node bitmap information. Since there are a total of 120 i-nodes in the file system, and each bit represents an i-node structure, the file system actually occupies  $120/8 = 15$  bytes in the 1 KB-sized disk block. The bit value of 0 indicates that the corresponding i-node structure is not occupied, and 1 indicates that it is occupied or reserved. The remaining unused byte bits in the disk block are initialized to 1 by the mkfs command.

From the data of the disk block 2, we can see that the first byte value is 0x07 (0b0000111), that is, the first three bits of the i-node bitmap have been occupied. As can be seen from the foregoing description, the first bit (bit 0) is reserved. The 2nd and 3rd bits respectively indicate that the file system's No. 1 and No. 2 i-nodes have been used, that is, the following i-node areas already contain two i-node structures. In fact, node 1 is used as the root i-node of the file system, and node 2 is used for the only file hello.c on the file system, the contents of which are described later.

Disk block 3 (0x0c00 - 0x0fff, 1 KB) is the logical block bitmap content. Since the disk capacity is only 360KB, the file system actually uses 360 bits, which is  $360/8 = 45$  bytes. The number of blocks for special purposes is: 1 boot block + 1 super block + 1 i-node bitmap block + 1 logical block bitmap block + 4 i-node data blocks = 8. Considering that the logic block bitmap only indicates that the disk block is occupied in the data area on the disk, after removing the number of function blocks that have been used, the actual number of bits required for the logic block bitmap is  $360 - 8 = 352$  bits (occupying 44 bytes), plus the remaining unused bit 0, the bitmap requires a total of 353 bits. This is why the last (45th) byte (0xfe) has only 1 bit being 0.

Therefore, when we know the bit offset value  $nr$  of a logic block in the logic block bitmap, then the corresponding actual disk block number  $block$  is equal to  $nr + 8 - 1$ , ie  $block = nr + s\_firstdatazone - 1$ . When we want to find the bit offset value (ie, the block number in the data area)  $nr$  in the logical block bitmap for a disk block number  $block$ , it is  $nr = block - s\_firstdatazone + 1$ .



Similar to the i-node bitmap, the first 3 bits of the first byte of the logic block bitmap are already occupied. The first (bit 0) bit is reserved, and the 2nd and 3rd bits indicate that 2 disk blocks (logical blocks) have been used in the disk data zone. In fact, the first disk block in the disk data zone represented by bit 1 is used for the root i-node to store data information (directory entries), and the second disk block represented by bit 2 is used for the number 2 i-node to save relevant data information. Please note that the data information mentioned here refers to the data content managed by the i-node, and is not the information of the i-node structure. The structure information of the i-node itself will be stored in the disk block in the i-node area dedicated to the i-node structure, that is, the disk disk block 4--7.

Disk block 4--7 (0x1000 - 0x1fff, 4KB) 4 disk blocks are used to store i-node structure information. Since the file system is provided with 120 i-nodes, and each i-node occupies 32 bytes (see Figure 12-4), a total of  $120 \times 32 = 3840$  bytes is required, that is, 4 disk blocks are required. From the data shown above, we can see that the first 32 bytes hold the contents of root i-node, and the subsequent 32 bytes store the contents of node 2, as shown in Table 12-2 and Table 12-3.

Table 12-2 No. 1 root i-node structure content

Field Name	Description	Value
i_mode	File's mode	0x41ed (drwxr-xr-x)
i_uid	File's owner id	0x0000
i_size	File size	0x00000030 (48 bytes)
i_mtime	Modified time	0x421cc200 (Feb 23 17:48)
i_gid	File group id	0x00
i_nlinks	Number of links	0x02
i_zone[9]	Array of logical block numbers	zone[0] = 0x0008, remaining items are all 0.

Table 12-3 No. 2 i-node structure content

Field Name	Description	Value
i_mode	File's mode	0x8180 (-rw-----)
i_uid	File's owner id	0x0000
i_size	File size	0x0000004a (74 bytes)
i_mtime	Modified time	0x421cc200 (Feb 23 17:48)
i_gid	File group id	0x00
i_nlinks	Number of links	0x01
i_zone[9]	Array of logical block numbers	zone[0] = 0x0009, remaining items are all 0.

It can be seen that the data block of root i node has only one block, and its logical block number is 8, located on the first block in the disk data zone, and the length is 30 bytes. From the previous section, a directory entry length is 16 (0x10) bytes, so there are 3 directory entries (0x30 bytes) coexisting in this logical block. Because it is a directory, its number of links is 2.

The data block of the No. 2 i-node is also only one block, and is located in the second block of the disk data zone, and the disk block number is 9. The length of the data stored therein is 74 bytes, which is the byte length of the hello.c file.

The disk block 8 (0x2000 - 0x23ff, 1 KB) is the data of the root i-node, and there are three directory item structure information (48 bytes), as shown in Table 12-4.

Table 12-4 Data content of root i-node

No.	I-node No.	File Name
1	0x0001	0x2e (.)
2	0x0001	0x2e,0x2e (..)
3	0x0002	0x68,0x65,0x6c,0x6c,0x6f,0x2e,0x63 (hello.c)

Disk block 9 (0x2400 - 0x27ff, 1 KB) is the content of the hello.c file. It contains 74 bytes of text information.

## 12.2 buffer.c

From this section, we will explain and annotate the programs in the fs/ directory one by one. According to the description in Section 1 of this chapter, the programs in this chapter can be divided into four parts: (1) cache management; (2) file underlying operations; (3) file data access; and (4) file high-level access control. Here we first describe the high-speed buffer (or buffer cache) management procedure of part 1. This section contains only one program buffer.c.

### 12.2.1 Function

Below we first explain the specific location of the cache in physical memory, and then describe the process of its initialization. Then we give the structure of the buffer cache, the linked list structure and the hash table structure used. Then, the important buffer block acquisition function `glblk()` and the block read function `bread()` are described in detail. Finally, the buffer cache access process and the synchronous operation method are explained from a system perspective.

#### 1. Buffer cache physical location

The buffer.c program is used to operate and manage the buffer cache (pool). The buffer cache is located between the kernel code block and the main memory area, as shown in Figure 12-15. The buffer cache acts as a bridge between the block device and other programs in the kernel. In addition to the block device driver, if the kernel program needs to access the data in the block device, it needs to go through the cache to operate indirectly.

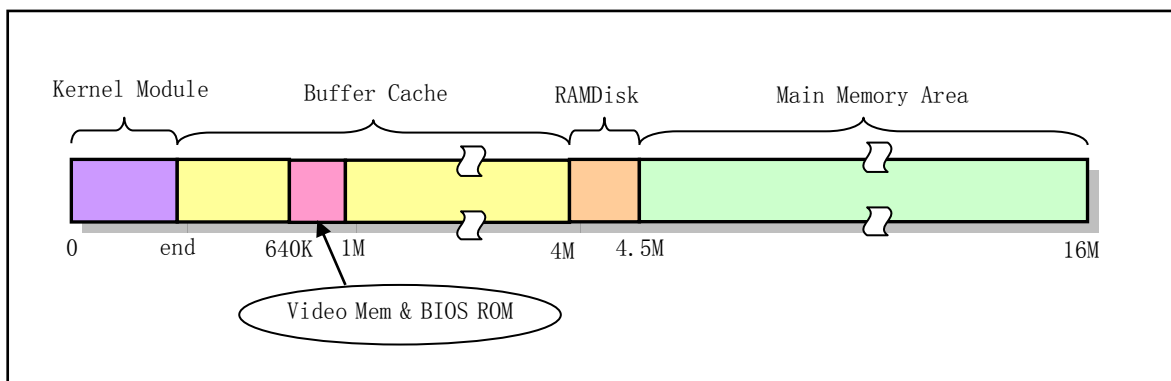


Figure 12-15 The location of the buffer cache in physical memory

The starting position of the cache in the figure begins with the end of the kernel module 'end', and 'end' is an external variable set by the linker ld during the kernel module link. This symbol is not defined in the kernel code. When the linker generates the system module, the ld program sets the address of 'end', which is equal to 'data\_start + datasize + bss\_size', which is the first effective address after the end of the bss segment, which is the end of the kernel module. In addition, the linker also sets two external variables, 'etext' and 'edata', which represent the first address after the code segment and the first address after the data segment.

## 2. Buffer cache initialization

The entire cache is divided into 1024-byte buffer blocks, which is exactly the same size as the disk logical block on the block device. The buffer cache is managed by a hash table and a linked list containing all the buffer blocks. During the buffer initialization process, the initialization program starts from both ends of the entire buffer, and sets the buffer block header structure and the corresponding buffer block respectively, as shown in Figure 12-16. The high end of the buffer is divided into buffer blocks of 1024 bytes, and the buffer header structure `buffer_head` (include/linux/fs.h, line 68) corresponding to each buffer block is respectively established at the low end. This header structure is used to describe the attributes of the corresponding buffer block and is used to join all buffer headers into a linked list. This setting operation continues until there is not enough memory in the buffer to divide the buffer block.

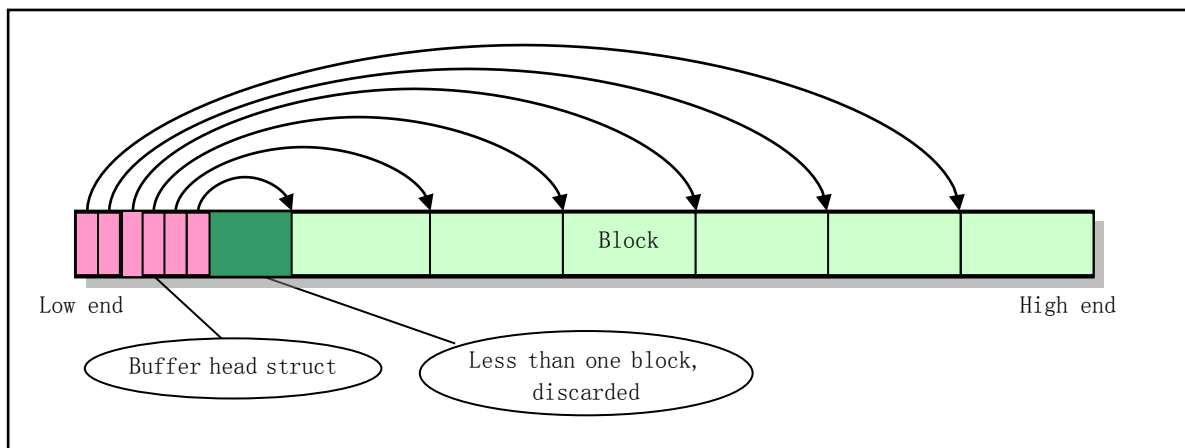


Figure 12-16 Initialization of the buffer cache

## 3. Buffer cache structure and linked list

The `buffer_head` of all buffer blocks is linked into a doubly linked list structure, as shown in Figure 12-17. The `free_list` pointer in the figure is the head pointer of the linked list, pointing to the first "most idle" buffer block in the free block list, which is the least recently used buffer block. The back pointer `b_prev_free` of the buffer block points to the last buffer block in the buffer block list, that is, the buffer block that has just been used recently. The buffer block header data structure is:

```
struct buffer_head {
    char * b_data;                // Points to buffer block data area (1024 bytes)
    unsigned long b_blocknr;      // Block No.
    unsigned short b_dev;        // Device no. of the data source (0 = free).
    unsigned char b_uptodate;    // Indicates if the data has been updated.
    unsigned char b_dirt;        // Modified flag: 0 -clean, 1 -modified (dirty).
    unsigned char b_count;       // The nr of users who use the block.
    unsigned char b_lock;        // Whether the block is locked. 0-ok, 1-lock
    struct task_struct * b_wait; // Point to the task waiting for the buffer.
```

```
struct buffer\_head * b_prev;           // The previous block on the hash queue.  
struct buffer\_head * b_next;           // The next block on the hash queue.  
struct buffer\_head * b_prev_free;       // The previous block on the free list.  
struct buffer\_head * b_next_free;       // The next block on the free list.  
};
```

---

The field `b_lock` is a lock flag indicating that the driver is modifying the contents of the buffer block, so the buffer block is busy and is being locked. This flag is independent of other flags of the buffer block and is mainly used in the `blk_drv/ll_rw_block.c` program to lock the buffer block when updating the data information in the buffer block. Because when updating the data in the buffer block, the current process will voluntarily go to sleep waiting, so that other processes have the opportunity to access the buffer block. Therefore, in order to prevent other processes from using the data, it is necessary to lock the buffer block before going to sleep.

The field `b_count` is the count used by the buffer manager, indicating the number of times the corresponding buffer block is being used (referenced) by each process, so this field is used for the program reference count management of the buffer block, and is also independent of other flags of the buffer block. When the reference count is not 0, the buffer manager cannot release the corresponding buffer block. A free block is a block with `b_count = 0`. When `b_count = 0`, it means that the corresponding buffer block is not used (free), otherwise it is being used. For the block of the program application, if the buffer manager can obtain the existing specified block from the hash table, the `b_count` of the block is incremented by 1 (`b_count++`). If the buffer block is an unused block that is re-applied, the `b_count` in its header structure is set equal to one. When a program releases its reference to a block, the number of references to that block is decremented accordingly (`b_count--`). Since the flag `b_lock` indicates that other programs are using and locking the specified buffer block, the `b_count` must be greater than 0 for the buffer block where `b_lock` is set.

The field `b_dirt` is a dirty flag indicating whether the content in the buffer block has been modified to be different from the corresponding data block on the block device (delayed write). The field `b_uptodate` is a data update (valid) flag indicating whether the data in the buffer block is valid. Both flags are set to 0 when the block is initialized or released, indicating that the buffer block is invalid at this time. `B_dirt = 1, b_uptodate = 0` when data is written to the buffer but has not yet been written to the device. The data becomes valid when the data is written to the block device or just after reading the buffer block from the block device, ie `b_uptodate = 1`. Please note that there is a special case where both `b_dirt` and `b_uptodate` are 1 when a new device buffer block is applied, indicating that the data in the buffer block is different from the block device, but the data is still valid (updated).

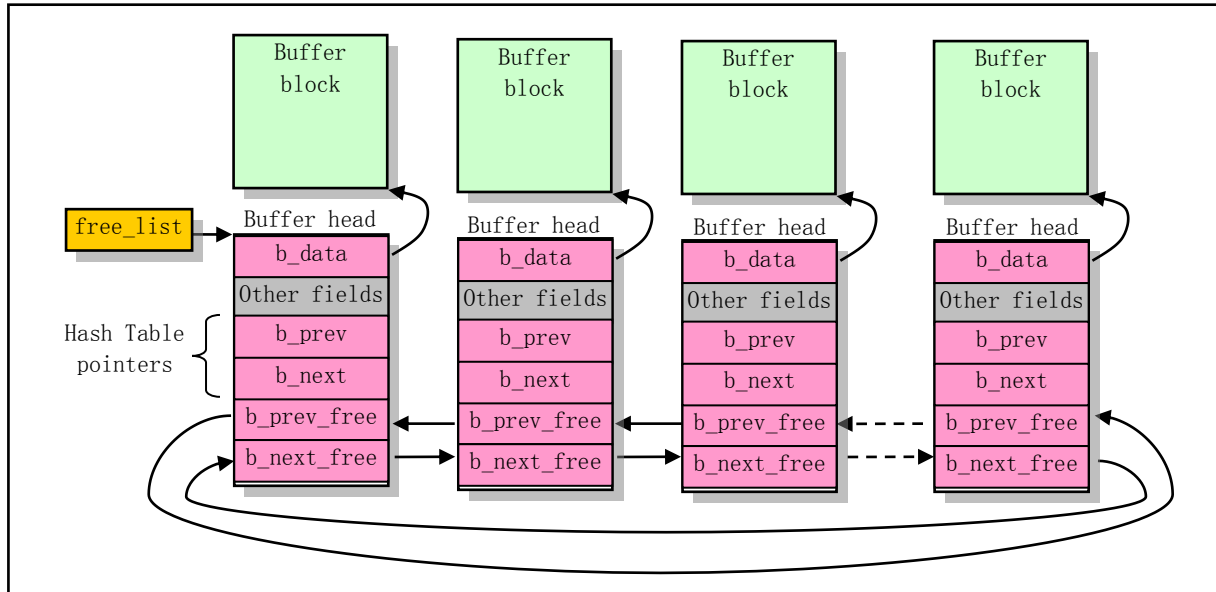


Figure 12-17 Bidirectional circular linked free list of all buffer blocks

The "other fields" in the buffer header structure in the figure include the block device number and the logical block number of the buffered data, and these two fields uniquely determine the block device and its data corresponding to the data in the buffer block. There are also several status flags: the data valid (update) flag, the modified flag, the number of processes the data is used, and whether the buffer block is locked.

When the kernel code uses the buffer block in the buffer cache, it specifies the device number (dev) and the logical block number of the data, and operates by calling the buffer block read function `bread()`, `bread_page()`, or `breada()`. These functions use the buffer search management function `getblk()` to find matching or free blocks in all buffer blocks. This function will be highlighted below. When the system releases the buffer block, you need to call the `brelse()` function. The call hierarchy of all these buffer block data access and management functions can be described in Figures 12-18.

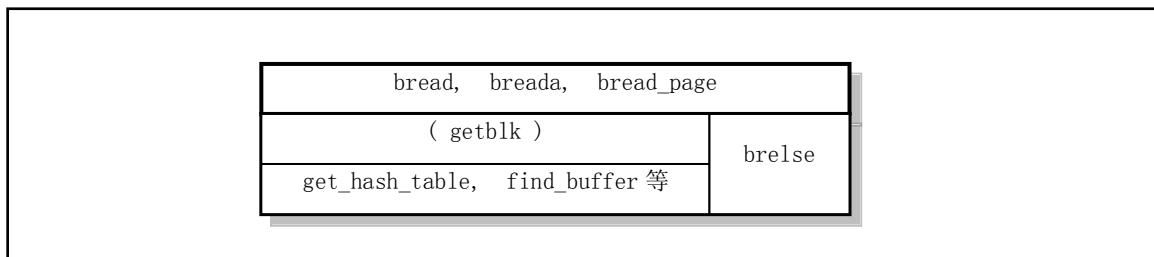


Figure 12-18 Hierarchical relationship between buffer management functions

#### 4. Hash queue of the buffer cache

In order to quickly and efficiently find out in the buffer cache whether the requested data block has been read into the buffer, the `buffer.c` program uses a hash array table structure with 307 `buffer_head` pointer entries. The hash function used by the hash table is a combination of the device number and the logical block number. The specific hash function used in the program is:  $(\text{device number} \wedge \text{logical block number}) \bmod 307$ . In Figure 12-17, the pointers `b_prev` and `b_next` are bidirectional links used in hash tables for hashing between multiple buffer blocks on the same item, that is, the buffer blocks with the same hash value calculated by the hash

function are linked in the same array item on the linked list. For details on how the buffer block operates on the hash queue, see the description in Chapter 3 of the Unix Operating System Design. For the state of a dynamically changing hash table structure at a certain time, see Figure 12-19.

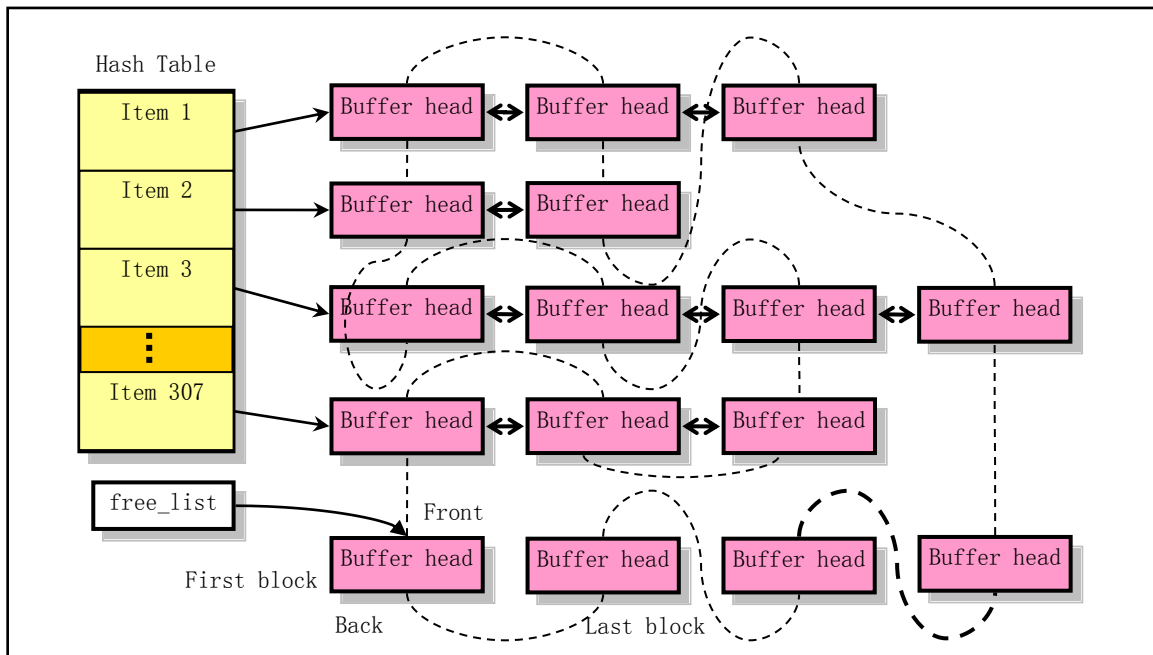


Figure 12-19 Schematic diagram of buffer block hash queue at a certain moment

Wherein, the double arrow horizontal line indicates a bidirectional link pointer that is hashed between the buffer block header structures in the same hash entry. The dotted line represents a bidirectional circular linked list of all the buffer blocks in the buffer cache (the so-called free linked list). The free\_list is the head pointer at the most free buffer block of the list. In fact, this doubly linked list is a least recently used (LRU) linked list. Below we will explain the buffer block search function getblk() in detail.

#### 5. Buffer block get function

The three functions mentioned above all call getblk() at execution time to get the appropriate free buffer block. The function first invokes the get\_hash\_table() function to search the hash table queue for the buffer block of the specified device number and logical block number. If the specified block exists, it immediately returns a pointer to the corresponding block header structure; If it doesn't exist, we start from the free-list header and scan the free-list to find a free buffer. In the search process, we also compare the found free buffer blocks, and according to the weights given by the combination of the modified flag and lock flag, it is determined which free block is most suitable. If the found free block is neither modified nor locked, then we don't have to keep looking. If no free block is found at this time, the current process is put to sleep, and it is searched again when it continues to execute. If the free block is locked, the process also needs to go to sleep, waiting for other (block driver) to unlock. If the buffer block is occupied by other processes during sleep waiting, then just start searching for the buffer block again. Otherwise, it is judged whether the buffer block has been modified, and if so, the block is written to the disk and waiting for the block to be unlocked. At this point, if the buffer block is once again occupied by another process, then once again, it is completely abandoned, so we have to start executing getblk() again.

After experiencing the above toss, there may be another unexpected situation at this time, that is, while we are sleeping, other processes may have added the buffer block we need to the hash queue, so we need to search

for the last time the hash queue here. If we really find the buffer block we need in the hash queue, then we have to make the above judgment on the buffer block we found, so again, we need to start executing `getblk()` again.

Finally, we found a free buffer block that was not used by the process, was not locked, and was clean (the modified flag was not set). So we set the number of references to the block, reset several other flags, and then remove the buffer header structure of the block from the free-list queue. After the device number and corresponding logical number to which the buffer block belongs are set, it is inserted into the header of the corresponding table entry of the hash table and linked to the end of the free-list queue. Since the search for free blocks starts from the free-list header, this operation of first removing from the free-list queue and using the most recently used buffer block and then reinserting into the tail of the free-list achieves the least recently used LRU algorithm. Finally, the pointer to the buffer block header is returned. The entire `getblk()` process can be seen in Figure 12-20.

From the above analysis, it can be seen that each time the function acquires a new free buffer block, it will move it to the end of the linked list pointed to by the `free_list` header pointer. That is, the closer the buffer block is to the end of the linked list, the closer time it is used. Therefore, if the corresponding buffer block is not found in the hash table, the search will start from the `free_list` header when searching for a new free buffer block. It can be seen that the algorithm for the kernel to obtain the buffer block uses the following strategy:

- If the specified buffer block exists in the hash table, it indicates that the available buffer block has been obtained, so it returns directly;
- Otherwise, we need to start the search from the `free_list` header in the linked list, that is starting from the least recently used buffer block.

Therefore, the most ideal situation is to find a buffer block that is completely free, that is, a buffer block with `b_dirt` and `b_lock` flags of 0; But if these two conditions are not met, then we need to calculate a value based on the `b_dirt` and `b_lock` flags. Because device operations are often time consuming, we need to increase the weight of `b_dirt` in your calculations. Then we wait on the buffer block with the smallest result value (if the buffer block is already locked). Finally, when the flag `b_lock` is 0, it indicates that the original content of the buffer block that has been waiting has been written to the block device. So `getblk()` gets a free buffer block.

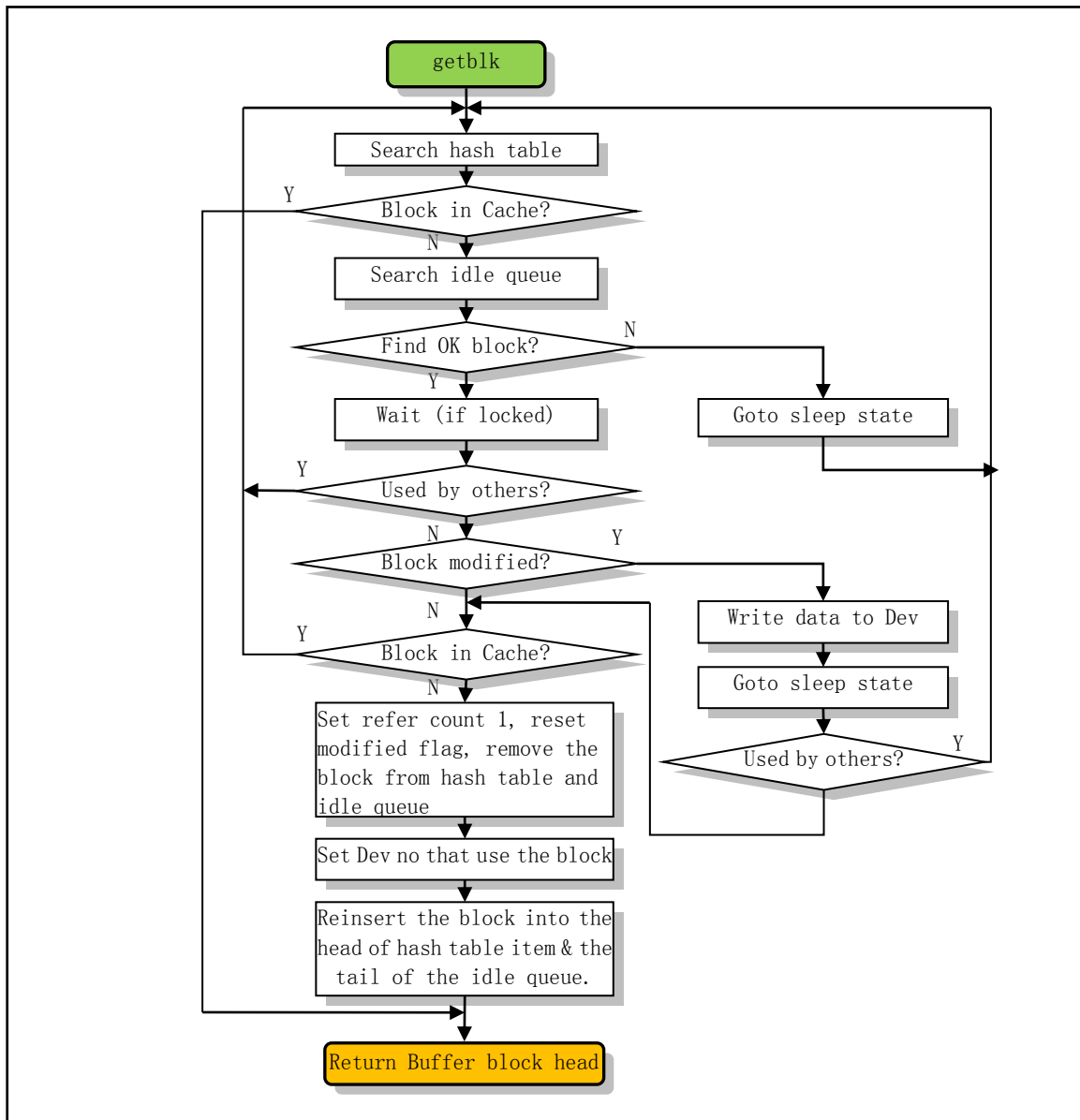


Figure 12-20 Getblk() function execution flow chart

## 6. Buffer block read function

From the above processing we can see that the buffer block returned by `getblk()` may be a new free block, or it may just be a buffer block containing the data we need, and it already exists in the buffer cache. Therefore, for the read data block operation (`bread()`), it is necessary to judge the update flag of the buffer block to see if the included data is valid. If it is valid, the data block can be directly returned to the applied program. Otherwise, you need to call the device's low-level block read-write function (`ll_rw_block()`), and at the same time let yourself go to sleep, waiting for the data to be read into the buffer block, and after waking up, judge whether the data is valid. If it is valid, this data can be returned to the application program, otherwise the read operation to the device fails and no data is fetched. Thus, the buffer block is released and a NULL value is returned. Figure 12-21 is a block diagram of the `bread()` function. The `breada()` and `bread_page()` functions are similar to the `bread()` function.



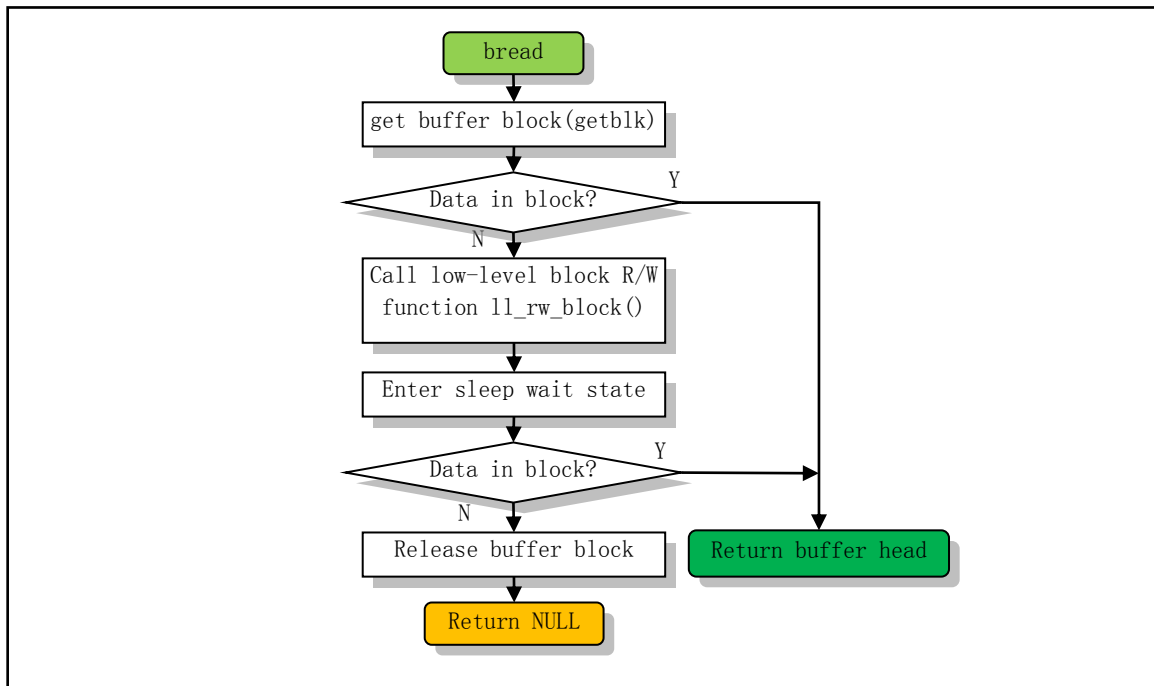


Figure 12-21 Bread() function execution flow diagram

When the program no longer needs to use data from a buffer block, the `brelse()` function can be invoked to free the buffer block and wake up the process that is asleep due to waiting for the buffer block. Note that the buffer blocks in the free list are not all free. Therefore, it can be used only when the disk is refreshed, unlocked, and no other process references (reference count = 0).

#### 7. buffer access method and synchronization operation

In summary, the high-speed buffer plays an important role in improving access efficiency and increasing data sharing for block devices. In addition to the driver, the read and write operations of other upper-level programs on the block device by the kernel need to be implemented indirectly through the cache manager. The main link between them is through the `bread()` function in the cache manager and the low-level interface function `ll_rw_block()` of the block device. If the upper program wants to access the block device data, it applies to the buffer manager through `bread()`. If the required data is already in the buffer cache, the buffer manager will return the data directly to the program. If the required data is not yet in the buffer, the buffer manager will apply to the block device driver through `ll_rw_block()`, and let the process corresponding to the program sleep and wait. After the block device driver puts the specified data into the cache, the buffer manager returns the data to the upper program, as shown in Figure 12-22.

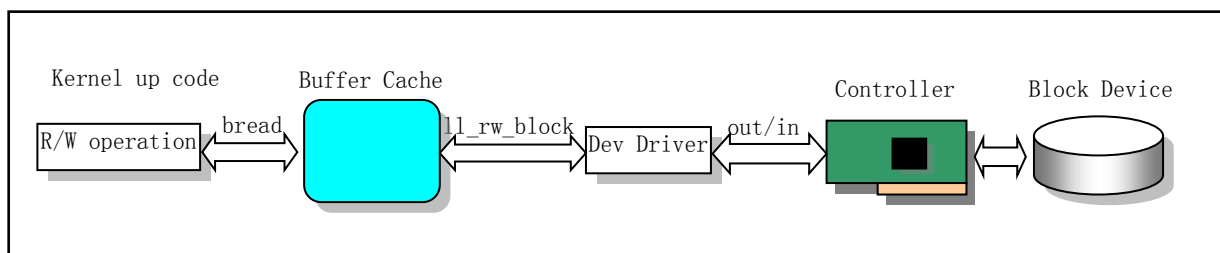


Figure 12-22 Kernel block device access operation

For update and synchronization operations, its main function is to make some buffer block contents in memory consistent with information on block devices such as disks. The main function of `sync_inodes()` is to match the i-node information in the `inode_table` with that on the disk, but it needs to go through the intermediate link of the system buffer cache. In fact, any synchronization operation is divided into two phases:

- (1) The data structure information is synchronized with the buffer block in the cache buffer and is independently responsible for the driver;
- (2) The synchronization problem between the data block and the corresponding block of the disk in the buffer cache is handled by the buffer management program here.

The `sync_inodes()` function does not deal directly with the disk. It can only advance to the buffer, which is only responsible for synchronizing with the information in the buffer. The rest of the operation requires the buffer manager to be responsible for execution. In order for `sync_inodes()` to know which i-nodes are different from those on the disk, you must first make the contents of the buffer consistent with the contents on the disk. This way `sync_inodes()` knows which disk inodes need to be modified and updated by comparing with the latest data in the buffer of the current disk. Finally, the second synchronization operation of the buffer cache and the disk device is performed, so that the data in the memory is truly synchronized with the data in the block device.

## 12.2.2 Code annotation

Program 12-1 linux/fs/buffer.c

---

```

1  /*
2  *  linux/fs/buffer.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  'buffer.c' implements the buffer-cache functions. Race-conditions have
9  *  been avoided by NEVER letting a interrupt change a buffer (except for the
10 *  data, of course), but instead letting the caller do it. NOTE! As interrupts
11 *  can wake up a caller, some cli-sti sequences are needed to check for
12 *  sleep-on-calls. These should be extremely quick, though (I hope).
13 */
14
15 /*
16 * NOTE! There is one discordant note here: checking floppies for
17 * disk change. This is where it fits best, I think, as it should
18 * invalidate changed floppy-disk-caches.
19 */
20
21 // <stdarg.h> Standard parameter header file. Define a list of variable parameters in the
22 //   form of macros. It mainly describes one type (va_list) and three macros (va_start,
23 //   va_arg and va_end) for the vsprintf, vprintf, and vfprintf functions.
24 // <linux/config.h> Kernel configuration header file. Define keyboard language and hard
25 //   disk type (HD_TYPE) options.
26 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
27 //   data of the initial task 0, and some embedded assembly function macro statements
28 //   about the descriptor parameter settings and acquisition.
29 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
30 //   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/io.h> Io header file. Defines the function that operates on the io port in the
//      form of a macro's embedded assembler.
21 #include <stdarg.h>
22
23 #include <linux/config.h>
24 #include <linux/sched.h>
25 #include <linux/kernel.h>
26 #include <asm/system.h>
27 #include <asm/io.h>
28
// The value of the variable 'end' below is generated by the linker ld at compile time to indicate
// the end position of the kernel module, as shown in Figure 12-15. We can find this value from
// the System.map file generated when the kernel was compiled. It is used here to indicate that
// the buffer cache starts at the end of the kernel code.
// The buffer_wait variable on line 33 is the queue head pointer for the task that is asleep
// while waiting for the free buffer block. It has a different effect than the b_wait pointer
// in the buffer block header structure. When a task requests a buffer block and happens to
// encounter a lack of available free buffer blocks, the task is added to the buffer_wait sleep
// wait queue. The b_wait is a wait queue header that is dedicated to the task waiting for the
// specified buffer block (ie, the buffer block corresponding to b_wait).
29 extern int end;
30 struct buffer_head * start_buffer = (struct buffer_head *) &end;
31 struct buffer_head * hash_table[NR_HASH];          // NR_HASH = 307
32 static struct buffer_head * free_list;             // free block link list head pointer.
33 static struct task_struct * buffer_wait = NULL;    // queue for waiting a free block.

// The following defines the number of buffer blocks contained in the system buffer cache. Here,
// NR_BUFFERS is a constant symbol defined on line 48 in the linux/fs.h file, which is defined
// as the variable nr_buffers, which is declared as a global variable on line 172 in the fs.h
// file. Mr. Linus wrote the code in such a way as to implicitly indicate that nr_buffers is
// a "constant" that does not change after the kernel buffer is initialized. It will be set
// in the initialization function buffer_init() (line 371).
34 int NR_BUFFERS = 0;                               // The number of buffer blocks.
35
//// Wait for the specified buffer block to unlock.
// If the specified buffer block bh has been locked, then we let the process sleep uninterrupted
// in the buffer queue b_wait. When the buffer block is unlocked, all processes on its waiting
// queue will be woken up. Although it goes to sleep after the interrupt is disabled (cli),
// doing so does not affect the response to interrupts in other process contexts. Because each
// process saves the value of the register EFLAGS in its own TSS segment, the value of the current
// EFLAGS in the CPU changes as the process switches. A process that uses sleep_on() to go to
// sleep needs to wake up explicitly with wake_up().
36 static inline void wait_on_buffer(struct buffer_head * bh)
37 {
// If it has been locked, the process goes to sleep and waits for it to unlock.
38     cli();                                           // disable int.
39     while (bh->b_lock)
40         sleep_on(&bh->b_wait);
41     sti();                                           // enable int.
42 }
43

```

```

//// Device data synchronization
// Synchronize data in device and cache. sync_inodes() is defined in file inode.c, line 59.
// The function first calls the i-node synchronization function to write all modified i-nodes
// in the memory i-node table to the buffer cache. Then, the entire cache buffer is scanned,
// and a write disk request is generated for the buffer block that has been modified, and the
// buffered data is written into the disk, so that the data in the cache is synchronized with
// the device.
44 int sys_sync(void)
45 {
46     int i;
47     struct buffer_head * bh;
48
49     sync_inodes();                /* write out inodes into buffers */
50     bh = start_buffer;            // points to the beginning of cache.
51     for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
52         wait_on_buffer(bh);        // Wait for the buffer to be unlocked (if locked).
53         if (bh->b_dirt)
54             ll_rw_block(WRITE,bh); // Generate a write device block request.
55     }
56     return 0;
57 }
58
//// Synchronize the cached data with data on the specified device.
// This function first searches all buffer blocks in the buffer cache and writes to the disk
// for the buffer of the specified device dev if its data has been modified (synchronous
// operation). The in-memory i-node table data is then written to the buffer cache. Then perform
// the same write operation as above for the specified device dev again.
59 int sync_dev(int dev)
60 {
61     int i;
62     struct buffer_head * bh;
63
64     // First, the data synchronization operation is performed on the device specified by the
65     // parameter, so that the data on the device is synchronized with the data in the buffer cache.
66     // The method is to scan all buffer blocks in the cache, and to check whether the buffer block
67     // of the specified device dev is locked, and if it is locked, sleep and wait for it to be unlocked.
68     // Then it is judged whether the buffer block is still the buffer block of the designated device,
69     // and it has been modified (the b_dirt flag is set), and if so, the write operation is performed
70     // on it. Because the buffer block may have been released or used for other purposes during
71     // our sleep, it is necessary to check again if the buffer block is still a buffer block for
72     // the specified device before proceeding.
73     bh = start_buffer;            // points to the beginning of cache.
74     for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
75         if (bh->b_dev != dev)      // ignore blocks that doesn't belong to dev.
76             continue;
77         wait_on_buffer(bh);        // wait for the buffer to be unlocked.
78         if (bh->b_dev == dev && bh->b_dirt)
79             ll_rw_block(WRITE,bh);
80     }
81
82     // Then write the i-node data into the buffer cache, and synchronize the inode in the inode_table
83     // with the information in the buffer cache. The data in the cache is then synchronized again
84     // with the data in the device after it has been updated. The two-pass sync operation is here
85     // to improve the efficiency of kernel execution. The first pass of the buffer synchronization

```

```

// can make many "dirty blocks" in the memory clean, so that the synchronization operation of
// the i-node can be performed efficiently. This second buffer synchronization synchronizes
// the buffer blocks that are dirty due to the i-node synchronization operation with the data
// in the device.
72     sync_inodes();
73     bh = start_buffer;
74     for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
75         if (bh->b_dev != dev)
76             continue;
77         wait_on_buffer(bh);
78         if (bh->b_dev == dev && bh->b_dirt)
79             ll_rw_block(WRITE, bh);
80     }
81     return 0;
82 }
83
///// Invalidate the data of the specified device in the buffer cache.
// Scan all buffer blocks in the cache. For the buffer block of the specified device, reset
// its valid (update) flag and the modified flag.
84 void inline invalidate_buffers(int dev)
85 {
86     int i;
87     struct buffer_head * bh;
88
89     bh = start_buffer;
90     for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
91         if (bh->b_dev != dev)           // Ignore blocks that doesn't belong to dev.
92             continue;
93         wait_on_buffer(bh);           // wait for the buffer to be unlocked.
94         // Since the process has performed a sleep wait, it is necessary to determine whether the buffer
95         // still belongs to the specified device.
96         if (bh->b_dev == dev)
97             bh->b_uptodate = bh->b_dirt = 0;
98     }
99 }
100 /*
101  * This routine checks whether a floppy has been changed, and
102  * invalidates all buffer-cache-entries in that case. This
103  * is a relatively slow routine, so we have to try to minimize using
104  * it. Thus it is called only upon a 'mount' or 'open'. This
105  * is the best way of combining speed and utility, I think.
106  * People changing diskettes in the middle of an operation deserve
107  * to loose :-))
108  *
109  * NOTE! Although currently this is only for floppies, the idea is
110  * that any additional removable block-device will use this routine,
111  * and that mount/open needn't know that floppies/whatever are
112  * special.
113  */
114
115 // Check if the disk has changed and invalidate the buffer block if it has been replaced.
116 void check_disk_change(int dev)
117 {

```

```

115         int i;
116
117         // First check if it is a floppy device, because the code now only supports floppy removable
118         // media. If not, quit. Then test if the floppy disk has been replaced, and if not, exit. The
119         // function floppy_change() is on line 139 of blk_drv/floppy.c.
120         if (MAJOR(dev) != 2)
121             return;
122         if (!floppy_change(dev & 0x03))
123             return;
124
125         // The floppy disk has been replaced, so the buffer blocks occupied by the i-node bitmap and
126         // the logical block bitmap of the corresponding device is released; and the i-node of the device
127         // and the cache block occupied by the data block information are invalidated.
128         for (i=0 ; i<NR_SUPER ; i++)
129             if (super_block[i].s_dev == dev)
130                 put_super(super_block[i].s_dev);
131         invalidate_inodes(dev);
132         invalidate_buffers(dev);
133 }
134
135 // Below is the hash function definition and the calculation macro for the hash table entry.
136 // The main role of the hash table is to reduce the time it takes to find elements. By establishing
137 // a correspondence between the storage location of the element and the keyword (hash function),
138 // we can immediately find the specified element directly through the function calculation.
139 // The guiding principle for building a hash function is to try to ensure that the probability
140 // of hashing to any array item is substantially equal. There are many ways to create a hash
141 // function. Here Linux 0.12 mainly uses the most commonly used keyword residual remainder
142 // method. Because the buffer block we are looking for has two conditions, the device number
143 // 'dev' and the buffer block number 'block', the hash function designed must contain these
144 // two key values. The XOR operation of the two keywords here is just one way to calculate the
145 // key value. Performing a modulo operation (%) on the key values ensures that the values
146 // calculated by the function are all within the range of the function array items.
147 #define hashfn(dev, block) (((unsigned)(dev^block))%NR_HASH)
148 #define hash(dev, block) hash_table[hashfn(dev, block)]
149
150 // Remove the buffer block from the hash queue and the free list.
151 // The hash queue is a doubly linked list structure, and the buffer block free list is a
152 // bidirectional circular linked list structure.
153 static inline void remove_from_queues(struct buffer_head * bh)
154 {
155     /* remove from hash-queue */
156     if (bh->b_next)
157         bh->b_next->b_prev = bh->b_prev;
158     if (bh->b_prev)
159         bh->b_prev->b_next = bh->b_next;
160
161     // If the buffer block is the first block of the queue, then the corresponding entry of the
162     // hash table points to the next buffer block in the queue.
163     if (hash(bh->b_dev, bh->b_blocknr) == bh)
164         hash(bh->b_dev, bh->b_blocknr) = bh->b_next;
165
166     /* remove from free list */
167     if (!(bh->b_prev_free) || !(bh->b_next_free))
168         panic("Free block list corrupted");
169     bh->b_prev_free->b_next_free = bh->b_next_free;
170     bh->b_next_free->b_prev_free = bh->b_prev_free;

```

```

// If the free list header points to this buffer block, it is directed to the next buffer block.
145     if (free\_list == bh)
146         free\_list = bh->b_next_free;
147 }
148
149 // Insert the buffer block at the end of the free list and put it into the hash queue.
149 static inline void insert\_into\_queues(struct buffer\_head * bh)
150 {
151     /* put at end of free list */
152     bh->b_next_free = free\_list;
153     bh->b_prev_free = free\_list->b_prev_free;
154     free\_list->b_prev_free->b_next_free = bh;
155     free\_list->b_prev_free = bh;
156     /* put the buffer in new hash-queue if it has a device */
157     // Note that when an item of the hash table is inserted for the first time, the hash() value
158     // is definitely NULL, so bh->b_next obtained on line 161 is definitely NULL. So on line 163,
159     // we should only assign b_prev a bh value if bh->b_next is not NULL. That is, before the 163th
160     // line, a sentence "if (bh->b_next)" should be added. The error was corrected after the kernel
161     // 0.96 version.
162     bh->b_prev = NULL;
163     bh->b_next = NULL;
164     if (!bh->b_dev)
165         return;
166     bh->b_next = hash(bh->b_dev, bh->b_blocknr);
167     hash(bh->b_dev, bh->b_blocknr) = bh;
168     bh->b_next->b_prev = bh;        // "if (bh->b_next)" should be added before this.
169 }
170
171 // Use hash table to look up the buffer block for a given device and a specified block number
172 // in the buffer cache. Returns the buffer block pointer if found, otherwise returns NULL.
173 static struct buffer\_head * find\_buffer(int dev, int block)
174 {
175     struct buffer\_head * tmp;
176
177     // Search the hash table for buffer block with the specified device number and block number.
178     for (tmp = hash(dev, block) ; tmp != NULL ; tmp = tmp->b_next)
179         if (tmp->b_dev==dev && tmp->b_blocknr==block)
180             return tmp;
181     return NULL;
182 }
183
184 /*
185  * Why like this, I hear you say... The reason is race-conditions.
186  * As we don't lock buffers (unless we are reading them, that is),
187  * something might happen to it while we sleep (ie a read-error
188  * will force it bad). This shouldn't really happen currently, but
189  * the code is ready.
190 */
191
192 // Use the hash table to find the buffer block.
193 // Use the hash table to find the specified buffer block in the buffer cache. If found, the
194 // block is locked and the block header pointer is returned.
195 struct buffer\_head * get\_hash\_table(int dev, int block)
196 {

```

```

185     struct buffer head * bh;
186
187     for (;;) {
188         // Look for the buffer block for the given device and the specified block in the buffer cache.
189         // If it is not found, return NULL and exit.
190         if (!(bh=find buffer(dev, block)))
191             return NULL;
192         // If the desired block is found, its reference count is incremented by one, and then waiting
193         // for the block to be unlocked (if it has been locked). Since it has gone through a sleep state,
194         // it is necessary to verify the correctness of the block and return the buffer head pointer.
195         // If the device or block number of the buffer block changes during sleep, its reference count
196         // is revoked and the search is made again.
197         bh->b_count++;
198         wait on buffer(bh);      // Wait for the buffer to be unlocked (if locked).
199         if (bh->b_dev == dev && bh->b_blocknr == block)
200             return bh;
201         bh->b_count--;
202     }
203 }
204
205 /*
206  * Ok, this is getblk, and it isn't very clear, again to hinder
207  * race-conditions. Most of the code is seldom used, (ie repeating),
208  * so it should be much more efficient than it looks.
209  *
210  * The algorithm is changed: hopefully better, and an elusive bug removed.
211  */
212 // The following macro is used to check the modified flag and lock flag of the block at the
213 // same time, and the weight of the modified flag is defined to be larger than the lock flag.
214 #define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
215
216 //// Get the specified block in the buffer cache.
217 // Check if the buffer block of the specified (device no. & block no.) is already in the cache.
218 // If the specified block is already in the buffer cache, the corresponding buffer block header
219 // is returned and exited; if not, a new entry corresponding to the device no. and block no.
220 // needs to be set in the cache and the corresponding buffer header pointer is returned.
221 struct buffer head * getblk(int dev, int block)
222 {
223     struct buffer head * tmp, * bh;
224
225     repeat:
226         // Search the hash table, if the specified block is already in the buffer cache, return the
227         // corresponding block head pointer and exit.
228         if (bh = get hash table(dev, block))
229             return bh;
230         // Otherwise, the free list is scanned to find the free block. First let 'tmp' point to the
231         // first free block header of the free list, and then loop to perform subsequent operations.
232         tmp = free list;
233         do {
234             // If the buffer block is being used (the reference count is not equal to 0), continue scanning
235             // the next item. For a block with b_count=0, that is, a block that is not currently referenced
236             // in the cache, it is not necessarily clean (b_dirt=0) or unlocked (b_lock=0). Therefore, we
237             // still need to continue the following judgments and choices. For example, for a block with

```



```

// b_count=0, when a task rewrites the content of a block and then releases it, the block b_count
// is still 0, but b_lock is not equal to 0; When a task executes breada() to pre-read several
// blocks, it will decrement b_count as soon as ll_rw_block() is called. However, at this time,
// the hard disk access operation may still be in progress, so b_lock=1, but b_count=0.
215         if (tmp->b_count)
216             continue;
// If the buffer block header pointer bh is empty, or the weight of the flag (modification,
// lock) of the block referred to by the current tmp is less than the weight of the bh's flag,
// then bh is pointed to the tmp's block header. If the buffer block header pointed by the tmp
// indicates that the buffer block is neither modified nor the lock flag is set (ie, BADNESS()
// = 0), it means that the corresponding cache block has been obtained on the specified device,
// and the loop is exited. Otherwise we will continue to execute this loop and see if we can
// find a buffer with the smallest BADNESS() value.
217         if (!bh || BADNESS(tmp)<BADNESS(bh)) {
218             bh = tmp;
219             if (!BADNESS(tmp))
220                 break;
221         }
222 /* and repeat until we find something good */
223         while ((tmp = tmp->b_next_free) != free_list);

// If the loop check finds that all buffer blocks are being used (all buffer block header reference
// counts are >0), sleep waits for free buffer blocks to be available. The process is explicitly
// woken up when a free buffer block is available, and then we jump to the beginning of the
// function to re-look for the free buffer block.
224         if (!bh) {
225             sleep_on(&buffer_wait);
226             goto repeat; // jump to line 210.
227         }
// Execution here, we have found a suitable free buffer block. Then wait for the buffer to unlock
// (if it has been locked). If the buffer is used by other tasks during our sleep phase, we
// have to repeat the above search process again.
228         wait_on_buffer(bh);
229         if (bh->b_count) // occupied again ??
230             goto repeat;
// If the buffer has been modified, write the data to the disk and wait for the buffer to unlock
// again. Similarly, if the buffer is used by other tasks again, the above search process is
// repeated.
231         while (bh->b_dirt) {
232             sync_dev(bh->b_dev);
233             wait_on_buffer(bh);
234             if (bh->b_count) // occupied again ??
235                 goto repeat;
236         }
237 /* NOTE!! While we slept waiting for this block, somebody else might */
238 /* already have added "this" block to the cache. check it */
// Check if the specified buffer block has been added to the hash table while we are sleeping.
// If so, repeat the above search process again.
239         if (find_buffer(dev, block))
240             goto repeat;
241 /* OK, FINALLY we know that this buffer is the only one of it's kind, */
242 /* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
// So let us occupy this buffer block, set the reference count to 1, reset the modified flag

```

```

// and the valid (update) flag.
243     bh->b_count=1;
244     bh->b_dirt=0;
245     bh->b_uptodate=0;
// The buffer block header is first removed from the hash queue and the free block list, and
// the buffer block is used for the specified device and the specified block thereon. The buffer
// block is then reinserted into the free list and the new location of the hash queue, and finally
// the buffer block header is returned.
246     remove\_from\_queues(bh);
247     bh->b_dev=dev;
248     bh->b_blocknr=block;
249     insert\_into\_queues(bh);
250     return bh;
251 }
252
//// Release the specified buffer block.
// Wait for the buffer block to unlock. The reference count is then decremented by one, and
// the process waiting for the free buffer block is explicitly awake.
253 void brelse(struct buffer head * buf)
254 {
255     if (!buf)                // Returns if the block head pointer is invalid.
256         return;
257     wait\_on\_buffer(buf);
258     if (!(buf->b_count--))
259         panic("Trying to free free buffer");
260     wake\_up(&buffer\_wait);
261 }
262
263 /*
264  * bread\(\) reads a specified block and returns the buffer that contains
265  * it. It returns NULL if the block was unreadable.
266  */
//// Read data blocks from the device.
// This function first requests a buffer block in the cache based on the specified device number
// dev and block number block. If the buffer block already contains valid data, it directly
// returns the buffer block pointer; otherwise, the specified data block is read from the device
// into the buffer block and the buffer block pointer is returned.
267 struct buffer head * bread(int dev, int block)
268 {
269     struct buffer head * bh;
270
// Apply a block of buffers in the cache. If the return value is NULL, it means that the kernel
// has an error and it stops. Then we determine if there is data available. Returns if the data
// in the buffer block is valid (updated) and can be used directly.
271     if (!(bh=getblk(dev, block)))
272         panic("bread: getblk returned NULL\n");
273     if (bh->b_uptodate)
274         return bh;
// Otherwise we call the read and write ll_rw_block() function of the underlying block device
// to generate a read device block request. Then we wait for the specified data block to be
// read in and wait for the buffer to unlock. After the sleep wakes up, if the buffer has been
// updated, the buffer head pointer is returned and exits. Otherwise, it indicates that the
// read device operation failed, so the buffer is released, NULL is returned, and exit.

```

```

275     ll_rw_block(READ, bh);
276     wait_on_buffer(bh);
277     if (bh->b_uptodate)
278         return bh;
279     brelse(bh);
280     return NULL;
281 }
282
283     /// Copy the memory block.
284     // Copy a block (1024 bytes) of data from the 'from' address to the 'to' position.
285 #define COPYBLK(from,to) \
286     __asm__( "cld\n\t" \
287             "rep\n\t" \
288             "movsl\n\t" \
289             :: "c" (BLOCK_SIZE/4), "S" (from), "D" (to) \
290             : "cx", "di", "si")
291
292 /*
293  * bread_page reads four buffers into memory at the desired address. It's
294  * a function of its own, as there is some speed to be got by reading them
295  * all at the same time, not waiting for one to be read, and then another
296  * etc.
297 */
298
299     /// Read the one page (4 buffer blocks) on the device to the specified memory address.
300     // The parameter 'address' is the address where the page data is saved; 'dev' is the specified
301     // device number; b[4] is an array containing 4 device data block numbers. This function is
302     // used in the do_no_page() function of the mm/memory.c file (line 428).
303 void bread_page(unsigned long address, int dev, int b[4])
304 {
305     struct buffer_head * bh[4];
306     int i;
307
308     // This function is executed 4 times. According to the 4 block numbers placed in the array b[],
309     // a page is read from the device dev and placed at the specified memory location 'address'.
310     // For the valid block number given by the parameter b[i], the function first gets the buffer
311     // block of the specified device and block number from the buffer cache. If the data in the
312     // buffer block is invalid (not updated) then a read device request is made to read the
313     // corresponding data block from the device. For the invalid block number b[i], we don't have
314     // to deal with it. Therefore, this function can freely read 1-4 data blocks according to the
315     // block number in the specified b[].
316     for (i=0 ; i<4 ; i++)
317     {
318         if (b[i]) { // If the block number is valid.
319             if (bh[i] = getblk(dev, b[i]))
320                 if (!bh[i]->b_uptodate)
321                     ll_rw_block(READ, bh[i]);
322             } else
323                 bh[i] = NULL;
324     }
325     // The contents of the 4 buffer blocks are then sequentially copied to the specified address.
326     // Before we can copy (use) the buffer block, we need to sleep and wait for the buffer block
327     // to be unlocked (if it is locked). In addition, because we may have slept, we also need to
328     // check if the data in the buffer block is valid before copying. We also need to release the
329     // buffer block after copying.
330     for (i=0 ; i<4 ; i++, address += BLOCK_SIZE)

```

```

309         if (bh[i]) {
310             wait\_on\_buffer(bh[i]);    // Wait for block to unlock (if it is locked).
311             if (bh[i]->b_uptodate)    // If the data is valid, copy it.
312                 COPYBLK((unsigned long) bh[i]->b_data, address);
313             brelse(bh[i]);          // Release the buffer block.
314         }
315     }
316
317     /*
318     * Ok, breada can be used as bread, but additionally to mark other
319     * blocks for reading as well. End the argument list with a negative
320     * number.
321     */
322     /// Reads some of the specified blocks from the specified device.
323     // The number of parameters is variable and is a series of specified block numbers. When the
324     // function succeeds, it returns the block header of the first block, otherwise it returns NULL.
325     struct buffer head * breada(int dev, int first, ...)
326     {
327         va\_list args;
328         struct buffer head * bh, *tmp;
329
330         // First take the first parameter (block number) in the variable parameter list. Then get the
331         // block of the specified device and block number from the cache. If the buffer block data is
332         // invalid (the update flag is not set), a read device data block request is issued.
333         va\_start(args, first);
334         if (!(bh=getblk(dev, first)))
335             panic("bread: getblk returned NULL\n");
336         if (!bh->b_uptodate)
337             ll\_rw\_block(READ, bh);
338
339         // Then take the other pre-read block numbers in the variable parameter list in order and do
340         // the same as above, but do not use them. Note that there is a bug on line 336. Where bh should
341         // be tmp. This bug was not corrected until the 0.96 kernel code. In addition, because this
342         // is a pre-read subsequent data block, it only needs to be read into the cache but not used
343         // immediately, so the 337th line statement needs to decrement its reference count to release
344         // the block (because the getblk() function will increase the block reference count).
345         while ((first=va\_arg(args, int))>=0) {
346             tmp=getblk(dev, first);
347             if (tmp) {
348                 if (!tmp->b_uptodate)
349                     ll\_rw\_block(READA, bh);    // here 'bh' should be 'tmp'.
350                 tmp->b_count--;    // release the pre-read block.
351             }
352         }
353
354         // At this point, all parameters in the variable parameter table are processed. Then wait for
355         // the first buffer block to be unlocked (if it has been locked). After the wait is over, if
356         // the data in the buffer block is still valid, the buffer block header pointer is returned.
357         // Otherwise the buffer is released and returns NULL.
358         va\_end(args);
359         wait\_on\_buffer(bh);
360         if (bh->b_uptodate)
361             return bh;
362         brelse(bh);
363         return (NULL);

```

```

346 }
347
348 // Buffer initialization function.
349 // The parameter buffer_end is the end of the buffer cache memory. For systems with 16MB of
350 // memory, the buffer cache end is set to 4MB. For systems with 8MB of memory, the buffer end
351 // is set to 2MB. The function sets (initializes) the block header structure and the corresponding
352 // data block from the start position of the buffer cache at the start_buffer and the buffer_end
353 // at the end of the buffer respectively until all the memory in the buffer cache is allocated.
354 void buffer_init(long buffer_end)
355 {
356     struct buffer_head * h = start_buffer;
357     void * b;
358     int i;
359
360     // First, determine the high-end position 'b' of the actual buffer based on the high-end position
361     // of the buffer provided by the parameter. If the high end of the buffer is equal to 1Mb, since
362     // the video memory and BIOS are used from 640KB - 1MB, the actual available buffer memory should
363     // be 640KB at the high end. Otherwise, the high end of the buffer cache memory must be greater
364     // than 1MB.
365     if (buffer_end == 1<<20)
366         b = (void *) (640*1024);
367     else
368         b = (void *) buffer_end;
369
370     // The following code is used to initialize the buffer cache, create a free block circular list,
371     // and get the number of blocks in the system. The operation process is to divide the buffer
372     // block of 1 KB size from the high end of the buffer, and at the same time, construct the
373     // buffer_head describing the buffer block at the low end of the buffer, and form the buffer_head
374     // into a doubly linked list.
375     // 'h' is a pointer to the block header structure, and 'h+1' is the next block header address
376     // that points to the memory address consecutively, or can be said to point to the end of the
377     // h block header. In order to ensure that there is enough memory to store a block header
378     // structure, the address of the memory block pointed to by 'b' needs to be greater than or
379     // equal to the end of the block header of 'h', that is, 'b >= h+1' is required.
380     while ( (b -= BLOCK_SIZE) >= ((void *) (h+1)) ) {
381         h->b_dev = 0; // device no.
382         h->b_dirt = 0; // dirt flag (block modified flag).
383         h->b_count = 0; // block reference count.
384         h->b_lock = 0; // block lock flag.
385         h->b_uptodate = 0; // Block update flag (or data valid flag).
386         h->b_wait = NULL; // block waiting queue.
387         h->b_next = NULL; // points to next header with same hash value.
388         h->b_prev = NULL; // points to previous header with same hash value.
389         h->b_data = (char *) b; // block data pointer.
390         h->b_prev_free = h-1; // point to the previous item in the free-list.
391         h->b_next_free = h+1; // point to the next item in the free-list.
392         h++; // h points to next new buffer header location.
393         NR_BUFFERS++; // counting buffer blocks.
394     }
395     // If b is decremented to 1MB, skip 384KB (video memory) and let b point to 0xA0000 (640KB).
396     if (b == (void *) 0x100000)
397         b = (void *) 0xA0000;
398 }
399
400 // Then let h point to the last valid buffer block header; let the free list header point to
401 // the first buffer block header; b_prev_free of the free list header points to the previous

```

```
// header (ie the last header); The next pointer of h points to the first header, so that the
// free list forms a bidirectional ring structure. Finally initialize the hash table, set all
// pointers in the table to NULL.
375     h--;                                // h points to the last block header.
376     free_list = start_buffer;           // free list header points to first block header.
377     free_list->b_prev_free = h;
378     h->b_next_free = free_list;
379     for (i=0; i<NR_HASH; i++)
380         hash_table[i]=NULL;
381 }
382
```

---

## 12.3 bitmap.c

Starting with this program, we begin to explore the second part of the file system composition, the underlying operational function part of the file system. This part consists of five files, super.c, bitmap.c, truncate.c, inode.c, and namei.c.

The super.c program mainly contains functions for accessing and managing file system superblocks; The bitmap.c program is used to process the logical block bitmap and the i-node bitmap of the file system; The truncate.c program only contains a function truncate() that cuts the file data length to zero; The inode.c program mainly involves the access and management of file system i-node information; The namei.c program is mainly used to complete the function of finding and loading its corresponding i-node information from a given file path name.

According to the order of the functional parts of a file system, we should describe them in the order of the programs given above, but because the super.c program also contains several high-level functions related to file system loading/unloading or system-calls, they need to use functions in several other programs, so we will explain them after introducing the inode.c program.

### 12.3.1 Function

The purpose and function of the bitmap.c program is simple and clear. It is mainly used to occupy/release the bits in the logical block bitmap or the i-node bitmap according to the usage of the logical block or the i-node structure in the file system. The operation functions of the logic block bitmap are free\_block() and new\_block(); the operation functions of the i-node bitmap are free\_inode() and new\_inode().

The function free\_block() is used to release the logical block in the data area on the specified device. The specific operation is to reset the bit in the logical block bitmap corresponding to the specified logical block. It first takes the super block of the specified device, and judges the validity of the logical block number according to the range of the device data logic block given in the super block. Then look in the buffer cache to see if the specified logical block is now in the buffer cache. If yes, the corresponding buffer block is released. Next, calculate the logical block number of the data from the beginning of the data zone (counting from 1), and operate on the logical block bitmap to reset the corresponding bit. Finally, in the buffer block containing the corresponding logical block bitmap, we set the modified bit flag according to the logical block number.

The function new\_block() is used to request a logical block from the block device, return the logical block number, and set the logical block bitmap bit corresponding to the block. It first gets the superblock of the specified device dev. Then, the entire logical block bitmap is searched for the first bit that is 0. If not found, the

disk device space is exhausted and the function returns 0. Otherwise, the first 0 bit position to be found is set to 1, indicating that the corresponding data logic block is occupied. At the same time, the code sets the modified flag of the buffer block in which the logical block bitmap containing the bit is located. Then calculate the disk block number of the data logical block, and apply the corresponding buffer block in the buffer cache, and clear the buffer block. Then set the updated and modified flags for this buffer block. Finally, the buffer block is released for use by other programs and returns the block number (logical block number).

The function `free_inode()` is used to release the specified i-node and reset the corresponding i-node bitmap bit; `New_inode()` is used to create a new i-node for device and return a pointer to the new i-node. The main operation process is to obtain an idle i-node entry in the memory i-node table and find an idle i-node from the i-node bitmap. The processing of these two functions is similar to the above two functions, so it will not be described here.

### 12.3.2 Code annotation

Program 12-2 linux/fs/bitmap.c

---

```

1  /*
2   *  linux/fs/bitmap.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /* bitmap.c contains the code that handles the inode and block bitmaps */
8  // <string.h> String header file. Defines some embedded functions about string operations.
9  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
10 //     data of the initial task 0, and some embedded assembly function macro statements
11 //     about the descriptor parameter settings and acquisition.
12 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
13 //     commonly used functions of the kernel.
14 #include <string.h>                // The memset() function is used here.
15
16 #include <linux/sched.h>
17 #include <linux/kernel.h>
18
19 // Clear a block of 1024 bytes of memory at the given address (addr).
20 // Input: eax = 0; ecx = length of the block in long words (BLOCK_SIZE/4); edi = specifies the
21 // starting address addr.
22 #define clear_block(addr) \
23     __asm__ ("cld\n\t" \
24             "rep\n\t" \
25             "stosl" \
26             :: "a" (0), "c" (BLOCK_SIZE/4), "D" ((long) (addr)): "cx", "di")
27
28 // Set the bit at the nr-th bit offset starting at the given address.
29 // The 'nr' can be greater than 32! This macro returns the original bit value.
30 // Input: %0 -eax (return value); %1 -eax(0); %2 -nr, bit offset; %3 -(addr), content of addr.
31 // Line 20 defines a local register variable 'res'. This variable will be saved in the specified
32 // EAX register for efficient access and operation. This method of defining variables is mainly
33 // used in inline assembly programs. The entire macro is a statement expression whose value
34 // is the value of the last 'res'.
35 // The BTL instruction on line 21 is used to test and set the bit. It saves the bit value specified
36 // by the base address (%3) and the bit offset (%2) to the carry flag CF, and then sets the

```

---

```

// bit to 1. The SETB instruction is used to set the operand (%al) according to the carry flag
// CF. If CF=1 then %al =1, otherwise %al =0.
19 #define set\_bit(nr,addr) ({\
20 register int res __asm__ ("ax"); \
21 __asm__ __volatile__ ("btsl %2,%3|n|tsetb %%al": \
22 "=a" (res): "" (0), "r" (nr), "m" (*(addr))); \
23 res;})
24
//// Resets the bit at the nr bit offset from the beginning of the specified address.
// Returns the inverse of the original bit value.
// Input: %0 -eax(return value); %1 -eax(0); %2 -nr, bit offset; %3 -(addr), content of addr.
// The BTRL instruction on line 27 is used to test and reset the bit. The command SETNB is used
// to set the operand (%al) according to the carry flag CF. If CF = 1, then %al = 0, otherwise %al
// = 1.
25 #define clear\_bit(nr,addr) ({\
26 register int res __asm__ ("ax"); \
27 __asm__ __volatile__ ("btrl %2,%3|n|tsetnb %%al": \
28 "=a" (res): "" (0), "r" (nr), "m" (*(addr))); \
29 res;})
30
//// Look for the first zero bit from address 'addr', return the bit offset from 'addr'.
// Input: %0 - ecx (return value); %1 - ecx(0); %2 - esi(addr).
// Look for the first bit that is 0 in the bitmap starting with the address specified by 'addr',
// and return the bit offset from the address addr. 'addr' is the address of the data area of
// the buffer block, and the range of the scan is 1024 bytes (8192 bits).
// The BSFL instruction on line 36 is used to scan the first non-zero bit in the EAX register
// and place its offset into EDX.
31 #define find\_first\_zero(addr) ({ \
32 int __res; \
33 __asm__ ("cld|n" \                                // clear direction flag.
34         "1:|tlodsl|n|t" \                          // obtain [esi] -> eax
35         "notl %%eax|n|t" \                          // inverse each bits in eax
36         "bsfl %%eax,%%edx|n|t" \                    // offset of first non-zero bit -> edx
37         "je 2f|n|t" \                               // jump forward to label 2 if eax = 0.
38         "addl %%edx,%%ecx|n|t" \                     // offset is added to ecx (first 0 bit in bitmap).
39         "jmp 3f|n" \                                // jump forward to label 3 (end).
40         "2:|taddl $32,%%ecx|n|t" \                   // ecx adds 32 bit offset if bit 0 not found.
41         "cmpl $8192,%%ecx|n|t" \                     // scanned 8192 bits (1024 bytes)?
42         "jl 1b|n" \                                 // jump backward to label 1 if not yet.
43         "3:" \                                       // end. At this time, ecx contains 0 bit offset.
44         : "=c" (__res): "c" (0), "S" (addr): "ax", "dx", "si"); \
45 __res;})
46
//// Free the logic block in the data zone on the device.
// Resets the bit in the logical block bitmap corresponding to the specified logical block.
// Returns 1 if successful, otherwise returns 0.
47 int free\_block(int dev, int block)
48 {
49     struct super\_block * sb;
50     struct buffer\_head * bh;
51
// First, the super block information of the file system on the device dev is taken, and the
// validity of the parameter block is checked according to the data block start logical block

```



```

// number and the total number of logical blocks in the file system. If the specified device
// super block does not exist, an error occurs. If the logical block number is smaller than
// the block number of the first logical block in the data zone on the disk or greater than
// the total number of logical blocks on the device, system stops too.
52     if (!(sb = get\_super(dev))) // in fs/super.c, line 56.
53         panic("trying to free block on nonexistent device");
54     if (block < sb->s_firstdatazone || block >= sb->s_nzones)
55         panic("trying to free block not in datazone");
56     bh = get\_hash\_table(dev, block);

// Then look for the block data from the hash table. If a buffer block is found, its validity
// is checked. At this time, if the number of references is greater than 1, it indicates that
// someone else is using the buffer block, so brelse() is called, and b_count is decremented
// by 1 and then exits. Otherwise, clear the modified and updated flags to release the data
// block. The main purpose of this piece of code is to detect that if the logic block currently
// exists in the buffer cache, the corresponding buffer block is released.
57     if (bh) {
58         if (bh->b_count > 1) {
59             brelse(bh); // ret after b_count--, still used by others.
60             return 0;
61         }
62         bh->b_dirt=0;
63         bh->b_uptodate=0;
64         if (bh->b_count) // if b_count=1, call brelse().
65             brelse(bh);
66     }

// Then we reset the bit of the block in the logic block bitmap (set to 0). First calculate
// the logical block number of the block starting from the data zone (counting from 1). The
// logic block bitmap is then operated to reset the corresponding bit. If the bit is already
// 0, it means system is faulty and stops. Since a block has 1024 bytes, which is 8192 bits,
// so 'block / 8192' can calculate which block in the logical bitmap is the specified block,
// and 'block & 8191' can get the bit offset position of the block in the current block of
// the logic block bitmap.
67     block -= sb->s_firstdatazone - 1; // block = block - (s_firstdatazone - 1);
68     if (clear\_bit(block&8191, sb->s_zmap[block/8192]->b_data)) {
69         printk("block (%04x:%d) ", dev, block+sb->s_firstdatazone-1);
70         printk("free_block: bit already cleared\n");
71     }

// Finally, set the modified flag of the buffer block where the logic block bitmap is located.
72     sb->s_zmap[block/8192]->b_dirt = 1;
73     return 1;
74 }
75

//// Request a logical disk block from the device.
// The function first takes the device's superblock and looks for the first zero value bit
// (representing an free block) in the logical block bitmap. This bit is then set to indicate
// that the corresponding logical block is expected to be obtained. Then, a corresponding buffer
// block is obtained in the buffer for the logic block. Finally, the buffer block is cleared,
// its updated flag and modified flag are set, and the logical block number is returned. If
// the execution is successful, the function returns the logical block number (disk block
// number), otherwise it returns 0.
76 int new\_block(int dev)
77 {

```

```

78     struct buffer head * bh;
79     struct super block * sb;
80     int i, j;
81
    // First get the super block of the device. Then scan the file system's 8 logical block bitmaps,
    // look for the first 0 value bit, find the free logic block, and get the block number where
    // the logical block is placed. If all the bits of the 8 block logic block bitmaps are scanned
    // (i >= 8 or j >= 8192), the 0 value bit is not found or the buffer block pointer of the bitmap
    // is invalid (bh = NULL), then exits with 0. (There is no free logic block).
82     if (!(sb = get\_super(dev)))
83         panic("trying to get new block from nonexistant device");
84     j = 8192;
85     for (i=0 ; i<8 ; i++)
86         if (bh=sb->s_zmap[i])
87             if ((j=find first zero(bh->b_data))<8192)
88                 break;
89     if (i>=8 || !bh || j>=8192)
90         return 0;
    // Next, the bits in the logical block bitmap corresponding to the found new logical block j
    // are set. If the corresponding bit is already set, it indicates that the system has an error
    // and stops. Otherwise set the modified flag of the corresponding buffer block that stores
    // the bitmap. Because the logic block bitmap only indicates the occupancy of the logic block
    // in the data zone on the disk, that is, the bit offset value in the logic block bitmap indicates
    // the block number from the beginning of the data zone, so the logical block number of the
    // first block of data zone needs to be added here to convert j to the logical block number.
    // If the new logical block number is greater than the total number of logical blocks on the
    // device, the block does not exist on the device. The application failed, return 0 & exit.
91     if (set\_bit(j, bh->b_data))
92         panic("new_block: bit already set");
93     bh->b_dirt = 1;
94     j += i*8192 + sb->s_firstdatazone-1;
95     if (j >= sb->s_nzones)
96         return 0;
    // Then, a buffer block is obtained in the buffer cache for the logical block number specified
    // on the device, and the buffer block header pointer is returned. Because the logical block
    // just obtained must have a reference count of 1 (set in getblk()), if it is not 1, it will
    // stop. Finally, the new logical block is cleared and its updated flag and modified flag are
    // set. Then release the corresponding buffer block and return the logical block number.
97     if (!(bh=getblk(dev, j)))
98         panic("new_block: cannot get block");
99     if (bh->b_count != 1)
100         panic("new_block: count is != 1");
101     clear\_block(bh->b_data);
102     bh->b_uptodate = 1;
103     bh->b_dirt = 1;
104     brelse(bh);
105     return j;
106 }
107
    /// Release the specified i-node.
    // This function first determines the validity and releasability of the i-node given by the
    // parameter. If the i-node is still in use, it cannot be released. Then, the i-node bitmap
    // is operated by using the super block information, the bit in the i-node bitmap corresponding

```

```

// to the i-node number is reset, and the i-node structure is cleared.
108 void free_inode(struct m_inode * inode)
109 {
110     struct super_block * sb;
111     struct buffer_head * bh;
112
113     // First determine the validity or legitimacy of the i-node that needs to be released. If the
114     // i-node pointer = NULL, then exit. If the device number field on the i-node is 0, the node
115     // is not used. Then clear the memory area occupied by the corresponding i-node and return.
116     // Memset() is defined at line 395 in file include/string.h. This means that the memory area
117     // specified by the inode pointer is filled with 0, and the sizeof(*inode) bytes are filled.
118     if (!inode)
119         return;
120     if (!inode->i_dev) {
121         memset(inode, 0, sizeof(*inode));
122         return;
123     }
124     // If there are other program references in this i-node, it cannot be released, indicating that
125     // there is a problem with the kernel code, and the system is down. If the number of file links
126     // is not 0, it means that there are other file directory entries that are using the node, so
127     // they should not be released, but should be put back.
128     if (inode->i_count>1) {
129         printk("trying to free inode with count=%d\n", inode->i_count);
130         panic("free_inode");
131     }
132     if (inode->i_nlinks)
133         panic("trying to free inode with links");
134
135     // After checking the validity of the i-node, we begin to operate on the i-node bitmap using
136     // its super-block information. First, take the super block of the device where the i-node is
137     // located, and test whether the device exists. Then determine whether the range of the i-node
138     // number is correct. If the i-node number is equal to 0 or greater than the total number of
139     // i-nodes on the device, an error occurs (the i-node of No. 0 is reserved). If the node bitmap
140     // corresponding to the i-node does not exist, an error occurs. Since the i-node bitmap of a
141     // buffer block has 8192 bits, i_num>>13 (ie, i_num/8192) can obtain the s_imap[] entry of the
142     // current i-node number, that is, the disk block.
143     if (!(sb = get_super(inode->i_dev)))
144         panic("trying to free inode on nonexistent device");
145     if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
146         panic("trying to free inode 0 or nonexistant inode");
147     if (!(bh=sb->s_imap[inode->i_num>>13]))
148         panic("nonexistent imap in superblock");
149
150     // Now we reset the bits in the node bitmap corresponding to the i-node. If the bit is already
151     // equal to 0, an error warning message is displayed. Finally, the buffer in which the i-node
152     // bitmap is located has been modified, and the memory area occupied by the i-node structure
153     // is cleared.
154     if (clear_bit(inode->i_num&8191, bh->b_data))
155         printk("free_inode: bit already cleared. |n|r");
156     bh->b_dirt = 1;
157     memset(inode, 0, sizeof(*inode));
158 }
159
160 // Create a new i-node for device, initialize and return a pointer to the new i-node.
161 // Obtain an free i-node entry in the memory i-node table and find an free i-node from the i-node

```

---

```

// bitmap.
137 struct m\_inode * new\_inode(int dev)
138 {
139     struct m\_inode * inode;
140     struct super\_block * sb;
141     struct buffer\_head * bh;
142     int i, j;
143
144     // First, obtain a free i-node entry from the in-memory i-node table (inode_table) and read
145     // the super block structure of the specified device. Then scan the 8-block i-node bitmap in
146     // the super block, look for the first 0-bit (look for a free node), and obtain the node number
147     // where the i-node is placed. If the i-node is not found after the scan, or the buffer block
148     // where the bitmap is located is invalid (bh=NULL), then the i-node in the previously requested
149     // i-node table is returned, and the null pointer is returned and exited (indicating no Idle
150     // i-node available).
151     if (!(inode=get\_empty\_inode()))                // fs/inode.c, line 197.
152         return NULL;
153     if (!(sb = get\_super(dev)))                    // fs/super.c, line 56.
154         panic("new_inode with unknown device");
155     j = 8192;
156     for (i=0 ; i<8 ; i++)
157         if (bh=sb->s_imap[i])
158             if ((j=find\_first\_zero(bh->b_data))<8192)
159                 break;
160     if (!bh || j >= 8192 || j+i*8192 > sb->s_ninodes) {
161         iput(inode);
162         return NULL;
163     }
164
165     // Now that we have found the i-node number j that has not been used, we set the corresponding
166     // bit of the i-node bitmap corresponding to j (if it is already set, it indicates that the
167     // kernel is faulty). Then set the modified flag of the buffer block where the i-node bitmap
168     // is located. Finally, the i-node structure is initialized (i_ctime is the time when the content
169     // of the i-node is changed).
170     if (set\_bit(j, bh->b_data))
171         panic("new_inode: bit already set");
172     bh->b_dirt = 1;
173     inode->i_count=1;                                // reference count.
174     inode->i_nlinks=1;                                // number of file directory entry links.
175     inode->i_dev=dev;                                // device number the i-node is located.
176     inode->i_uid=current->euid;                        // user id of the i-node.
177     inode->i_gid=current->egid;                        // group id.
178     inode->i_dirt=1;                                // set the modified flag.
179     inode->i_num = j + i*8192;                        // Corresponds to i-node nr in the device.
180     inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT\_TIME;
181     return inode;
182 }

```

---

## 12.4 truncate.c

### 12.4.1 Function

The truncate.c program is used to release all logical blocks occupied by the specified i-node on the device, including direct blocks, primary indirect blocks, and secondary indirect blocks. Therefore, the file length corresponding to the node of the file is cut to 0, and the occupied device space is released. For the convenience of reading, the schematic diagram of the direct block and indirect block organization structures in the i-node is given again, as shown in Figure 12-23.

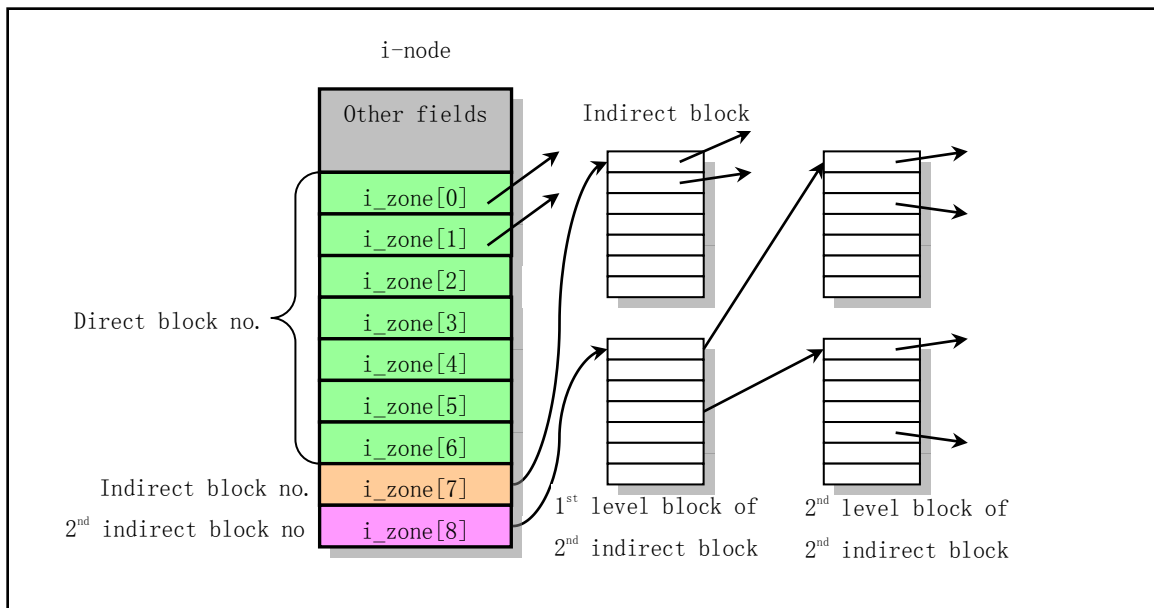


Figure 12-23 Schematic diagram of logical block connections of a i-node

The `i_zone[]` array in the i-node stores the disk block number of the logical block on the device. The first 7 items of the array (`i_zone[0]`--`i_zone[6]`) directly store the numbers of the first 7 data blocks in the relevant file. `i_zone[7]` stores the block number of the primary indirect block. Because the disk size is 1024 bytes, each disk block can store  $(1024 / 2) = 512$  disk block numbers, that is, one indirect block number can address up to 512 device disk blocks. Accordingly, the secondary indirect block number `i_zone[8]` can addresses  $(512 * 512) = 261,144$  disk blocks.

### 12.4.2 Code annotation

Program 12-3 linux/fs/truncate.c

```

1 /*
2  * linux/fs/truncate.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 // <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.
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//      and constants.
7 #include <linux/sched.h>
8
9 #include <sys/stat.h>
10
11 // Release all one indirect blocks.
12 // The parameter dev is the device number of the file system; block is the logical block number.
13 // Returns 1 if successful, otherwise returns 0.
14 static int free_ind(int dev, int block)
15 {
16     struct buffer_head * bh;
17     unsigned short * p;
18     int i;
19     int block_busy;           // The logic block has not been released.
20
21 // First determine the validity of the parameters. Returns if the logical block number is 0.
22 // Then the primary block is read, and all the logical blocks used on it are released, and then
23 // the buffer block of this primary indirect block is released. The function free_block() is
24 // used to release the disk block of the specified logical block number on the device (fs/bitmap.c
25 // line 47).
26 if (!block)
27     return 1;
28 block_busy = 0;
29 if (bh=bread(dev, block)) {
30     p = (unsigned short *) bh->b_data; // points to the block data area.
31     for (i=0; i<512; i++, p++)        // a block can contains 512 block nr.
32         if (*p)
33             if (free_block(dev, *p)) {
34                 *p = 0;
35                 bh->b_dirt = 1; // set the modified flag.
36             } else
37                 block_busy = 1; // block is not released.
38     brelse(bh); // release the buffer block occupied by the indirect block.
39 }
40 // Finally release an indirect block on the device. But if there is a logical block that is
41 // not released, it returns 0 (failure).
42 if (block_busy)
43     return 0;
44 else
45     return free_block(dev, block); // 1 if successful, otherwise return 0.
46 }
47
48 // Release all secondary indirect blocks.
49 // The parameter dev is the device number; 'block' is the logical block number of the secondary
50 // indirect block.
51 static int free_dind(int dev, int block)
52 {
53     struct buffer_head * bh;
54     unsigned short * p;
55     int i;

```

```

43         int block_busy;                                // The logic block has not been released.
44
45         // First determine the validity of the parameters. Returns if the logical block number is 0.
46         // The code then reads the first-level block of the secondary indirect block and releases all
47         // the logical blocks on it indicating that it has been used, and then releases the buffer block
48         // of the first-level block.
49         if (!block)
50             return 1;
51         block_busy = 0;
52         if (bh=bread(dev, block)) {
53             p = (unsigned short *) bh->b_data; // points to the block data area.
54             for (i=0; i<512; i++, p++)        // 512 second-level blocks
55                 if (*p)
56                     if (free_ind(dev, *p)) { // release primary indirect block.
57                         *p = 0;
58                         bh->b_dirt = 1; // set modified flag.
59                     } else
60                         block_busy = 1; // block not released.
61             brelse(bh); // release buffer block occupied by secondary indirect block.
62         }
63         // Finally, the secondary indirect block on the device is released. But if there is a logic
64         // block that is not released, it returns 0 (failure).
65         if (block_busy)
66             return 0;
67         else
68             return free_block(dev, block);
69     }
70
71     // Truncates file data.
72     // The file length corresponding to the node is cut to 0, and the occupied device space is
73     // released.
74     void truncate(struct m_inode * inode)
75     {
76         int i;
77         int block_busy;
78
79         // First determine the validity of the specified i-node. Returns if it is not a regular file,
80         // a directory, or a link entry. Then release 7 direct blocks of the i-node and set all 7 logical
81         // block items to zero. If the block is busy and not released, the block_busy flag is set.
82         if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode) ||
83             S_ISLNK(inode->i_mode)))
84             return;
85         repeat:
86         block_busy = 0;
87         for (i=0; i<7; i++)
88             if (inode->i_zone[i]) { // release if block nr is not zero.
89                 if (free_block(inode->i_dev, inode->i_zone[i]))
90                     inode->i_zone[i]=0;
91                 else
92                     block_busy = 1; // set if not released.
93             }
94         if (free_ind(inode->i_dev, inode->i_zone[7])) // release primory indirect blocks.
95             inode->i_zone[7] = 0;

```

```
84     else
85         block_busy = 1;                // set if not released.
86     if (free\_dind(inode->i_dev, inode->i_zone[8])) // release secondary indirect blocks
87         inode->i_zone[8] = 0;
88     else
89         block_busy = 1;

    // After that, the i-node modified flag is set, and if there is still a logical block that is
    // not released due to "busy", the current process running time slice is set to 0 to switch
    // to another process to run, and then wait for a while to re-execute the release operation.
    // Finally, the file modification time and the i-node change time are set to the current time.
    // The macro CURRENT_TIME is defined in line 142 of the header file linux/sched.h and is defined
    // as (startup_time + jiffies/HZ) for the seconds value since 1970:0:0:0.
90     inode->i_dirt = 1;
91     if (block_busy) {
92         current->counter = 0;            // current process time slice is set to 0.
93         schedule();
94         goto repeat;
95     }
96     inode->i_size = 0;                // the file size is set to zero.
97     inode->i_mtime = inode->i_ctime = CURRENT\_TIME;
98 }
99
100
```

---

## 12.5 inode.c

### 12.5.1 Function

The inode.c program includes functions `iget()`, `iput()`, and block mapping function `bmap()` that process the i-node, as well as other auxiliary functions. The `iget()`, `iput()`, and `bmap()` functions are mainly used in the `namei()` mapping function of the `namei.c` program to find the corresponding i-node from the file path name.

#### 1. `iget()` function

The `iget()` function is used to read the i-node of the specified node number `nr` from the device `dev`, and increment the reference count field value `i_count` of the node by one. The operation flow is shown in Figure 12-24. The function first determines the validity of the parameter `dev`, and takes an idle i-node from the i-node table, then scans the i-node table, finds the i-node with the specified node number `nr`, and increments the reference number of the i-node. If the device of the currently scanned is not the specified device or the node number is not equal to the specified node, continue scanning. Otherwise, the i-node with the specified device number and node number has been found, and the node is waiting to be unlocked (if it is locked right now). While waiting for the i-node to be unlocked, the node table may change. So if the device number of the i-node is now not equal to the specified device number or the node number is not equal to the specified node number, the entire i-node table needs to be re-scanned again. Subsequently, the reference count value of the i-node is incremented by 1, and it is determined whether the i-node is a mount point of another file system.



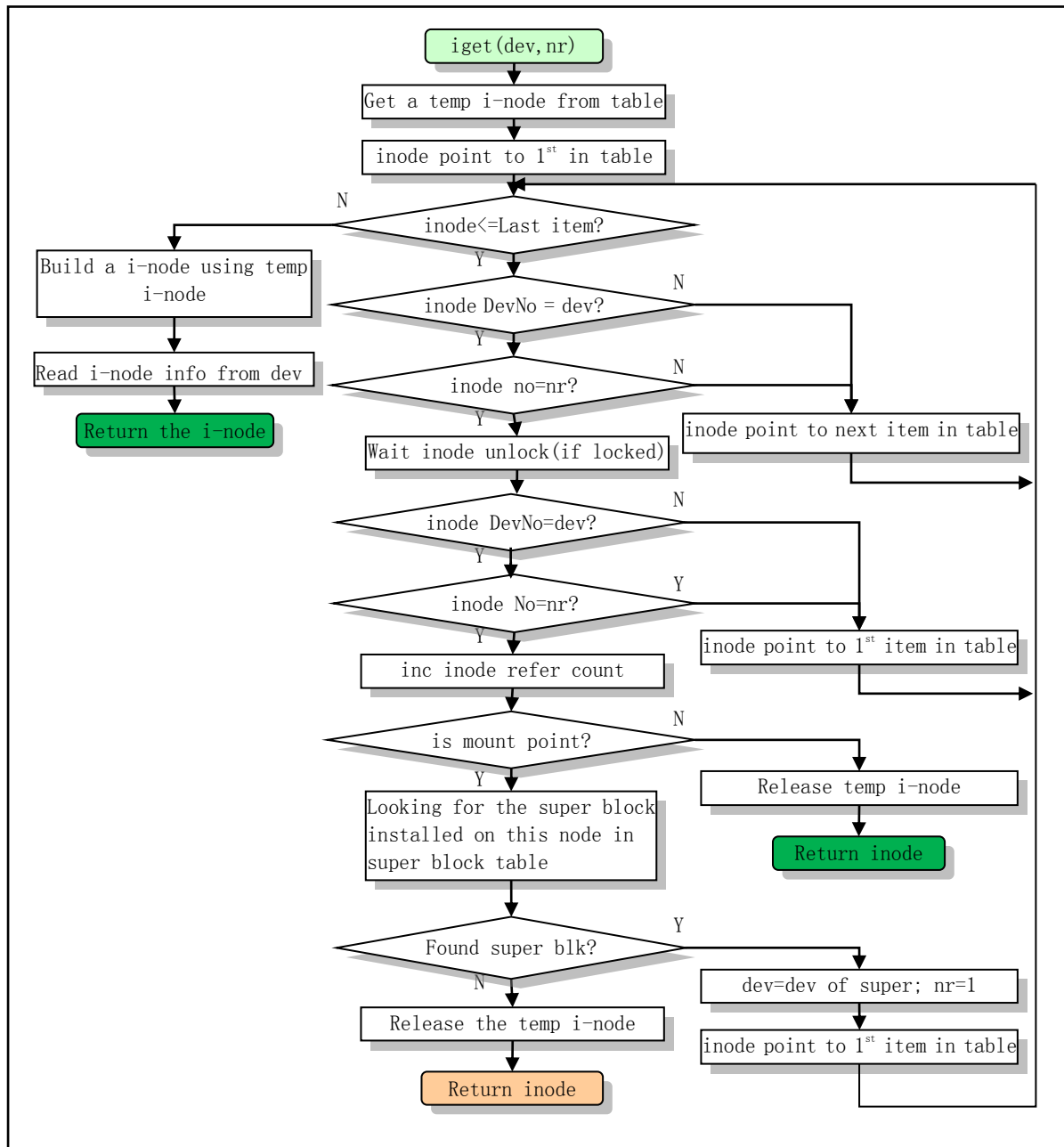


Figure 12-24 iget() function operation flow chart

If the i-node is a mount point of a file system, the super block table is searched for the super block installed on the i-node. If the corresponding super block is not found, an error message is displayed, and the free node that the function starts to acquire is released, and the i-node pointer is returned. If the corresponding super block is found, the i-node is written to the disk. Then get the device number from the super block installed in the i-node file system, and let the i-node number be 1. Then scan the entire i-node table again to fetch the root node of the mounted file system.

If the i-node is not the installation point of other file systems, it indicates that the corresponding i-node has been found, so the idle i-node that is temporarily applied can be discarded at this time, and the found i-node pointer is returned.

If the specified i-node is not found in the i-node table, the node is established in the i-node table using the idle i-node of the previous application, and the i-node information is read from the corresponding device, and

the i-node pointer is returned.

## 2. iput() function

The function performed by the iput() function is exactly the opposite of iget(). It is mainly used to decrement the i-node reference count value by 1, and if it is a pipe i-node, wake up the waiting process. If the i-node is the i-node of the block device file, the device is refreshed, and if the link count of the i-node is 0, all the disk logical blocks occupied by the i node are released, and are returned after the i-node is released. If the i-node reference count value i\_count is 1, the number of links is not zero, and the content has not been modified, then the i-node reference count is decremented by one at this time, and then returned. Because if an i node has i\_count=0, it means it has been released. The operation flow of this function is also similar to iget().

If the process does not need to continuously use an i-node at a certain time, the iput() function should be called to decrement the value of the reference count field i\_count of the i-node, and also let the kernel perform some other processing. Therefore, after performing one of the following operations, the kernel code should normally call the iput() function:

- Increase the value of the i-node reference count field i\_count by one;
- Called the namei(), dir\_namei(), or open\_namei() function;
- Called the iget(), new\_inode(), or get\_empty\_inode() function;
- When closing a file, if no other process has used the file;
- When unmounting a file system (replace the device file i-node, etc.).

In addition, when a process is created, its current working directory pwd, current root directory root, and executable file directory executable fields are initialized to point to three i-nodes, and the reference count field of three i-nodes are also set accordingly. Therefore, when the process executes a system-call that changes the current working directory, the iput() function needs to be called in code of the system-call to first put back the i-node in use, and then let the pwd of the process point to the new i-node of the path name. Similarly, to modify the root and executable fields of a process, you also need to execute the iput() function.

## 3. bmap() function

The \_bmap() function is used to map a file data block to the corresponding disk block. The parameter 'inode' is the i-node pointer of the file, 'block' is the data block number in the file, and 'create' is the creation flag, indicating whether the corresponding disk block needs to be established on the disk if the corresponding file data block does not exist. The return value of this function is the logical block number (disk block number) corresponding to the file data block on the device. When create=0, the function is the bmap() function. When create=1, it is the create\_block() function.

The data in a regular file is placed in the data zone of the disk, and a file name is associated with these data disk blocks through the corresponding i-node. The numbers of these disk blocks are stored in the logical block array of the i-node. The \_bmap() function mainly processes the logical block array i\_zone[] of the i-node, and sets the occupancy of the logical block bitmap according to the logical block number in i\_zone[]. See Figure 12-6 in Section 12.1. As mentioned earlier, i\_zone[0] to i\_zone[6] are used to store the direct logical block number of the file; i\_zone[7] is used to store the indirect logical block number; and i\_zone[8] is used to store the secondary indirect logical block number. When the file size is small (less than 7K), the disk block number used by the file can be directly stored in the 7 direct block items in the i-node; when the file is slightly larger (not more than 7K+512K), it needs to use primary indirect block item i\_zone[7]; when the file is larger, the secondary indirect block item i\_zone[8] is needed. Therefore, when the file is relatively small, the speed of the Linux addressing disk block is faster.

## 12.5.2 Code annotation

Program 12-4 linux/fs/inode.c

```

1  /*
2   *  linux/fs/inode.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  // <string.h> String header file. Defines some embedded functions about string operations.
8  // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
9  //   and constants.
10 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
11 //   of the initial task 0, and some embedded assembly function macro statements about the
12 //   descriptor parameter settings and acquisition.
13 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
14 //   used functions of the kernel.
15 // <linux/mm.h> Memory management header file. Contains page size definitions and some page
16 //   release function prototypes.
17 // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
18 //   descriptors/interrupt gates, etc. is defined.
19 #include <string.h>
20 #include <sys/stat.h>
21
22 #include <linux/sched.h>
23 #include <linux/kernel.h>
24 #include <linux/mm.h>
25 #include <asm/system.h>
26
27 // An array of pointers to the total number of device data blocks. Each pointer item points
28 // to the total number of blocks of the given major device number, hd_sizes[]. Each item of
29 // the total number of block arrays corresponds to the total number of data blocks owned by
30 // a sub-device determined by the sub-device number.
31 extern int *blk_size[];
32
33 struct m_inode inode_table[NR_INODE]={0,},}; // In-memory i-node table (NR_INODE=32).
34
35 static void read_inode(struct m_inode * inode); // read i-node information, line 297.
36 static void write_inode(struct m_inode * inode); // write i-node info to cache, line 324.
37
38 // Wait for the specified i-node to be available.
39 // If the i-node has been locked, the current task is placed in an uninterruptible wait state
40 // and added to the i-wait queue until the i-node unlocks and explicitly wakes up the task.
41 static inline void wait_on_inode(struct m_inode * inode)
42 {
43     cli();
44     while (inode->i_lock)
45         sleep_on(&inode->i_wait); // kernel/sched.c, line 199.
46     sti();
47 }
48
49 // Lock the i-node (lock the given i-node).
```

```

// If the i-node is locked, the current task is placed in an uninterruptible wait state and
// added to the i-wait queue i_wait. Until the i-node unlocks and explicitly wakes up the task,
// then locks it.
30 static inline void lock_inode(struct m_inode * inode)
31 {
32     cli();
33     while (inode->i_lock)
34         sleep_on(&inode->i_wait);
35     inode->i_lock=1;                // set locked flag.
36     sti();
37 }
38
////// Unlock the specified i node.
// Reset the lock flag of the i-node and explicitly wake up all processes waiting on the i-wait
// queue i_wait.
39 static inline void unlock_inode(struct m_inode * inode)
40 {
41     inode->i_lock=0;
42     wake_up(&inode->i_wait);        // kernel/sched.c, line 204.
43 }
44
////// Release all i-nodes of device dev in the memory i-node table.
// Scans the i-node table array in memory and releases it if an item is used by the specified
// device.
45 void invalidate_inodes(int dev)
46 {
47     int i;
48     struct m_inode * inode;
49
// First let the pointer point to the first item in the memory i-node table array, and then
// scan all the i-nodes in it. For each of the i-nodes, wait for the i-node to unlock (if it
// is currently locked), and then determine if it belongs to the specified device. If yes, it
// is released, that is, the device number field i_dev of the i-node is set to 0, and the modified
// flag is also reset. In the meantime, it will also check if it is still being used, that is,
// whether its reference count is not 0, and if so, a warning message is displayed.
// The pointer assignment sentence "0+inode_table" on line 50 is equivalent to "inode_table",
// "&inode_table[0]", but this may be more straightforward.
50     inode = 0+inode_table;        // points to the first item of i-node table.
51     for(i=0 ; i<NR_INODE ; i++,inode++) {
52         wait_on_inode(inode);      // wait for i-node to be available(unlocked).
53         if (inode->i_dev == dev) {
54             if (inode->i_count)      // if references is not 0, error warning.
55                 printk("inode in use on removed disk\n\r");
56             inode->i_dev = inode->i_dirt = 0;    // release i-node.
57         }
58     }
59 }
60
////// Synchronize all i-nodes.
// Synchronize all i-nodes in the memory i-node table with i-nodes on the device.
61 void sync_inodes(void)
62 {
63     int i;

```

```

64     struct m\_inode * inode;
65
66     // First, let the pointer of the memory i-node type point to the first item of the i-node table,
67     // and then scan all the nodes in the table. For each of the i-nodes, wait for the i-node to
68     // be unlocked (if it is currently locked), and then check if the i-node has been modified and
69     // is not a pipe node. If this is the case, the i-node is written to the buffer cache. The buffer
70     // manager program will write them to the disk at the appropriate time.
71     inode = 0 + inode\_table; // points to the first item of the table.
72     for(i=0 ; i<NR\_INODE ; i++,inode++) {
73         wait\_on\_inode(inode); // wait for i-node to be available.
74         if (inode->i_dirt && !inode->i_pipe) // modified and not a pipe node.
75             write\_inode(inode); // write to the buffer cache.
76     }
77
78     ///// File data block mapping to disk block (bmap - block map).
79     // Parameters: inode - i-node pointer of the file; block - data block number in the file;
80     // create - the block creation flag. This function maps the specified file data block to the
81     // logical block on the device and returns the logical block number. If the block creation flag
82     // is set, a new disk block is requested when the corresponding logical block does not exist
83     // on the device, and the logical block number (disk block number) corresponding to the file
84     // data block is returned. The function is processed in four parts: (1) parameters validity
85     // check; (2) direct block processing; (3) primary indirect block processing; and (4) secondary
86     // indirect block processing.
87     static int bmap(struct m\_inode * inode, int block, int create)
88     {
89         struct buffer\_head * bh;
90         int i;
91
92         // (1) First check the validity of the parameters. If the file data block number is less than
93         // 0, the kernel will stop. If the block number is greater than (direct block nr + indirect
94         // block nr + second indirect block nr), it is out of range of the file system.
95         if (block<0)
96             panic("_bmap: block<0");
97         if (block >= 7+512+512*512)
98             panic("_bmap: block>big");
99
100        // (2) Then, according to the size of the file block number and whether the creation flag is
101        // set, the processing is performed separately. If the block number is less than 7, it is
102        // represented by a direct block. At this time, if the creation flag is set and the logical
103        // block field in the i-node is 0, a disk block (logical block) is requested from the device,
104        // and the logical block number on the disk is filled in the logical block field. Then set
105        // i-node change time and modified flag, and finally return the logical block number. The function
106        // new_block() is defined at line 76 in the bitmap.c program.
107        if (block<7) {
108            if (create && !inode->i_zone[block])
109                if (inode->i_zone[block]=new\_block(inode->i_dev)) {
110                    inode->i_ctime=CURRENT\_TIME; // ctime - change time
111                    inode->i_dirt=1; // set modified flag.
112                }
113            return inode->i_zone[block];
114        }
115    }

```

```

// (3) If the block number is >= 7, and less than (7 + 512), it means that it uses an indirect
// block. Therefore, an indirect block is processed below. If it is a create operation, and
// the indirect block field i_zone[7] of the i-node is 0, it indicates that the file is the
// first time to use the indirect block. Therefore, it is necessary to apply for a disk block
// for storing the indirect block information, and fill the actual disk block number into the
// indirect block field. Then set the i-node modified flag and modification time. If the disk
// block fails to be created at the time of creation, the i-node indirect block field i_zone[7]
// is 0 at this time, and 0 is returned. Or the parameter creation flag is 0, but i_zone[7]
// is also 0, indicating that there is no indirect block in the i-node, so the mapping disk
// block fails, and returns 0 to exit.
91     block -= 7;
92     if (block < 512) {
93         if (create && !inode->i_zone[7])
94             if (inode->i_zone[7] = new_block(inode->i_dev)) {
95                 inode->i_dirt = 1;
96                 inode->i_ctime = CURRENT_TIME;
97             }
98         if (!inode->i_zone[7])
99             return 0;
// Now read an indirect block of the i-node on the device and get the logical block number i
// in the 'block' item on it (each block number occupies 2 bytes). If it is a create operation
// and the obtained logical block number is 0, then it is necessary to apply for a disk block
// and set the 'block' entry in the indirect block equal to the logical block number of the
// new disk block. Then set the modified flag of the indirect block. If it is not a create
// operation, then 'i' is the logical block number that needs to be mapped (find). Finally,
// the buffer block occupied by the indirect block is released, and the logical block number
// of the new application or the corresponding block on the disk is returned.
100         if (!(bh = bread(inode->i_dev, inode->i_zone[7])))
101             return 0;
102         i = ((unsigned short *) (bh->b_data))[block];
103         if (create && !i)
104             if (i = new_block(inode->i_dev)) {
105                 ((unsigned short *) (bh->b_data))[block] = i;
106                 bh->b_dirt = 1;
107             }
108         brelse(bh);
109         return i;
110     }

```

// (4) If the program runs to this place, it indicates that the data block belongs to the secondary indirect block. Its processing is similar to the primary indirect block. The following is the processing of the secondary indirect block. First, the block is subtracted from the number of blocks (512) accommodated by the primary indirect block, and then created or searched according to whether the creation flag is set. If it is a create operation and the secondary indirect block field of the i node is 0, then a new disk block is required to store the primary block information of the secondary indirect block, and the actual disk block number is filled in the secondary indirect block field. After that, set the i-node modified flag and modification time. Similarly, if the disk block fails to be created at the time of creation, then the i-node secondary indirect block field i\_zone[8] is 0, and 0 is returned. Or it is not a create operation, but i\_zone[8] turns out to be 0, indicating that there is no indirect block in the i-node, so the mapped disk block fails, and returns 0 to exit.

```

111     block -= 512;
112     if (create && !inode->i_zone[8])

```

```

113         if (inode->i_zone[8]=new_block(inode->i_dev)) {
114             inode->i_dirt=1;
115             inode->i_ctime=CURRENT_TIME;
116         }
117         if (!inode->i_zone[8])
118             return 0;
119         // Now read the secondary indirect block of the i-node on the device and get the logical block
120         // number 'i' in the (block / 512) entry on the first-level block of the secondary indirect
121         // block. If it is a create operation, and the logical block number in the (block / 512) item
122         // on the first-level block of the secondary indirect block is 0, then it is necessary to apply
123         // for a disk block as the second-level block 'i' of the secondary indirect block, and let the
124         // (block / 512) item in the first-level block of the secondary indirect block to be equal to
125         // the block number 'i' of the second-level block. Then set the first-level block modified flag
126         // of the secondary indirect block and release this first-level block of the secondary indirect
127         // block. If it is not created, then 'i' is the logical block number that needs to be found.
119         if (!(bh=bread(inode->i_dev, inode->i_zone[8])))
120             return 0;
121         i = ((unsigned short *)bh->b_data)[block>>9];
122         if (create && !i)
123             if (i=new_block(inode->i_dev)) {
124                 ((unsigned short *) (bh->b_data))[block>>9]=i;
125                 bh->b_dirt=1;
126             }
127         brelse(bh);
128         // If the second-level block number of the secondary indirect block is 0, it means that the
129         // application disk block fails or the original corresponding block number is 0, then 0 is
130         // returned. Otherwise, the second-level block of the secondary indirect block is read from
131         // the device, and the logical block number in the block item on the second-level block is taken.
128         if (!i)
129             return 0;
130         if (!(bh=bread(inode->i_dev, i)))
131             return 0;
132         i = ((unsigned short *)bh->b_data)[block&511]; // limits to 511 (0x1fff).
133         // If the creation flag is set, and the logical block number in the block of the second-level
134         // block is 0, then apply a disk block as the block that finally stores the data information,
135         // and let the block item in the second-level block equal the new logical block number 'i'.
136         // Then set the modified flag of the second-level block. Finally, the second-level block of
137         // the secondary indirect block is released, and the logical block number of the newly applied
138         // or original corresponding block on the disk is returned.
133         if (create && !i)
134             if (i=new_block(inode->i_dev)) {
135                 ((unsigned short *) (bh->b_data))[block&511]=i;
136                 bh->b_dirt=1;
137             }
138         brelse(bh);
139         return i;
140     }
141
142     // Get the logical block number of the file data block on the device.
143     // Parameters: inode - i-node pointer of the file; block - the data block number in the file.
144     // If the operation is successful, the logical block number is returned, otherwise returns 0.
142 int bmap(struct m_inode * inode, int block)
143 {

```

```

144         return bmap(inode,block,0);        // create flag = 0.
145     }
146
147     // Find or create a disk block on the device for the file block.
148     // Find the corresponding disk block of the file data block on the device. If it does not exist,
149     // create a block and return the corresponding disk block number.
150     // Parameters: inode - i-node pointer of the file; block - the data block number in the file.
151     // If the operation is successful, the logical block number is returned, otherwise returns 0.
152     int create_block(struct m_inode * inode, int block)
153     {
154         return bmap(inode,block,1);
155     }
156
157     // Put back an i-node (write back to device).
158     // This function is mainly used to decrement the i-node reference count by 1, and if it is a
159     // pipe i-node, wake up the waiting process. If it is a block device file i-node, the device
160     // is refreshed, and if the link count of the i-node is 0, all the disk logical blocks occupied
161     // by the i-node are released, and the i-node is released.
162     void iput(struct m_inode * inode)
163     {
164         // First check the validity of the i-node given by the parameter and wait for the i-node to
165         // unlock (if it is locked). If the reference count of the i-node is 0, it means that the i-node
166         // is already idle. The kernel then asks for a put back operation on it, indicating that there
167         // is a problem with code in the kernel. The error message is then displayed and the machine
168         // is shut down.
169         if (!inode)
170             return;
171         wait_on_inode(inode);
172         if (!inode->i_count)
173             panic("iput: trying to free free inode");
174         // If it is a pipe i-node, wake up the process waiting for the pipe, the number of references
175         // is decremented by 1, and if the reference count is still not 0, the function returned.
176         // Otherwise, the memory page occupied by the pipeline is released, and the reference count,
177         // modified flag, and pipe flag of the node are reset, then function returned. For the pipe
178         // i-nodes, inode->i_size stores the memory page address. See get_pipe_inode(), lines 231,237.
179         if (inode->i_pipe) {
180             wake_up(&inode->i_wait);
181             wake_up(&inode->i_wait2);        //
182             if (--inode->i_count)
183                 return;
184             free_page(inode->i_size);
185             inode->i_count=0;
186             inode->i_dirt=0;
187             inode->i_pipe=0;
188             return;
189         }
190         // If the device number field of the i-node is 0, the reference count for this node is decremented
191         // by one and returned. For example, an i-node for pipeline operation has a device number of
192         // 0 for the i-node. In addition, if it is the i-node of the block device file, then in the
193         // logical block field 0 (i_zone[0]) is the device number, then refresh the device and wait
194         // for the i-node to unlock.
195         if (!inode->i_dev) {
196             inode->i_count--;

```



```

172         return;
173     }
174     if (S_ISBLK(inode->i_mode)) {
175         sync_dev(inode->i_zone[0]);
176         wait_on_inode(inode);
177     }
    // If the reference count of the i-node is greater than 1, the count is decremented by 1 and
    // returned directly (because the i-node is still in use and cannot be released), otherwise
    // the reference count value of the i-node is 1 (because it has been checked on line 157 when
    // it is zero). If the number of links of the i-node is 0, it indicates that the corresponding
    // file is deleted. All logical blocks of the i-node are then released and the i-node is released.
    // The function free_inode() is used to actually release the i-node operation, that is, reset
    // the i-node bitmap bit, and clear the i-node structure content.
178 repeat:
179     if (inode->i_count>1) {
180         inode->i_count--;
181         return;
182     }
183     if (!inode->i_nlinks) {          // at this point, i_count = 1.
184         truncate(inode);
185         free_inode(inode);         // bitmap.c, line 108.
186         return;
187     }
    // If the i-node has been modified, write back to update the i-node and wait for the i-node
    // to unlock (if locked). Since it may sleep when writing the i-node, other processes may modify
    // the i-node at this time, so the above-mentioned checking procedure (label repeat) needs to
    // be repeated again after the process is woken up.
188     if (inode->i_dirt) {
189         write_inode(inode);        /* we can sleep - so do again */
190         wait_on_inode(inode);
191         goto repeat;
192     }
    // If the program is executed here, the i-node reference count i_count is 1, the number of links
    // is not zero, and the content has not been modified. Therefore, as long as the i-node reference
    // count is decremented by 1, it can be returned. At this time, i_count=0 indicates that the
    // i-node has been released.
193     inode->i_count--;
194     return;
195 }
196
197 struct m_inode * get_empty_inode(void)
198 {
199     struct m_inode * inode;
200     static struct m_inode * last_inode = inode_table; // point to the first item.
201     int i;
202
    // The entire i-node table is scanned after initializing the last_inode pointer to point to
    // the i-node table header. If the last_inode has already pointed to the last item of the i-node
    // table, let it re-point to the beginning of the i-node table to continue looping through the
    // table. If the reference count of the i-node pointed to by last_inode is 0, it indicates that

```

```

// an idle i-node item may be found, and then the inode is pointed to the i-node. If the modified
// flag and the lock flag of the i-node are both 0, then we can use the i-node and so we exit
// the for() loop.
203     do {
204         inode = NULL;
205         for (i = NR\_INODE; i ; i--) { // NR_INODE = 32.
206             if (++last_inode >= inode\_table + NR\_INODE)
207                 last_inode = inode\_table;
208             if (!last_inode->i_count) {
209                 inode = last_inode;
210                 if (!inode->i_dirt && !inode->i_lock)
211                     break;
212             }
213         }
// If no free i-node is found (inode = NULL), the i-node table is printed for debugging use
// and stops the system.
214         if (!inode) {
215             for (i=0 ; i<NR\_INODE ; i++)
216                 printk("%04x: %6d\t", inode\_table[i].i_dev,
217                     inode\_table[i].i_num);
218                 panic("No free inodes in mem");
219         }
// Otherwise wait for the i-node to unlock (if it is locked again). If the i-node modified flag
// is set, the i-node is refreshed (synchronized). Because it may sleep when refreshing, you
// need to cycle again to wait for the i-node to unlock.
220         wait\_on\_inode(inode);
221         while (inode->i_dirt) {
222             write\_inode(inode);
223             wait\_on\_inode(inode);
224         }
225     } while (inode->i_count);

// If the i-node is occupied by others (the reference count of the i-node is not 0), then the
// idle i-node needs to be searched again. Otherwise, it indicates that an idle i-node item
// that meets the requirements has been found. Then, the content of the i-node item is cleared,
// and the reference count is set to 1, and the i-node pointer is returned.
226     memset(inode, 0, sizeof(*inode));
227     inode->i_count = 1;
228     return inode;
229 }
230
//// Get the pipe i-node.
// First scan the i-node table, look for an idle i-node entry, and then get a page of free memory
// for the pipeline to use. Then the reference count of the obtained i-node is set to 2 (reader
// and writer), the pipe head and tail are initialized, and the pipe type representation of
// the i-node is set. Returns the pointer to the i-node and returns NULL if it fails.
231 struct m\_inode * get\_pipe\_inode(void)
232 {
233     struct m\_inode * inode;
234
// First we get an free i-node from the memory i-node table and return NULL if we can't find
// one. Then apply for a page of memory for the i-node and have the node's i_size field point
// to the page. If there is no free memory, the i-node is released and NULL is returned.

```

```

235     if (!(inode = get\_empty\_inode()))
236         return NULL;
237     if (!(inode->i_size=get\_free\_page())) {        // i_size points to a buffer page.
238         inode->i_count = 0;
239         return NULL;
240     }
    // Then set the reference count of the i-node to 2 and reset the pipe head and tail pointer.
    // The i_zone[0] and i_zone[1] of the i-node logical block number array are used to store the
    // pipe header and the pipe tail pointer, respectively. Finally, set the i-node flag to be the
    // pipe i-node type and return the i-node.
241     inode->i_count = 2;                          /* sum of readers/writers */
242     PIPE\_HEAD(*inode) = PIPE\_TAIL(*inode) = 0; // Reset the pipe head and tail.
243     inode->i_pipe = 1;                            // pipe i-node.
244     return inode;
245 }
246
    ///// Get an i-node from device.
    // Parameters: dev - device number; nr - i-node number.
    // The i-node structure content of the specified i-node number is read from the device into
    // the memory i-node table, and the i-node pointer is returned. First search in the i-node table
    // located in the buffer cache. If the i-node with the specified node number is found, the i-node
    // pointer is returned after some check processing. Otherwise, the i-node is read from the device
    // and placed in the i-node table, and the i-node pointer is returned.
247 struct m\_inode * iget(int dev,int nr)
248 {
249     struct m\_inode * inode, * empty;
250
    // First determine the validity of the parameters. If the device number is 0, it indicates a
    // kernel code problem, displays an error message and stops. Otherwise, an idle i-node is taken
    // in advance from the i-node table for backup. Then scan the entire i-node table, look for
    // the i-node with the specified node number nr, and increment the reference number by one.
    // However, if the device number of the currently scanned i-node is not equal to the specified
    // device number or the node number is not equal to the specified node number, the scanning
    // is continued.
251     if (!dev)
252         panic("iget with dev==0");
253     empty = get\_empty\_inode();
254     inode = inode\_table;                        // points to the first item of table.
255     while (inode < NR\_INODE+inode\_table) {
256         if (inode->i_dev != dev || inode->i_num != nr) {
257             inode++;
258             continue;
259         }
    // If an i-node with the specified dev and nr is found, the node is awaited to be unlocked (if
    // locked). The i-node table may change while waiting for the node to be unlocked. Therefore,
    // the same check as above is required again when the code continues to execute. If a change
    // has occurred, the entire i-node table is rescanned again.
260         wait\_on\_inode(inode);
261         if (inode->i_dev != dev || inode->i_num != nr) {
262             inode = inode\_table;
263             continue;
264         }
    // This means that the corresponding i-node has been found. The i-node reference count is then

```

```

// incremented by one and then further checked to see if it is the mount point for another file
// system. If so, look for the root node of the mounted filesystem. If the i-node is indeed
// the mount point of other file systems, the super block table is searched for the super block
// installed on this i-node. If the super block is not found, an error message is displayed,
// and the idle node empty obtained at the beginning of the function is put back, and the i-node
// pointer is returned.
265         inode->i_count++;
266         if (inode->i_mount) {
267             int i;
268
269             for (i = 0 ; i<NR_SUPER ; i++)
270                 if (super_block[i].s_imount==inode)
271                     break;
272             if (i >= NR_SUPER) {
273                 printk("Mounted inode hasn't got sb\n");
274                 if (empty)
275                     iput(empty);
276                 return inode;
277             }
// The code execution to this point indicates that the file system superblock installed to the
// inode has been found. Then, we write the i-node to the disk to put it back, and replace the
// device number with the one taken from the super block installed on the i-node, and the i-node
// number needed is re-set to ROOT_INO, which is 1. The entire i-node table is then rescanned
// to obtain the root i-node information of the mounted file system.
278         iput(inode);
279         dev = super_block[i].s_dev;
280         nr = ROOT_INO;
281         inode = inode_table;
282         continue;
283     }
// Finally we found the corresponding i-node. Therefore, you can discard the idle i-node
// temporarily applied at the beginning of this function and return the found i-node.
284     if (empty)
285         iput(empty);
286     return inode;
287 }
// If we do not find the specified i-node in the table, the i-node is created in the table by
// using the idle i-node 'empty' of the previous application, and the i-node information is
// read from the corresponding device, and the i-node pointer is returned.
288     if (!empty)
289         return (NULL);
290     inode=empty;
291     inode->i_dev = dev;           // set the device of the new i-node.
292     inode->i_num = nr;           // set the i-node number.
293     read_inode(inode);
294     return inode;
295 }
296
///// Read the specified i-node information.
// The disk block containing the specified i-node information is read from the device and then
// copied to the specified i-node structure. In order to determine the logical block number
// of the disk block where the i-node is located, the super block on the corresponding device
// must first be read to obtain the information of INODES_PER_BLOCK for calculating the logical

```

```

// block number. After calculating the logical block number where the i-node is located, the
// logical block is read into a buffer block, and then the contents of the i-node at the
// corresponding position in the buffer block are copied to the specified position.
297 static void read\_inode(struct m\_inode * inode)
298 {
299     struct super\_block * sb;
300     struct buffer\_head * bh;
301     int block;
302
303     // First lock the i-node and get the super-block of the device where the node is located.
304     lock\_inode(inode);
305     if (!(sb=get\_super(inode->i_dev)))
306         panic("trying to read inode without dev");
307
308     // The logical block number of the device where the i-node is located = (boot block + super
309     // block) + i-node bitmap blocks + logical block bitmap blocks + (i node number - 1) / i-nodes
310     // per block, refer to Figure 12-1. Although the i-node number is numbered from 0, the first
311     // node 0 is not used, and the corresponding node 0 structure is not saved on the disk. Therefore,
312     // the first disk block storing the i-node holds the i-node structure with the i-node number
313     // 1--32 instead of 0--31. Therefore, it is necessary to decrement by one when calculating the
314     // disk block number of the i-node corresponding to the i-node number. Here we read the logical
315     // block where the i-node is located from the device and copy the contents of the i-node to
316     // the location indicated by the inode pointer.
317     block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
318         (inode->i_num-1)/INODES\_PER\_BLOCK;
319     if (!(bh=bread(inode->i_dev, block)))
320         panic("unable to read i-node block");
321     *(struct d\_inode *)inode =
322         ((struct d\_inode *)bh->b_data)
323         [(inode->i_num-1)%INODES\_PER\_BLOCK];
324
325     // Finally, the buffer block read in is released and the i-node is unlocked. For block device
326     // files, you also need to set the maximum length of the file for the i-node.
327     brelse(bh);
328     if (S\_ISBLK(inode->i_mode)) {
329         int i = inode->i_zone[0]; // i_zone[0] contains dev nr for device file.
330         if (blk\_size[MAJOR(i)])
331             inode->i_size = 1024*blk\_size[MAJOR(i)][MINOR(i)];
332         else
333             inode->i_size = 0x7fffffff;
334     }
335     unlock\_inode(inode);
336 }
337
338 // Write the i-node information to the buffer block.
339 // This function writes the specified i-node to the buffer block of the cache, and writes it
340 // to the disk when the cache is refreshed. In order to determine the logical block number of
341 // the device where the i-node is located, you need to first read the super-block on the device.
342 // After calculating the logical block number where the i-node is located, the logical block
343 // is read into a buffer block, and then the contents of the i-node are copied to the corresponding
344 // positions of the buffer block.
345 static void write\_inode(struct m\_inode * inode)
346 {
347     struct super\_block * sb;
348     struct buffer\_head * bh;

```

---

```

328         int block;
329
330         // The i-node is first locked. If the i-node has not been modified or the device number of the
331         // i-node is equal to zero, the i-node is unlocked and exits. For an i-node that has not been
332         // modified, its content is the same as in the buffer or in the device. Then get the super-block
333         // of the i-node.
334         lock_inode(inode);
335         if (!inode->i_dirt || !inode->i_dev) {
336             unlock_inode(inode);
337             return;
338         }
339         if (!(sb=get_super(inode->i_dev)))
340             panic("trying to write inode without device");
341         // Then we read the logical block where the i-node is located from the device, and copy the
342         // i-node information to the logical block corresponding to the location of the i-node item.
343         block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
344             (inode->i_num-1)/INODES_PER_BLOCK;
345         if (!(bh=bread(inode->i_dev, block)))
346             panic("unable to read i-node block");
347         ((struct d_inode *)bh->b_data)
348             [(inode->i_num-1)%INODES_PER_BLOCK] =
349             *(struct d_inode *)inode;
350         // Then we set the buffer modified flag, and the i-node content is already consistent with the
351         // buffer, so the i-node dirty flag is set to zero. Finally, the buffer containing the i-node
352         // is released and the i-node is unlocked.
353         bh->b_dirt=1;
354         inode->i_dirt=0;
355         brelse(bh);
356         unlock_inode(inode);
357     }
358 }
359

```

---

### 12.5.3 Information

None :)

## 12.6 super.c

### 12.6.1 Function

The super.c program contains functions that operate on the superblocks in the file system. These functions belong to the file system low-level functions and are used by the upper-level functions that deal with the filename and directories or pathes, mainly get\_super(), put\_super(), and read\_super(). There are also file system load/unload system calls sys\_umount() and sys\_mount(), and the root file system load function mount\_root(). Some other auxiliary functions are similar to those in buffer.c.

The super block mainly stores information about the entire file system. For its structure, see Figure 12-3 in Section 12.1, “Overall Function Description”.

The get\_super() function is used to search the corresponding super block in the memory super block array and return the pointer of the super block under the condition of the specified device. Therefore, when the

function is called, the corresponding file system must have been mounted, or at least the superblock has taken up an entry in the superblock array, otherwise it returns NULL.

The `put_super()` function is used to release the super block of the specified device. It releases the buffer block occupied by the i-node bitmap and the logical block bitmap of the file system corresponding to the super block, and releases the corresponding operation block item in the super block table (array) `super_block[]`. This function is called when the user calls `umount()` to unload a file system or replace a disk.

The `read_super()` function is used to read the super block of the file system of the specified device into the buffer and register it into the super block table. At the same time, the i-node bitmap and the logical block bitmap of the file system are also read into the super block structure in the memory array. Finally return a pointer to the super block structure.

The `sys_umount()` system-call is used to unmount a file system with a specified device file name, while `sys_mount()` is used to load a file system to a directory name. The last function in the program, `mount_root()`, is the root filesystem used to install the system and will be called when the system is initialized. The specific operation process is shown in Figure 12-25.

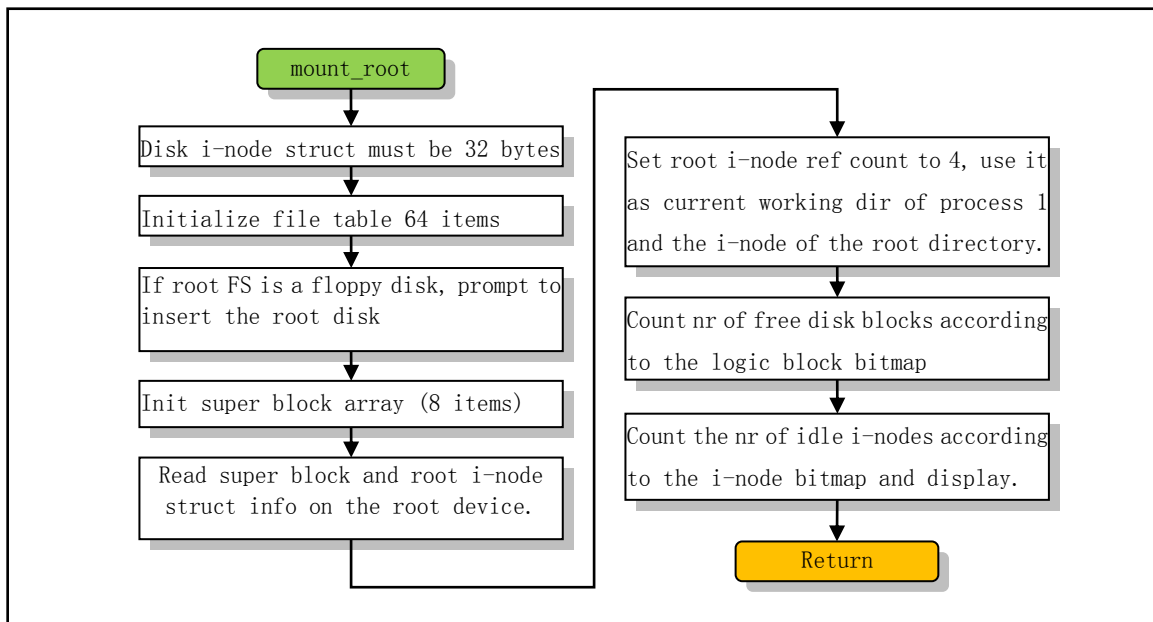


Figure 12-25 The purpose of the `mount_root()` function

In addition to installing the root file system, this function also initializes the kernel using the file system. It initializes the in-memory superblock array, initializes the file descriptor array `file_table[]`, and displays the number of free disk blocks and idle i-nodes in the root file system.

The `mount_root()` function is called in the system initialization file `main.c`, after process 0 creates the first child process (process 1), and the system only calls it once. The specific location to be called is in the `setup()` function of the initialization function `init()` in the `main.c` program. `setup()` is actually a system-call whose implementation code starts at line 74 of `/kernel/blk_drv/hd.c`.

## 12.6.2 Code annotation

Program 12-5 `linux/fs/super.c`

```

1 /*
2  * linux/fs/super.c

```

```

3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * super.c contains code to handle the super-block tables.
9  */
10 // <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk
11 //      type (HD_TYPE) options.
12 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
13 //      of the initial task 0, and some embedded assembly function macro statements about the
14 //      descriptor parameter settings and acquisition.
15 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
16 //      used functions of the kernel.
17 // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
18 //      descriptors/interrupt gates, etc. is defined.
19 // <errno.h> Error number header file. Contains various error numbers in the system.
20 // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
21 //      and constants.
22 #include <linux/config.h>
23 #include <linux/sched.h>
24 #include <linux/kernel.h>
25 #include <asm/system.h>
26
27 #include <errno.h>
28 #include <sys/stat.h>
29
30 // Perform a buffer cache synchronization with the data in the device (fs/buffer.c, line 59).
31 int sync_dev(int dev);
32 void wait_for_keypress(void);      // kernel/chr_drv/tty_io.c, line 140.
33
34 /* set_bit uses setb, as gas doesn't recognize setc */
35 // Tests the bit at the specified bit offset and returns the bit value.
36 // Similar to the inline assembly macro defined on line 19 of the bitmap.c program, but this
37 // macro only tests and returns the bit settings, without any changes to the bits (so the name
38 // test_bit() may be more appropriate).
39 // Input: %0 - ax(__res); %1 - eax(0); %2 - bitnr, bit offset value; %3 - (addr), starting address.
40 // Line 23 defines a local register variable '__res'. This variable will be saved in the eax
41 // register for efficient access and operation. The entire macro definition is a statement
42 // expression whose value is the value of the last '__res'. The BT instruction on line 24 is
43 // used to test the bit. It puts the value of the bit specified by the address addr (%3) and
44 // the bit offset bitnr (%2) into the carry flag CF. The SETB instruction is used to set the
45 // operand %al according to the carry flag CF. If CF = 1, then %al = 1, otherwise %al = 0.
46 #define set_bit(bitnr, addr) ({ \
47     register int __res __asm__("ax"); \
48     __asm__("bt %2,%3;setb %%al": "=a" (__res): "a" (0), "r" (bitnr), "m" (*(addr))); \
49     __res; })
50
51 struct super_block super_block[NR_SUPER];    // Super block table array (NR_SUPER = 8).
52 /* this is initialized in init/main.c */
53 int ROOT_DEV = 0;                          // Root file system device number.
54
55 // Lock the super block.

```



```

// If the superblock has been locked, the current task is placed in an uninterruptible wait
// state and added to the superblock wait queue s_wait until the superblock is unlocked and
// the task is explicitly woken up. Then lock it.
// The following three functions (lock_super(), free_super(), and wait_on_super()) have the
// same effect as the first three functions in the inode.c file, except that the object being
// manipulated here is replaced by a super block.
31 static void lock\_super(struct super\_block * sb)
32 {
33     cli(); // disable interrupt
34     while (sb->s_lock) // sleep if super block is already locked.
35         sleep\_on(&(sb->s_wait)); // kernel/sched.c, line 199.
36     sb->s_lock = 1; // lock it!
37     sti(); // enable interrupt
38 }
39
//// Unlock the specified super block.
// Resets the lock flag of the super block and explicitly wakes up all processes waiting on
// this wait queue s_wait (If we use the name unlock_super(), it might be more appropriate).
40 static void free\_super(struct super\_block * sb)
41 {
42     cli();
43     sb->s_lock = 0; // reset the lock.
44     wake\_up(&(sb->s_wait)); // kernel/sched.c, line 188.
45     sti();
46 }
47
//// Sleep waiting for the super block to unlock.
// If the superblock has been locked, the current task is placed in an uninterruptible wait
// state and added to the wait queue s_wait of the superblock. Until the super block unlocks
// and explicitly wakes up the task.
48 static void wait\_on\_super(struct super\_block * sb)
49 {
50     cli();
51     while (sb->s_lock) // sleep if super block is already locked.
52         sleep\_on(&(sb->s_wait));
53     sti();
54 }
55
//// Get super block of the specified device.
// Search for the super block structure of the specified device dev in the super block table
// (array). Returns a pointer to the superblock if found, otherwise returns a null pointer.
56 struct super\_block * get\_super(int dev)
57 {
58     struct super\_block * s; // superblock data structure pointer.
59
// First check the validity of the device given by the parameter. If the device number is 0,
// the null is returned. Then let s point to the beginning of the superblock array and start
// searching the entire superblock array to find the superblock of the specified device dev.
60     if (!dev)
61         return NULL;
62     s = 0+super\_block;
63     while (s < NR\_SUPER+super\_block)
// If the current search term is a superblock of the specified device, ie the device number

```

```

// field value of the superblock is the same as specified by the function parameter, then the
// superblock is awaited to be unlocked (if it has been locked by another process). During the
// waiting period, the super block item may be used by other devices, so it is necessary to
// check again after the sleep returns whether it is still the super block of the specified
// device. If it is, return the pointer of the super block, otherwise search the super block
// array again, so s needs to point to the beginning of the super block array again.
64         if (s->s_dev == dev) {
65             wait_on_super(s);
66             if (s->s_dev == dev)
67                 return s;
68             s = 0+super_block;
// If the current search term is not, check the next item. Finally, if the specified superblock
// is not found, a null pointer is returned.
69         } else
70             s++;
71     return NULL;
72 }
73
//// Release (put back) super block of the specified device.
// Releases the superblock array item used by the device (sets its s_dev=0) and releases the
// buffer cache blocks occupied by the device's inode bitmap and logic block bitmap. If the
// file system corresponding to the super block is the root file system, or another file system
// is already installed on one of its i nodes, the super block cannot be released.
74 void put_super(int dev)
75 {
76     struct super_block * sb;
77     int i;
78
// First check the validity and legality of the parameters. If the specified device is a root
// file system device, the warning message is displayed and returned. Then look for the file
// system superblock of the specified device in the superblock table. If the super block indicates
// that the i-node to which the file system is installed has not been processed, a warning message
// is displayed and returned. In the file system unmount operation, s_imount will be set to
// null before calling this function, see line 192.
79     if (dev == ROOT_DEV) {
80         printk("root diskette changed: prepare for armageddon\n\r");
81         return;
82     }
83     if (!(sb = get_super(dev)))
84         return;
85     if (sb->s_imount) {
86         printk("Mounted disk changed - tssk, tssk\n\r");
87         return;
88     }
// After finding the super block of the specified device, we first lock it, and then set its
// device number field s_dev to 0, that is, release the file system super block on the device.
// Then release the other kernel resources occupied by the super block, that is, release the
// buffer blocks occupied by the file system i-node bitmap and the logical block bitmap in the
// buffer cache. The following constant symbols I_MAP_SLOTS and Z_MAP_SLOTS are both equal to
// 8, which are used to indicate the number of disk logical blocks occupied by the i-node bitmap
// and the logical block bitmap, respectively. Note that if the contents of these buffer blocks
// have been modified, a synchronization operation is required to write the data in the buffer
// block to the device. The function finally unlocks the super block and returns.

```

```

89     lock_super(sb);
90     sb->s_dev = 0;                                // Set the super block to be idle.
91     for(i=0;i<I_MAP_SLOTS;i++)
92         brelse(sb->s_imap[i]);
93     for(i=0;i<Z_MAP_SLOTS;i++)
94         brelse(sb->s_zmap[i]);
95     free_super(sb);
96     return;
97 }
98
99     /// Read the super block of the specified device.
100    // If the file system superblock on the specified device dev is already in the superblock table,
101    // the pointer to the superblock entry is returned directly. Otherwise, the super block is read
102    // from the device dev into the buffer block and copied into the super block table. Finally
103    // return the super block pointer.
104    static struct super_block * read_super(int dev)
105    {
106        struct super_block * s;
107        struct buffer_head * bh;
108        int i, block;
109
110        // First check the validity of the parameter and then check if the device has replaced the disc
111        // (ie whether it is a floppy device). If the disk is replaced, all the buffer blocks in the
112        // buffer cache are invalid and need to be invalidated, that is, the original loaded file system
113        // is released.
114        if (!dev)
115            return NULL;
116        check_disk_change(dev);
117        // If the device's superblock is already in the superblock table, the pointer to the superblock
118        // is returned directly. Otherwise, find a free item in the super block array (that is, the
119        // item with the field s_dev=0). Returns a null pointer if the array is already full.
120        if (s = get_super(dev))
121            return s;
122        for (s = 0+super_block ;; s++) {
123            if (s >= NR_SUPER+super_block)
124                return NULL;
125            if (!s->s_dev)
126                break;
127        }
128        // After a free entry is found in the superblock array, it is used to specify the file system
129        // on device dev. The memory fields in the super block structure are then partially initialized.
130        s->s_dev = dev;
131        s->s_isup = NULL;
132        s->s_imount = NULL;
133        s->s_time = 0;
134        s->s_rd_only = 0;
135        s->s_dirt = 0;
136        // The superblock is then locked and its information is read from the device into the buffer
137        // block pointed to by bh. The super block is located in the second logical block (block 1)
138        // of the block device. If the read superblock operation fails, the item in the super block
139        // array selected above is released (ie, s_dev=0), and the item is unlocked, and a null pointer
140        // is returned. Otherwise, the read super block is copied from the buffer block data area to
141        // the corresponding item of the super block array, and the buffer block storing the read

```

```

// information is released.
122     lock\_super(s);
123     if (!(bh = bread(dev, 1))) {
124         s->s_dev=0;
125         free\_super(s);
126         return NULL;
127     }
128     *((struct d\_super\_block *) s) =
129         *((struct d\_super\_block *) bh->b_data);
130     brelse(bh);
// Now that we have obtained the superblock of the file system from device dev, we start checking
// the validity of this superblock and read the i-node bitmap and logical block bitmap from
// the device. If the file system magic number field of the read super block is incorrect, it
// means that the file system is not the correct one. Therefore, as in the above, the item in
// the super block array selected above is released, and the item is unlocked, and the empty
// pointer is returned. For this version of the Linux kernel, only the MINIX file system version
// 1.0 is supported, and the magic number is 0x137f.
131     if (s->s_magic != SUPER\_MAGIC) {
132         s->s_dev = 0;
133         free\_super(s);
134         return NULL;
135     }

// The following begins to read the i-node bitmap and logic block bitmap data on the device.
// First initialize the bitmap space in the memory superblock structure. Then, the i-node bitmap
// and the logical block bitmap information are read from the device and stored in the
// corresponding field of the super block. The i-node bitmap is stored in the logical block
// starting from block 2 on the device, occupying a total of s_imap_blocks blocks. The logic
// block bitmap occupies s_zmap_blocks blocks in subsequent blocks.
136     for (i=0; i<I\_MAP\_SLOTS; i++)                // initialize.
137         s->s_imap[i] = NULL;
138     for (i=0; i<Z\_MAP\_SLOTS; i++)
139         s->s_zmap[i] = NULL;
140     block=2;
141     for (i=0 ; i < s->s_imap_blocks ; i++)        // read i-node bitmap in the device.
142         if (s->s_imap[i]=bread(dev, block))
143             block++;
144     else
145         break;
146     for (i=0 ; i < s->s_zmap_blocks ; i++)        // Read logic block bitmap in the device.
147         if (s->s_zmap[i]=bread(dev, block))
148             block++;
149     else
150         break;
// If the number of bitmap blocks read is not equal to the number of logical blocks that the
// bitmap should occupy, it indicates that there is a problem with the file system bitmap, and
// the super block initialization fails. Therefore, all resources that were previously applied
// and occupied can be released, that is, the cache block occupied by the i-node bitmap and
// the logical block bitmap is released, the super block array item selected above is released,
// the super block item is unlocked, and a null pointer is returned. .
151     if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {
152         for(i=0; i<I\_MAP\_SLOTS; i++)                // release buffer blocks used by bitmap.
153             brelse(s->s_imap[i]);

```

```

154         for(i=0;i<Z_MAP_SLOTS;i++)
155             brelse(s->s_zmap[i]);
156         s->s_dev=0; // releases the selected super block item.
157         free_super(s); // unlock the super block item.
158         return NULL;
159     }
    // Otherwise everything is OK. In addition, for a function that requests an idle i-node, if
    // all i-nodes on the device have been used, the lookup function will return a value of zero.
    // Therefore, the 0th i-node is not available, so the lowest bit of the 1st block in the bitmap
    // is set to 1 to prevent the file system from assigning the 0th i node. For the same reason,
    // the lowest bit of the logic block bitmap is also set to 1. The last function unlocks the
    // superblock and returns the superblock pointer.
160     s->s_imap[0]->b_data[0] |= 1;
161     s->s_zmap[0]->b_data[0] |= 1;
162     free_super(s);
163     return s;
164 }
165
    /// Unmount file system system-call.
    // The parameter 'dev_name' is the file name of the device where the file system is located.
    // The function first obtains the device number based on the given block device file name, then
    // resets the corresponding field in the file system super block, releasing the buffer block
    // occupied by the super block and the bitmap. Finally, the buffer cache is synchronized with
    // the data on the device. Returns 0 if the unmount operation succeeds, otherwise returns an
    // error code.
166 int sys_umount(char * dev_name)
167 {
168     struct m_inode * inode;
169     struct super_block * sb;
170     int dev;
171
    // First find its i-node according to the device file name, and get the device number. The
    // device number defined by the device file is stored in i_zone[0] of its i-node. See line 445
    // of the system call sys_mknod() in the fs/namei.c program. In addition, since the file system
    // needs to be stored on the block device, if it is not a block device file, the i-node dev_i
    // just returned is put back, and the error code is returned.
172     if (!(inode=namei(dev_name)))
173         return -ENOENT;
174     dev = inode->i_zone[0];
175     if (!S_ISBLK(inode->i_mode)) {
176         iput(inode); // fs/inode.c, line 150.
177         return -ENOTBLK;
178     }
    // OK, now the i-node obtained to get the device number has completed its mission, so put back
    // the i-node here. Then we check if the conditions for unmounting the file system are met.
    // If the device is a root file system, it cannot be unmounted, and a busy error is returned.
    // If the super block of the file system on the device is not found in the super block table,
    // or the file system on the device is not installed, an error code is returned. If the i-node
    // to which it is mounted does not set its flag i_mount, a warning message is displayed. Then
    // look up the i-node table to see if any processes are using the files on the device, and if
    // so, return a busy error code.
179     iput(inode);
180     if (dev==ROOT_DEV)

```

```

181         return -EBUSY;
182     if (!(sb=get_super(dev)) || !(sb->s_imount))
183         return -ENOENT;
184     if (!sb->s_imount->i_mount)
185         printk("Mounted inode has i_mount=0\n");
186     for (inode=inode_table+0 ; inode<inode_table+NR_INODE ; inode++)
187         if (inode->i_dev==dev && inode->i_count)
188             return -EBUSY;
189     // The unmounting conditions for the file system on the device are now met, so we can start
190     // implementing the actual unmount operation. First reset the mount flag of the i-node to which
191     // it is mounted, release the i-node, then set the mounted i-node field in the superblock to
192     // be NULL, and put back the root i-node of the device file system. Then set the pointer of
193     // the root i node of the mounted file system in the super block to be NULL.
194     sb->s_imount->i_mount=0;
195     iput(sb->s_imount);
196     sb->s_imount = NULL;
197     iput(sb->s_isup);
198     sb->s_isup = NULL;
199     // Finally, we release the superblock on the device and the buffer block occupied by the bitmap,
200     // and perform a synchronization operation between the cache and the data on the device, and
201     // then return 0 (unmounted successfully).
202     put_super(dev);
203     sync_dev(dev);
204     return 0;
205 }
206
207 // Mount (new) file system system-call.
208 // The parameter dev_name is the device file name, dir_name is the directory name to be mounted,
209 // and rw_flag is the read/write flag of the mounted file system. The location to be mounted
210 // must be a directory name, and the corresponding i-node is not occupied by other programs.
211 // Returns 0 if successful, otherwise returns an error number.
212 int sys_mount(char * dev_name, char * dir_name, int rw_flag)
213 {
214     struct m_inode * dev_i, * dir_i;
215     struct super_block * sb;
216     int dev;
217
218     // First, find the corresponding i-node according to the device file name to get the device
219     // number. For block special device files, the device number is in i_zone[0] of its i-node.
220     // In addition, since the file system must be in the block device, if not, the i-node dev_i
221     // just obtained is put back and the error code is returned.
222     if (!(dev_i=namei(dev_name)))
223         return -ENOENT;
224     dev = dev_i->i_zone[0];
225     if (!S_ISBLK(dev_i->i_mode)) {
226         iput(dev_i);
227         return -EPERM;
228     }
229
230     // OK, now the device file i-node dev_i obtained in order to get the device number has completed
231     // its mission, so put it back here. Then let's check if the directory name to which the file
232     // system will be mounted on is valid. Then obtaining the corresponding i-node dir_i according
233     // to the given directory file name, and if the reference count of the i-node is not 1 (only
234     // referenced here), or the node number is 1 of the root file system, then putting back the

```

```

// i node and returns an error code. In addition, if the node is not a directory file node,
// the i-node is also put back, and the error code is returned because the new file system can
// only be mounted on a directory name.
212     iput(dev_i);
213     if (!(dir_i=namei(dir_name)))
214         return -ENOENT;
215     if (dir_i->i_count != 1 || dir_i->i_num == ROOT_INO) {
216         iput(dir_i);
217         return -EBUSY;
218     }
219     if (!S_ISDIR(dir_i->i_mode)) {           // mount point needs to be a directory entry.
220         iput(dir_i);
221         return -EPERM;
222     }
// Now that the mount point is checked, we start reading the super block of the new file system.
// A file system's superblock will first search from the superblock table, and read from the
// device if it is not in the superblock table. After getting the super block of the new file
// system, we check it first. We want to make sure that the new file system has not been mounted
// elsewhere, and that the i-node to be mounted is not occupied (by other file system), otherwise
// error occurred and returned.
223     if (!(sb=read_super(dev))) {
224         iput(dir_i);
225         return -EBUSY;
226     }
227     if (sb->s_imount) {                     // if new fs has been mounted elsewhere.
228         iput(dir_i);
229         return -EBUSY;
230     }
231     if (dir_i->i_mount) {                   // if the mount point is occupied.
232         iput(dir_i);
233         return -EPERM;
234     }
// Finally, we set the "mounted i-node" field 's_imount' of the new file system superblock to
// point to the i-node of the directory to which it is mounted, and set the mount flag and node
// modified flag of the mount location i-node, and then return 0 (installation is successful) .
235     sb->s_imount=dir_i;
236     dir_i->i_mount=1;
237     dir_i->i_dirt=1;                        /* NOTE! we don't iput(dir_i) */
238     return 0;                             /* we do that in umount */
239 }
240
///// Mount root file system.
// This function is part of the system initialization operation. It first initializes the file
// table file_table[] and the super block table, then reads the root file system super block
// and gets the root i node. Finally, the available resources (the number of free blocks and
// the number of idle i nodes) on the root file system are counted and displayed. This function
// is called at system initialization (sys_setup() in file blk_drv/hd.c, line 157).
241 void mount_root(void)
242 {
243     int i, free;
244     struct super_block * p;
245     struct m_inode * mi;
246

```

```

// First check if the size of the disk i-node structure meets the requirements (32 bytes) to
// prevent inconsistencies when modifying the code. Then initialize the file table (64 items)
// and the super block table (8 items). Here, the reference count in all file structures is
// set to 0 (indicating idle), and the device field of each structure in the superblock table
// is initialized to 0 (also indicating idle). If the device where the root file system is located
// is a floppy disk, it prompts "Insert root floppy and press ENTER" and wait for the key.
247     if (32 != sizeof (struct d_inode))
248         panic("bad i-node size");
249     for(i=0;i<NR_FILE;i++)                                // initialize file table (64 items).
250         file_table[i].f_count=0;
251     if (MAJOR(ROOT_DEV) == 2) {
252         printk("Insert root floppy and press ENTER");
253         wait_for_keypress();
254     }
255     for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
256         p->s_dev = 0;                                       // initialize super block table (8).
257         p->s_lock = 0;
258         p->s_wait = NULL;
259     }
// After doing the "extra" initialization work above, we started to mount the root file system.
// The file system superblock is then read from the root device and a pointer to the root i-node
// (node 1) of the file system in the memory i-node table is obtained. If the superblock on
// the device fails or the root node fails, the message is displayed and the device is down.
260     if (!(p=read_super(ROOT_DEV)))
261         panic("Unable to mount root");
262     if (!(mi=iget(ROOT_DEV,ROOT_INO)))                    // ROOT_INO is defined as 1 in fs.h.
263         panic("Unable to read root i-node");
// Now we setup the super block and the root i-node, and increment the reference number of the
// root i-node by 3 times. Because the i-node is also referenced in the next few lines of code,
// in addition, the i-node reference count in the iget() function has already been set to 1.
// Then, the mounted file system root i-node field (s_isup) and the i-node field (s_imount)
// of the super block are set to this i-node, and then the current working directory and the
// root i-node of the current process are set too. At this point, the current process is the
// number 1 process (init process).
264     mi->i_count += 3 ;    /* NOTE! it is logically used 4 times, not 1 */
265     p->s_isup = p->s_imount = mi;
266     current->pwd = mi;
267     current->root = mi;
// Then we count the resources on the root file system and count the number of free blocks and
// the number of idle i-nodes on the device. First, let 'i' be equal to the total number of
// logical blocks of the device indicated in the super block, and then count the number of free
// blocks according to the occupancy of the corresponding bits in the logical block bitmap.
// Here the macro function set_bit() is only used to test the bits. "i&8191" is used to obtain
// the corresponding bit offset value of the i-node number in the current bitmap block. "i>>13"
// divides i by 8192, which is the number of bits contained in a disk block.
268     free=0;
269     i=p->s_nzones;
270     while (-- i >= 0)
271         if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
272             free++;
// After displaying the number of free logical blocks on the device, we then count the number
// of free i-nodes on the device. First, let 'i' be equal to the total number of i-nodes on
// the device indicated in the super block +1 (add 1 to count the 0 nodes as well), and then

```



---

```

// calculate the number of idle i-nodes according to the occupancy of the corresponding bits
// in the i-node bitmap. Finally, the number of free i-nodes and the total number of i-nodes
// available on the device are displayed.
273     printk("%d/%d free blocks\n\r", free, p->s_nzones);
274     free=0;
275     i=p->s_ninodes+1;
276     while (-- i >= 0)
277         if (!set_bit(i&8191, p->s_imap[i>>13]->b_data))
278             free++;
279     printk("%d/%d free inodes\n\r", free, p->s_ninodes);
280 }
281

```

---

## 12.7 namei.c

### 12.7.1 Function

The namei.c program is the longest file in the Linux 0.12 kernel, but it only has about 900 lines of code :). This file mainly implements the function namei() which finds the corresponding i-node according to the directory name or file name, as well as some operation functions and system calls about the creation and deletion of the directory, the creation and deletion of the directory entries.

The Linux 0.12 system uses the MINIX file system version 1.0. Its directory entry structure is the same as that of a traditional UNIX file, and is defined in the include/linux/fs.h file. In a directory of the file system, the directory entries corresponding to all file names are stored in the data block of the directory file name. For example, the directory entry of all file names under the directory name home/ is stored in the data block of the home/ directory name file; and all files under the file system root directory are stored in the data block of the specified i-node (ie node 1). Each directory entry includes only a file name string of 14 bytes in length and a 2-byte i-node number corresponding to the file name, as shown below.

---

```

// Defined in the include/linux/fs.h file.
36 #define NAME_LEN 14 // file name maximum length.
37 #define ROOT_INO 1 // root i-node number.

// File directory entry structure.
157 struct dir_entry {
158     unsigned short inode; // i-node number.
159     char name[NAME_LEN]; // filename string
160 };

```

---

Other information about the file is saved in the i-node structure specified by the i-node number. The i-node structure mainly includes information such as file access attributes, host, length, access save time, and disk block. The i-node of each inode number is located at a fixed location on the disk.

When a file is opened, the file system finds its i-node number based on the given file name, thereby finding the disk block location where the file is located. For example, to find the i-node number of the file '/usr/bin/vi', the file system first starts from the root directory with a fixed i-node number (1). That is, from the data block of the i-node number 1, the directory entry with the file name 'usr' is found, thereby obtaining the i-node number of

the file '/usr'. According to the i-node number file system, the directory '/usr' can be successfully obtained, and the directory entry of the file name 'bin' can be found therein. This also knows the i-node number of '/usr/bin', so we can know the location of the directory '/usr/bin' and find the directory entry for the 'vi' file in that directory. Finally, we obtain the i-node number of the file path name '/usr/bin/vi', so that the i-node structure information of the i-node number can be obtained from the disk.

There are also two special file directory entries in each directory, whose names are fixed to '.' and '..' respectively. The '.' directory entry gives the i-node number of the current directory, and the '..' directory entry gives the i-node number of the direct parent directory of the current directory. Therefore, when a relative path name is given, the file system can use these two special directory entries for lookup operations. For example, to find './kernel/Makefile', you can first get the i-node number of the parent directory according to the '..' directory entry of the current directory, and then perform the lookup operation according to the process described above.

Since several main functions in the program have detailed comments in front of them, and the usages of each function and system-calls are clear, they will not be described here.

## 12.7.2 Code annotation

Program 12-6 linux/fs/namei.c

---

```

1  /*
2  *  linux/fs/namei.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  Some corrections by tytso.
9  */
10
    // <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/segment.h> Segment operation header file. An embedded assembly function is defined
    //      for segment register operations.
    // <string.h> String header file. Defines some embedded functions about string operations.
    // <fcntl.h> File control header file. The definition of the operation control constant symbol
    //      used for the file and its descriptors.
    // <errno.h> Error number header file. Contains various error numbers in the system.
    // <const.h> The constant symbol file currently defines the flags of i_mode field in i-node.
    // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
    //      and constants.
11 #include <linux/sched.h>
12 #include <linux/kernel.h>
13 #include <asm/segment.h>
14
15 #include <string.h>
16 #include <fcntl.h>
17 #include <errno.h>
18 #include <const.h>

```

```

19 #include <sys/stat.h>
20
21 // Internal function that finds the corresponding i-node by file name.
22 static struct m_inode * namei(const char * filename, struct m_inode * base,
23     int follow_links);
24
25 // The right expression in the macro below is a special way to access an array. It is based
26 // on the fact that the value of an array item (such as a[b]) represented by an array name and
27 // an array subscript is equivalent to the value of the array pointer (address) plus the offset
28 // address* (a + b), at the same time, it can be known that the term a[b] can also be expressed
29 // in the form of b[a]. So for the character array item form "LoveYou"[2] (or 2["LoveYou"])
30 // is equivalent to *("LoveYou" + 2). In addition, the position where the string "LoveYou" is
31 // stored in the memory is its address, so the value of the array item "LoveYou"[2] is the
32 // character "v" whose index value is 2 in the string, and its ASCII code value is 0x76, or
33 // 0166 in octal notation. In C, characters can also be represented by their ASCII value by
34 // adding a backslash to the ASCII value of the character. For example, the character "v" can
35 // be expressed as "\x76" or "\166". Therefore, characters that cannot be displayed (for example,
36 // control characters with an ASCII code value in range of 0x00--0x1f) can be represented by
37 // their ASCII code values.
38 //
39 // Below is the file access mode macro. Where 'x' is the file access (open) mode flag defined
40 // in line 7 of the header file include/fcntl.h. This macro indexes the corresponding value
41 // in the double quoted string based on the value of the file access token x. The double quotation
42 // marks have four octal control characters: "\004\002\006\377", which indicate the read, write,
43 // and execute permissions: r, w, rw, and wxrwxrwx, respectively, and correspond to the index
44 // value of x: 0--3. For example, if x is 2, the macro returns an octal value of 006, indicating
45 // readability and writability (rw). In addition, where 0_ACCMODE = 00003, is the mask of the
46 // index value x.
47 #define ACC MODE(x) ("004\002\006\377"[(x)&0_ACCMODE])
48
49 /*
50  * comment out this line if you want names > NAME_LEN chars to be
51  * truncated. Else they will be disallowed.
52  */
53 /* #define NO_TRUNCATE */
54
55 #define MAY EXEC 1
56 #define MAY WRITE 2
57 #define MAY READ 4
58
59 /*
60  * permission()
61  *
62  * is used to check for read/write/execute permissions on a file.
63  * I don't know if we should look at just the euid or both euid and
64  * uid, but that should be easily changed.
65  */
66
67 // Detect file access permissions.
68 // Parameters: inode - the i-node pointer of the file; mask - access attribute mask.
69 // Returns: access allowed returns 1, otherwise returns 0.
70 static int permission(struct m_inode * inode, int mask)
71 {
72     int mode = inode->i_mode; // File access mode.

```

```

46
47 /* special case: not even root can read/write a deleted file */
    // If the i-node has a corresponding device, but the link count is equal to 0, indicating that
    // the file has been deleted, then returns. Otherwise, if the effective user id (euid) of the
    // process is the same as the user id of the i-node, the access permission of the file owner
    // is taken. Otherwise, if the effective group id (egid) of the process is the same as the group
    // id of the i-node, the access rights of the group user is taken.
48     if (inode->i_dev && !inode->i_nlinks)
49         return 0;
50     else if (current->euid==inode->i_uid)
51         mode >>= 6;
52     else if (in_group_p(inode->i_gid))
53         mode >>= 3;
    // Finally, it is judged that if the access right is the same as the mask code, or is a super
    // user, it returns 1; otherwise, it returns 0.
54     if (((mode & mask & 0007) == mask) || suser())
55         return 1;
56     return 0;
57 }
58
59 /*
60 * ok, we cannot use strncmp, as the name is not in our data space.
61 * Thus we'll have to use match. No big problem. Match also makes
62 * some sanity tests.
63 *
64 * NOTE! unlike strncmp, match returns 1 for success, 0 for failure.
65 */
    ///// String matching and comparison function.
    // Parameters: len - the length of the string to compare; name - pointer to the file name;
    // de - the structure of the directory entry.
    // Returns: Returns 1 if they are the same, otherwise returns 0.
66 static int match(int len, const char * name, struct dir_entry * de)
67 {
68     register int same __asm__("ax"); // register variable.
69
    // First determine the validity of the function parameters. Returns 0 if the directory entry
    // pointer is empty, or if the directory entry i-node is null, or if the length to be compared
    // exceeds the file name length NAME_LEN. If the length of the comparison is equal to 0, but
    // the file name in the directory entry is '.', then we consider the same, so return 1 (match).
    // If the given length 'len' to be compared is less than NAME_LEN, but the file name length
    // exceeds the len, then 0 (no match) is also returned.
    // The method for judging whether the file name length exceeds 'len' on line 75 is to detect
    // whether name[len] is NULL. If the length exceeds len, then name[len] is a normal character
    // that is not NULL. For a string name of length len, the character name[len] should be NULL.
70     if (!de || !de->inode || len > NAME_LEN)
71         return 0;
72     /* "" means "." --> so paths like "/usr/lib/libc.a" work */
73     if (!len && (de->name[0]=='.' && (de->name[1]=='\0')))
74         return 1;
75     if (len < NAME_LEN && de->name[len]) // the length of the given name > len.
76         return 0;
    // Then use the embedded assembly statement for a quick comparison. It performs string
    // comparisons in the user data space (fs segment).

```

```

// %0 - eax (comparison result, same); %1 - eax (eax initial value 0); %2 - esi (name);
// %3 - edi (directory name pointer); %4 - ecx (compared bytes) len).
77     __asm__( "cld\n\t" // clear direction.
78             "fs ; repe ; cmpsb\n\t" // compare [esi++] and [edi++] in user space.
79             "setz %al" // set al = 1 if identical (same = eax).
80             : "=a" (same)
81             : "" (0), "S" ((long) name), "D" ((long) de->name), "c" (len)
82             : "cx", "di", "si");
83     return same; // return the result.
84 }
85
86 /*
87  * find_entry()
88  *
89  * finds an entry in the specified directory with the wanted name. It
90  * returns the cache buffer in which the entry was found, and the entry
91  * itself (as a parameter - res_dir). It does NOT read the inode of the
92  * entry - you'll have to do that yourself if you want to.
93  *
94  * This also takes care of the few special cases due to '..'-traversal
95  * over a pseudo-root and a mount point.
96  */
// Finds the directory entry for the specified file name in the specified directory.
// Parameters: *dir - directory i-node pointer; name - file name; namelen - filename length.
// This function searches the data (file) of the specified directory for the directory entry
// of the given file name, and performs special processing according to the current related
// settings for the case where the file name is '..'. See the comments before line 151 of
// linux/sched.c for the role of pointers to pointers in function arguments.
// Returns: the cache block pointer, if successful, and the directory entry pointer returned
// at *res_dir. Failure returns a null pointer.
97 static struct buffer_head * find_entry(struct m_inode ** dir,
98     const char * name, int namelen, struct dir_entry ** res_dir)
99 {
100     int entries;
101     int block, i;
102     struct buffer_head * bh;
103     struct dir_entry * de;
104     struct super_block * sb;
105
106     // Similarly, this function also needs to judge and verify the validity of the function
107     // parameters. If we define the symbol constant NO_TRUNCATE on line 30 above, then if the file
108     // name length exceeds the maximum length NAME_LEN, it will not be processed. If NO_TRUNCATE
109     // is not defined, it is truncated when the file name length exceeds the maximum length NAME_LEN.
110 #ifdef NO_TRUNCATE
111     if (namelen > NAME_LEN)
112         return NULL;
113 #else
114     if (namelen > NAME_LEN)
115         namelen = NAME_LEN;
116 #endif

117     // First calculate the number of directory entries in this directory. The i_size field of the
118     // i-node of the directory contains the size of the data contained in this directory, so it

```

```

// is divided by the size of a directory entry (16 bytes) to get the number of directory entries.
// Then null the pointer to the directory entry to be returned.
113     entries = (*dir)->i_size / (sizeof (struct dir\_entry));
114     *res_dir = NULL;

// Next, we specialize in the case where the directory entry file name is '..'. If the root
// i-node specified by the current process is the same directory given by the function parameter,
// it means that for this process, this directory is its pseudo root directory, that is, the
// process can only access the items in the directory and cannot fall back to its parent directory.
// That is, for this process, this directory is now the root directory of the file system. So
// we need to change the file name to '..'.
// Otherwise, if the i-node number of the directory is equal to ROOT_INO (number 1), it is indeed
// the root i-node of the file system, so the superblock of the file system is taken. If the
// i-node (s_imount) to which the file system is mounted in the superblock exists, the current
// operation is in the file system mounted on the s_imount node, then the original i-node is
// put back first, and then the i-node to be mounted is dealt with. So we let '*dir' point
// to the i-node to which it is mounted; and the number of references to this i-node is incremented
// by one. That is to say, in response to this situation, we quietly carried out the "stealing
// the column" project :)
115 /* check for '..', as we might have to do some "magic" for it */
116     if (namelen==2 && get\_fs\_byte(name)=='.' && get\_fs\_byte(name+1)=='.') {
117 /* '..' in a pseudo-root results in a faked '.' (just change namelen) */
118         if ((*dir) == current->root)
119             namelen=1;
120         else if ((*dir)->i_num == ROOT\_INO) {
121 /* '..' over a mount-point results in 'dir' being exchanged for the mounted
122 directory-inode. NOTE! We set mounted, so that we can input the new dir */
123             sb=get\_super((*dir)->i_dev);
124             if (sb->s_imount) { // mounted point.
125                 iput(*dir);
126                 (*dir)=sb->s_imount;
127                 (*dir)->i_count++;
128             }
129         }
130     }

// Now let's get started and find out where the directory entry for the specified filename is.
// Therefore, we need to read the data of the directory, that is, take out the data block (logical
// block) in the data zone of the block device corresponding to the directory i-node. The block
// numbers of these logical blocks are stored in the i_zone[] array of the i-node structure.
// We first take the first direct block number saved in it, and then read the specified directory
// item data block from the device.
131     if (!(block = (*dir)->i_zone[0]))
132         return NULL;
133     if (!(bh = bread((*dir)->i_dev, block)))
134         return NULL;

// At this point we can search for the directory entry matching the given file name in the read
// data block. First let 'de' point to the data block part of the buffer block, and loop the
// search without exceeding the number of directory entries in the directory. Where i is the
// index of the directory entry.
135     i = 0;
136     de = (struct dir\_entry *) bh->b_data;

```

```

137     while (i < entries) {
138         // If the current directory entry data block has been searched and no matching entry has been
139         // found, the current directory entry data block is released and then read into the next logical
140         // block of the directory. If the block is empty, then as long as all the entries in the directory
141         // have not been searched, the block is skipped and the next logical block of the directory
142         // is read. If the block is not empty, let de point to the data block and continue searching
143         // in it. On the 141th line, i/DIR_ENTRIES_PER_BLOCK can get the data block number in the
144         // directory file where the directory item currently searched, and the bmap() function (inode.c,
145         // line 142) can calculate the corresponding logical block number on the device.
146         if ((char *)de >= BLOCK_SIZE+bh->b_data) {
147             brelse(bh);
148             bh = NULL;
149             if (!(block = bmap(*dir, i/DIR_ENTRIES_PER_BLOCK)) ||
150                 !(bh = bread((*dir)->i_dev, block))) {
151                 i += DIR_ENTRIES_PER_BLOCK;
152                 continue;
153             }
154             de = (struct dir_entry *) bh->b_data;
155         }
156         // If a matching directory entry is found, the directory entry structure pointer 'de' and the
157         // directory entry i-node pointer '*dir' and the directory entry data block pointer 'bh' are
158         // returned, and the function is exited. Otherwise continue to compare the next directory entry
159         // in the directory entry data block.
160         if (match(namelen, name, de)) {
161             *res_dir = de;
162             return bh;
163         }
164         de++;
165         i++;
166     }
167     // If all the directory entries in the specified directory have been searched and the
168     // corresponding directory entry has not been found, the data block of the directory is released,
169     // and finally NULL is returned (failed).
170     brelse(bh);
171     return NULL;
172 }
173
174 /*
175  *      add_entry()
176  *
177  * adds a file entry to the specified directory, using the same
178  * semantics as find_entry(). It returns NULL if it failed.
179  *
180  * NOTE!! The inode part of 'de' is left at 0 - which means you
181  * may not sleep between calling this and putting something into
182  * the entry, as someone else might have used it while you slept.
183  */
184
185 // Adds a file entry to the specified directory.
186 // Parameters: dir - i-node of the directory; name - file name; namelen - file name length.
187 // Returns: buffer block pointer; res_dir - the pointer to the returned directory entry.
188 static struct buffer_head * add_entry(struct m_inode * dir,
189     const char * name, int namelen, struct dir_entry ** res_dir)
190 {

```

```

172     int block, i;
173     struct buffer head * bh;
174     struct dir\_entry * de;
175
    // Similarly, this function also needs to judge and verify the validity of the function
    // parameters. If we define the symbol constant NO_TRUNCATE on line 30 above, then if the file
    // name length exceeds the maximum length NAME_LEN, it will not be processed. If NO_TRUNCATE
    // is not defined, it is truncated when the file name length exceeds the maximum length NAME_LEN.
176     *res_dir = NULL; // the result directory entry pointer.
177 #ifdef NO_TRUNCATE
178     if (namelen > NAME\_LEN)
179         return NULL;
180 #else
181     if (namelen > NAME\_LEN)
182         namelen = NAME\_LEN;
183 #endif
    // Now let's get started and add a directory entry with the given filename to the specified
    // directory. Therefore, we need to read the data of the directory, that is, take out the data
    // block (logical block) in the data zone of the block device corresponding to the directory
    // i-node. The block numbers of these logical blocks are stored in the i_zone[] array of the
    // i-node structure. We first take the first direct block number saved in it, and then read
    // the specified directory item data block from the device. In addition, if the file name length
    // provided by the parameter is equal to 0, it returns with a NULL.
184     if (!namelen)
185         return NULL;
186     if (!(block = dir->i_zone[0]))
187         return NULL;
188     if (!(bh = bread(dir->i_dev, block)))
189         return NULL;
    // At this point, we loop through the data blocks of the i-node of this directory to find the
    // last unused empty directory entry. First, let the directory entry pointer point to the data
    // block part of the buffer block, that is, the first directory entry. Where i is the index
    // of the directory entry in the directory.
190     i = 0;
191     de = (struct dir\_entry *) bh->b_data;
192     while (1) {
    // If the current directory entry data block has been searched but the required empty directory
    // entry has not been found, the current directory entry data block is released and then the
    // next logical block of the directory is read. Create a block if the corresponding logical
    // block does not exist. Returns null if the read or create operation fails. If the buffer block
    // pointer returned by the disk logical block data read this time is empty, it indicates that
    // the logical block may be a newly created empty block because it does not exist. So we add
    // the directory entry index value to the number of directory entries DIR_ENTRIES_PER_BLOCK
    // that a logical block can hold, to skip the block and continue searching. Otherwise, the newly
    // read block has directory entry data, so the directory entry structure pointer points to the
    // buffer block data portion of the block, and then continues the search there.
    // The i/DIR_ENTRIES_PER_BLOCK on line 196 can be used to calculate the block number in the
    // directory file where the currently searched directory entry i is located, and the function
    // create_block() (inode.c, line 147) can be used to read or created logic block on the device.
193     if ((char *)de >= BLOCK\_SIZE+bh->b_data) {
194         brelse(bh);
195         bh = NULL;
196         block = create\_block(dir, i/DIR\_ENTRIES\_PER\_BLOCK);

```



```

197         if (!block)
198             return NULL;
199         if (!(bh = bread(dir->i_dev, block))) {    // skip if empty.
200             i += DIR\_ENTRIES\_PER\_BLOCK;
201             continue;
202         }
203         de = (struct dir\_entry *) bh->b_data;
204     }
    // If the size of the directory entry number i currently multiplied by the directory entry size
    // exceeds the directory data size i_size indicated by the i-node of the directory, it means
    // that there is no empty entries left due to the deletion of file. So we can only append new
    // directory entry that need to be added to the end of the directory data file. Therefore, we
    // need to set the directory entry for this directory, set the i-node number of the directory
    // entry to be empty, and update the size of the directory file (plus the length of a directory
    // entry), and then set the i-node of the directory to be modified, and then update the change
    // time of this directory to the current time.
205     if (i*sizeof(struct dir\_entry) >= dir->i_size) {
206         de->inode=0;
207         dir->i_size = (i+1)*sizeof(struct dir\_entry);
208         dir->i_dirt = 1;
209         dir->i_ctime = CURRENT\_TIME;
210     }
    // If the i-node number of the currently searched directory entry 'de' is empty, it means that
    // a free directory entry that has not been used or a new directory entry added is found.
    // Therefore, we update the directory modification time to the current time, and copy the file
    // name from the user data space to the file name field of the directory entry, and set the
    // corresponding cache buffer block modified flag of the directory entry. Returns with a pointer
    // to the directory entry and a pointer to the cache buffer block.
211     if (!de->inode) {
212         dir->i_mtime = CURRENT\_TIME;
213         for (i=0; i < NAME\_LEN ; i++)
214             de->name[i]=(i<namelen)?get\_fs\_byte(name+i):0;
215         bh->b_dirt = 1;
216         *res_dir = de;
217         return bh;
218     }
219     de++;          // if entry is in use, continue to check the next entry.
220     i++;
221 }
    // This function cannot be executed here. This may be because Mr. Linus copied the code of the
    // find_entry() function above and then modified it into this function :).
222     brelse(bh);
223     return NULL;
224 }
225
226 static struct m\_inode * follow\_link(struct m\_inode * dir, struct m\_inode * inode)
227 {
228     unsigned short fs;          // Used to temporarily save FS segment register.
229     struct buffer\_head * bh;
230

```

```

// First determine the validity of the function parameters. If the directory i-node is not given,
// we use the root i-node set in the process task structure and increase the number of links
// by one. If the directory entry i-node is not given, the directory i-node is put back and
// NULL is returned. If the specified directory entry is not a symbolic link, we directly return
// the i-node corresponding to the directory entry.
231     if (!dir) {
232         dir = current->root;
233         dir->i_count++;
234     }
235     if (!inode) {
236         iput(dir);
237         return NULL;
238     }
239     if (!S\_ISLNK(inode->i_mode)) {
240         iput(dir);
241         return inode;
242     }
// Then get the FS segment register value. The FS usually holds the selector 0x17 pointing to
// the task (user) data segment. If FS does not point to the user data segment, or if the first
// direct block number of the given directory entry i-node is equal to 0, or an error occurs
// when reading the first direct block, put back the dir and inode nodes and return NULL.
// Otherwise, the FS is now pointing to the user data segment, and we have successfully read
// the file contents of this symbolic link directory entry, and the file content is already
// in the buffer block data area pointed to by bh. In fact, this buffer block data area contains
// only one file path name string pointed to by the link.
243     __asm__ ("mov %%fs, %0": "=r" (fs)); // get FS contents.
244     if (fs != 0x17 || !inode->i_zone[0] ||
245         !(bh = bread(inode->i_dev, inode->i_zone[0]))) {
246         iput(dir);
247         iput(inode);
248         return NULL;
249     }
// At this point we can use the data content of the symbolic link file to find the i-node of
// the file to which it is linked. Now we no longer need the i-node information of the symbolic
// link directory entry, so we put it back. Now we have a problem, that is, the user data processed
// by the kernel function should be stored in the user data space by default, and use the FS
// segment register to transfer data from the user space to the kernel space. But the data that
// needs to be processed here is in kernel space. Therefore, in order to correctly process the
// user data located in the kernel space, we need to temporarily point the FS segment register
// to the kernel space, that is, let FS = 0x10, and restore the original FS after the function
// namei() is called. Finally, the buffer block is released, and the file i-node pointed to
// by the symbol link obtained by _namei() is returned.
250     iput(inode);
251     __asm__ ("mov %0, %%fs": "=r" ((unsigned short) 0x10));
252     inode = \_namei(bh->b_data, dir, 0);
253     __asm__ ("mov %0, %%fs": "=r" (fs));
254     brelse(bh);
255     return inode;
256 }
257
258 /*
259 *   get\_dir()
260 *

```

```

261 * Getdir traverses the pathname until it hits the topmost directory.
262 * It returns NULL on failure.
263 */
    //// Search for the i-node of the topmost directory of the given pathname.
    // Parameters: pathname - the path name; inode - the i-node specifying the starting directory.
    // Returns: the i-node pointer of the directory, or NULL if it fails.
264 static struct m\_inode * get\_dir(const char * pathname, struct m\_inode * inode)
265 {
266     char c;
267     const char * thisname;
268     struct buffer head * bh;
269     int namelen, inr;
270     struct dir\_entry * de;
271     struct m\_inode * dir;
272
    // First determine the validity of the parameters. If the i-node pointer inode of the given
    // directory is empty, the current working directory i-node of the current process is used.
    // If the user specifies that the first character of the path name is '/', the path name is
    // an absolute path name, and then we should start from the root (or pseudo root) i-node set
    // in the current process task structure. So we need to put back the directory i-node specified
    // or set by the parameter and get the root i-node used by the process, then increment the
    // reference count of the i-node and delete the first character '/' of the path name. This ensures
    // that the current process can only use the root i-node it sets as the starting point for the
    // search.
273     if (!inode) {
274         inode = current->pwd;                // i-node of the working directory.
275         inode->i_count++;
276     }
277     if ((c=get\_fs\_byte(pathname))== '/') {
278         iput(inode);                        // put back the original i-node.
279         inode = current->root;            // root i-node specified for the process.
280         pathname++;
281         inode->i_count++;
282     }
    // Then loop through the various directory names and file name in the path name. In the loop
    // processing, we first determine the validity of the i-node of the directory name currently
    // being processed, and point the variable 'thisname' to the part of the directory name currently
    // being processed. If the i-node indicates that the currently processed directory name portion
    // is not a directory type, or there is no access permission to enter the directory, then the
    // i-node is put back and a NULL is returned. Of course, when entering the loop, the i-node
    // 'inode' of the current directory is the process root i-node or the i-node of the current
    // working directory, or the i-node of a search starting directory specified by the parameter.
283     while (1) {
284         thisname = pathname;
285         if (!S\_ISDIR(inode->i_mode) || !permission(inode, MAY\_EXEC)) {
286             iput(inode);
287             return NULL;
288         }
    // In each loop we process one directory name in the pathname. So in each loop we have to separate
    // a directory name from the pathname string. The method is to search for the detected character
    // from the current pathname pointer 'pathname' until the character is a trailing character
    // (NULL) or a '/' character. At this point the variable 'namelen' is exactly the length of
    // the currently processed directory name part, and the variable 'thisname' is pointing to the

```

```

// beginning of the directory name part. At this time, if the character is the end character
// NULL, it indicates that the end of the path name has been searched, and the last specified
// directory name or file name has been reached, then the i-node pointer is returned.
// NOTE: If the last name in the path name is also a directory name, but the '/' character is
// not appended to it, the function will not return the i-node of the last directory name!
// For example: For the path name '/usr/src/linux', this function will only return the i-node
// of the 'src/' directory name.
289         for(namelen=0; (c=get_fs_byte(pathname++))&&(c!='/'); namelen++)
290             /* nothing */;
291         if (!c)
292             return inode;
// After getting the current directory name part (or file name), we call the lookup directory
// entry function find_entry() to find the directory entry with the specified name in the
// currently processed directory. (If not found, put back the i-node and return NULL.) Then,
// the i-node number 'inr' and the device number 'idev' are taken out in the found directory
// entry, the cache block containing the directory entry is released, and the i-node is put
// back. Then take the i-node 'inode' of the node number 'inr', and continue to loop the next
// directory name part in the path name with the directory entry as the current directory. If
// the currently processed directory entry is a symbolic link name, then follow_link() is used
// to get the i-node of the directory entry name it points to.
293         if (!(bh = find_entry(&inode, thisname, namelen, &de))) {
294             iput(inode);
295             return NULL;
296         }
297         inr = de->inode;          // i-node number of the current directory name part.
298         brelse(bh);
299         dir = inode;
300         if (!(inode = iget(dir->i_dev, inr))) {          // get i-node content.
301             iput(dir);
302             return NULL;
303         }
304         if (!(inode = follow_link(dir, inode)))
305             return NULL;
306     }
307 }
308
309 /*
310  *      dir_namei()
311  *
312  * dir_namei() returns the inode of the directory of the
313  * specified name, and the name within that directory.
314  */
//// Gets the i-node of the specified directory name, and the top-level directory name.
// Parameters: pathname - the directory path name; namelen - the path name length; name - the
// topmost directory name returned; base - the i-node of the directory to start searching.
// Returns: the i-node and name and length of the top-level directory of the specified directory
// name. Returns NULL on error.
315 static struct m_inode * dir_namei(const char * pathname,
316     int * namelen, const char ** name, struct m_inode * base)
317 {
318     char c;
319     const char * basename;
320     struct m_inode * dir;

```

```

321 // First, get the i-node of the top-level directory of the specified path name, then search
// and detect the pathname, find the name string after the last '/' character, calculate its
// length, and return the i-node pointer of the top-level directory. Note that if the last
// character of the path name is the slash character '/', then the returned directory name is
// empty and the length is 0. However, the returned i-node pointer still points to the i-node
// of the directory name before the last '/' character.
322     if (!(dir = get\_dir(pathname, base))) // base is the i-node of the starting dir.
323         return NULL;
324     basename = pathname;
325     while (c=get\_fs\_byte(pathname++))
326         if (c=='/')
327             basename=pathname;
328     *namelen = pathname-basename-1;
329     *name = basename;
330     return dir;
331 }
332
333 // Get the i-node of the specified pathname (internal function).
334 // Parameters: pathname - path name; base - search start directory i-node;
335 // follow_links - whether to follow the symbolic link, 1 - yes, 0 - no.
336 // Returns: the corresponding i-node.
337 struct m\_inode * namei(const char * pathname, struct m\_inode * base,
338     int follow_links)
339 {
340     const char * basename;
341     int inr, namelen;
342     struct m\_inode * inode;
343     struct buffer\_head * bh;
344     struct dir\_entry * de;
345
346     // First find the directory name of the topmost directory in the specified path name and get
347     // its i-node. If it does not exist, it returns NULL and exits. If the length of the topmost
348     // name returned is 0, it means that the path name is named after the last item in a directory.
349     // Therefore, we have found the i-node of the corresponding directory, so we can return the
350     // i-node directly. If the length of the returned name is not 0, then we call the dir\_namei()
351     // function again to search for the top-level directory name with the newly specified start
352     // directory base, and make a similar judgment based on the returned information.
353     if (!(dir = dir\_namei(pathname, &namelen, &basename)))
354         return NULL;
355     if (!namelen) // special case: '/usr/' etc */
356         return dir;
357     if (!(base = dir\_namei(pathname, &namelen, &basename, base)))
358         return NULL;
359     if (!namelen) // special case: '/usr/' etc */
360         return base;
361
362     // Then look for the i-node of the specified file name directory entry in the returned top-level
363     // directory. Please pay attention again! If it is also a directory name at the end, but there
364     // is no '/' added after it, it will not return the i-node of the last directory! For example:
365     // '/usr/src/linux' will only return the i-node of the 'src/' directory name. Since the function
366     // dir\_namei() treats the last name that does not end with '/' as a file name, it is necessary
367     // to use the find directory entry i-node function find\_entry() to handle this situation
368     // separately. At this point, 'de' contains the directory entry pointer found, and 'dir' is

```

```

// the i-node pointer of the directory containing the directory entry.
346     bh = find\_entry(&base, basename, namelen, &de);
347     if (!bh) {
348         iput(base);
349         return NULL;
350     }
// Then take the i-node number of the directory entry, release the cache buffer block containing
// the directory entry, and put back the i-node of the directory. Then take the i-node
// corresponding to the node number, modify the access time to be the current time, and set
// the modified flag. Finally, the i-node pointer inode is returned. In addition, if the currently
// processed directory entry is a symbolic link name, then follow_link() is used to get the
// i-node of the directory entry it points to.
351     inr = de->inode;
352     brelse(bh);
353     if (!(inode = iget(base->i_dev, inr))) {
354         iput(base);
355         return NULL;
356     }
357     if (follow_links)
358         inode = follow\_link(base, inode);
359     else
360         iput(base);
361     inode->i_atime=CURRENT\_TIME;
362     inode->i_dirt=1;
363     return inode;
364 }
365
////// Take the i-node of the specified path name without following the symbolic link.
// Parameters: pathname - the path name.
// Returns: the corresponding i-node.
366 struct m\_inode * lnamei(const char * pathname)
367 {
368     return \_namei(pathname, NULL, 0);
369 }
370
371 /*
372  *      namei ()
373  *
374  * is used by most simple commands to get the inode of a specified name.
375  * Open, link etc use their own routines, but this is enough for things
376  * like 'chmod' etc.
377  */
////// Take the i-node of the specified path name and follow the symbolic link.
// Parameters: pathname - the path name.
// Returns: the corresponding i-node.
378 struct m\_inode * namei(const char * pathname)
379 {
380     return \_namei(pathname, NULL, 1);
381 }
382
383 /*
384  *      open_namei ()
385  *

```

```

386 * namei for open - this is in fact almost the whole open-routine.
387 */
    /// The namei function used to open the file.
    // Parameters: filename - is the file path name; flag - is the open file flag, which can take
    // values O_RDONLY (read only), O_WRONLY (write only) or O_RDWR (read and write), and O_CREAT
    // (create), O_EXCL (file must not exist), O_APPEND (add at the end of the file) and other
    // combinations of other flags; Mode - the permission attribute of the specified file when
    // creating a new file. These attributes are S_IRWXU (the file owner has read, write, and execute
    // permissions), S_IRUSR (user has read file permissions), S_IRWXG (group members have read,
    // write, and execute permissions), and so on. For newly created files, these attributes are
    // only used for future access to the file, and open calls that create read-only files will
    // also return a readable and writable file handle. See the include files sys/stat.h, fcntl.h.
    // Returns: 0 is returned successfully, otherwise the error code is returned; res_inode - returns
    // the i-node pointer of the corresponding file path name.
388 int open_namei(const char * pathname, int flag, int mode,
389                struct m_inode ** res_inode)
390 {
391     const char * basename;
392     int inr, dev, namelen;
393     struct m_inode * dir, *inode;
394     struct buffer head * bh;
395     struct dir entry * de;
396
    // First, the function parameters are handled reasonably. If the file access mode flag is
    // read-only (O), but the file truncate flag O_TRUNC is set, the write-only flag O_WRONLY is
    // added to the file open flag. The reason for this is because the truncate flag O_TRUNC is
    // valid only if the file is writable. Then use the file access permission mask of the current
    // process, mask the corresponding bit in the given mode, and add the normal file flag I_REGULAR.
    // This flag will be used as the default property for new files when the open file does not
    // exist and needs to be created. See the note on line 411 below.
397     if ((flag & O_TRUNC) && !(flag & O_ACCMODE))
398         flag |= O_WRONLY;
399     mode &= 0777 & ~current->umask;
400     mode |= I_REGULAR;           // regular file flag. See file include/const.h.

    // Then find the corresponding i-node according to the specified path name, and the top directory
    // name and its length. At this time, if the top directory name length is 0 (for example, '/usr/'),
    // then if the operation is not read/write, creation, and file truncation 0, it means that a
    // directory name file operation is open. So directly return the i-node of the directory, and
    // return 0 to exit. Otherwise, the process is illegal, so the i-node is put back and the error
    // code is returned.
401     if (!(dir = dir_namei(pathname, &namelen, &basename, NULL)))
402         return -ENOENT;
403     if (!namelen) {                /* special case: '/usr/' etc */
404         if (!(flag & (O_ACCMODE | O_CREAT | O_TRUNC))) {
405             *res_inode = dir;
406             return 0;
407         }
408         iput(dir);
409         return -EISDIR;
410     }

    // Then, according to the i-node of the top-level directory name obtained above, the directory
    // entry structure corresponding to the last file name in the path name is searched, and the

```

```

// cache block of the directory entry is obtained at the same time. If the cache block pointer
// is NULL, it means that the directory entry corresponding to the file name is not found, so
// it is only possible to create a file. At this time, if the operation is not to create a file,
// or the user does not have the right to write in the directory, put back the i-node of the
// directory and return the corresponding error code to exit.
411     bh = find\_entry(&dir, basename, namelen, &de);
412     if (!bh) {
413         if (!(flag & O\_CREAT)) {
414             iput(dir);
415             return -ENOENT;
416         }
417         if (!permission(dir, MAY\_WRITE)) {
418             iput(dir);
419             return -EACCES;
420         }
// Now we have determined that it is a create operation and has a write permission. Therefore,
// we apply for a new i-node on the device to use the file name specified on the path name,
// and make initial settings for the new i-node: set the user id; access mode; set the modified
// flag. Then add a new directory entry in the specified directory dir.
421         inode = new\_inode(dir->i_dev);
422         if (!inode) {
423             iput(dir);
424             return -ENOSPC;
425         }
426         inode->i_uid = current->euid;
427         inode->i_mode = mode;
428         inode->i_dirt = 1;
429         bh = add\_entry(dir, basename, namelen, &de);
// If the returned cache block pointer that should contain a new directory entry is NULL, it
// means the add directory entry operation failed. Then the error rollback process is executed:
// the reference connection count of the new i-node is decremented by 1, the i-nodes are put
// back, and the error code exit is returned. Otherwise, the operation of adding a directory
// entry is successful. So we set some initial values of the new directory entry: set the i-node
// number to the number of the newly applied i-node; set the cache block modified flag. Then
// release the cache block and the i-node of the directory is put back. Then the i-node pointer
// of the new directory entry is returned and exits successfully.
430         if (!bh) {
431             inode->i_nlinks--;
432             iput(inode);
433             iput(dir);
434             return -ENOSPC;
435         }
436         de->inode = inode->i_num;
437         bh->b_dirt = 1;
438         brelse(bh);
439         iput(dir);
440         *res_inode = inode;
441         return 0;
442     }
// If the above (line 411) succeeds in obtaining the directory entry corresponding to the file
// name (ie, bh is not NULL), this indicates that the specified file already exists. Then, the
// i-node number of the directory entry and the device number are taken, and the cache block
// and the i-node of the directory are released. If the exclusive operation flag O_EXCL is set,

```



```

// but the file already exists, then the error code (file exists) is returned and exits.
443     inr = de->inode;
444     dev = dir->i_dev;
445     brelse(bh);
446     if (flag & O_EXCL) {
447         iput(dir);
448         return -EEXIST;
449     }
// Then we read the i-node content of the directory entry. If the i-node is an i-node of a directory
// and the access mode is write-only or read-write, or has no access permission, then put back
// the i-node and return the access permission error code.
450     if (!(inode = follow_link(dir, iget(dev, inr))))
451         return -EACCES;
452     if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
453         !permission(inode, ACC_MODE(flag))) {
454         iput(inode);
455         return -EPERM;
456     }
// Then we update the access time field of the i-node to the current time. If the truncate flag
// is set, the file size is truncated to zero. Finally returns a pointer to the i-node of the
// directory entry and returns 0 (success).
457     inode->i_atime = CURRENT_TIME;
458     if (flag & O_TRUNC)
459         truncate(inode);
460     *res_inode = inode;
461     return 0;
462 }
463
///// Create a device special file or a regular file node.
// This function creates a file system node (common file, device special file, or named pipe)
// named filename, specified by mode and dev.
// Parameters: filename - the pathname of filename; mode - specifies the permission to use and
// the type of node created; dev - the device number.
// Returns: 0 if successful, otherwise returns an error code.
464 int sys_mknod(const char * filename, int mode, int dev)
465 {
466     const char * basename;
467     int namelen;
468     struct m_inode * dir, * inode;
469     struct buffer_head * bh;
470     struct dir_entry * de;
471
// First check the validity of the operation permissions and parameters and get the i-node of
// the top-level directory in the path name. If it is not a superuser, return the access permission
// error code. If the i-node corresponding to the top-level directory in the path name is not
// found, an error code is returned. If the top file name length is 0, it means that the given
// path name does not specify a file name at the end or the process has no write permission
// in the directory, then we put back the i-node of the directory and return an error code.
472     if (!suser())
473         return -EPERM;
474     if (!(dir = dir_namei(filename, &namelen, &basename, NULL)))
475         return -ENOENT;
476     if (!namelen) {

```

```

477         iput(dir);
478         return -ENOENT;
479     }
480     if (!permission(dir, MAY\_WRITE)) {
481         iput(dir);
482         return -EPERM;
483     }
484     // Then we search for the file specified by the path name already exists, if it already exists,
485     // you can not create a file node with the same name. Therefore, if the directory entry of the
486     // last file name corresponding to the path name already exists, the buffer block containing
487     // the directory entry is released and the i-node of the directory is put back, and the error
488     // code that the file already exists is returned.
489     bh = find\_entry(&dir, basename, namelen, &de);
490     if (bh) {
491         brelse(bh);
492         iput(dir);
493         return -EEXIST;
494     }
495     // If the directory entry for the file name is not found, then we apply for a new i-node and
496     // set the attribute mode of the i-node. If you are creating a block device file or a character
497     // device file, make the direct logical block pointer 0 of the i-node equal to the device number.
498     // That is, for the device file, the i_zone[0] of the i-node stores the device number of the
499     // device defined by the device file. Then set the modification time and access time of the
500     // i-node to the current time, and set the i-node modified flag.
501     inode = new\_inode(dir->i_dev);
502     if (!inode) { // put back inode and return error code if failed.
503         iput(dir);
504         return -ENOSPC;
505     }
506     inode->i_mode = mode;
507     if (S\_ISBLK(mode) || S\_ISCHR(mode))
508         inode->i_zone[0] = dev;
509     inode->i_mtime = inode->i_atime = CURRENT\_TIME;
510     inode->i_dirt = 1;
511     // Then add a new directory entry for this new i-node. If it fails (the cache block pointer
512     // containing the directory entry is NULL), put back the i-node of the directory; reset the
513     // requested i-node reference connection count, put back the i-node, and return the error code.
514     bh = add\_entry(dir, basename, namelen, &de);
515     if (!bh) {
516         iput(dir);
517         inode->i_nlinks=0;
518         iput(inode);
519         return -ENOSPC;
520     }
521     // Now adding the directory entry operation is also successful, so we set the contents of this
522     // directory item. Let the i-node field of the directory entry be equal to the new i-node number,
523     // and set the cache block modified flag, put back the directory and the new i-node, release
524     // the cache block, and finally return 0 (success).
525     de->inode = inode->i_num;
526     bh->b_dirt = 1;
527     iput(dir);
528     iput(inode);
529     brelse(bh);

```

```

512         return 0;
513     }
514     ///// Make a directory (used in system-call).
515     // Parameters: pathname - the path name; mode - permission attribute used by the directory.
516     // Returns: 0 if successful, otherwise returns an error code.
517     int sys_mkdir(const char * pathname, int mode)
518     {
519         const char * basename;
520         int namelen;
521         struct m_inode * dir, * inode;
522         struct buffer_head * bh, *dir_block;
523         struct dir_entry * de;
524
525         // First check the validity of the operation permissions and parameters and get the i-node of
526         // the top-level directory in the path name. If the i-node corresponding to the top-level
527         // directory in the path name is not found, an error code is returned. If the top file name
528         // length is 0, it means that the given path name does not specify a file name at the end or
529         // the process has no write permission in the directory, then we put back the i-node of the
530         // directory and return an error code.
531         if (!(dir = dir_namei(pathname, &namelen, &basename, NULL)))
532             return -ENOENT;
533         if (!namelen) {
534             iput(dir);
535             return -ENOENT;
536         }
537         if (!permission(dir, MAY_WRITE)) {
538             iput(dir);
539             return -EPERM;
540         }
541
542         // Then we search for the directory name specified by the path name already exists, if it already
543         // exists, we can not create a directory node with the same name. Therefore, if the directory
544         // entry of the last directory name on the path name already exists, the buffer block containing
545         // the directory entry is released and the i-node of the directory is put back, and the error
546         // code that the file already exists is returned. Otherwise, we apply for a new i-node and set
547         // the attribute mode of the i-node: set the file size corresponding to the new i-node to 32
548         // bytes (the size of two directory entries), set the modified flag of the node, and the node's
549         // modify time and access time. Two directory entries are used for the '.' and '..' directories
550         // respectively.
551         bh = find_entry(&dir, basename, namelen, &de);
552         if (bh) {
553             brelse(bh);
554             iput(dir);
555             return -EEXIST;
556         }
557         inode = new_inode(dir->i_dev);
558         if (!inode) { // put back inode and return error code if failed.
559             iput(dir);
560             return -ENOSPC;
561         }
562         inode->i_size = 32;
563         inode->i_dirt = 1;
564         inode->i_mtime = inode->i_atime = CURRENT_TIME;

```

```

// Next, we apply for a disk block for storing the directory entry data for the new i-node,
// and let the first direct block pointer of the i-node be equal to the block number. If the
// application fails, put back the i-node of the directory; reset the newly requested i-node
// link count; put back the new i-node, and return no space error code. Otherwise, set the new
// i-node modified flag.
547     if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
548         iput(dir);
549         inode->i_nlinks--;
550         iput(inode);
551         return -ENOSPC;
552     }
553     inode->i_dirt = 1;
// Now we read the newly requested disk block from the device (the purpose is to put the
// corresponding block into the buffer cache). Similarly, if an error occurs, put back the i-node
// of the directory; release the requested disk block; reset the newly requested i-node link
// count; put back the new i-node, and return no space error code to exit.
554     if (!(dir_block=bread(inode->i_dev,inode->i_zone[0]))) {
555         iput(dir);
556         inode->i_nlinks--;
557         iput(inode);
558         return -ERROR;
559     }
// Then we create two default new directory entries ('.' and '..') data in the created directory
// file in the buffer block. First, let 'de' point to the data block that holds the directory
// entry, and then set the i-node number field of the directory entry equal to the newly applied
// i-node number, and the name field is set to ".". Then 'de' points to the next directory entry
// structure, and stores the i-node number of the parent directory and name ".." in the structure.
// Then set the cache block modified flag and release the buffer block. Re-initializes the mode
// field of the new i-node and sets the i-node modified flag.
560     de = (struct dir_entry *) dir_block->b_data;
561     de->inode=inode->i_num; // set the '.' directory entry.
562     strcpy(de->name, ".");
563     de++;
564     de->inode = dir->i_num; // set the '..' directory entry.
565     strcpy(de->name, "..");
566     inode->i_nlinks = 2;
567     dir_block->b_dirt = 1;
568     brelse(dir_block);
569     inode->i_mode = I_DIRECTORY | (mode & 0777 & ~current->umask);
570     inode->i_dirt = 1;
// Now we add a new directory entry in the specified directory to store the i-node and directory
// name of the newly created directory. If the failure (including the buffer block pointer of
// the directory entry is NULL), put back the i-node of the directory; the requested i-node
// reference link count is reset and put back to the i-node. Return error code to exit.
571     bh = add_entry(dir, basename, namelen, &de);
572     if (!bh) {
573         iput(dir);
574         inode->i_nlinks=0;
575         iput(inode);
576         return -ENOSPC;
577     }
// Finally, the i-node field of the new directory entry is equal to the new i-node number, the
// cache block modified flag is set, the directory and the new i-node are put back, the buffer

```

```

// block is released, and finally 0 (success) is returned.
578     de->inode = inode->i_num;
579     bh->b_dirt = 1;
580     dir->i_nlinks++;
581     dir->i_dirt = 1;
582     iput(dir);
583     iput(inode);
584     brelse(bh);
585     return 0;
586 }
587
588 /*
589  * routine to check that the specified directory is empty (for rmdir)
590  */
591 static int empty_dir(struct m_inode * inode)
592 {
593     int nr, block;
594     int len;
595     struct buffer_head * bh;
596     struct dir_entry * de;
597
598     // First calculate the number of existing directory entries in the specified directory and check
599     // if the information in the two specific directory entries is correct. There should be at least
600     // 2 directory entries in a directory: the entries whose names are "." and "..". If the number
601     // of directory entries is less than 2 or the first direct block of the i-node of the directory
602     // does not point to any disk block number, or the direct block cannot be read, then the warning
603     // message is displayed, and 0 is returned. (failure).
604     len = inode->i_size / sizeof (struct dir_entry); // number of directory entries.
605     if (len < 2 || !inode->i_zone[0] ||
606         !(bh = bread(inode->i_dev, inode->i_zone[0]))) {
607         printk("warning - bad directory on dev %04x\n", inode->i_dev);
608         return 0;
609     }
610
611     // At this time, the buffer block indicated by 'bh' contains directory entry data. We let the
612     // directory entry pointer 'de' point to the first directory entry in the buffer block. For
613     // the first directory entry ( "." ), its i-node number field should be equal to the i-node number
614     // of the current directory; for the second directory entry ( ".. " ), its i-node number field
615     // should be equal to the one of up layer directory, and will not be 0. Therefore, if the i-node
616     // number of the first directory entry is not equal to the that of the current directory, or
617     // the i-node number of the second directory entry is zero, or the name fields of the two directory
618     // entries are not equal to "." and "..", an error warning message is displayed and returns 0.
619     de = (struct dir_entry *) bh->b_data;
620     if (de[0].inode != inode->i_num || !de[1].inode ||
621         strcmp(".", de[0].name) || strcmp("..", de[1].name)) {
622         printk("warning - bad directory on dev %04x\n", inode->i_dev);
623         return 0;
624     }
625
626     // Then we make 'nr' equal to the directory entry number (counted from 0); 'de' points to the
627     // third directory entry, and loop through all the remaining (len - 2) directory entries in
628     // the directory to see if the i-node number field is not 0 (used).

```

```

610     nr = 2;
611     de += 2;
612     while (nr < len) {
// If the directory entries in the disk block have been completely checked, and the directory
// entry being used is not found, the buffer block of the disk block is released, and the next
// disk block containing the directory entries in the directory data file is read in. The reading
// method is to calculate the data block number (nr/DIR_ENTRIES_PER_BLOCK) of the corresponding
// directory entry item in the directory file according to the currently detecting directory
// item number 'nr', and then use the bmap() function to obtain the block number, and then use
// the block device function bread() reads the disk block into the buffer cache and returns
// a pointer to the buffer block. If the disk block is not used (or has not been used, such
// as the file has been deleted, etc.), continue to read the next block, if it cannot be read
// out, then error returns 0. Otherwise let 'de' point to the first directory entry in the block.
613         if ((void *) de >= (void *) (bh->b_data+BLOCK_SIZE)) {
614             brelse(bh);
615             block=bmap(inode,nr/DIR_ENTRIES_PER_BLOCK);
616             if (!block) {
617                 nr += DIR_ENTRIES_PER_BLOCK;
618                 continue;
619             }
620             if (!(bh=bread(inode->i_dev,block)))
621                 return 0;
622             de = (struct dir_entry *) bh->b_data;
623         }
// For the current directory entry, if its i-node number field is not equal to 0, it means that
// the directory item is being used before, indicating that the directory is not empty, then
// the buffer block is released, and 0 is returned. Otherwise, if all the directory entries
// in the directory have not been queried, the directory entry number 'nr' is incremented, and
// 'de' is pointed to the next directory entry, and the detection continues.
624         if (de->inode) {
625             brelse(bh);
626             return 0;
627         }
628         de++;
629         nr++;
630     }
// If the code is executed here, it means that the used directory entry is not found in the
// directory (except for the first two, of course), then the buffer block is released and 1
// is returned.
631     brelse(bh);
632     return 1;                // directory is empty!
633 }
634
///// Remove the directory.
// Parameters: name - the directory name (path name).
// Returns: Returns 0 for success, otherwise returns an error number.
635 int sys_rmdir(const char * name)
636 {
637     const char * basename;
638     int namelen;
639     struct m_inode * dir, * inode;
640     struct buffer_head * bh;
641     struct dir_entry * de;

```

```

642 // First check the validity of the operation permissions and parameters and get the i-node of
// the top-level directory in the path name. If the i-node corresponding to the top-level
// directory in the path name is not found, an error code is returned. If the top file name
// length is 0, it means that the given path name does not specify a file name at the end or
// the process has no write permission in the directory, then we put back the i-node of the
// directory and return an error code.
643     if (!(dir = dir\_namei(name, &namelen, &basename, NULL)))
644         return -ENOENT;
645     if (!namelen) {
646         iput(dir);
647         return -ENOENT;
648     }
649     if (!permission(dir, MAY\_WRITE)) {
650         iput(dir);
651         return -EPERM;
652     }
// Then, according to the i-node and directory name of the specified directory, the function
// find_entry() is used to find the directory entry, and the buffer block 'bh' containing the
// directory entry, the i-node 'dir' of the directory, and the directory entry 'de' are returned.
// Then, according to the i-node number in the directory entry 'de', the corresponding i-node
// is obtained by using the iget() function. In the course of specific operations, if the directory
// entry of the last directory name in the path name does not exist, the buffer block containing
// the directory entry is released, and the i-node of the directory is put back, the error code
// of file already exists is returned. If the operation of obtaining the directory entry's i-node
// is wrong, the i-node of the directory is put back, and the buffer block containing the directory
// entry is released, and the error code is returned.
653     bh = find\_entry(&dir, basename, namelen, &de);
654     if (!bh) {
655         iput(dir);
656         return -ENOENT;
657     }
658     if (!(inode = iget(dir->i_dev, de->inode))) {
659         iput(dir);
660         brelse(bh);
661         return -EPERM;
662     }
// At this point we have the directory i-node 'dir', the directory entry 'de' to be deleted
// and its corresponding i-node. Below we verify the feasibility of the deletion by checking
// the information in these three objects.
//
// If the directory has the restricted delete flag set and the effective user id (euid) of the
// process is not root, and the effective user id (euid) of the process is not equal to the
// user id of the i-node, it means that the current process does not have permission to remove
// the directory, so we put back the directory i-node and the entry's i-node, then release the
// buffer block and return the error code.
663     if ((dir->i_mode & S\_ISVTX) && current->euid &&
664         inode->i_uid != current->euid) {
665         iput(dir);
666         iput(inode);
667         brelse(bh);
668         return -EPERM;
669     }

```

```

// If the device number of the directory entry i-node is not equal to the device number of the
// directory containing this entry, or the referenced link count of the removing directory is
// greater than 1 (indicating there is symbol link, etc.), the directory cannot be removed.
// So the directory i-node containing the directory name to be deleted and the entry's i-node
// are then released, the buffer block is released, and an error code is returned.
670     if (inode->i_dev != dir->i_dev || inode->i_count>1) {
671         iput(dir);
672         iput(inode);
673         brelse(bh);
674         return -EPERM;
675     }
// If the directory entry i-node is equal to the i-node of the directory, indicating that the
// "." directory is attempted to be deleted, this is not allowed. Then the directory entry i-node
// and the i-node of the directory to be deleted are put back, the buffer block is released,
// and the error code is returned.
676     if (inode == dir) {      /* we may not delete ".", but "../dir" is ok */
677         iput(inode);
678         iput(dir);
679         brelse(bh);
680         return -EPERM;
681     }
// If the attribute of the i-node of the directory to be deleted indicates that this is not
// a directory, the premise of this deletion operation does not exist at all. Then, the i-node
// of the directory entry to be deleted and the i-node of its directory are put back, the buffer
// block is released, and an error code is returned.
682     if (!S\_ISDIR(inode->i_mode)) {
683         iput(inode);
684         iput(dir);
685         brelse(bh);
686         return -ENOTDIR;
687     }
// If the directory to be deleted is not empty, it cannot be deleted. Then, the directory i-node
// containing the directory name to be deleted and the i-node of the directory to be deleted
// are put back, the buffer block is released, and an error code is returned.
688     if (!empty\_dir(inode)) {
689         iput(inode);
690         iput(dir);
691         brelse(bh);
692         return -ENOTEMPTY;
693     }
// For an empty directory, the number of directory entry links should be 2 (link to the upper
// directory and itself). If the number of links of the i-node to be deleted is not equal to
// 2, a warning message is displayed, but the deletion operation continues. Then, the i-node
// number field of the directory entry of the directory to be deleted is set to 0, indicating
// that the directory entry is no longer used, and the buffer block modified flag is set, and
// the buffer block is released. Then set the number of links of the i-node of the deleted
// directory to 0 (indicating idle), and set the i-node modified flag.
694     if (inode->i_nlinks != 2)
695         printk("empty directory has nlink!=2 (%d)", inode->i_nlinks);
696     de->inode = 0;
697     bh->b_dirt = 1;
698     brelse(bh);
699     inode->i_nlinks=0;

```



```

700     inode->i_dirt=1;
// Then reduce the i-node link count of the directory containing the deleted directory name
// by 1, modify the change time and modification time to the current time, and set the modified
// flag of the node. Finally, the directory i-node containing the directory name to be deleted
// and the i-node of the directory to be deleted are put back, and 0 is returned (the deletion
// operation is successful).
701     dir->i_nlinks--;
702     dir->i_ctime = dir->i_mtime = CURRENT\_TIME;
703     dir->i_dirt=1;
704     iput(dir);
705     iput(inode);
706     return 0;
707 }
708
///// Delete (release) the directory entry corresponding to a file name.
// Remove a name from the file system. If it is the last link to the file and no process is
// opening the file, the file will also be deleted and the occupied device space will be freed.
// Parameters: name - the file name (path name).
// Returns: 0 if successful, otherwise returns an error code.
709 int sys\_unlink(const char * name)
710 {
711     const char * basename;
712     int namelen;
713     struct m\_inode * dir, * inode;
714     struct buffer head * bh;
715     struct dir\_entry * de;
716
// First check the validity of the operation permissions and parameters and get the i-node of
// the top-level directory in the path name. If the i-node corresponding to the top-level
// directory in the path name is not found, an error code is returned. If the top file name
// length is 0, it means that the given path name does not specify a file name at the end or
// the process has no write permission in the directory, then we put back the i-node of the
// directory and return an error code.
717     if (!(dir = dir\_namei(name, &namelen, &basename, NULL)))
718         return -ENOENT;
719     if (!namelen) {
720         iput(dir);
721         return -ENOENT;
722     }
723     if (!permission(dir, MAY\_WRITE)) {
724         iput(dir);
725         return -EPERM;
726     }
// Then, according to the i-node and directory name of the specified directory, the function
// find_entry() is used to find the directory entry, and the buffer block 'bh' containing the
// directory entry, the i-node 'dir' of the directory, and the directory entry 'de' are returned.
// Then, according to the i-node number in the directory entry 'de', the corresponding i-node
// is obtained by using the iget() function.
727     bh = find\_entry(&dir, basename, namelen, &de);
728     if (!bh) {
729         iput(dir);
730         return -ENOENT;
731     }

```

```

732         if (!(inode = iget(dir->i_dev, de->inode))) {
733             iput(dir);
734             brelse(bh);
735             return -ENOENT;
736         }
// At this point we have the directory i-node 'dir' , the directory entry 'de' to be deleted
// and its corresponding i-node. Below we verify the feasibility of the deletion by checking
// the information in these three objects.
//
// If the directory has the restricted deletion flag set and the valid user id (euid) of the
// process is not root, and the euid of the process is not equal to the user id of the i-node,
// and the euid of the process is not equal to the user id of the directory i-node, Indicates
// that the current process does not have permission to delete the directory. Then put back
// the directory i-node containing the directory name to be deleted and the i-node of the
// directory to be deleted, then release the buffer block and return the error code.
737         if ((dir->i_mode & S\_ISVTX) && !suser() &&
738             current->euid != inode->i_uid &&
739             current->euid != dir->i_uid) {
740             iput(dir);
741             iput(inode);
742             brelse(bh);
743             return -EPERM;
744         }
// If the specified file name is a directory, it cannot be deleted. Then the i-node of the
// directory i-node and the file name directory entry is put back, the buffer block containing
// the directory entry is released, and the error code is returned.
745         if (S\_ISDIR(inode->i_mode)) {
746             iput(inode);
747             iput(dir);
748             brelse(bh);
749             return -EPERM;
750         }
// If i-node's link count is already 0, a warning message is displayed and corrected to 1.
751         if (!inode->i_nlinks) {
752             printk("Deleting nonexistent file (%04x:%d), %d\n",
753                 inode->i_dev, inode->i_num, inode->i_nlinks);
754             inode->i_nlinks=1;
755         }
// Now we can delete the directory entry corresponding to the file name. Then, the i-node number
// field in the file name directory entry is set to 0, indicating that the directory entry is
// released, and the buffer block modified flag is set, and the buffer block is released.
756         de->inode = 0;
757         bh->b_dirt = 1;
758         brelse(bh);
// Then, the number of links of the i-node corresponding to the file name is decremented by
// 1, the modified flag is set, and the update time is set to the current time. Finally, put
// back the i-node and directory's i-node, and return 0 (success). If it is the last link of
// the file, that is, the number of i-node links minus 1 is equal to 0, and no process is opening
// the file at this time, the file will also be deleted when iput() is called to put back the
// i-node and the occupied device space is released. See fs/inode.c, line 183.
759         inode->i_nlinks--;
760         inode->i_dirt = 1;
761         inode->i_ctime = CURRENT\_TIME;

```

```

762     iput(inode);
763     iput(dir);
764     return 0;
765 }
766
767 // Create a symbolic link.
768 // Create a symbolic link (also known as a soft link) for an existing file.
769 // Parameters: oldname - the original path name; newname - the new path name.
770 // Returns: 0 if successful, otherwise returns an error code.
771 int sys\_symlink(const char * oldname, const char * newname)
772 {
773     struct dir\_entry * de;
774     struct m\_inode * dir, * inode;
775     struct buffer\_head * bh, * name_block;
776     const char * basename;
777     int namelen, i;
778     char c;
779
780     // First find the i-node 'dir' of the top-level directory of the new pathname and return the
781     // last file name and its length. If the i-node of the directory is not found, an error code
782     // is returned. If the file name is not included in the new path name, the i-node of the new
783     // path name directory is put back and the error code is returned. In addition, if the user
784     // does not have permission to write in the new directory, then the link cannot be established.
785     // Then put back the i-node of the new path name directory and return the error code.
786     dir = dir\_namei(newname, &namelen, &basename, NULL);
787     if (!dir)
788         return -EACCES;
789     if (!namelen) {
790         iput(dir);
791         return -EPERM;
792     }
793     if (!permission(dir, MAY\_WRITE)) {
794         iput(dir);
795         return -EACCES;
796     }
797     // Now we apply for a new i-node on the device specified by the directory, and set the i-node
798     // mode to the symbolic link type and the mode mask code specified by the process, and set the
799     // i-node modified flag.
800     if (!(inode = new\_inode(dir->i_dev))) {
801         iput(dir);
802         return -ENOSPC;
803     }
804     inode->i_mode = S\_IFLNK | (0777 & ~current->umask);
805     inode->i_dirt = 1;
806     // In order to save the symbolic link path name string, we need to apply for a disk block for
807     // the i-node, and let the first direct block number i_zone[0] equal to the obtained logical
808     // block number, and then set the i-node modified flag. If the application fails, put back the
809     // i-node of the corresponding directory; reset the newly requested i-node link count; put back
810     // the new i-node, and return no space error code.
811     if (!(inode->i_zone[0] = new\_block(inode->i_dev))) {
812         iput(dir);
813         inode->i_nlinks--;
814         iput(inode);

```

```

797         return -ENOSPC;
798     }
799     inode->i_dirt = 1;
// The newly requested disk block is then read from the device (the purpose is to place the
// block in the buffer cache). If there is an error, put back the i-node of the directory; reset
// the newly requested i-node link count; put back the new i-node, and return error code.
800     if (!(name_block=bread(inode->i_dev, inode->i_zone[0]))) {
801         iput(dir);
802         inode->i_nlinks--;
803         iput(inode);
804         return -ERROR;
805     }
// Now we can put the symbolic link name string into this disk block. The disk block length
// is 1024 bytes, so the default symbol link name maximum length can only be 1024 bytes. We
// copy the symbolic link name string in the user space to the buffer block where the disk block
// is located, and set the buffer block modified flag. To prevent the user-supplied string from
// ending without a null, we need to put a NULL at the last byte of the buffer block data area.
// Then release the buffer block, and set the size of the data in the corresponding file of
// the i-node to be equal to the length of the symbolic link name string, and set the i-node
// modified flag.
806     i = 0;
807     while (i < 1023 && (c=get_fs_byte(oldname++)))
808         name_block->b_data[i++] = c;
809     name_block->b_data[i] = 0;
810     name_block->b_dirt = 1;
811     brelse(name_block);
812     inode->i_size = i;
813     inode->i_dirt = 1;
// Then we search for the name of the symbolic link file specified by the path name. If it already
// exists, you cannot create a directory entry i-node with the same name. If the corresponding
// symbolic link file name already exists, the buffer block containing the directory entry is
// released, the newly requested i-node link count is reset, and the i-node of the directory
// is put back, and the error code that the file already exists is returned.
814     bh = find_entry(&dir, basename, namelen, &de);
815     if (bh) {
816         inode->i_nlinks--;
817         iput(inode);
818         brelse(bh);
819         iput(dir);
820         return -EEXIST;
821     }
// Now we add a new directory entry in the specified directory, which is used to store the i-node
// number and directory name of the newly created symbolic link file name. If it fails (the
// buffer block pointer containing the directory entry is NULL), the i-node of the directory
// is put back; the requested i-node reference link count is reset, and the i-node is put back,
// and the error code is returned.
822     bh = add_entry(dir, basename, namelen, &de);
823     if (!bh) {
824         inode->i_nlinks--;
825         iput(inode);
826         iput(dir);
827         return -ENOSPC;
828     }

```

```

// Finally, the i-node field of the new directory entry is equal to the new i-node number, the
// buffer block modified flag is set, the buffer block is released, the directory and the new
// i-node are put back, and finally 0 (success) is returned.
829     de->inode = inode->i_num;
830     bh->b_dirt = 1;
831     brelse(bh);
832     iput(dir);
833     iput(inode);
834     return 0;
835 }
836
///// Create a file name directory entry for an existing file.
// Create a new link (also known as hard link) for an existing file.
// Parameters: oldname - the original path name; newname - the new path name.
// Returns: 0 if successful, otherwise returns an error code.
837 int sys_link(const char * oldname, const char * newname)
838 {
839     struct dir_entry * de;
840     struct m_inode * oldinode, * dir;
841     struct buffer_head * bh;
842     const char * basename;
843     int namelen;
844
// First validate the original file name, it should exist and not a directory name. So we first
// take the i-node 'oldinode' corresponding to the original file path name. If it is 0, it means
// an error and returns the error code. If the original path name corresponds to a directory
// name, the i-node is put back and an error code is also returned.
845     oldinode=namei(oldname);
846     if (!oldinode)
847         return -ENOENT;
848     if (S_ISDIR(oldinode->i_mode)) {
849         iput(oldinode);
850         return -EPERM;
851     }
// Then find the i-node 'dir' of the top-level directory of the new pathname and return the
// last file name and its length. If the i-node of the directory is not found, the i-node of
// the original path name is put back and the error code is returned. If the file name is not
// included in the new path name, the i-node of the original path name and the new path name
// directory are put back, and the error code is returned.
852     dir = dir_namei(newname, &namelen, &basename, NULL);
853     if (!dir) {
854         iput(oldinode);
855         return -EACCES;
856     }
857     if (!namelen) {
858         iput(oldinode);
859         iput(dir);
860         return -EPERM;
861     }
// We can't build hard links across devices. Therefore, if the device number of the top directory
// of the new path name is different from the device number of the original path name, the i-node
// of the new path name directory and the i-node of the original path name are put back, and
// the error code is returned. In addition, if the user does not have the right to write in

```

---

```

// the new directory, the link cannot be established, so the i-node of the new path name directory
// and the i-node of the original path name are put back, and the error code is returned.
862     if (dir->i_dev != oldinode->i_dev) {
863         iput(dir);
864         iput(oldinode);
865         return -EXDEV;
866     }
867     if (!permission(dir, MAY\_WRITE)) {
868         iput(dir);
869         iput(oldinode);
870         return -EACCES;
871     }
// Now check if the new pathname already exists and if it does, it will not be able to establish
// a link. Then, the buffer block containing the existing directory entry is released, and the
// i-node of the new path name directory and the i-node of the original path name are put back,
// and the error code is returned.
872     bh = find\_entry(&dir, basename, namelen, &de);
873     if (bh) {
874         brelse(bh);
875         iput(dir);
876         iput(oldinode);
877         return -EEXIST;
878     }
// Now that all the conditions are met, we add a directory entry in the new directory. If it
// fails, put back the i-node of the directory and the i-node of the original pathname, and
// return the error code. Otherwise, the i-node number of the directory entry is initially set
// equal to the i-node number of the original path name, and the buffer block modified flag
// containing the newly added directory entry is set, the buffer block is released, and the
// i-node of the directory is put back.
879     bh = add\_entry(dir, basename, namelen, &de);
880     if (!bh) {
881         iput(dir);
882         iput(oldinode);
883         return -ENOSPC;
884     }
885     de->inode = oldinode->i_num;
886     bh->b_dirt = 1;
887     brelse(bh);
888     iput(dir);
// Then increase the link count of the original node by 1, modify its change time to the current
// time, and set the i-node modified flag. Finally, put back the i-node of the original path
// name and return 0 (success).
889     oldinode->i_nlinks++;
890     oldinode->i_ctime = CURRENT\_TIME;
891     oldinode->i_dirt = 1;
892     iput(oldinode);
893     return 0;
894 }
895

```

---

## 12.8 file\_table.c

### 12.8.1 Function

The file\_table.c program is currently empty, only the file table array is defined.

### 12.8.2 Code annotation

Program 12-7 linux/fs/file\_table.c

---

```
1 /*  
2  *  linux/fs/file_table.c  
3  *  
4  *  (C) 1991  Linus Torvalds  
5  */  
6  
7 // <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,  
8 //      m_inode, etc.).  
9 #include <linux/fs.h>  
10  
11 struct file file_table[NR_FILE]; // File table array (64 items in total).  
12
```

---

## 12.9 block\_dev.c

From here on is the third part of the file system program, which includes five programs: block\_dev.c, char\_dev.c, pipe.c, file\_dev.c, and read\_write.c. The first four programs provide services for read\_write.c, which mainly implements data access operations of the file system. The read\_write.c program mainly implements the system-calls sys\_write() and sys\_read(). These five programs can be thought of as interface drivers for system-calls with block devices, character devices, pipe "devices", and file system "devices." The relationship between them can be represented by Figures 12-26. The system-calls sys\_write() or sys\_read() will determine which type of file is based on the attributes of the file descriptor provided by the parameter, and then call the read/write functions in the corresponding device interface program, and these functions will execute driver codes accordingly.

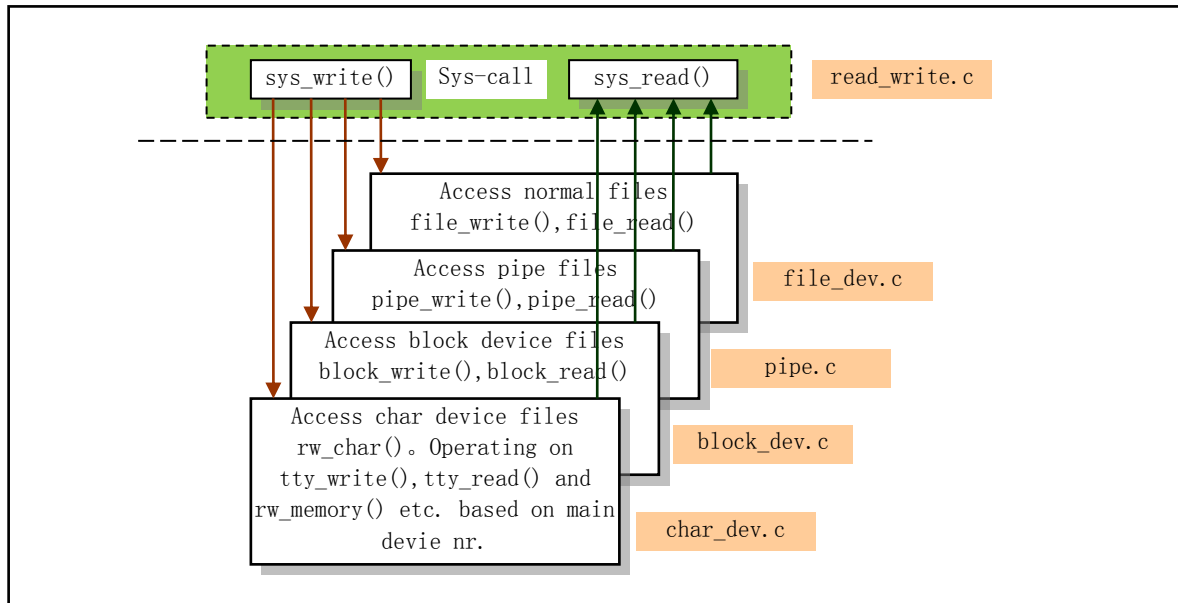


Figure 12-26 Interface functions for various types of files and system-calls

### 12.9.1 Function

The `block_dev.c` program belongs to the block device file data access operation program. The file includes two block device read and write functions, `block_read()` and `block_write()`, which are used to directly read and write the original data on the block device. These two functions are called by the system call function `read()` and `write()`, and they are not referenced elsewhere.

Since the block device reads and writes to the disk each time in units of disk blocks (same size as the buffer block), the function `block_write()` first maps the file pointer 'pos' position to the block number and the offset value in the block, and then use the block read function `bread()` or the block read-ahead function `breada()` to read the data block where the file pointer is located into the buffer block of the buffer cache. The data is then copied from the user data buffer to the offset position of the current buffer block according to the length 'chars' of the data to be written. If there is still data to be written, the next block is read into the buffer block in the cache, and the user data is copied into the buffer block. When the data is written for the second time and later, the offset offset is 0. See Figure 12-27.

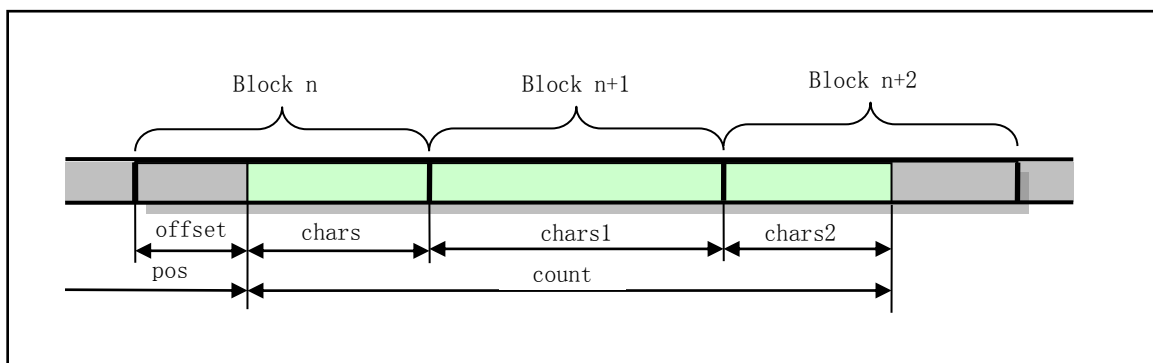


Figure 12-27 Block data read and write operation pointer position diagram

The user's buffer is allocated by the system when the user program starts executing, or dynamically applied



during execution. Before calling this function, the system maps the virtual linear address used by the user buffer to the corresponding memory page in the main memory area.

The function `block_read()` operates in the same way as `block_write()` except that the data is copied from the buffer to the location specified by the user.

## 12.9.2 Code annotation

Program 12-8 linux/fs/block\_dev.c

---

```

1  /*
2   *  linux/fs/block_dev.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  // <errno.h> Error number header file. Contains various error numbers in the system.
8  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
9  //   of the initial task 0, and some embedded assembly function macro statements about the
10 //   descriptor parameter settings and acquisition.
11 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
12 //   used functions of the kernel.
13 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
14 //   segment register operations.
15 // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
16 //   descriptors/interrupt gates, etc. is defined.
17 #include <errno.h>
18
19 #include <linux/sched.h>
20 #include <linux/kernel.h>
21 #include <asm/segment.h>
22 #include <asm/system.h>
23
24 // An array of pointers to the total number of device data blocks. Each of its pointers points
25 // to an array of total blocks of the specified major device number, hd_sizes[]. Each item of
26 // the total number of block arrays corresponds to the total number of data blocks owned by
27 // a sub-device determined by the sub-device number.
28 extern int *blk_size[];           // blk_drv/ll_rw_blk.c, line 49.
29
30 // Block write function - Writes data of given size to the specified offset on a device.
31 // Parameters: dev - device number; pos - offset pointer in the device file;
32 // buf - buffer in user space; count - number of bytes to transfer.
33 // Returns: number of bytes written. If no data is written or errors, returns error code.
34 // For the kernel, the write operation is to write data to the cache. When the data is finally
35 // written to the device is determined and processed by the cache manager. In addition, since
36 // the block device reads and writes in units of blocks, when the write position is not at the
37 // edge of the block, the entire block in which the data is located needs to be read out first,
38 // and then the data is filled from the write position. Then write a complete block of data
39 // to the disk (ie, handle it to the buffer cache).
40 int block_write(int dev, long * pos, char * buf, int count)
41 {
42     // First, the position 'pos' in the file is converted into a block number 'block' in which the
43     // disk block is started to be read and written, and the offset position 'offset' at which the

```

```

// first byte is to be written in the block is obtained.
18     int block = *pos >> BLOCK_SIZE_BITS;    // the block number where pos is located.
19     int offset = *pos & (BLOCK_SIZE-1);    // offset value in the data block.
20     int chars;
21     int written = 0;
22     int size;
23     struct buffer_head * bh;
24     register char * p;                      // local register variable.
25
// When writing a block device file, the total number of data blocks required to be written
// must of course not exceed the maximum number of data blocks allowed on the specified device.
// Therefore, the total number of blocks of the specified device is first taken out to compare
// and limit the write data length given by the function parameter. If no length is specified
// for the device in the system, the default length of 0x7fffffff (2GB blocks) is used.
26     if (blk_size[MAJOR(dev)])
27         size = blk_size[MAJOR(dev)][MINOR(dev)];
28     else
29         size = 0x7fffffff;
// Then, for the number of bytes to be written 'count', the following operations are looped
// until the data is completely written. During the loop execution, if the block number of the
// currently written data is greater than or equal to the total number of blocks of the specified
// device, the number of bytes written is returned and exited. The number of bytes that can
// be written in the currently processed data block is then calculated. If the number of bytes
// to be written is less than one block, then we only need to write the count byte; if we just
// want to write 1 block of data, then directly apply for a buffer block and put the user data
// into it. Otherwise, we need to read in the data block that will be written to the partial
// data, and pre-read the next two blocks of data. Then increment the block number and prepare
// for the next operation. If the buffer block operation fails, the number of bytes written
// is returned, and if no bytes are written, an error code (negative number) is returned.
30     while (count>0) {
31         if (block >= size)
32             return written?written:-EIO;
33         chars = BLOCK_SIZE - offset;    // bytes that can be written to this block.
34         if (chars > count)
35             chars=count;
36         if (chars == BLOCK_SIZE)
37             bh = getblk(dev,block);    // buffer.c, lines 206 and 322.
38         else
39             bh = breada(dev,block,block+1,block+2,-1);
40         block++;
41         if (!bh)
42             return written?written:-EIO;
// Next, the pointer p is first pointed to the position in the buffer block where the data is
// to be written. If there is less than one block of data written in the last loop, we need
// to fill in (modify) the required bytes from the beginning of the block, so the offset must
// be set to zero by default. After that, the offset pointer 'pos' in the file is forwarded
// by the number of bytes to be written this time, and the number of bytes to be written is
// accumulated into the statistical value 'written'. Then subtract the number of bytes 'counts'
// that need to be written from the number of bytes (chars) to be written this time. Then we
// copy the 'chars' bytes from the user buffer to the location in the buffer block where the
// write begins. After the copy is completed, the buffer block modified flag is set and the
// buffer block is released (that is, the buffer block reference count is decremented by 1).
43         p = offset + bh->b_data;

```

```

44         offset = 0;
45         *pos += chars;
46         written += chars;                // The total number of bytes written.
47         count -= chars;
48         while (chars-->0)
49             *(p++) = get\_fs\_byte(buf++);
50         bh->b_dirt = 1;
51         brelse(bh);
52     }
53     return written;                      // Returns the number of bytes written.
54 }
55
56 // Block read function - Reads data from the specified device into the user buffer.
57 // Parameters: dev - device number; pos - offset in the device file; buf - buffer in user space;
58 // count - number of bytes to transfer.
59 // Returns: number of bytes read. If no bytes are read or error, an error code is returned.
60 int block\_read(int dev, unsigned long * pos, char * buf, int count)
61 {
62     int block = *pos >> BLOCK\_SIZE\_BITS;
63     int offset = *pos & (BLOCK\_SIZE-1);
64     int chars;
65     int size;
66     int read = 0;
67     struct buffer head * bh;
68     register char * p;                  // local register variable.
69
70     // When reading a block device file, the total number of blocks required to be read cannot exceed
71     // the maximum number of blocks allowed on the specified device. Therefore, the total number
72     // of blocks of the device is first taken out to compare and limit the read data length given
73     // by the function parameter. If no length is specified for the device in the system, the default
74     // length of 0x7fffffff (2GB blocks) is used.
75     if (blk\_size[MAJOR(dev)])
76         size = blk\_size[MAJOR(dev)][MINOR(dev)];
77     else
78         size = 0x7fffffff;
79
80     // Then, for the number of bytes to be read in 'count', the following operations are looped
81     // until the data is completely read. During the loop execution, if the block number of the
82     // currently read data is greater than or equal to the total number of blocks of the specified
83     // device, the number of read bytes is returned and exited. Then calculate the number of bytes
84     // to read in the currently processed data block. If the number of bytes that need to be read
85     // is less than one block, then just read 'count' bytes. Then invoke the read block function
86     // breada () to read in the required data block, and pre-read the next two blocks of data. If
87     // the read operation is in error, the number of bytes read is returned. If no bytes are read,
88     // an error code is returned. Then increment the block number by one to prepare for the next
89     // operation.
90     while (count>0) {
91         if (block >= size)
92             return read?read:-EIO;
93         chars = BLOCK\_SIZE-offset;
94         if (chars > count)
95             chars = count;
96         if (!(bh = breada(dev, block, block+1, block+2, -1)))
97             return read?read:-EIO;

```

```
78         block++;
// Next, the pointer p is first pointed to the position in the buffer block of the read disk
// block where the data is read. If the data of the last loop read operation is less than one block,
// the required byte needs to be read from the beginning of the block, so the offset must be set to
// zero beforehand. After that, the offset pointer 'pos' in the file is moved forward by the number
// of bytes 'chars' to be read, and the number of bytes to be read is accumulated into the statistical
// value 'read'. Then subtract the count number 'chars' to be read from the count value 'count' that
// needs to be read. Then we copy the 'chars' bytes from the starting point pointed to by the 'p' in
// the buffer block to the user buffer, and move the user buffer pointer forward. The buffer block
// is released after this copy is completed.
79         p = offset + bh->b_data;
80         offset = 0;
81         *pos += chars;
82         read += chars;           // The total number of bytes read in.
83         count -= chars;
84         while (chars-->0)
85             put_fs_byte(*(p++), buf++);
86         brelse(bh);
87     }
88     return read;                // returns the number of bytes read.
89 }
90
```

---

## 12.10 file\_dev.c

### 12.10.1 Function

The file\_dev.c file contains two functions, file\_read() and file\_write(). They are also used by the system-call functions read() and write() to read and write ordinary files. Similar to the previous file block\_dev.c, this file is also used to access file data, but the functions in this program are operated by specifying the file path name. The file i-node and file structure are given in the function parameters. Through the information in the i node, the corresponding device number can be obtained, and by the file structure, we can obtain the current read and write pointer position of the file. The function in the previous file block\_dev.c specifies the device number and the read/write location in the file directly in the parameter. It is specifically used to operate on the block device file, such as the /dev/hd0 device file.

### 12.10.2 Code annotation

Program 12-9 linux/fs/file\_dev.c

---

```
1  /*
2  *  linux/fs/file_dev.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
// <errno.h> Error number header file. Contains various error numbers in the system.
// <fcntl.h> File control header file. The definition of the operation control constant symbol
//     used for the file and its descriptors.
```

```

// <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/segment.h> Segment operation header file. An embedded assembly function is defined for
// segment register operations.
7 #include <errno.h>
8 #include <fcntl.h>
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h>
12 #include <asm/segment.h>
13
14 #define MIN(a,b) (((a)<(b))?(a):(b)) // get the minimum value in a, b.
15 #define MAX(a,b) (((a)>(b))?(a):(b)) // get the maximum value in a, b.
16
///// File Read Function - Reads data in the file based on the i-node and file structure.
// From the i-node we can know the device number, and the file structure can know the current
// read-write pointer position in the file. 'Buf' specifies the location of the buffer in user
// space, and 'count' is the number of bytes that need to be read. The return value is the number
// of bytes actually read, or the error code (less than 0).
17 int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
18 {
19     int left, chars, nr;
20     struct buffer_head * bh;
21
// First determine the validity of the parameters. Returns 0 if the byte count to be read is
// less than or equal to zero. If the number of bytes to be read is not equal to 0, the following
// loop operation is performed until the data is completely read or a problem is encountered.
// During the read loop operation, we use bmap() to get the corresponding logical (disk) block
// number 'nr' of the data block on the device based on the i-node and file table structure
// information. If nr is not 0, the disk block is read from the device. Exit the loop if the
// read operation fails. If nr is 0, it means that the specified data block does not exist,
// so the buffer block pointer is set to NULL. (filp->f_pos)/BLOCK_SIZE is used to calculate
// the data block number where the current pointer of the file is located.
22     if ((left=count)<=0)
23         return 0;
24     while (left) {
25         if (nr = bmap(inode, (filp->f_pos)/BLOCK_SIZE)) { // inode.c, line 140.
26             if (!(bh=bread(inode->i_dev,nr)))
27                 break;
28         } else
29             bh = NULL;
// Then we calculate the offset value nr of the file read/write pointer in the data block, then
// the number of bytes we want to read in the data block is (BLOCK_SIZE - nr). Then compare
// with the number of bytes that need to be read now, 'left', where the small value is the number
// of bytes 'chars' to be read for this operation. If '(BLOCK_SIZE - nr) > left', the block
// is the last piece of data to be read, otherwise the next block of data needs to be read.
// Then adjust the read and write file pointers. The pointer advances the number of bytes 'chars'
// that will be read this time. The remaining byte count 'left' also needs to be subtracted
// by 'chars' bytes.
30         nr = filp->f_pos % BLOCK_SIZE;

```

```

31         chars = MIN( BLOCK_SIZE-nr , left );
32         filp->f_pos += chars;
33         left -= chars;
// If the above data is read from the device, point p to the location in the buffer block where
// the data is to be read, and copy the 'chars' bytes into the user buffer 'buf', otherwise
// fill in the user buffer with 'chars' bytes of zero.
34         if (bh) {
35             char * p = nr + bh->b_data;
36             while (chars-->0)
37                 put_fs_byte(*(p++), buf++);
38             brelse(bh);
39         } else {
40             while (chars-->0)
41                 put_fs_byte(0, buf++);
42         }
43     }
// Finally, the access time of the i-node is modified to the current time, and the number of
// bytes read is returned. If the number of read bytes is 0, the error code is returned.
// CURRENT_TIME is a macro defined on line 142 of include/linux/sched.h and is used to calculate
// UNIX time. That is, from 0:00 on January 1, 1970, to the current time. The unit is seconds.
44     inode->i_atime = CURRENT_TIME;
45     return (count-left)?(count-left):-ERROR;
46 }
47
//// File Write Function - Writes user data to a file based on the i-node and file structure.
// From the i-node we can know the device number, and from the file structure we can know the
// current read-write pointer position in the file. 'buf' specifies the location of the buffer
// in the user space, and 'count' is the number of bytes to be written. The return value is
// the number of bytes actually written, or the error code (less than 0).
48 int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
49 {
50     off_t pos;
51     int block, c;
52     struct buffer_head * bh;
53     char * p;
54     int i=0;
55
56     /*
57      * ok, append may not work when many processes are writing at the same time
58      * but so what. That way leads to madness anyway.
59      */
// First determine where the data is written to the file. If you want to append data to the
// file, move the file write pointer to the end of the file, otherwise it will be written at
// the current read and write pointer of the file.
60     if (filp->f_flags & O_APPEND)
61         pos = inode->i_size;
62     else
63         pos = filp->f_pos;
// Then, when the number of bytes that have been written 'i' (zero at the beginning) is less
// than the specified number of write bytes 'count', the following operations are performed
// cyclically. In the loop operation, we first get the logical (disk) block number 'block'
// corresponding to the file data block (pos/BLOCK_SIZE) on the device. If the disk block does
// not exist, create one. If the resulting disk block number equal to 0, the creation failed

```

---

```

// and the loop is exited. Otherwise, we read the disk block on the device according to the
// disk block number, and exit the loop if an error occurs.
64     while (i < count) {
65         if (!(block = create\_block(inode, pos/BLOCK_SIZE)))
66             break;
67         if (!(bh = bread(inode->i_dev, block)))
68             break;
// At this time, the buffer block pointer 'bh' is pointing to the file data block just read.
// Now we can find the offset 'c' of the current read/write pointer of the file in the data
// block, and point the pointer 'p' to the position in the buffer block where the data is started
// to be written, and set the buffer block modified flag. For the current pointer in the block,
// a total of c = (BLOCK_SIZE - c) bytes can be written from the start of the write position
// to the end of the block. If c is greater than the remaining number of bytes to be written
// (count - i), then simply write c = (count - i) bytes this time.
69         c = pos % BLOCK_SIZE;
70         p = c + bh->b_data;
71         bh->b_dirt = 1;
72         c = BLOCK_SIZE - c;
73         if (c > count - i) c = count - i;
// Before writing the data, we pre-set the position in the file to be written in the next loop
// operation. So we move the 'pos' pointer forward by the number of bytes that need to be written.
// If the value of the 'pos' position exceeds the current length of the file at this time, the
// file length field in the i-node is modified, and the i-node modified flag is set. Then, the
// number of bytes to be written this time is added to the written byte count 'i' for loop judgment.
// Then 'c' bytes are copied from the user buffer 'buf' to the start position pointed to by
// 'p' in the cache buffer block. The buffer block is released after copying.
74         pos += c;
75         if (pos > inode->i_size) {
76             inode->i_size = pos;
77             inode->i_dirt = 1;
78         }
79         i += c;
80         while (c-- > 0)
81             *(p++) = get\_fs\_byte(buf++);
82         brelse(bh);
83     }
// The loop exits when the data has all been written to the file or if a problem occurs during
// the write operation. At this point we change the file modification time to the current time
// and adjust the file read and write pointer. That is, if the operation does not append data
// to the end of the file, the file read/write pointer is adjusted to the current read/write
// position 'pos', and the modification time of the file i-node is changed to the current time.
// Finally, the number of bytes written is returned. If the number of bytes written is 0, the
// error code -1 is returned.
84     inode->i_mtime = CURRENT_TIME;
85     if (!(filp->f_flags & O_APPEND)) {
86         filp->f_pos = pos;
87         inode->i_ctime = CURRENT_TIME;
88     }
89     return (i?i:-1);
90 }
91

```

---

## 12.11 pipe.c

### 12.11.1 Function

Pipeline operations are the most basic way of communicating between processes. The pipe.c program includes the pipe file read and write operation functions `read_pipe()` and `write_pipe()`, and implements the pipe system-call `sys_pipe()`. These two functions are also low-level implementation functions for system-calls `read()` and `write()`, and are only used in `read_write.c`.

When creating and initializing a pipeline, the program specifically requests a pipeline i-node and allocates a page buffer (4KB) to the pipeline. The `i_size` field of the pipe i-node is set to point to the pipe buffer, the pipe data header pointer is stored in the `i_zone[0]` field, and the pipe data tail pointer is stored in the `i_zone[1]` field. For read pipeline operation, the data is read from the end of the pipeline and the pipe tail pointer is moved forward a number of bytes read; for write to the pipeline operation, the data is written to the pipe header, and the head pointer is moved forward a number of locations (pointing to a null byte), as shown in Figure 12-28.

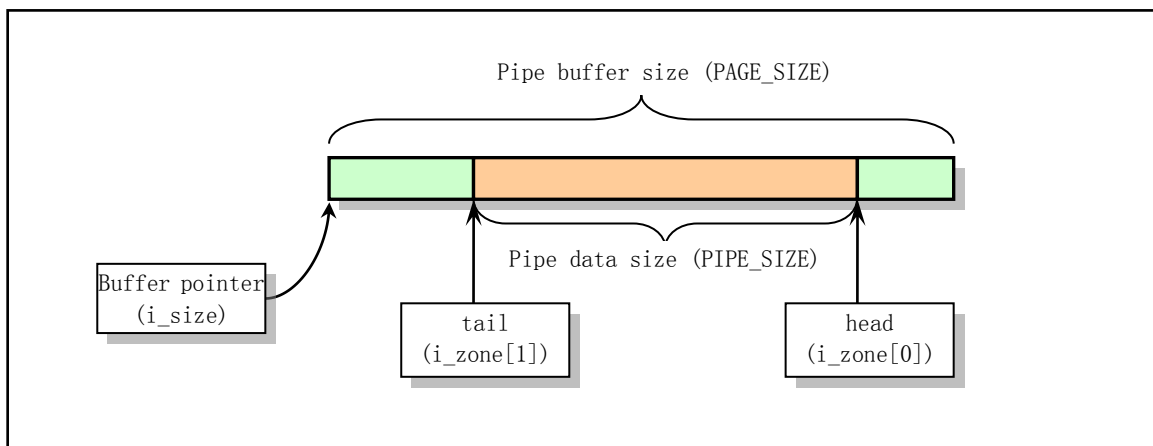


Figure 12-28 pipe buffer operation diagram

`read_pipe()` is used to read the data in the pipeline. If there is no data in the pipeline, the process of writing the pipeline is awakened, and itself goes to sleep. If the data is read, the pipe head pointer is adjusted accordingly and the data is passed to the user buffer. When all data in the pipeline are taken away, the process waiting for the write pipeline is also woken up, and the number of bytes of read data is returned. When the pipeline write process has exited the pipeline operation, the function exits immediately and returns the number of bytes read.

The `write_pipe()` function operates like a read pipeline function.

The system-call `sys_pipe()` is used to create an unnamed pipe. It first obtains two entries in the system's file table, and then looks for two unused descriptor entries in the file descriptor table of the current process to save the corresponding file structure pointer. Then apply for an idle i-node in the system and get a buffer block for the pipeline. The corresponding file structure is then initialized, one file structure is set to read-only mode, and the other is set to write-only mode. Finally, the two file descriptors are passed to the user.

In addition, several macros related to pipeline operations (such as `PIPE_HEAD()`, `PIPE_TAIL()`, etc.) used in the above functions are defined on lines 57--64 of the `include/linux/fs.h` file.



## 12.11.2 Code annotation

Program 12-10 linux/fs/pipe.c

```

1  /*
2  *  linux/fs/pipe.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  // <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
8  //      manipulation function prototypes.
9  // <errno.h> Error number header file. Contains various error numbers in the system.
10 // <termios.h> Terminal input and output function header file. It mainly defines the terminal
11 //      interface that controls the asynchronous communication port.
12 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
13 //      of the initial task 0, and some embedded assembly function macro statements about the
14 //      descriptor parameter settings and acquisition.
15 // <linux/mm.h> Memory management header file. Contains page size definitions and some page
16 //      release function prototypes.
17 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
18 //      segment register operations.
19 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
20 //      used functions of the kernel.
21 #include <signal.h>
22 #include <errno.h>
23 #include <termios.h>
24
25 #include <linux/sched.h>
26 #include <linux/mm.h>          /* for get_free_page */
27 #include <asm/segment.h>
28 #include <linux/kernel.h>
29
30 // Pipe read function.
31 // The parameter 'inode' is the i-node of the pipe, 'buf' is the user data buffer pointer, and
32 // 'count' is the number of bytes read.
33 int read_pipe(struct m_inode * inode, char * buf, int count)
34 {
35     int chars, size, read = 0;
36
37     // If the byte count that needs to be read is greater than 0, we loop through the following
38     // operations. During the loop operation, if there is no data in the current pipe (size=0),
39     // the process waiting for the node is woken up, which is usually a write pipeline process.
40     // If the pipe writer is no longer there, that is, the i-node reference count value is less
41     // than 2, the number of read bytes is returned. If there is a non-blocking signal currently
42     // received, it will immediately return the number of bytes read and exit; if no data has been
43     // received yet, it will return to restart the system-call number and exit. Otherwise, let the
44     // process sleep on the pipe to wait for the arrival of information. The macro PIPE_SIZE is
45     // defined in include/linux/fs.h. For the meaning of "restart system-call", see the
46     // kernel/signal.c program.
47     while (count>0) {
48         while (!(size=PIPE_SIZE(*inode))) {          // get data length in the pipe.
49             wake_up(& PIPE_WRITE_WAIT(*inode));

```

```

23         if (inode->i_count != 2)    /* are there any writers? */
24             return read;
25         if (current->signal & ~current->blocked)
26             return read?read:-ERESTARTSYS;
27         interruptible sleep on(& PIPE_READ_WAIT(*inode));
28     }
    // At this point, there is data in the pipeline (buffer), so we get the number of 'chars' in
    // the pipe from the tail pointer to the end of the buffer. If it is greater than the number
    // of bytes that need to be read, 'count', make it equal to 'count'. If 'chars' is greater than
    // the 'size' of the data in the current pipeline, it is equal to 'size'. Then, the number of
    // bytes to be read 'count' is subtracted from the number of bytes readable by this time, and
    // added to the already read number of bytes 'read'.
29     chars = PAGE_SIZE-PIPE_TAIL(*inode);
30     if (chars > count)
31         chars = count;
32     if (chars > size)
33         chars = size;
34     count -= chars;
35     read += chars;
    // Then let 'size' point to the pipe tail pointer and adjust the current pipe tail pointer (forward
    // 'chars' bytes). Wraps around if the tail pointer exceeds the end of the pipe. The data in
    // the pipe is then copied into the user buffer. For a pipe i-node, the pipe buffer block pointer
    // is in the i_size field.
36     size = PIPE_TAIL(*inode);
37     PIPE_TAIL(*inode) += chars;
38     PIPE_TAIL(*inode) &= (PAGE_SIZE-1);
39     while (chars-->0)
40         put_fs_byte((char *)inode->i_size)[size++], buf++);
41 }
    // When the read pipe operation ends, the process waiting for the pipeline is woken up and the
    // number of bytes read is returned.
42     wake_up(& PIPE_WRITE_WAIT(*inode));
43     return read;
44 }
45
    /// Pipe write function.
    // The parameter 'inode' is the i-node of the pipe, 'buf' is the data buffer pointer, and 'count'
    // is the number of bytes that will be written to the pipe.
46 int write_pipe(struct m_inode * inode, char * buf, int count)
47 {
48     int chars, size, written = 0;
49
    // If the number of bytes to be written 'count' is still greater than 0, then we loop through
    // the following operations. During the loop operation, if the current pipe is full (free space
    // size = 0), the process waiting for the pipeline is woken up, usually the read pipe process
    // is woken up. If there is no pipe reader, that is, the i-node reference count value is less
    // than 2, the SIGPIPE signal is sent to the current process, and the number of bytes written
    // is returned. If 0 bytes are written, -1 is returned. Otherwise, let the current process sleep
    // on the pipe, waiting for the pipe process to read the data, thus making the pipe free the
    // space. Macros PIPE_SIZE(), PIPE_HEAD(), etc. are defined in the file include/linux/fs.h.
50     while (count>0) {
51         while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
52             wake_up(& PIPE_READ_WAIT(*inode));

```

```

53         if (inode->i_count != 2) { /* no readers */
54             current->signal |= (1<<(SIGPIPE-1));
55             return written?written:-1;
56         }
57         sleep_on(& PIPE_WRITE_WAIT(*inode));
58     }
    // The program executes here, indicating that there is a writable space 'size' in the pipeline
    // buffer. So we take the number of bytes 'chars' from the pipe head pointer to the end of the
    // buffer. The write pipe operation begins from the pipe head pointer. If 'chars' is greater
    // than the number of bytes that need to be written, 'count', make it equal to 'count'. If 'chars'
    // is greater than the free space 'size' in the current pipeline, it is equal to 'size'. Then
    // we let the number of bytes to be written 'count' minus the number of bytes written to this
    // time 'chars', and add the number of bytes written this time to 'written'.
59     chars = PAGE_SIZE-PIPE_HEAD(*inode);
60     if (chars > count)
61         chars = count;
62     if (chars > size)
63         chars = size;
64     count -= chars;
65     written += chars;
    // Then let 'size' point to the pipe head pointer and adjust the current pipe data head pointer
    // (forward chars bytes). Wraps around if the head pointer exceeds the end of the pipe. Then
    // copy the 'chars' bytes from the user buffer to the beginning of the pipe head pointer. For
    // a pipe i-node, the pipe buffer block pointer is in the i_size field.
66     size = PIPE_HEAD(*inode);
67     PIPE_HEAD(*inode) += chars;
68     PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
69     while (chars-->0)
70         ((char *)inode->i_size)[size++] = get fs byte(buf++);
71 }
    // When the write pipe operation ends, the process waiting for the pipe is woken up, and the
    // number of bytes written is returned.
72     wake_up(& PIPE_READ_WAIT(*inode));
73     return written;
74 }
75
76 int sys_pipe(unsigned long * fildes)
77 {
78     struct m_inode * inode;
79     struct file * f[2];           // array of file structures.
80     int fd[2];                    // array of file handles.
81     int i, j;
82
83     // First, take two free items from the system file table (items with a reference count field
84     // of 0), and set the reference count to 1. If there is only one free entry, the item is released
85     // (reset reference count). Returns -1 if no two free items are found.
86     j=0;
87     for(i=0; j<2 && i<NR_FILE; i++)
88         if (!file_table[i].f_count)

```

```

86         (f[j++]=i+file_table)->f_count++;
87     if (j==1)
88         f[0]->f_count=0;
89     if (j<2)
90         return -1;
    // For each of the two file table structure items obtained, a file handle number is assigned,
    // and two items of the process file structure array are respectively pointed to the two file
    // structures, and the file handle is the index number of the array. Similarly, if there is
    // only one free file handle, the handle is released. If no two free handles are found, the
    // two file structure items obtained above are released and -1 is returned.
91     j=0;
92     for(i=0;j<2 && i<NR_OPEN;i++)
93         if (!current->filp[i]) {
94             current->filp[ fd[j]=i ] = f[j];
95             j++;
96         }
97     if (j==1)
98         current->filp[fd[0]]=NULL;
99     if (j<2) {
100         f[0]->f_count=f[1]->f_count=0;
101         return -1;
102     }
    // Then use the function get_pipe_inode() to request an i-node used by the pipe and allocate
    // a page of memory as a buffer for the pipe. If unsuccessful, the two file handles and file
    // structure items are released accordingly, and -1 is returned.
103     if (!(inode=get_pipe_inode())) { // fs/inode.c, line 231.
104         current->filp[fd[0]] =
105             current->filp[fd[1]] = NULL;
106         f[0]->f_count = f[1]->f_count = 0;
107         return -1;
108     }
    // If the pipe i-node application is successful, the two file structures are initialized so
    // that they both point to the same pipe i-node and set the read and write pointers to zero.
    // The file mode of the first file structure is set to read, and the file mode of the second
    // file structure is set to write. Finally, the file handle array is copied to the user space
    // array. If it succeeds, it returns 0 and exits.
109     f[0]->f_inode = f[1]->f_inode = inode;
110     f[0]->f_pos = f[1]->f_pos = 0;
111     f[0]->f_mode = 1; // read */
112     f[1]->f_mode = 2; // write */
113     put_fs_long(fd[0],0+fildes);
114     put_fs_long(fd[1],1+fildes);
115     return 0;
116 }
117
    /// Pipe io control function.
    // Parameters: pino - pipe i node pointer; cmd - control command; arg - arguments.
    // The function returns 0 for success, otherwise it returns an error code.
118 int pipe_ioctl(struct m_inode *pino, int cmd, int arg)
119 {
    // If the command is to get the current readable data length in the pipeline, put the pipe data
    // length value into the location specified by the user argument and return 0. Otherwise an
    // invalid command error code is returned.

```

```
120     switch (cmd) {
121         case FIONREAD:
122             verify_area((void *) arg, 4);
123             put_fs_long(PIPE_SIZE(*pino), (unsigned long *) arg);
124             return 0;
125         default:
126             return -EINVAL;
127     }
128 }
129
```

---

## 12.12 char\_dev.c

### 12.12.1 Function

The char\_dev.c program includes access functions for character device files, mainly rw\_ttyx(), rw\_tty(), rw\_memory(), and rw\_char(). There is also a device read/write function pointer table whose item number represents the major device number.

rw\_ttyx() is a serial terminal device read/write function whose main device number is 4. It implements read and write operations on the serial terminal by calling the tty driver.

rw\_tty() is the console terminal read/write function, and the major device number is 5. The implementation principle is the same as rw\_ttyx(), but it is limited to whether the process can perform console operations.

rw\_memory() is a memory device file read/write function, and the major device number is 1. It implements byte operations on the memory image, but the Linux 0.12 kernel has not implemented the operation of the minor device number 0, 1, and 2, until the 0.96 version begins to implement the read and write operations of the minor device numbers 1 and 2.

rw\_char() is an interface function for character device read and write operations. The other character device performs the operation of the corresponding character device on the character device read/write function pointer table through this function. The file system's operation functions open(), read(), etc. operate on all character device files.

### 12.12.2 Code annotation

Program 12-11 linux/fs/char\_dev.c

---

```
1  /*
2  *  linux/fs/char_dev.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
// <errno.h> Error number header file. Contains various error numbers in the system.
// <sys/types.h> Type header file. The basic system data types are defined.
// <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/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 assembly language.
7 #include <errno.h>
8 #include <sys/types.h>
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h>
12
13 #include <asm/segment.h>
14 #include <asm/io.h>
15
16 extern int tty\_read(unsigned minor, char * buf, int count);    // terminal read.
17 extern int tty\_write(unsigned minor, char * buf, int count);    // terminal write.
18
// Defines the character device read/write function pointer type.
19 typedef (*crw\_ptr)(int rw, unsigned minor, char * buf, int count, off\_t * pos);
20
//// Serial terminal read/write operation function.
// Parameters: rw - read/write command; minor - terminal subdevice number; buf - buffer;
// cout - number of read/write bytes; pos - read/write operation current pointer, this pointer
// is useless for terminal operations.
// Returns: the number of bytes actually read or written. If it fails, it returns an error code.
21 static int rw\_ttyx(int rw, unsigned minor, char * buf, int count, off\_t * pos)
22 {
23     return ((rw==READ)?tty\_read(minor, buf, count):
24             tty\_write(minor, buf, count));
25 }
26
//// Terminal read/write operation functions.
// Same as rw\_ttyx(), but it add detection of whether the process has a control terminal.
27 static int rw\_tty(int rw, unsigned minor, char * buf, int count, off\_t * pos)
28 {
// If the process does not have a corresponding control terminal, an error code is returned.
// Otherwise, the terminal invokes the read/write function rw\_ttyx() and returns the actual
// number of read or write bytes.
29     if (current->tty<0)
30         return -EPERM;
31     return rw\_ttyx(rw, current->tty, buf, count, pos);
32 }
33
//// Read/write memory data. Not implemented.
34 static int rw\_ram(int rw, char * buf, int count, off\_t * pos)
35 {
36     return -EIO;
37 }
38
//// Physical memory data read/write function. Not implemented.
39 static int rw\_mem(int rw, char * buf, int count, off\_t * pos)
40 {

```

```

41         return -EIO;
42     }
43
44     /// Kernel virtual memory data read/write function. Not implemented.
45 static int rw_kmem(int rw, char * buf, int count, off_t * pos)
46 {
47     return -EIO;
48 }
49
50 /// Port read/write operation function.
51 /// Parameters: rw - read or write commands; buf - buffer; cout - number of read/write bytes;
52 /// pos - port address. Returns: the number of bytes actually read or written.
53 static int rw_port(int rw, char * buf, int count, off_t * pos)
54 {
55     int i=*pos;
56
57     /// For the number of bytes required to be read or written, and the port address is less than
58     /// 64k, a single byte read or write operation is performed cyclically. If the command is read,
59     /// one byte of content is read from port i and placed in the user buffer. If the command is
60     /// written, one byte is taken from the user data buffer and output to port i.
61     while (count-->0 && i<65536) {
62         if (rw==READ)
63             put_fs_byte(inb(i), buf++);
64         else
65             outb(get_fs_byte(buf++), i);
66         i++; // increase port address. [??]
67     }
68     /// Then count the number of bytes read/written, adjust the corresponding read and write pointers,
69     /// and return the number of bytes read/written.
70     i -= *pos;
71     *pos += i;
72     return i;
73 }
74
75 /// Memory read/write operation functions. The memory master number is 1. Only the processing
76 /// of the 0-5 subdevices is implemented here.
77 static int rw_memory(int rw, unsigned minor, char * buf, int count, off_t * pos)
78 {
79     /// According to the memory subdevice number, different memory read/write functions are called.
80     switch(minor) {
81         case 0: // device file name is /dev/ram0 or /dev/ramdisk.
82             return rw_ram(rw, buf, count, pos);
83         case 1: // device file name is /dev/ram1 or /dev/mem or ram.
84             return rw_mem(rw, buf, count, pos);
85         case 2: // device file name is /dev/ram2 or /dev/kmem.
86             return rw_kmem(rw, buf, count, pos);
87         case 3: // device file name is /dev/null.
88             return (rw==READ)?0:count; /* rw_null */
89         case 4: // device file name is /dev/port
90             return rw_port(rw, buf, count, pos);
91         default:
92             return -EIO;
93     }
94 }

```

```
81 }
82
83 // Define the number of devices in the system.
84 #define NRDEVS ((sizeof (crw_table))/(sizeof (crw_ptr)))
85
86 // Character device read/write function pointer table.
87 static crw_ptr crw_table[]={
88     NULL,          /* nodev */
89     rw_memory,     /* /dev/mem etc */
90     NULL,          /* /dev/fd */
91     NULL,          /* /dev/hd */
92     rw_ttyx,       /* /dev/ttyx */
93     rw_tty,        /* /dev/tty */
94     NULL,          /* /dev/lp */
95     NULL};         /* unnamed pipes */
96
97 // Character device read/write functions.
98 // Parameters: rw - read or write commands; dev - device number; buf - buffer; count - number
99 // of read or write bytes; pos - read or write pointer.
100 // Returns: the actual number of read/write bytes.
101 int rw_char(int rw,int dev, char * buf, int count, off_t * pos)
102 {
103     crw_ptr call_addr;
104
105     // If the device number exceeds the number of system devices, an error code is returned. If
106     // the device does not have a corresponding read/write function, an error code is also returned.
107     // Otherwise, the read/write operation function of the corresponding device is called, and the
108     // actual number of bytes read/written is returned.
109     if (MAJOR(dev)>=NRDEVS)
110         return -ENODEV;
111     if (!(call_addr=crw_table[MAJOR(dev)]))
112         return -ENODEV;
113     return call_addr(rw, MINOR(dev), buf, count, pos);
114 }
115
```

---

## 12.13 read\_write.c

### 12.13.1 Function

The read\_write.c program implements the file operating system-calls read(), write(), and lseek(). Read() and write() will call the corresponding read or write functions implemented in the previous 4 files according to different file types. Therefore, this program is the upper interface implementation of the functions in the previous four files. lseek () is used to set the file read/write pointer.

The read() system-call first determines the validity of the given parameter, and then checks the type of the file based on the i-node information of the file. If it is a pipe, call the read function in the program pipe.c; if it is a character device file, call the rw\_char() character read function in char\_dev.c; if it is a block device file, execute the block device read operation in the block\_dev.c and return the number of bytes read; if it is a



directory file or a normal formal file, call the file read function `file_read()` in `file_dev.c`. The implementation of the `write()` system-call is similar to `read()`.

The `lseek()` system-call modifies the current read/write pointers in the file structure corresponding to the file handle. For files and pipe files where the read/write pointers cannot be moved, an error code will be given and returned immediately.

## 12.13.2 Code annotation

Program 12-12 linux/fs/read\_write.c

---

```

1  /*
2   *  linux/fs/read_write.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
8  //      and constants.
9  // <errno.h> Error number header file. Contains various error numbers in the system.
10 // <sys/types.h> Type header file. The basic system data types are defined.
11 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
12 //      used functions of the kernel.
13 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
14 //      of the initial task 0, and some embedded assembly function macro statements about the
15 //      descriptor parameter settings and acquisition.
16 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
17 //      segment register operations.
18 #include <sys/stat.h>
19 #include <errno.h>
20 #include <sys/types.h>
21
22 #include <linux/kernel.h>
23 #include <linux/sched.h>
24 #include <asm/segment.h>
25
26 // Character device read/write functions. fs/char_dev.c, line 95.
27 extern int rw_char(int rw,int dev, char * buf, int count, off_t * pos);
28 // Read pipe operation function. fs/pipe.c, line 16.
29 extern int read_pipe(struct m_inode * inode, char * buf, int count);
30 // Write pipe operation function. fs/pipe.c, line 46.
31 extern int write_pipe(struct m_inode * inode, char * buf, int count);
32 // Block device read operation function. fs/block_dev.c, line 56.
33 extern int block_read(int dev, off_t * pos, char * buf, int count);
34 // Block device write operation function. fs/block_dev.c, line 16.
35 extern int block_write(int dev, off_t * pos, char * buf, int count);
36 // Read file manipulation function. fs/file_dev.c, line 17.
37 extern int file_read(struct m_inode * inode, struct file * filp,
38                     char * buf, int count);
39 // Write a file manipulation function. fs/file_dev.c, line 48.
40 extern int file_write(struct m_inode * inode, struct file * filp,
41                     char * buf, int count);
42
43 //// Relocate file read/write pointer system-call.

```

---

```

// The parameter fd is the file handle, offset is the new file read or write pointer offset,
// and origin is the starting position of the offset. There are three options: SEEK_SET (0,
// from the beginning of the file), SEEK_CUR (1, from the current read/ write location), SEEK_END
// (2, from the end of the file).
25 int sys_lseek(unsigned int fd, off_t offset, int origin)
26 {
27     struct file * file;
28     int tmp;
29
// First determine the validity of the parameters provided by the function. If the file handle
// value is greater than the maximum number of files opened by the program NR_OPEN(20), or the
// file structure pointer of the handle is empty, or the i-node field of the corresponding file
// structure is empty, or the specified device file pointer is not positionable, an error code
// is returned and quit. If the i-node corresponding to the file is a pipe node, an error code
// is returned to exit. Because the pipe head and tail pointers are not free to move!
30     if (fd >= NR_OPEN || !(file=current->filp[fd]) || !(file->f_inode)
31         || !IS_SEEKABLE(MAJOR(file->f_inode->i_dev)))
32         return -EBADF;
33     if (file->f_inode->i_pipe)
34         return -ESPIPE;
// Then relocate the file read/write pointer according to the set positioning flag.
// Origin = SEEK_SET, which means that the file read and write pointer is required to be set
// as the origin at the beginning of the file. If the offset value is less than zero, an error
// code is returned, otherwise we set the file read/write pointer equal to offset.
35     switch (origin) {
36         case 0:
37             if (offset<0) return -EINVAL;
38             file->f_pos=offset;
39             break;
// Origin = SEEK_CUR, which means that the read/write pointer is required to be relocated as
// the origin of the file's current read/write pointer. If the current pointer of the file plus
// the offset value is less than 0, an error code is returned. Otherwise, an offset value is
// added to the current read/write pointer.
40         case 1:
41             if (file->f_pos+offset<0) return -EINVAL;
42             file->f_pos += offset;
43             break;
// Origin = SEEK_END, which means that the read/write pointers are required to be relocated
// from the end of the file. At this time, if the file size plus the offset value is less than
// zero, the error code is returned. Otherwise relocate the read/write pointer to the file length
// plus the offset value.
44         case 2:
45             if ((tmp=file->f_inode->i_size+offset) < 0)
46                 return -EINVAL;
47             file->f_pos = tmp;
48             break;
// If the origin setting is invalid, an error code is returned. Finally, the file read/write
// pointer value after relocation is returned.
49         default:
50             return -EINVAL;
51     }
52     return file->f_pos;
53 }

```

```

54  // Read file system-call.
    // The parameter 'fd' is the file handle, 'buf' is the user buffer, and 'count' is the number
    // of bytes to read.
55  int sys_read(unsigned int fd, char * buf, int count)
56  {
57      struct file * file;
58      struct m_inode * inode;
59
    // The function first determines the validity of the parameter. If the file handle value is
    // greater than the program maximum open file number NR_OPEN, or the byte count value to be
    // read is less than 0, or the file structure pointer of the handle is null, an error code is
    // returned and exits. If the number of bytes to be read is equal to 0, then 0 is returned.
60      if (fd >= NR_OPEN || count < 0 || !(file = current->filp[fd]))
61          return -EINVAL;
62      if (!count)
63          return 0;
    // Next verify the buffer memory limit used to hold the data. Then take the i-node of the file
    // and call the corresponding read operation function according to the attribute (mode) of the
    // i-node. If it is a pipe file, and it is in the read pipe mode, the read pipe operation is
    // executed. If it succeeds, the number of bytes read is returned. Otherwise, the error code
    // is returned. If it is a character file, we perform a read character device operation and
    // return the number of characters read. If it is a block device file, a block device read
    // operation is performed and the number of bytes read is returned.
64      verify_area(buf, count);
65      inode = file->f_inode;
66      if (inode->i_pipe)
67          return (file->f_mode & 1) ? read_pipe(inode, buf, count) : -EIO;
68      if (S_ISCHR(inode->i_mode))
69          return rw_char(READ, inode->i_zone[0], buf, count, &file->f_pos);
70      if (S_ISBLK(inode->i_mode))
71          return block_read(inode->i_zone[0], &file->f_pos, buf, count);
    // If it is a directory file or a regular file, first verify the validity of the read byte 'count'
    // and adjust it: if the number of read bytes plus the file current read/write pointer value
    // is greater than the file size, set the number of read bytes to be the file size subtracts
    // the current read/write pointer value; if the read number is equal to 0, 0 is returned. Then
    // perform the file read operation, return the number of bytes read and exit.
72      if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
73          if (count + file->f_pos > inode->i_size)
74              count = inode->i_size - file->f_pos;
75          if (count <= 0)
76              return 0;
77          return file_read(inode, file, buf, count);
78      }
    // If executed here, it means that we can't judge the attributes of the file. Then print the
    // node file mode and return an error code.
79      printk("(Read) inode->i_mode=%06o\n", inode->i_mode);
80      return -EINVAL;
81 }
82
    // Write file system-call.
    // The parameter 'fd' is the file handle, 'buf' is the user buffer, and 'count' is the number
    // of bytes to be written.

```

---

```

83 int sys_write(unsigned int fd, char * buf, int count)
84 {
85     struct file * file;
86     struct m_inode * inode;
87
88     // Similarly, we first check the validity of the function parameters. If the file handle value
89     // is greater than the program maximum open file number NR_OPEN, or the byte count value to
90     // be read is less than 0, or the file structure pointer of the handle is null, an error code
91     // is returned and exits. If the number of bytes to be read is equal to 0, then 0 is returned.
92     if (fd >= NR_OPEN || count < 0 || !(file = current->filp[fd]))
93         return -EINVAL;
94     if (!count)
95         return 0;
96
97     // Then we get the i-node of the file, which is used to call the corresponding write operation
98     // function according to the mode of the i-node. If it is a pipe file and is in the write pipe
99     // mode, the write pipe operation is executed. If it succeeds, the number of bytes written is
100    // returned, otherwise the error code is returned; If it is a character device file, write a
101    // character device operation and return the number of characters written; If it is a block
102    // device file, a block device write operation is performed and the number of bytes written
103    // is returned; If it is a regular file, it performs a file write operation and returns the
104    // number of bytes written, and exits.
105    inode = file->f_inode;
106    if (inode->i_pipe)
107        return (file->f_mode & 2) ? write_pipe(inode, buf, count) : -EIO;
108    if (S_ISCHR(inode->i_mode))
109        return rw_char(WRITE, inode->i_zone[0], buf, count, &file->f_pos);
110    if (S_ISBLK(inode->i_mode))
111        return block_write(inode->i_zone[0], &file->f_pos, buf, count);
112    if (S_ISREG(inode->i_mode))
113        return file_write(inode, file, buf, count);
114
115    // If executed here, it means that we can't judge the attributes of the file. Then print the
116    // i-node file attribute and return an error code to exit.
117    printk("(Write) inode->i_mode=%06o\n", inode->i_mode);
118    return -EINVAL;
119 }

```

---

### 12.13.3 User Program Read/Write Operation

After reading the above program, we should be able to understand how a read/write operation in a user program is performed. Below we take the read function as an example to illustrate how a read file function call in the user program is executed and completed.

Usually, applications do not directly invoke Linux system-calls, but instead call subroutines in a function library (such as libc.a). But if you want to improve some efficiency, you can of course call it directly. For a basic library, you usually need to provide a collection of the following basic functions or subroutines:

- A. System call interface functions;
- B. Memory allocation management functions;
- C. Set of signal processing functions;
- D. String processing functions;
- E. Standard input/output functions;

- F. Other function sets, such as `bsd` functions, encryption and decryption functions, arithmetic operations functions, terminal operation functions, and network socket function sets.

In these function sets, the system-call is the underlying interface function of the operating system. Many functions that involve system calls will invoke system functions with standard names in the system-call interface function set instead of directly using the Linux system-call interface. This can greatly make a function library independent of the operating system it is in, making the function library highly portable and allowing the user program to have higher system independence. For a new library source code, simply replace the part of the system-call (system interface part) with the one of the new operating system, you can basically complete the porting of the function library.

Subroutines in the library can be thought of as an intermediate layer between the application and the kernel system. Their main role is to provide "wrapper functions" for application executing system-calls, in addition to providing functional functions such as computational functions that are not part of the kernel. In this way, the calling interface can be simplified, the interface is more simple and easy to remember, and some parameter verification and error processing can be performed in these wrapping functions, thereby making the program more reliable and stable.

For Linux systems, all input and output operations are done by reading and writing files. Because all peripherals are presented as files in the system, all access between the program and the peripherals can be handled using a uniform file handle. Under normal circumstances, before we can read or write a file, we need to first use the open file operation to notify the operating system of the action to be started. If you want to perform a write operation on a file, you may first need to create this file or delete the previous content in the file. The operating system also needs to check if you have permission to perform these operations. If everything is ok, the open operation will return a file descriptor to the program. The file descriptor will replace the file name to determine the file being accessed. It works just like the file handle in MS-DOS. At this point, all information about an open file is maintained by the system, and the user program only needs to use the file descriptor to access the file.

File read and write operations use `read` and `write` system-calls, respectively, and user programs typically execute these two system-calls by accessing the `read` and `write` functions in the library. The definition of these two functions are as follows:

---

```
int read(int fd, char *buf, int n);
int write(int fd, char *buf, int n);
```

---

The first parameter of these two functions is the file descriptor; the second parameter is a character buffer array for storing data that is read or written; the third parameter is the number of data bytes that need to be read or written. The function return value is the byte count transmitted at the time of one call. For read file operations, the returned value may be smaller than the data you want to read. If the return value is 0, it means that it has reached the end of the file; if it returns -1, it means that the read operation encountered an error. For write operations, the value returned is the number of bytes actually written. If the value is not equal to the value specified by the third argument, this indicates that the write operation encountered an error. For a read function, its implementation in the library is as follows:

---

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

---

---

```
_syscall3(int, read, int, fd, char *, buf, off_t, count)
```

---

Where `_syscall3()` is a macro defined at line 189 of the `include/unistd.h` header file. If the macro is expanded with the specific parameters above, we can get the following code:

---

```
int read(int fd, char *buf, off_t count)
{
    long __res;
    __asm__ volatile (
        "int $0x80"
        : "=a" (__res)
        : "" (__NR_read), "b" ((long)(fd)), "c" ((long)(buf)), "d" ((long)(count)));
    if (__res >= 0)
        return __res;
    errno = -__res;
    return -1;
}
```

---

It can be seen that this expanded macro is a concrete implementation of a read operation function, from which the program enters the system kernel for execution. It uses the embedded assembly statement to execute the Linux system-call interrupt 0x80 with the function number `__NR_read(3)`. This interrupt call returns the actual number of bytes read in the `eax(__res)` register. If the returned value is less than 0, it means that the read operation error occurs, so the error number is inverted and stored in the global variable `errno`, and the value of -1 is returned to the calling program.

In the Linux kernel, read operations are implemented in the file system's `read_write.c` file. When the above system-call interrupt is executed, the `sys_read()` function starting from line 55 in the `read_write.c` file is called. The prototype of the `sys_read()` function is defined as follows:

---

```
int sys_read(unsigned int fd, char *buf, int count)
```

---

This function first determines the validity of the parameter. If the file descriptor value is greater than the maximum number (20) of files opened by the system at the same time, or the size to be read is less than 0, or the file has not been opened yet (the file structure pointer indexed by the file descriptor is null), returns a negative error code. The kernel then verifies that the buffer size that will hold the read data is appropriate. During the verification process, the kernel program verifies the size of the buffer `buf` according to the specified number of read bytes. If the `buf` is too small, the system will expand it. Therefore, if the memory buffer opened by the user program is too small, it is possible to destroy the latter data.

Then the kernel code obtains the i-node of the file from the internal file table corresponding to the file descriptor, and classifies and judges the file according to the flag information in the node, calls the following type of read operation function, and returns the actual read number of bytes.

- ♦ If the file is a pipe file, call the read pipe function `read_pipe()` (implemented in `fs/pipe.c`).
- ♦ If it is a character device file, the read character device function `rw_char()` is called (implemented in `fs/char_dev.c`). The function then calls the character device driver or operates on the memory character device based on the specific character device subtype.
- ♦ If it is a block device file, the block device read function `block_read()` is called (implemented in

fs/block\_dev.c). This function calls the read block function `bread()` in the buffer cache manager `fs/buffer.c`, and finally calls the `ll_rw_block()` function in the block device driver to perform the actual block device read operation.

- ♦ If the file is a normal regular file, the regular file read function `file_read()` (implemented in `fs/file_read.c`) is called to perform the read data operation. This function is similar to the operation of the read block device. Finally, it also calls the underlying driver access function `ll_rw_block()` of the block device where the file system is located, but `file_read()` also needs to maintain information about the internal file table structure, such as moving current pointers of the file.

When the system-call of the read operation returns, the `read()` function in the library can determine whether the operation is correct according to the return value of the system-call. If the returned value is less than 0, it means that the read error occurs, so the error number is inverted and stored in the global variable `errno`, and the value of -1 is returned to the application. See Figure 12-29 for the entire process from the execution of the `read()` function by the user to the actual operation in the kernel.

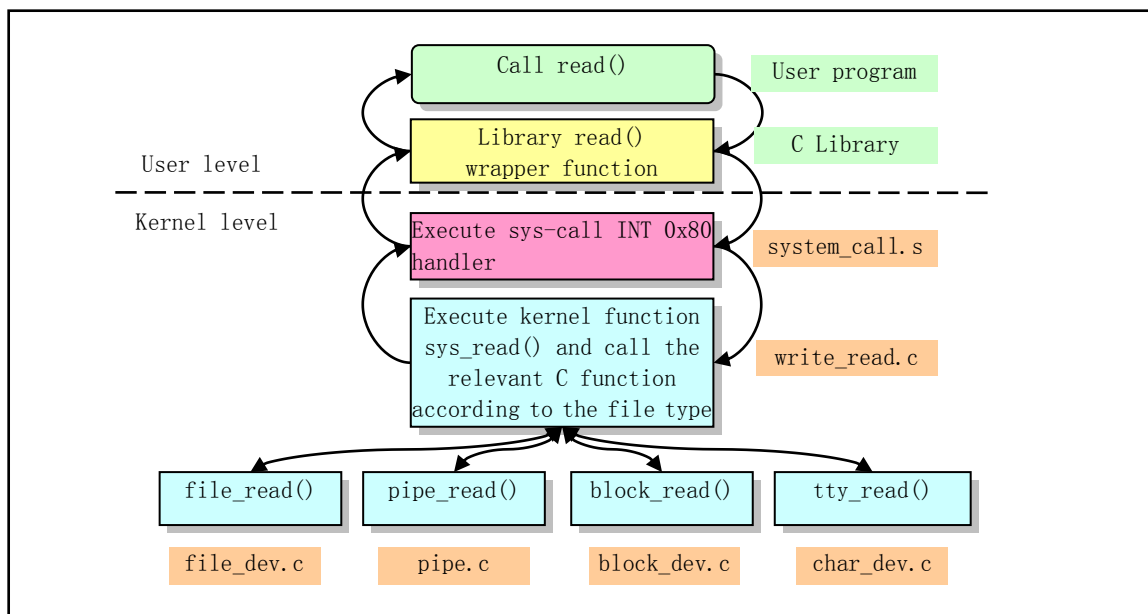


Figure 12-29 Read() function call execution procedure

## 12.14 open.c

From the beginning of this section, all the procedures described are part of the high-level operations and management part of the file system, which is the fourth part of the procedure in this chapter. This section includes five programs, `open.c`, `exec.c`, `stat.c`, `fcntl.c`, and `ioctl.c`.

The `open.c` program mainly includes file access operating system-calls; The `open.c` program mainly includes file access operating system calls; `exec.c` mainly contains program loading and execution functions `execve()`; `stat.c` program is used to obtain status information of a file; `fcntl.c` program implements file access control management; `ioctl.c` program is used to control access to the device.

## 12.14.1 Function

The `open.c` program implements many system calls related to file operations, including file creation, opening and closing, file host and attribute modification, file access permission modification, file operation time modification, and system file system root change.

## 12.14.2 Code annotation

Program 12-13 `linux/fs/open.c`

---

```
1  /*
2   *  linux/fs/open.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  // <string.h> String header file. Defines some embedded functions about string operations.
8  // <errno.h> Error number header file. Contains various error numbers in the system.
9  // <fcntl.h> File control header file. The definition of the operation control constant symbol
10 //     used for the file and its descriptors.
11 // <sys/types.h> Type header file. The basic system data types are defined.
12 // <utime.h> User time header file. The access and modification time structures and the utime()
13 //     prototype are defined.
14 // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
15 //     and constants.
16 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
17 //     of the initial task 0, and some embedded assembly function macro statements about the
18 //     descriptor parameter settings and acquisition.
19 // <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
20 //     communication.
21 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
22 //     used functions of the kernel.
23 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
24 //     segment register operations.
25
26 #include <string.h>
27 #include <errno.h>
28 #include <fcntl.h>
29 #include <sys/types.h>
30 #include <utime.h>
31 #include <sys/stat.h>
32
33 #include <linux/sched.h>
34 #include <linux/tty.h>
35 #include <linux/kernel.h>
36
37 #include <asm/segment.h>
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```



```

// to the ustat structure pointed to by ubuf. The ustat structure is defined in
// include/sys/types.h.
20 int sys_ustat(int dev, struct ustat * ubuf)
21 {
22     return -ENOSYS;           // Return error code, not been implemented yet.
23 }
24
///// Set file access and modification time.
// The parameter filename is the file name, and the times is the access and modification time
// that needs to be set. If the times pointer is not NULL, the time information in the utimbuf
// structure is used to set the access and modification time of the file; if the times pointer
// is NULL, the current time of the system is taken to set the access and modification time
// domain of the specified file.
25 int sys_utime(char * filename, struct utimbuf * times)
26 {
27     struct m_inode * inode;
28     long actime, modtime;
29
// The file's time is saved in its i-node, so we first get the corresponding i-node based on
// the file name. If the supplied access and modification time structure pointer times are not
// NULL, the time value set by the user is read from the structure. Otherwise, the current time
// of the system is used to set the access and modification time of the file.
30     if (!(inode=namei(filename)))
31         return -ENOENT;
32     if (times) {
33         actime = get_fs_long((unsigned long *) &times->actime);
34         modtime = get_fs_long((unsigned long *) &times->modtime);
35     } else
36         actime = modtime = CURRENT_TIME;
// Then modify the access time field and the modification time field in the i-node, set the
// i-node modified flag, put back the i-node, and return 0.
37     inode->i_atime = actime;
38     inode->i_mtime = modtime;
39     inode->i_dirt = 1;
40     iput(inode);
41     return 0;
42 }
43
44 /*
45  * XXX should we use the real or effective uid?  BSD uses the real uid,
46  * so as to make this call useful to setuid programs.
47  */
///// Check the access rights of the file.
// The parameter mode is the checked access attribute. It consists of three valid bits: R_OK
// (value 4), W_OK(2), X_OK(1), and F_OK(0), which indicate that the file being checked is
// readable, writable, executable, or file exists. Returns 0 if access is allowed, otherwise
// returns an error code.
48 int sys_access(const char * filename, int mode)
49 {
50     struct m_inode * inode;
51     int res, i_mode;
52
// The access permission of the file is also stored in the i-node of the file, so we must first

```

```

// obtain the i-node corresponding to the file name. The detected access attribute mode consists
// of the lower 3 bits, so "AND" octal 07 is required to clear all high bits. If the i-node
// corresponding to the file name does not exist, an error code with no permission is returned.
// If the i-node exists, the file attribute code, and the i-node is put back. In addition, the
// statement "iput(inode);" on line 57 is preferably placed after line 61.
53     mode &= 0007;
54     if (!(inode=namei(filename)))
55         return -EACCES; // error code: No access rights.
56     i_mode = res = inode->i_mode & 0777;
57     iput(inode);
// If the current process user is the owner of the file, the file owner attribute is taken;
// otherwise, if the current process user belongs to the same group as the file owner, the file
// group attribute is taken; otherwise, the lowest 3 bits of the res is 'Others' access rights
// of the file.
58     if (current->uid == inode->i_uid)
59         res >>= 6;
60     else if (current->gid == inode->i_gid)
61         res >>= 6; // Here should be "res >>= 3;" [??]
// At this time, the lowest 3 bits of res are the access attribute bits selected according to
// the relationship between the current process and the file. Now let's check these 3 bits.
// If the file has the attribute bit mode queried by the parameter, the access is granted and
// returns 0.
62     if ((res & 0007 & mode) == mode)
63         return 0;
64     /*
65     * XXX we are doing this test last because we really should be
66     * swapping the effective with the real user id (temporarily),
67     * and then calling suser() routine. If we do call the
68     * suser() routine, it needs to be called last.
69     */
// If the current user ID is 0 (super user) and the mask execution bit is 0 or the file can
// be executed and searched by anyone, it returns 0, otherwise it returns an error code.
70     if ((!current->uid) &&
71         (!(mode & 1) || (i_mode & 0111)))
72         return 0;
73     return -EACCES; // error code: No access rights.
74 }
75
//// Change the current working directory.
// The parameter filename is the directory name.
// Returns 0 if the operation is successful, otherwise it returns an error code.
76 int sys_chdir(const char * filename)
77 {
78     struct m_inode * inode;
79
// Changing the current working directory requires that the current working directory field
// of the process task structure be pointed to the i-node of the given directory name. So we
// first take the i-node of the given name. If the i-node is not a directory i-node, the i-node
// is put back and an error code is returned.
80     if (!(inode = namei(filename)))
81         return -ENOENT; // error code: file or directory does not exist.
82     if (!S_ISDIR(inode->i_mode)) {
83         iput(inode);

```

```

84         return -ENOTDIR;           // error code: not a directory name.
85     }
    // Then release the i-node of the original working directory of the process and point it to
    // the newly set working directory i-node and return 0.
86     iput(current->pwd);
87     current->pwd = inode;
88     return (0);
89 }
90
    ///// Change the root directory.
    // Set the specified directory name to be the root directory '/' of the current process.
    // Returns 0 if the operation is successful, otherwise returns an error code.
91 int sys_chroot(const char * filename)
92 {
93     struct m_inode * inode;
94
    // This system-call is used to change the root field in the current process task structure to
    // point to the i-node of the given directory name. If the i-node of the given name does not
    // exist, an error code is returned. If the i-node is not a directory i-node, the i-node is
    // put back and an error code is returned.
95     if (!(inode=namei(filename)))
96         return -ENOENT;
97     if (!S_ISDIR(inode->i_mode)) {
98         iput(inode);
99         return -ENOTDIR;
100    }
    // Then release the original root i-node of the current process and reset it to the i-node of
    // the specified directory name and return 0.
101     iput(current->root);
102     current->root = inode;
103     return (0);
104 }
105
    ///// Modify file attribute (mode).
    // The parameter mode is the new file mode.
    // Returns 0 if the operation is successful, otherwise returns an error code.
106 int sys_chmod(const char * filename, int mode)
107 {
108     struct m_inode * inode;
109
    // This function sets a new access mode for the specified file. Because the mode of the file
    // is in the i-node of the file, we need to first get the i-node corresponding to the file name.
    // If the effective user id of the current process is different from the user id of the file
    // i-node and is not a superuser, then the file i-node is put back and an error code is returned
    // (no access rights).
110     if (!(inode=namei(filename)))
111         return -ENOENT;
112     if ((current->euid != inode->i_uid) && !suser()) {
113         iput(inode);
114         return -EACCES;
115     }
    // Otherwise, the given mode is used to modify the file mode of the i-node, and the i-node modified
    // flag is set, the i-node is put back, and 0 is returned.

```

```

116     inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777);
117     inode->i_dirt = 1;
118     iput(inode);
119     return 0;
120 }
121
122     // Modify the file owner.
123     // The parameter uid is the user identifier (user ID) and the gid is the group ID.
124     // Returns 0 if the operation is successful, otherwise returns an error code.
125 int sys\_chown(const char * filename, int uid, int gid)
126 {
127     struct m\_inode * inode;
128
129     // This call is used to set the user and group IDs in the file i-node, so we first need to get
130     // the i-node of the given file name. If the current process is not a superuser, put back the
131     // i-node and return an error code (no access rights).
132     if (!(inode=namei(filename)))
133         return -ENOENT;
134     if (!suser()) {
135         iput(inode);
136         return -EACCES;
137     }
138     // Otherwise, we use the value provided by the parameter to set the user ID and group ID of
139     // the file i-node, and set the i-node to modify the flag, put back the i-node, and return 0.
140     inode->i_uid=uid;
141     inode->i_gid=gid;
142     inode->i_dirt=1;
143     iput(inode);
144     return 0;
145 }
146
147     // Check and set terminal character device properties.
148     // This auxiliary function is only used for the following system-call sys\_open(). It is used
149     // to check the modification and setting of the current process attributes and system tty table
150     // when the open file is a tty terminal character device.
151     // Returns 0 if the operation is successful. A return of -1 indicates a failure, and the
152     // corresponding character device cannot be opened.
153 static int check\_char\_dev(struct m\_inode * inode, int dev, int flag)
154 {
155     struct tty\_struct *tty;
156     int min; // subdevice number.
157
158     // This function only handles situations where the major device number is 4 (/dev/ttyxx file)
159     // or 5 (/dev/tty file). The subdevice number of /dev/tty is 0. In fact, if a process has a
160     // controlling terminal, /dev/tty is the synonymous name of the process control terminal device,
161     // that is, the /dev/tty device is a virtual device, which corresponds to one of the /dev/ttyxx
162     // devices actually used by the process. For a process, if it has a control terminal, the tty
163     // field in its task structure will be a sub-device number of device No.4.
164     // Now let's get the control terminal device of the current process and temporarily store it
165     // in the min variable for later use. If the parameter specifies device 5 (/dev/tty), then let
166     // min be equal to the tty field value in the process structure, that is, get the sub-device
167     // number of device No. 4. Otherwise, if a sub-device of a device No. 4 is given in the parameter,
168     // the sub-device number is directly taken. If the obtained sub-device number of device No.

```

```

// 4 is less than 0, then the process does not have a control terminal, or the device number
// is incorrect, then -1 is returned.
144     if (MAJOR(dev) == 4 || MAJOR(dev) == 5) {
145         if (MAJOR(dev) == 5)
146             min = current->tty;
147         else
148             min = MINOR(dev);
149         if (min < 0)
150             return -1;
// The primary pseudo terminal device can only be used exclusively by a process. Therefore,
// if the sub-device number indicates a primary pseudo terminal and the open file i-node reference
// count is greater than 1, it indicates that the device has been used by other processes. so
// the character device can no longer be opened again, and -1 is returned. Otherwise, we let
// the pointer tty point to the corresponding structure item in the tty table.
// If the open file operation flag does not contain the flag O_NOCTTY, and the process
// is the group leader, and has no control terminal, and the session field in the tty structure
// is 0 (indicating that the terminal is not yet the control terminal of any process group),
// then It is allowed to set this terminal device min for the process as its control terminal.
// Therefore, we set the terminal device number field tty value in the process task structure
// to be equal to min, and set the session number and group number of the tty structure to be
// equal to the session number and group number of the process, respectively.
151         if ((IS_A_PTY_MASTER(min)) && (inode->i_count>1))
152             return -1;
153         tty = TTY_TABLE(min);
154         if (!(flag & O_NOCTTY) &&
155             current->leader &&
156             current->tty<0 &&
157             tty->session==0) {
158             current->tty = min;
159             tty->session= current->session;
160             tty->pgrp = current->pgrp;
161         }
// If the open file operation flag contains the O_NONBLOCK flag, we need to make the relevant
// settings for the character terminal device: To satisfy the read operation, the minimum number
// of characters to be read is 0, the timeout timing value is set to 0, and the terminal device
// is set to the non-canonical mode. The non-blocking mode works only in non-canonical mode.
// In this mode, when both VMIN and VTIME are set to 0, no matter how many characters are in
// the auxiliary queue, the process reads how many characters and returns immediately.
162         if (flag & O_NONBLOCK) {
163             TTY_TABLE(min)->termios.c_cc[VMIN] =0;
164             TTY_TABLE(min)->termios.c_cc[VTIME] =0;
165             TTY_TABLE(min)->termios.c_lflag &= ~ICANON;
166         }
167     }
168     return 0;
169 }
170
//// Open (or create) a file.
// The parameter 'flag' is the open file flag, which can take values: O_RDONLY, O_WRONLY or
// O_RDWR, and O_CREAT, O_EXCL, O_APPEND and other combinations of flags. If this call creates
// a new file, mode is used to specify the permission properties of the file. These attributes
// are S_IRWXU (the file owner can read, write, and execute), S_IRUSR (user can read), S_IRWXG
// (group members can read, write, and execute), and so on. For newly created files, these

```

```

// properties apply only to future access to the file. An open call that creates a read-only
// file will also return a readable and writable file handle. If the call operation succeeds,
// the file handle (file descriptor) is returned, otherwise an error code is returned. See
// the files sys/stat.h, fcntl.h.
171 int sys_open(const char * filename, int flag, int mode)
172 {
173     struct m_inode * inode;
174     struct file * f;
175     int i, fd;
176
// The parameters are processed first. We combine the file mode set by the user with the process
// mode mask to generate the allowed file mode. In order to create a file handle for an open
// file, we need to search the array of file structure pointers in the process structure to
// find a free entry. The index number fd of the free item is the handle value. If there is
// no free item, an error code is returned (invalid parameter).
177     mode &= 0777 & ~current->umask;
178     for(fd=0 ; fd<NR_OPEN ; fd++)
179         if (!current->filp[fd]) // free item found.
180             break;
181     if (fd>=NR_OPEN)
182         return -EINVAL;
// Then we set the current process's close_on_exec file handle bitmap and reset the corresponding
// bit. close_on_exec is a bitmap flag for all file handles of a process. Each bit represents
// an open file descriptor that determines the file handle that needs to be closed when the
// system-call execve() is called. When a program creates a child process using the fork()
// function, it usually calls the execve() function in the child process to load another new
// program, and the new program starts executing in the child process. If the bit in the
// close_on_exec of a file handle is set, the corresponding file handle will be closed when
// execve() is executed, otherwise the file handle will always be open.
// When a file is opened, the file handle is also open by default in the child process, so the
// corresponding bit needs to be reset here. Then we look for a free entry in the file table
// (the entry with a reference count of 0) for the open file, and we let f point to the file
// table array, and start searching. In addition, the pointer assignment "0+file_table" on line
// 184 is equivalent to "file_table" and "&file_table[0]", but this may be more clear.
183     current->close_on_exec &= ~(1<<fd);
184     f=0+file_table;
185     for (i=0 ; i<NR_FILE ; i++,f++)
186         if (!f->f_count) break; // idle item found.
187     if (i>=NR_FILE)
188         return -EINVAL;
// At this point we let the file structure pointer of the process's file handle fd point to
// the searched file structure, increment the file reference count by 1, and then call the
// function open_namei() to perform the open operation. If the return value is less than 0,
// it indicates an error, so the file structure just applied is released, and the error code
// is returned. If the file open operation is successful, the inode is the i-node pointer of
// the opened file.
189     (current->filp[fd]=f)->f_count++;
190     if ((i=open_namei(filename, flag, mode, &inode))<0) {
191         current->filp[fd]=NULL;
192         f->f_count=0;
193         return i;
194     }
// Based on the mode field of the opened file i-node, we can know the type of the file. For

```

```

// different types of files, we need to do some special processing. If the character device
// file is opened, then we need to call the check_char_dev() function to check if the current
// process can open the file. If allowed (the function returns 0), then in the check_char_dev(),
// the control terminal will be set for the process according to the file open flag; if the
// character device file is not allowed to be opened, then we can only release the file item
// and handle resource requested above, and returns the error code.
195 /* ttys are somewhat special (ttyxx major==4, tty major==5) */
196     if (S_ISCHR(inode->i_mode))
197         if (check_char_dev(inode, inode->i_zone[0], flag)) {
198             iput(inode);
199             current->filp[fd]=NULL;
200             f->f_count=0;
201             return -EAGAIN;          // resource is temporarily unavailable.
202         }
// If a block device file is open, check if the disc has been replaced. If it is replaced, all
// the buffer blocks of the device in the cache buffer need to be invalidated.
203 /* Likewise with block-devices: check for floppy_change */
204     if (S_ISBLK(inode->i_mode))
205         check_disk_change(inode->i_zone[0]);
// Now we initialize the file structure of the opened file. Set the file structure properties
// and flags, set the handle reference count to 1, and set the i-node field to the i-node of
// the open file, the file read-write pointer to 0, and finally the file handle is returned.
206     f->f_mode = inode->i_mode;
207     f->f_flags = flag;
208     f->f_count = 1;
209     f->f_inode = inode;
210     f->f_pos = 0;
211     return (fd);
212 }
213
///// Create file system call.
// The parameter pathname is the path name, and the mode is the same as the sys_open() above.
// If successful, the file handle is returned, otherwise the error code is returned.
214 int sys_creat(const char * pathname, int mode)
215 {
216     return sys_open(pathname, O_CREAT | O_TRUNC, mode);
217 }
218
// Close file system call.
// The parameter fd is the file handle.
// Returns 0 if successful, otherwise returns an error code.
219 int sys_close(unsigned int fd)
220 {
221     struct file * filp;
222
// First check the validity of the parameters. If the given file handle value is greater than
// the number of files NR_OPEN that the program can open at the same time, an error code (invalid
// parameter) is returned. Then reset the corresponding bit of the file handle in the
// close_on_exec bitmap of the process. In addition, if the file structure pointer of the file
// handle is NULL, an error code is returned.
223     if (fd >= NR_OPEN)
224         return -EINVAL;
225     current->close_on_exec &= ~(1<<fd);

```

```
226         if (!(filp = current->filp[fd]))
227             return -EINVAL;
// Now we set the file structure pointer of the file handle to NULL. If the handle reference
// count in the file structure is already 0 before the file is closed, the kernel error occurs
// and the machine is stopped. Otherwise, the reference count of the file structure is decremented
// by 1. At this point if it is not yet 0, then there are other processes that are using the
// file, so it returns 0 (success). If the reference count is equal to 0 now, the file has no
// process references and the file structure has become free. Then release the file i-node and
// return 0.
228         current->filp[fd] = NULL;
229         if (filp->f_count == 0)
230             panic("Close: file count is 0");
231         if (--filp->f_count)
232             return (0);
233         iput(filp->f_inode);
234         return (0);
235     }
236 }
```

---

## 12.15 exec.c

### 12.15.1 Function

The exec.c program implements the loading and execution of binary executables and shell script files. The main function is do\_execve(), which is the C handler for the system-call interrupt (int 0x80) function number \_\_NR\_execve(), and the kernel implementation function for the exec() function cluster. The other five related exec functions are generally implemented in the library function, and eventually this function is called.

The Linux kernel version 0.12 only supports executable files in the a.out format. This format is simple and straightforward and is suitable for introductory learning. The execution file header of the a.out format holds an exec data structure, as shown below. It describes the basic format and content of the executable file. See the include/a.out.h header file in Chapter 14 for a detailed description of the a.out format.

---

```
// The header structure of the executable.
6 struct exec {
7     unsigned long a_magic;           /* Use macros N_MAGIC, etc for access */
8     unsigned a_text;                /* length of text, in bytes */
9     unsigned a_data;                /* length of data, in bytes */
10    unsigned a_bss;                  /* length of uninitialized data area for file, in bytes */
11    unsigned a_syms;                 /* length of symbol table data in file, in bytes */
12    unsigned a_entry;                /* start address */
13    unsigned a_trsize;               /* length of relocation info for text, in bytes */
14    unsigned a_drsize;               /* length of relocation info for data, in bytes */
15 };
```

---

When a program creates a child process using the fork(), one of the exec() cluster functions is usually



called in the child process to load and execute another new program. At this point, the code, data segment (including heap, stack content) of the child process will be completely replaced by the new program, and the new program will be executed in the child process. The main uses of the `execve()` function are:

- Perform initialization operations on command line parameters and environment parameter space pages -- set the initial space start pointer; initialize the space page pointer array to (NULL); get the i-node of the file according to the execution file name; calculate the number of parameters and environment variables number; check file type and execute permission;
- According to the header data structure at the beginning of the executable file, the information is processed therein - the file header is read according to the executed file i-node; If it is a shell script (the first line starts with #!), analyze the shell program name and its parameters, and execute the shell program with the executable file as a parameter; check whether it is executable according to the magic number of the file and the length of the segment;
- The initialization operation before running the new file on the current calling process -- point to the i-node of the new executable file; reset the signal processing handle; set the local descriptor base address and segment length according to the header information; set the parameter and environment parameter page pointer; modify the contents of each field of the process.
- Replace the return address of the original call `execve()` program on the stack with the new executable running address and run the newly loaded executable program.

During the execution of `execve()`, the kernel clears the page directory and page table entries of the original program copied by `fork()` and releases the corresponding pages. The kernel only re-sets the information in the process structure for the newly loaded program code, and requests and maps the memory pages occupied by the command line parameters and environment parameter blocks, and sets the code execution point. At this point, the kernel does not immediately load the code and data of the new program from the block device where the executable file is located. When `execve()` exits and returns, the new program starts executing, but the initial execution will definitely cause a page fault exception to occur. Because the code and data have not yet been read into memory from the block device. At this time, the page fault exception processing procedure requests a memory page (memory frame) for the new program in the main memory area according to the linear address causing the exception, and reads the specified page from the block device, and also set corresponding memory page directory entries and page table entries for the linear address. This method of loading executable files is called load on demand, as described in the memory management chapter.

In addition, since the new program is executed in the child process, the child process is the process of the new program, and the process ID of the new program is the process ID of the child process. Similarly, the properties of the child process become properties of the new program process, and the processing of the open file is related to the close on exec flag of each file descriptor, refer to description in the file `linux/fs/fcntl.c`. Each open file descriptor in the process has an close- on- execution flag. In the process control structure, the flag is represented by an unsigned long integer `close_on_exec`, each bit of which represents the flag for each file descriptor. If a file descriptor is set in the corresponding bit in `close_on_exec`, the descriptor will be closed when `execve()` is executed, otherwise the descriptor will always be open. Unless we use the file control function `fcntl()` to specifically set this flag, the kernel default operation will keep the descriptor open after `execve()` execution.

The meaning of the command line parameters and environment parameters is explained below. When a user types a command at the command prompt, the program specified to execute accepts the typed command

line arguments from that command line. For example, when a user types the following file name list command:

---

```
ls -l /home/john/
```

---

The shell process creates a new process and executes the `/bin/ls` command there. The three parameters `ls`, `-l`, and `/home/john/` on the command line when the `/bin/ls` executable is loaded will be inherited by the new process. In an environment that supports C, when calling the main function `main()` of the program, it takes two arguments.

---

```
int main(int argc, char *argv[])
```

---

The first is the number of arguments on the command line when the program is executed, usually `argc` (argument count); the second is an array of pointers to the string arguments (`argv` -- argument vector). Each string represents a parameter, and the end of the `argv` array always ends with a null. Usually, `argv[0]` is the name of the program being executed, so the value of `argc` is at least 1. For the above example, `argc=3`, `argv[0]`, `argv[1]`, and `argv[2]` are `'ls'`, `'-l'`, and `'/home/john/'`, respectively, and `argv[3] = NULL`, as shown in Figure 12-30.

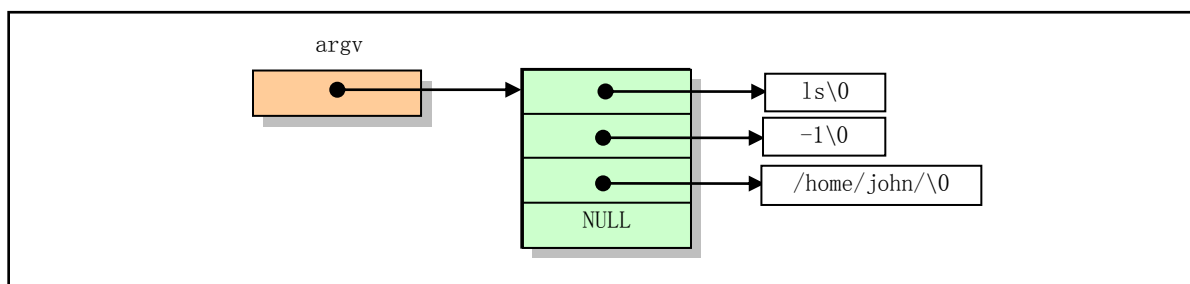


Figure 12-30 Command line argument pointer array `argv[]`

`Main()` also has an optional third parameter that contains environment variable parameters that are used to customize the environment settings of the executable and provide environment setting parameter values for it. It is also an array of pointers to string arguments and ends with `NUL`, except that they are environment variable values. When a program needs to explicitly use an environment variables, `main()` can be declared as:

---

```
int main(int argc, char *argv[], char *envp[])
```

---

The environment string is in the form:

---

```
VAR_NAME=somevalue
```

---

Where `VAR_NAME` represents the name of an environment variable, and the string after the equal sign represents the value assigned to this environment variable. At the command line prompt, type the shell internal command `'set'` to display a list of environment parameters in the current environment. Command line parameters and environment strings are placed at the top of the user stack in the program memory area before the program begins execution, as explained below.

The `execve()` function has a lot of processing operations on command line arguments and environment space. The parameters and environment space of each process or task can have a total of `MAX_ARG_PAGES` pages, and the total size can reach 128kB bytes. The way data is stored in this space is similar to the stack operation, where parameters or environment variable strings are stored backwards from the end of the assumed 128kB space. Initially, the program defines an offset value `p` in the space pointing to the end of the space (128kB-4 bytes). This offset will fall back as the data is stored, as shown in Figure 12-31. As can be seen from the figure, `p` clearly indicates how much free space is left in the current parameter environment space. The `copy_string()` function is used to copy command line arguments and environment strings from the user memory space to the kernel free page. When analyzing `copy_string()`, we can refer to this figure.

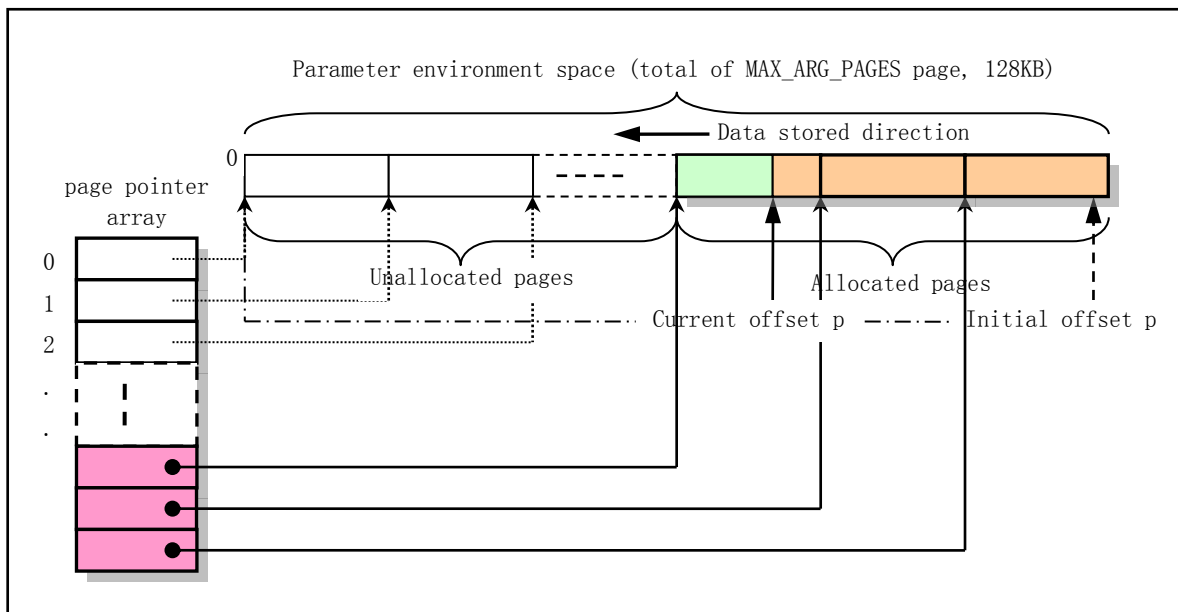


Figure 12-31 Parameter and environment variable string space

After `copy_string()` is executed, the kernel code will adjust `p` to be the parameter and environment variable pointers from the beginning of the process's logical address space, as shown in Figure 12-32. The method is to subtract the size occupied by the parameters and environment variables (128kB - `p`) from the maximum logical space size of 64MB occupied by a process. The left part of `p1` will also use the `create_tables()` function to hold a pointer table of parameters and environment variables, and `p1` will be adjusted to the left again to point to the beginning of the pointer table. Then the resulting pointer is page aligned, and finally the initial stack pointer `sp` is obtained.

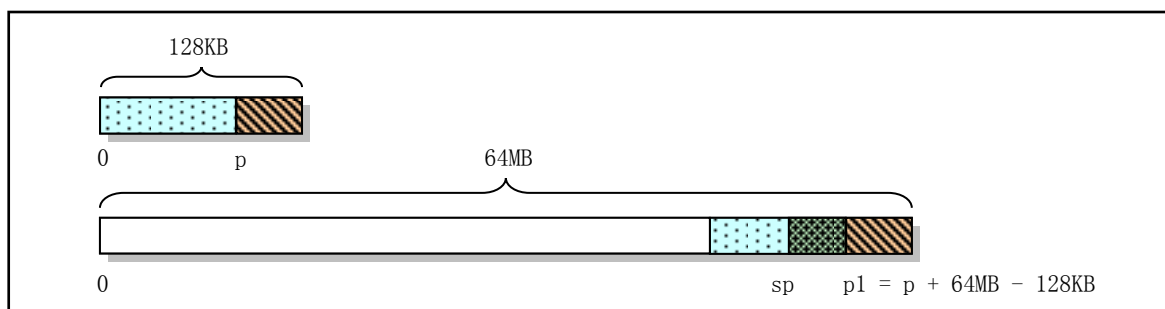


Figure 12-32 Method for converting p to the initial stack pointer

The `create_tables()` function is used to create an environment and parameter variable pointer table in the new program stack based on the given current stack pointer value `p` and the parameter variable value `argc` and the number of environment variables `envc`, and return the adjusted stack pointer value. Then the pointer is page aligned, and finally the initial stack pointer `sp` is obtained. The form of the stack pointer table after creation is shown in Figure 12-33.

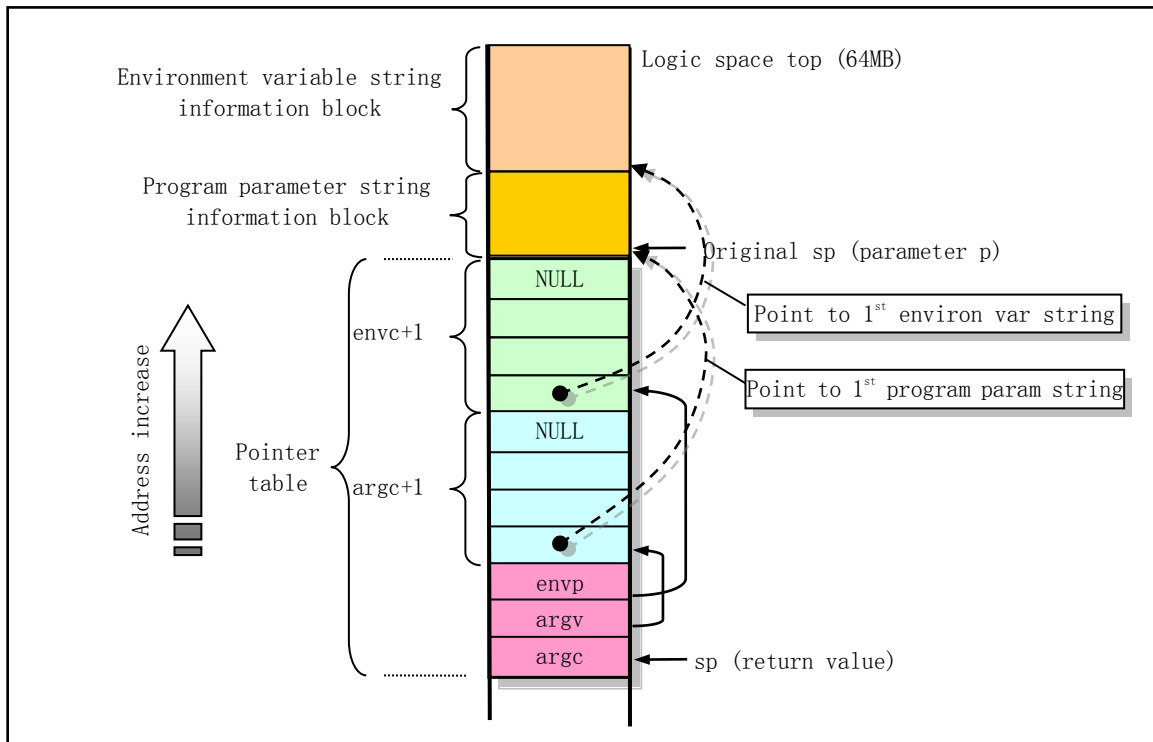


Figure 12-33 Schematic diagram of the pointer table in the new program stack

When the function `do_execve()` returns last, it replaces the code pointer `eip` on the stack of the original call system interrupt program with the code entry point pointing to the new executable program, and replaces the stack pointer with the stack pointer `esp` of the new execution file. Thereafter, the return instruction of this system-call will eventually pop up the data in the stack and cause the CPU to execute the new executable file, as shown in Figure 12-34.

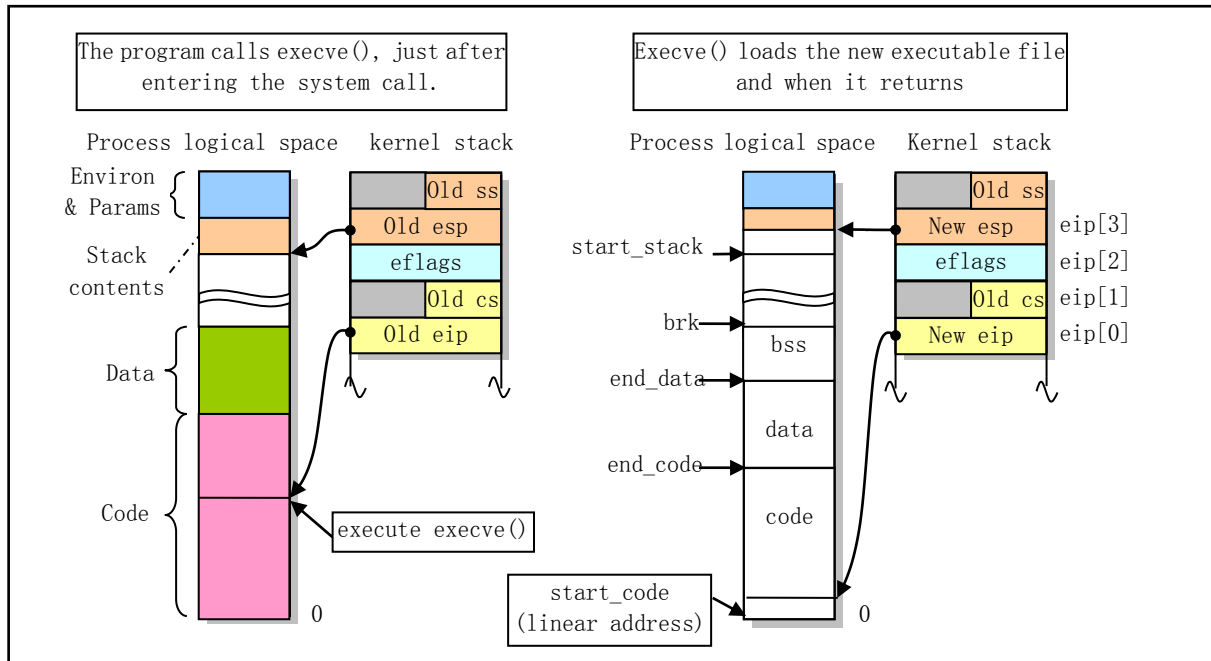


Figure 12-34 Changes in the esp and eip in the stack during loading a execution file

The left half of the figure is the case when the process logic 64MB space still contains the original execution program; the right half is the case when the original execution code and data are released and the stack and code pointers are updated. The shaded (color) section of the figure contains code or data information. The start\_code in the process task structure is the address in the linear space of the CPU, and the remaining variable values are the addresses in the process logical space.

## 12.15.2 Code annotation

Program 12-14 linux/fs/exec.c

```

1  /*
2  *  linux/fs/exec.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  #-checking implemented by tytso.
9  */
10
11 /*
12 *  Demand-loading implemented 01.12.91 - no need to read anything but
13 *  the header into memory. The inode of the executable is put into
14 *  "current->executable", and page faults do the actual loading. Clean.
15 *
16 *  Once more I can proudly say that linux stood up to being changed: it
17 *  was less than 2 hours work to get demand-loading completely implemented.
18 */
19
20 // <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.
// <string.h> String header file. Defines some embedded functions about string operations.
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//      and constants.
// <a.out.h> The a.out header file defines the a.out executable file format and some macros.
// <linux/fs.h> File system header file. Define the file table structure (file,buffer_head,
//      m_inode, etc.).
// <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/mm.h> Memory management header file. Contains page size definitions and some page
//      release function prototypes.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//      segment register operations.
20 #include <signal.h>
21 #include <errno.h>
22 #include <string.h>
23 #include <sys/stat.h>
24 #include <a.out.h>
25
26 #include <linux/fs.h>
27 #include <linux/sched.h>
28 #include <linux/kernel.h>
29 #include <linux/mm.h>
30 #include <asm/segment.h>
31
32 extern int sys\_exit(int exit_code);          // kernel/exit.c, line 365.
33 extern int sys\_close(int fd);                // fs/open.c, line 219.
34
35 /*
36  * MAX_ARG_PAGES defines the number of pages allocated for arguments
37  * and envelope for the new program. 32 should suffice, this gives
38  * a maximum env+arg of 128kB !
39  */
40 #define MAX\_ARG\_PAGES 32
41
42 // Use the library file.
43 // Parameters: library - the name of the library file.
44 // Select a library file for the process and replace the i-node field of the current library
45 // file of the process with the i-node of the library file specified here. If the given library
46 // of the parameter is empty, the current library file of the process is released.
47 // Returns: 0 if successful, otherwise returns an error code.
48 int sys\_uselib(const char * library)
49 {
50     struct m\_inode * inode;
51     unsigned long base;
52
53     // First determine if the current process is a normal process. This is done by looking at the
54     // size of the current process space, because the space size of the normal process is set to
55     // TASK_SIZE (64MB). Therefore, if the size of the process logical address space is not equal

```

```

// to TASK_SIZE, an error code (invalid parameter) is returned, otherwise the library file inode
// is taken. If the library file name pointer is empty, set the inode equal to NULL.
47     if (get_limit(0x17) != TASK_SIZE)
48         return -EINVAL;
49     if (library) {
50         if (!(inode=namei(library)))           /* get library inode */
51             return -ENOENT;
52     } else
53         inode = NULL;
54 /* we should check filetypes (headers etc), but we don't */
// Then put back the process original library file i-node, and preset the process library i-node
// field to be null. Then get the location of the library code of the process, and release the
// page table of the original library code and the memory page occupied. Finally, let the process
// library i-node field point to the new library i-node and return 0 (success). Similar to loading
// an executable file, the actual library code will be loaded into memory when it is actually
// used. In addition, the library file code is placed at the end of the process space, the size
// is a multiple of 4MB, see the linux/sched.h file.
55     iput(current->library);
56     current->library = NULL;
57     base = get_base(current->ldt[2]);
58     base += LIBRARY_OFFSET;                     // linux/sched.h, line 26.
59     free_page_tables(base, LIBRARY_SIZE);
60     current->library = inode;
61     return 0;
62 }
63
64 /*
65  * create_tables() parses the env- and arg-strings in new user
66  * memory and creates the pointer tables from them, and puts their
67  * addresses on the "stack", returning the new stack pointer value.
68  */
// Create a parameter and environment variable pointer table in the new task stack.
// See Figure 12-31 through Figure 12-33 above.
// Parameters: p - the parameter and environment information offset pointer in the data segment;
// argc - the number of parameters; envc - the number of environment variables.
// This function returns the stack pointer.
69 static unsigned long * create_tables(char * p, int argc, int envc)
70 {
71     unsigned long *argv, *envp;
72     unsigned long * sp;
73
// The stack pointer is addressed with a 4-byte boundary, so here we need to make sp a multiple
// of 4. At this point sp is at the end of the parameter environment table. Then we first move
// sp down (low address direction), empty the space occupied by the environment variable pointers
// on the stack, and let the environment pointer envp point to it. One more place here is used
// to store a NULL value at the end. Move sp down again, leaving room for the command line argument
// pointer and pointing the argv pointer to it. Similarly, a more reserved location is used
// to hold a NULL value. At this point sp points to the beginning of the parameter pointer block,
// and then we push the environment parameter block pointer envp and the command line parameter
// block pointer, as well as the number of command line parameters, onto the stack. Note that
// the following pointer plus 1 operation causes sp to increment 4 bytes.
74     sp = (unsigned long *) (0xffffffff & (unsigned long) p);
75     sp -= envc+1;                               // That is sp = sp - (envc+1);

```

```

76     envp = sp;
77     sp -= argc+1;
78     argv = sp;
79     put_fs_long((unsigned long)envp, --sp);      // put in user space.
80     put_fs_long((unsigned long)argv, --sp);
81     put_fs_long((unsigned long)argc, --sp);
    // Then put the pointers of the command line and the pointers of the environment variables into
    // the corresponding places reserved above, and finally place a NULL pointer.
82     while (argc-->0) {
83         put_fs_long((unsigned long) p, argv++);
84         while (get_fs_byte(p++)) /* nothing */; // p points to next param string.
85     }
86     put_fs_long(0, argv);
87     while (envc-->0) {
88         put_fs_long((unsigned long) p, envp++);
89         while (get_fs_byte(p++)) /* nothing */; // p points to next param string.
90     }
91     put_fs_long(0, envp);
92     return sp;                                // returns the current new stack pointer constructed.
93 }
94
95 /*
96  * count() counts the number of arguments/envelopes
97  */
    /// Count the number of parameters.
    // Parameters: argv - an array of parameter pointers, the last pointer is NULL.
    // Counts the number of pointers in the pointer array, and then returns the count value.
98 static int count(char ** argv)
99 {
100     int i=0;
101     char ** tmp;
102
103     if (tmp = argv)
104         while (get_fs_long((unsigned long *) (tmp++)))
105             i++;
106
107     return i;
108 }
109
110 /*
111  * 'copy_string()' copies argument/envelope strings from user
112  * memory to free pages in kernel mem. These are in a format ready
113  * to be put directly into the top of new user memory.
114  *
115  * Modified by TYT, 11/24/91 to add the from_kmem argument, which specifies
116  * whether the string and the string array are from user or kernel segments:
117  *
118  * from_kmem    argv *      argv **
119  * 0            user space  user space
120  * 1            kernel space user space
121  * 2            kernel space kernel space
122  *
123  * We do this by playing games with the fs segment register. Since it

```



```

124 * it is expensive to load a segment register, we try to avoid calling
125 * set_fs() unless we absolutely have to.
126 */
127 //// Copy the parameter strings into the parameter and environment space of the process.
128 //// Parameters: argc - the number of arguments to be added; argv - an array of argument pointers;
129 //// page - an array of arguments to the environment space page pointer; p - an offset pointer
130 //// in the space of the arguments table, always pointing to the header of the copied string;
131 //// from_kmem - the character string source flag.
132 //// In the do_execve() function, p is initialized to point to the last longword in the argument
133 //// table (128kB) space, and the argument strings are reversely copied to it in the stack operation
134 //// mode. Therefore, the p pointer will gradually decrease as the copy information increases,
135 //// and always points to the head of the parameter string. The string source flag from_kmem should
136 //// be a new parameter added by TYT(Tytso) to give execve() the ability to execute script files.
137 //// When execve() does not have the ability to run script files, all parameter strings will be
138 //// in the user data space.
139 //// Returns: the current head pointer of the argument/environment space, or 0 if error.
140 static unsigned long copy_strings(int argc, char ** argv, unsigned long *page,
141 unsigned long p, int from_kmem)
142 {
143     char *tmp, *pag;
144     int len, offset = 0;
145     unsigned long old_fs, new_fs;
146
147     //// First, the current segment register DS (pointing to the kernel data segment) and FS (user
148 //// data segment) values are taken and stored in the variables new_fs and old_fs, respectively.
149 //// If the string and its pointers are in kernel space, set fs to point to kernel space.
150     if (!p)
151         return 0; /* bullet-proofing */ // offset pointer verification
152     new_fs = get_ds();
153     old_fs = get_fs();
154     if (from_kmem==2) // string and string pointers all in kernel space.
155         set_fs(new_fs);
156     //// Then loop through the parameters, start copying backwards from the last parameter, and copy
157 //// to the specified offset address. In the loop, first take the current string pointer that
158 //// needs to be copied. If the string is in user space and the string array (string pointer)
159 //// is in kernel space, first set the FS segment register to point to the kernel data segment
160 //// (DS), and restore the FS immediately after taking the string pointer tmp in the kernel data
161 //// space, otherwise the string pointer is directly taken from the user space to tmp without
162 //// modifying the FS value.
163     while (argc-- > 0) {
164         if (from_kmem == 1) // FS points to kernel space if pointers in it.
165             set_fs(new_fs);
166         if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
167             panic("argc is wrong");
168         if (from_kmem == 1) // if string pointers in kernel.
169             set_fs(old_fs); // FS points back to user space.
170         //// The string is then taken from the user space and the length len of the parameter string is
171 //// calculated, after which tmp points to the end of the string. If the string length exceeds
172 //// the free length remaining in the parameter and environment space at this time, the space
173 //// is not enough. The FS segment register value is then restored (if changed) and returns 0.
174 //// However, this shouldn't happened because the parameters/environment space are 128KB.
175         len=0; /* remember zero-padding */
176         do {

```

```

149         len++;
150     } while (get_fs_byte(tmp++));
151     if (p-len < 0) {                /* this shouldn't happen - 128kB */
152         set_fs(old_fs);
153         return 0;
154     }
    // Then we reversely copy the string char by char to the end of the parameter and environment
    // space. In the process of looping through the characters of a string, we first need to determine
    // whether there is a memory page at the corresponding position in the parameter and environment
    // space. If it does not exist, first apply for a page of memory. The 'offset' is used as the
    // current pointer offset value in a page. Since the offset variable 'offset' is initialized
    // to 0 at the beginning of this function, the following check for '(offset-1 < 0)' is definitely
    // true, so that 'offset' is re-set to the current p pointer within the page range.
155     while (len) {
156         --p; --tmp; --len;
157         if (--offset < 0) {
158             offset = p % PAGE_SIZE;
159             if (from_kmem==2)        // If string in kernel space,
160                 set_fs(old_fs);    // FS points back to user space.
    // If the string space page pointer array item (page[p/PAGE_SIZE]) is zero, it means that the
    // space memory page where the p pointer is located does not exist yet, then you need to apply
    // for a free memory page and fill the page pointer into the array. At the same time, we also
    // make the page pointer pag point to the new page. Returns 0 if no free page is available.
161             if (!(pag = (char *) page[p/PAGE_SIZE]) &&
162                 !(pag = (char *) page[p/PAGE_SIZE] =
163                     (unsigned long *) get_free_page()))
164                 return 0;
165             if (from_kmem==2)        // If string in kernel space,
166                 set_fs(new_fs);    // FS points to kernel space.
167         }
168     }
    // Then copy one byte of the string from the FS segment to the offset of the parameter and
    // environment space in the memory page 'pag'.
169     *(pag + offset) = get_fs_byte(tmp);
170 }
171 }
    // Finally, if the string and string array are in kernel space, the original value of the FS
    // segment register is restored, and then the header offset of the copied parameter in parameter
    // and environment space is returned.
172     if (from_kmem==2)
173         set_fs(old_fs);
174     return p;
175 }
176
    //// Modify the local descriptor table of the task.
    // Modify the segment base address and length limit of the descriptor in the local descriptor
    // table LDT, and place the parameter and environment space page at the end of the data segment.
    // Parameters: text_size - the length limit of the code segment given by the a_text field in
    // the file header; page - an array of parameters and environment space page pointers.
    // Returns: the data segment limit value (64MB).
177 static unsigned long change_ldt(unsigned long text_size, unsigned long * page)
178 {
179     unsigned long code_limit, data_limit, code_base, data_base;

```

```

180         int i;
181
182         // First set the code and data segment length limit to 64MB, and then take the code segment
183         // base address in the code segment descriptor in the LDT of the current process. The code segment
184         // base address is the same as the data segment base address. These new values are then used
185         // to re-set the base and segment lengths in the code segment and data segment descriptors in
186         // the LDT. Please note here that since the code and data segment base address of the new program
187         // being loaded is the same as the original program, there is no need to repeat them. That is,
188         // the two statements on the lines 186, 188 that set the base address of the segment are redundant
189         // and can be omitted.
190         code_limit = TASK_SIZE;
191         data_limit = TASK_SIZE;
192         code_base = get_base(current->ldt[1]);          // include/linux/sched.h, line 277
193         data_base = code_base;
194         set_base(current->ldt[1], code_base);
195         set_limit(current->ldt[1], code_limit);
196         set_base(current->ldt[2], data_base);
197         set_limit(current->ldt[2], data_limit);
198
199         /* make sure fs points to the NEW data segment */
200         // The selector of the local table data segment descriptor (0x17) is put in the FS register,
201         // that is, by default, the FS points to the task data segment. Then put pages with data stored
202         // in parameter and environment space (up to MAX_ARG_PAGES page, 128kB) at the end of the data
203         // segment. The method is: from the beginning of AA, put it backwards page by page. The method
204         // is: From the beginning of the library code location of the process space, put backwards into
205         // the segment page by page. The library file code occupies the end of the process space and
206         // is a multiple of 4MB. The function put_dirty_page() is used to map physical pages into the
207         // process logic space. See the mm/memory.c file.
208         __asm__ ("pushl $0x17\n\ttop %%fs":);
209         data_base += data_limit - LIBRARY_SIZE;
210         for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
211             data_base -= PAGE_SIZE;
212             if (page[i])                // put the page if exists.
213                 put_dirty_page(page[i], data_base);
214         }
215         return data_limit;              // return the length limit of data segment (64MB).
216     }
217
218     /*
219     * 'do_execve()' executes a new program.
220     *
221     * NOTE! We leave 4MB free at the top of the data-area for a loadable
222     * library.
223     */
224
225     /// Load and execute other programs. Called in execve() system-call.
226     // This is a function called by the system call interrupt (int 0x80), function number __NR_execve.
227     // The argument of the function is the value that is gradually pushed onto the stack after
228     // the system-call is executed until the function is called (kernel/system_call.s, line 217).
229     // These values (in system_call.s file) include:
230     // (1) The edx, ecx, and ebx register values pushed onto the stack on line 89--91 correspond
231     // to **envp, **argv, and *filename, respectively; (2) The return address (tmp) of the function
232     // pushed onto the stack when the sys_execve function in sys_call_table is called on line 99;
233     // (3) On line 216, the program code pointer eip that calls the system interrupt that is pushed
234     // onto the stack before calling this function do_execve().

```

```

// Parameters: eip - the program code pointer to call the system interrupt;
// tmp - the return address of the system interrupt when calling _sys_execve, not used;
// filename - the pointer to the executable file name;
// argv - a pointer to an array of command line argument pointers;
// envp - a pointer to an array of environment variable pointers.
// Returns: if successful, it does not return; otherwise sets the error code and returns -1.
207 int do_execve(unsigned long * eip, long tmp, char * filename,
208             char ** argv, char ** envp)
209 {
    // The meaning of some of the following local variables is as follows:
    // Line 213 - An array of arguments & environmental string (A&E) space page pointers.
    // Line 217 - A flag to control if we will execute a shell script file.
    // Line 218 - A integer used to point to the tail of arguments & environmental space.
210     struct m_inode * inode;
211     struct buffer_head * bh;
212     struct exec ex;
213     unsigned long page[MAX_ARG_PAGES];           // array of page pointers in A&E space.
214     int i, argc, envc;
215     int e_uid, e_gid;                             // effective user and group ID.
216     int retval;
217     int sh_bang = 0;                               // control if we execute script file.
218     unsigned long p = PAGE_SIZE * MAX_ARG_PAGES - 4; // p points to the end of the A&E space.
219
    // Before we officially set up the environment for the execution files, let's do some preparatory
    // work. The kernel prepares 128KB (32 pages) of space to store command line arguments and
    // environment string variables for the executable file. The previous line sets p to be initially
    // placed at the last long word of the 128KB space. During the initialization parameter and
    // environment space operations, p will be used to indicate the current location in 128KB space.
    // In addition, the parameter eip[1] is the original user program code segment register CS value
    // that calls this system call, and the segment selector must of course be the code segment
    // selector (0x000f) of the current task. If it is not the value, CS can only be the selector
    // 0x0008 of the kernel code segment. But this is absolutely not allowed, because the kernel
    // code is resident and cannot be replaced. Therefore, the following is based on the value of
    // eip[1] to confirm whether it is normal. Then we initialize the 128KB arguments & environment
    // string space, clear all bytes, and get the i-node of the execution file. Then according to
    // the function parameters, calculate the number of command line parameters argc and the number
    // of environment strings envc. In addition, the executable file must be a regular file.
220     if ((0xffff & eip[1]) != 0x000f)
221         panic("execve called from supervisor mode");
222     for (i=0 ; i<MAX_ARG_PAGES ; i++)             /* clear page-table */
223         page[i]=0;
224     if (!(inode=namei(filename)))                  /* get executables inode */
225         return -ENOENT;
226     argc = count(argv);                           // The number of command line arguments.
227     envc = count(envp);                           // The number of environment variables.
228
229 restart_interp:
230     if (!S_ISREG(inode->i_mode)) {                 /* must be regular file */
231         retval = -EACCES;
232         goto exec_error2;                          // if not regular file, jump to line 376.
233     }

    // Then we check whether the current process has the right to run the specified executable file,
    // that is, according to the mode in the execution file i-node, to see if the process has the

```

```

// permission to execute it. We first check to see if the "set-user-id" flag and the
// "set-group-id" flag are set in the mode. These two flags are mainly used to enable general
// users to execute programs of privileged users (such as super user root), such as passwd,
// which changes the password. If the set-user-id flag is set, the euid of the subsequent
// execution process is set to the user ID of the execution file, otherwise it is set to the
// euid of the current process. If the execution file set-group-id is set, the effective group
// ID (egid) of the execution process is set to the group ID of the execution file, otherwise
// it is set to the egid of the current process. Here, the two determined values are temporarily
// stored in the variables e_uid and e_gid.
234     i = inode->i_mode;
235     e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
236     e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;

// Now compare the euid and egid of the process with the access mode of the executable file.
// If the execution file belongs to the user running the process, the file mode value is shifted
// to the right by 6 bits, and the lowest 3 bits are the access permission flags of the file
// owner. Otherwise, if the execution file belongs to the same group as the user of the current
// process, the attribute is shifted to the right by 3 bits, so that the lowest 3 bits are the
// access permission flags of the execution file group. Otherwise, the lowest 3 bits of the
// mode at this time are the permissions of other users to access the executable file.
// Then we determine whether the current process has permission to run the executable file based
// on the lowest 3-bit value. If the selected user does not have the right to run the file (bit
// 0 is the execution permission), and the other users do not have any rights, or the current
// process user is not the super user, it indicates that the current process does not have the
// authority to run the executable file. Then set the unexecutable error code, and jump to
// exec_error2 to do the exit processing.
237     if (current->euid == inode->i_uid)
238         i >>= 6;
239     else if (in\_group\_p(inode->i_gid))
240         i >>= 3;
241     if (!(i & 1) &&
242         !((inode->i_mode & 0111) && suser())) {
243         retval = -ENOEXEC;
244         goto exec_error2;
245     }

// If the code can be executed here, this means that the current process has permission to run
// the specified executable file. So from here we need to extract the data from the execution
// file header and analyze and set the runtime environment based on the information, or run
// another shell program to execute this script file. First, the first block of the execution
// file is read into the buffer block, and the buffer block data is copied into the ex structure.
// If the first two bytes of the execution file are the characters '#!', it is a script file.
// If we want to run a script file, we need to execute an interpreter for it (such as a shell
// program). Usually the first line of the script file is "#!/bin/bash", which specifies the
// interpreter needed to run the script file. The running method is to take the interpreter
// name and the following parameters (if any) from the first line of the script file, and then
// put these parameters and the script file (the executable file) name into the interpreter's
// command line arguments space.
// Before that, we of course need to put the original command line parameters and environment
// strings specified by the function into the 128KB space, and the command line parameters
// established here are placed in front of them (because they are reversed). Finally, let the
// kernel execute the interpreter of the script file. Then, after setting the parameters such
// as the script file name, the i-node of the interpreter is taken out and jumped to line 229
// to execute the interpreter. Since we need to jump to the executed code line 229, we need

```

```

// to set a flag sh_bang that prohibits the execution of the following script processing code
// again after confirming and processing the script file. In the code that follows, this flag
// is also used to indicate that we have set the command line arguments for the executable file,
// and no need to repeat such settings.
246     if (!(bh = bread(inode->i_dev, inode->i_zone[0]))) {
247         retval = -EACCES;
248         goto exec_error2;
249     }
250     ex = *((struct exec *) bh->b_data);    /* read exec-header */
251     if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
252         /*
253          * This section does the #! interpretation.
254          * Sorta complicated, but hopefully it will work.  -TYT
255          */
256
257         char buf[128], *cp, *interp, *i_name, *i_arg;
258         unsigned long old_fs;
259
260         // From here on, we extract the interpreter name and its parameters from the script file, and
261         // put the interpreter name, its parameters, and script file name into the environment parameter
262         // block. First copy the string after the character '#' on the first line of the script file
263         // into the buffer buf, which contains the script interpreter name (for example /bin/sh), and
264         // possibly several parameters of the interpreter. Then process the contents of buf: replace
265         // the first newline with NULL and remove the space tab.
266         strncpy(buf, bh->b_data+2, 127);
267         brelse(bh); // release the buffer block.
268         iput(inode); // put back the i-node of the script file.
269         buf[127] = '\0';
270         if (cp = strchr(buf, '\n')) {
271             *cp = '\0'; // delete the space, tabs.
272             for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
273         }
274         if (!cp || *cp == '\0') { // if the line contains nothing, error!
275             retval = -ENOEXEC; /* No interpreter name found */
276             goto exec_error1;
277         }
278
279         // At this point, we get a line of content (string) starting with the name of the script
280         // interpreter. The line is analyzed below. First we get the first string, which should be the
281         // interpreter name, at which point i_name points to the name. If there are characters after
282         // the interpreter name, they should be the argument string, so that i_arg points to the string.
283         interp = i_name = cp;
284         i_arg = 0;
285         for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
286             if (*cp == '/')
287                 i_name = cp+1;
288         }
289         if (*cp) {
290             *cp++ = '\0'; // add a NULL to the end of interpreter name.
291             i_arg = cp; // i_arg points to the arguments.
292         }
293         /*
294          * OK, we've parsed out the interpreter name and
295          * (optional) argument.

```

```

285         */
// Now we need to put the parsed interpreter name i_name, its parameter i_arg and script file
// name as parameters into the interpreter's environment and parameter block. But first we need
// to put some of the original argument and environment strings provided by the function first,
// and then put the contents of the first line parsed here. For example, if the command line
// is "example.sh -arg1 -arg2", that is, the execution file is a script file. If the first line
// of content is "#!/bin/bash -iarg1 -iarg2", then after placing the parameters in the first
// line here, the new command line looks like this:
//      "bash -iarg1 -iarg2 example.sh -arg1 -arg2"
// Here we set the sh_bang flag, and then put the original parameters and environment strings
// provided by the function parameters into the space. The number of environment strings and
// arguments are envc and argc-1 respectively. One of the original arguments that are not copied
// is the original executable file name, which is the name of the script file here, and will
// be handled below.
// Please pay attention here! The pointer p gradually moves toward the small address as the
// copy information increases, so after the two copy string functions are executed, the
// environment string block is located above the program command line argument string block,
// and p points to the first of the program argument string. In addition, the last parameter
// (0) of copy_strings() indicates that the parameter string is in user space.
286         if (sh_bang++ == 0) {
287             p = copy_strings(envc, envp, page, p, 0);
288             p = copy_strings(--argc, argv+1, page, p, 0);
289         }
290     /*
291     * Splice in (1) the interpreter's name for argv[0]
292     *          (2) (optional) argument to interpreter
293     *          (3) filename of shell script
294     *
295     * This is done in reverse order, because of how the
296     * user environment and arguments are stored.
297     */
// Next we reverse copy the script file name, interpreter parameters, and interpreter file names
// into the arguments and environment space. If an error occurs, the error code is set and jump
// to exec_error1. In addition, since the script file name provided by this function parameter
// is in user space, the pointer to the script file name given to copy_strings() is in kernel
// space, so the last parameter of this copy string function (string source flag) needs to be
// Set to 1. If the string is in kernel space, the last parameter of copy_strings() needs to
// be set to 2, as shown in lines 301 and 304 below.
298         p = copy_strings(1, &filename, page, p, 1);
299         argc++;
300         if (i_arg) { // copy multiple arguments of the interpreter.
301             p = copy_strings(1, &i_arg, page, p, 2);
302             argc++;
303         }
304         p = copy_strings(1, &i_name, page, p, 2);
305         argc++;
306         if (!p) {
307             retval = -ENOMEM;
308             goto exec_error1;
309         }
310     /*
311     * OK, now restart the process with the interpreter's inode.
312     */

```



```

// Finally, we get the i-node pointer of the interpreter and then jump to line 229 to execute
// the interpreter. In order to get the i-node of the interpreter, we need to use the namei()
// function, but the parameters (filename) used by the function are obtained from the user data
// space, that is, from the space pointed to by the segment register FS. So before calling the
// namei() function, we need to temporarily let FS point to the kernel data space so that the
// function can get the name of the interpreter from kernel space and restore the default settings
// of FS after namei() returns. Then jump to restart_interp (line 229) to reprocess the new
// executable file -- the script file's interpreter.
313         old_fs = get\_fs();
314         set\_fs(get\_ds());
315         if (!(inode=namei(interp))) {           /* get executables inode */
316             set\_fs(old_fs);
317             retval = -ENOENT;
318             goto exec_error1;
319         }
320         set\_fs(old_fs);
321         goto restart_interp;
322     }

// At this point, the execution file header structure data in the buffer block has been copied
// to ex. So we release the buffer block first and start checking the execution header information
// in ex. For this kernel, it only supports the ZMAGIC executable file format, and the execution
// file code is executed from logical address 0, so execution files containing code or data
// relocation information are not supported. Of course, if the executable file is too large
// or the executable file is incomplete, then we can't run it. Therefore, the program will not
// be executed for the following situations: The executable file is not an executable file
// (ZMAGIC), or the code and data relocation portion is not equal to 0, or (code section + data
// section + heap) is longer than 50MB, or the execution file size is less than (code section
// + data section + symbol table + execution header).
323     brelse(bh);
324     if (N\_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
325         ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
326         inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N\_TXTOFF(ex)) {
327         retval = -ENOEXEC;
328         goto exec_error2;
329     }

// In addition, if the code at the beginning of the execution file is not located at the boundary
// of a page (1024 bytes), it cannot be executed. Because the demand paging technique requires
// that the contents of the executable file be loaded in page units, the code and data in the
// executable file image are required to start at the page boundary.
330     if (N\_TXTOFF(ex) != BLOCK\_SIZE) {
331         printk("s: N\_TXTOFF != BLOCK\_SIZE. See a.out.h ", filename);
332         retval = -ENOEXEC;
333         goto exec_error2;
334     }

// If the sh_bang flag is not set, copy the specified number of command line arguments and
// environment strings into the parameter and environment space. If the sh_bang flag is already
// set, it means that the script interpreter will be run, and the environment variable page
// has been copied, no need to copy again. Similarly, if sh_bang is not set and needs to be
// copied, then the pointer p gradually moves toward the small address as the copy information
// increases. Therefore, after the two copy string functions are executed, the environment string
// block is located above the program argument string block, and p points to the first argument
// string of the program. In fact, p is the offset value in the 128KB parameter and environment

```



```

// space, So if p=0, it means that the environment variable and the parameter space page are
// already full.
335     if (!sh_bang) {
336         p = copy\_strings(envc, envp, page, p, 0);
337         p = copy\_strings(argc, argv, page, p, 0);
338         if (!p) {
339             retval = -ENOMEM;
340             goto exec_error2;
341         }
342     }
343 /* OK, This is the point of no return */
344 /* note that current->library stays unchanged by an exec */
// In the previous section, we set the command line arguments and environment space for running
// the execution file according to the information provided by the function parameters, but
// we have not done any substantive work for it, that is, we have not done the initialization
// of process task structure, creating the page table, and etc. Now let's do the job. Since
// the execution file directly uses the "body" of the current process, that is, the current
// process will be transformed into a process that executes the file, we need to first release
// some system resources occupied by the current process, including closing the specified open
// file and releasing occupied page table and memory pages, etc. Then, according to the execution
// file header structure, the content of the descriptor in the local descriptor table LDT used
// by the current process is modified, the length limit of the code segment and the data segment
// descriptor is re-set, and the e_uid and e_gid obtained are used to set the related fields
// in the process task. Finally, the return address eip[] of the program that executes this
// system call is pointed to the beginning of the code in the execution file. This way, when
// the system-call exits and returns, it will run the code of the new executable file.
// Note that although the new executable file code and data have not yet been loaded into memory
// from the file, its parameters and environment blocks have used get\_free\_page\(\) in
// copy\_strings\(\) to get the physical memory page to hold the data, and used put\_page\(\) in
// change\_ldt\(\) to store the data at the end of the process logic space. In addition, in
// create\_tables\(\), a page fault exception is also caused by storing arguments and environment
// pointer tables on the user stack, so that the memory manager also maps physical memory pages
// for the user stack space.
//
// Here we first put back the i-node of the process's original execution program, and let the
// process executable field point to the i-node of the new executable file. Then reset all signal
// processing handles of the original process, but there is no need to reset for the SIG_IGN
// handle. Then according to the close_on_exec bitmap flag, close the specified open file and
// reset the flag.
345     if (current->executable)
346         iput(current->executable);
347     current->executable = inode;
348     current->signal = 0;
349     for (i=0 ; i<32 ; i++) {
350         current->sigaction[i].sa_mask = 0;
351         current->sigaction[i].sa_flags = 0;
352         if (current->sigaction[i].sa_handler != SIG\_IGN)
353             current->sigaction[i].sa_handler = NULL;
354     }
355     for (i=0 ; i<NR\_OPEN ; i++)
356         if ((current->close_on_exec>>i)&1)
357             sys\_close(i);
358     current->close_on_exec = 0;

```

```

// Then, according to the base address and the length limit specified by the current process,
// the physical memory page and the page table itself specified by the page table corresponding
// to the code and data segment of the original program are released. At this moment, the new
// execution file does not occupy any pages in the main memory area, so the page fault abort
// will occur when the processor actually runs the new execution file code. The memory management
// program then performs page fault processing to apply for a memory page and set related page
// table entries for the new execution file, and reads the relevant execution file page into
// the memory. Also, if the "last task used coprocessor" points to the current process, it is
// set to be NULL and the used math flag is reset.
359     free_page_tables(get_base(current->ldt[1]), get_limit(0x0f));
360     free_page_tables(get_base(current->ldt[2]), get_limit(0x17));
361     if (last_task_used_math == current)
362         last_task_used_math = NULL;
363     current->used_math = 0;
// Then we modify the descriptor base address and the segment length in the LDT according to
// the code length field a_text in the new execution header structure, and place the 128KB
// arguments and environment space page at the end of the data segment. After executing the
// following statement, p is now changed to be the offset starting at the beginning of the data
// segment, but still points to the beginning of the data in the arguments and environment space,
// that is, p has been converted to be the stack pointer value. Then call the internal function
// create_tables() to create an environment and parameter variable pointer table in the stack
// space for the program's main() as parameters and return the stack pointer.
364     p += change_ldt(ex.a_text, page);
365     p -= LIBRARY_SIZE + MAX_ARG_PAGES*PAGE_SIZE;
366     p = (unsigned long) create_tables((char *)p, argc, envc);

// Then modify the process field values to become the information of the new executable file.
// That is, the process task structure code tail field end_code is equal to the code segment
// length a_text of the execution file; the data tail field end_data is equal to the code segment
// length of the execution file plus the data segment length (a_data + a_text); and the process
// heap end field brk = a_text + a_data + a_bss. Brk is used to indicate the end position of
// the current data segment (including the uninitialized portion) of the process, and the kernel
// specifies the starting position of the allocation when allocating memory for the process.
// Then set the process stack start field to the page where the stack pointer is located, and
// re-set the effective user id and effective group id of the process.
367     current->brk = ex.a_bss +
368         (current->end_data = ex.a_data +
369         (current->end_code = ex.a_text));
370     current->start_stack = p & 0xfffff000;
371     current->suid = current->euid = e_uid;
372     current->sgid = current->egid = e_gid;
// Eventually, the program pointer that originally invoked the system-call interrupt is replaced
// with the code pointer on the stack to point to the entry point of the new executable, and
// the stack pointer is replaced with the stack pointer of the new execution file. Thereafter,
// the return instruction will pop up the data on the stack and cause the CPU to execute the
// new execution file, so it will not return to the program that originally called the system
// interrupt.
373     eip[0] = ex.a_entry;           /* eip, magic happens :-) */
374     eip[3] = p;                   /* stack pointer */
375     return 0;
376 exec_error2:
377     iput(inode);                 // put back the i-node.
378 exec_error1:

```

```
379     for (i=0 ; i<MAX_ARG_PAGES ; i++)
380         free_page(page[i]);        // release memory pages if error occurs.
381     return(retval);                 // return error code.
382 }
383
```

---

## 12.16 stat.c

### 12.16.1 Function

The stat.c program implements the system-call functions stat() and fstat() for obtaining file status information, and stores the obtained information in the user buffer. The file status information is stored in the stat structure as shown below, where all field information can be obtained from the i-node of the file. Stat() uses the file name to get the information, and fstat() uses the file handle (descriptor) to get the information. Please refer to the file include/sys/stat.h.

---

```
    // File status structure. All fields are available from the file's i-node structure.
6 struct stat {
7     dev_t    st_dev;        // device number that contains the file.
8     ino_t    st_ino;        // file i-node number.
9     umode_t  st_mode;       // file type and mode.
10    nlink_t   st_nlink;      // number of links to the file.
11    uid_t     st_uid;        // user identification number of the file.
12    gid_t     st_gid;        // group number of the file.
13    dev_t     st_rdev;       // device number (if the file is a char or block device file).
14    off_t     st_size;       // file size (bytes) if the file is a regular file.
15    time_t    st_atime;      // last access time.
16    time_t    st_mtime;      // last modified time.
17    time_t    st_ctime;      // i-node last changed time.
18 };
```

---

### 12.16.2 Code annotation

#### Program 12-15 linux/fs/stat.c

---

```
1 /*
2  *  linux/fs/stat.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 // <errno.h> Error number header file. Contains various error numbers in the system.
8 // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
9 //    and constants.
10 // <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,
11 //    m_inode, etc.).
12 // <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/segment.h> Segment operation header file. An embedded assembly function is defined for
// segment register operations.
7 #include <errno.h>
8 #include <sys/stat.h>
9
10 #include <linux/fs.h>
11 #include <linux/sched.h>
12 #include <linux/kernel.h>
13 #include <asm/segment.h>
14
// Use the i-node to get file status information.
// The parameter inode is the file i-node, and the statbuf is the stat file state structure
// pointer in the user data space, which is used to store the obtained state information.
15 static void cp_stat(struct m_inode * inode, struct stat * statbuf)
16 {
17     struct stat tmp;
18     int i;
19
// First verify (or allocate) enough memory space to store the data, and then temporarily copy
// the information on the corresponding file i-node to tmp. Finally, these status information
// is copied into the user buffer.
20     verify_area(statbuf, sizeof (struct stat));
21     tmp.st_dev = inode->i_dev; // device number that contains the file.
22     tmp.st_ino = inode->i_num; // file i-node number.
23     tmp.st_mode = inode->i_mode; // file type and mode.
24     tmp.st_nlink = inode->i_nlinks; // number of links to the file.
25     tmp.st_uid = inode->i_uid; // user id.
26     tmp.st_gid = inode->i_gid; // group id.
27     tmp.st_rdev = inode->i_zone[0]; // device no. (if the file is a char/block file).
28     tmp.st_size = inode->i_size; // file size (bytes).
29     tmp.st_atime = inode->i_atime; // last access time.
30     tmp.st_mtime = inode->i_mtime; // last modified time.
31     tmp.st_ctime = inode->i_ctime; // i-node last changed time.
32     for (i=0 ; i<sizeof (tmp) ; i++)
33         put_fs_byte(((char *) &tmp)[i], i + (char *) statbuf);
34 }
35
// File status system-call function.
// Get related file status information according to the given file name.
// The parameter statbuf is the user buffer pointer for storing state information.
// Returns 0 if successful, and returns an error code if an error occurs.
36 int sys_stat(char * filename, struct stat * statbuf)
37 {
38     struct m_inode * inode;
39
// First, find the corresponding i-node according to the file name, then copy the file state
// information on the i-node into the user buffer, and finally put back the i-node.
40     if (!(inode=namei(filename)))
41         return -ENOENT;

```

```

42     cp\_stat(inode, statbuf);
43     iput(inode);
44     return 0;
45 }
46
47 // Symbolic link file status system-call function.
48 // Get related file status information according to the given file name. If there is a symbolic
49 // link file name in the file path name, the status information of the symbol file itself is
50 // taken, and the link is not followed.
51 // The parameter statbuf is the user buffer pointer for storing file status information.
52 // Returns 0 if successful, and returns an error code if an error occurs.
53 int sys\_lstat(char * filename, struct stat * statbuf)
54 {
55     struct m\_inode * inode;
56
57     // First find the corresponding i-node according to the file name. If it is a symbolic link
58     // file, it does not follow the link. The file status information on the i-node is then copied
59     // into the user buffer, and the i-node is put back.
60     if (!(inode = lnamei(filename))) // get the i-node without following the link.
61         return -ENOENT;
62     cp\_stat(inode, statbuf);
63     iput(inode);
64     return 0;
65 }
66
67 // Use the file handle to get the file state.
68 // Get status information about the file based on the given file handle.
69 // The parameter fd is the handle (descriptor) of the specified file, and the statbuf is the
70 // user buffer pointer that holds the status information.
71 // Returns 0 if successful, and returns an error code if an error occurs.
72 int sys\_fstat(unsigned int fd, struct stat * statbuf)
73 {
74     struct file * f;
75     struct m\_inode * inode;
76
77     // First take the file structure corresponding to the file handle, and then get the i-node of
78     // the file. The file status information on the i-node is then copied into the user buffer.
79     // If the file handle value is greater than the maximum number of files that can be opened by
80     // a program NR_OPEN, or the file structure pointer of the handle is NULL, or the i-node field
81     // of the corresponding file structure is NULL, an error occurs and an error code is returned.
82     if (fd >= NR\_OPEN || !(f=current->filp[fd]) || !(inode=f->f_inode))
83         return -EBADF;
84     cp\_stat(inode, statbuf);
85     return 0;
86 }
87
88 // Read symbolic link file system-call.
89 // This function reads the contents of the symbolic link file (that is, the pathname string
90 // of the file pointed to by the symbolic link) and places it in the user buffer. If the buffer
91 // is too small, the contents of the symbolic link will be truncated.
92 // Parameters: path is the symbolic link file path name; buf is the user buffer; bufsiz is the
93 // buffer length.
94 // Returns: the number of characters in the buffer if successful, or error code if it fails.

```

---

```

69 int sys_readlink(const char * path, char * buf, int bufsiz)
70 {
71     struct m_inode * inode;
72     struct buffer head * bh;
73     int i;
74     char c;
75
76     // First check and verify the validity of the function parameters and adjust them. The user
77     // buffer size bufsi must be between 1 and 1023. Then get the i-node of the symbolic link file
78     // and read the first block of data content of the file. Then put back the i-node.
79     if (bufsiz <= 0)
80         return -EBADF;
81     if (bufsiz > 1023)
82         bufsiz = 1023;
83     verify_area(buf, bufsiz);
84     if (!(inode = lnamei(path)))
85         return -ENOENT;
86     if (inode->i_zone[0])
87         bh = bread(inode->i_dev, inode->i_zone[0]);
88     else
89         bh = NULL;
90     iput(inode);
91     // If the file data content is successfully read, the most bufsiz characters are copied from
92     // the file into the user buffer , and NULL characters are not copied. Finally release the buffer
93     // block and return the number of bytes copied.
94     if (!bh)
95         return 0;
96     i = 0;
97     while (i < bufsiz && (c = bh->b_data[i])) {
98         i++;
99         put_fs_byte(c, buf++);
100     }
101     brelse(bh);
102     return i;
103 }

```

---

## 12.17 fcntl.c

### 12.17.1 Function

The fcntl.c program implements the file control system call fcntl(), and the two file handle (descriptor) duplication system-calls dup() and dup2(). Dup2() specifies the minimum value of the new handle, while dup() returns the unused handle with the smallest current value. Fcntl() is used to modify the state of an opened file, or to copy a file handle. Handle duplication operations are primarily used for standard input/output redirection and pipe operations of files. Some of the constant symbols used in this program are defined in the include/fcntl.h file. It is recommended that you also refer to this header file when reading this program.

The new file handle returned by the functions dup() and dup2() will use the same file table entry as the

original handle being copied or duplicated. For example, when a process does not open any other file, if you use the `dup()` or `dup2()` function but specify a new handle of 3, the file handles after the function is executed are shown in Figure 12-35.

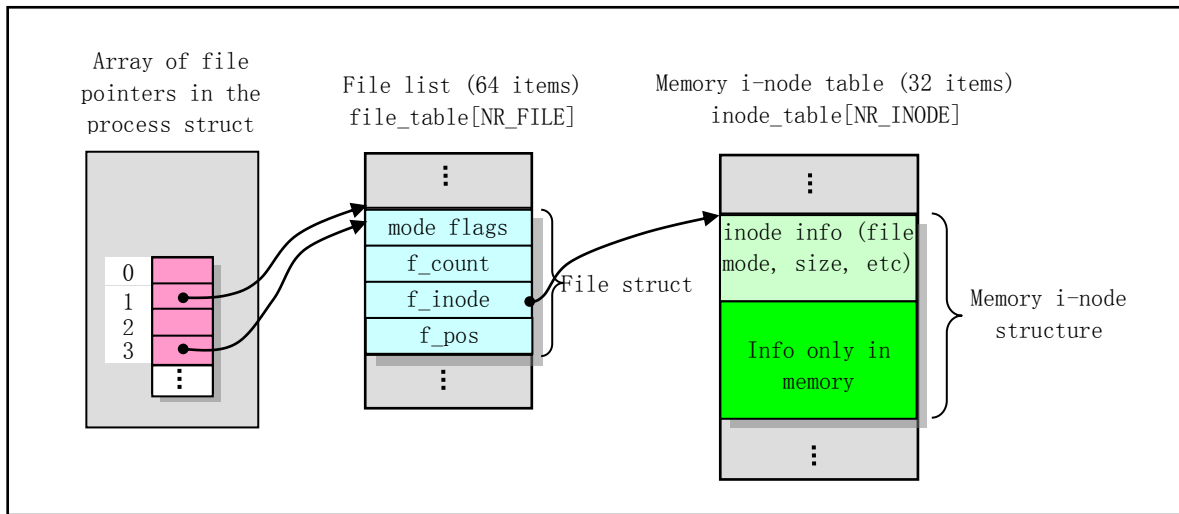


Figure 12-35 File structures after executing `dup(1)` or `dup2(1,3)` function

In addition, it can be seen from the internal function `dupfd()` in this program that for the file handle newly created by `dup()` or `dup2()` function, the flag `close_on_exec` will be cleared, that is, when the `exec()` class function is run, it will not close the file handle created with `dup()`.

The `fcntl()` function, which was adopted by AT&T's System III, is mainly used to modify the properties of an open file. It integrates four functionalities in the function with the control command `cmd` in the parameter:

- `cmd = F_DUPFD`, duplicate the file handle. At this time, the return value of `fcntl()` is a new file handle whose value is greater than or equal to the value specified by the third parameter. The newly created file handle will use the same file entry as the original handle, but its execution close flag bit will be reset. For this command, the function `dup(fd)` is equivalent to `fcntl(fd, F_DUPFD, 0)`; and the function `dup2(fd, newfd)` is equivalent to the statements `"close(newfd); fcntl(fd, F_DUPFD, newfd);"`

- `cmd = F_GETFD` or `F_SETFD`. These two commands are used to read or set the corresponding bits in the flag `close_on_exec` of the file handle. When setting is flag, the third argument of the function is the new bit value of the flag.

- `cmd = F_GETFL` or `F_SETFL`. These two commands are used to read or set file operations and access flags, respectively. These flags are `RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, and `O_NONBLOCK`. For details, see the `include/fcntl.h` file. When setting the operation, the third parameter of the function is the new value of the file operation and access flags, and only the `O_APPEND` and `O_NONBLOCK` flags can be changed.

- `cmd = F_GETLK`, `F_SETLK` or `F_SETLKW`. These commands are used to read or set the file lock flag, but the file record lock function is not implemented in the Linux 0.12 kernel.

## 12.17.2 Code annotation

Program 12-16 `linux/fs/fcntl.c`

1 /\*

```

2  * linux/fs/fcntl.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
// <string.h> String header file. Defines some embedded functions about string operations.
// <errno.h> Error number header file. Contains various error numbers in the system.
// <sys/types.h> Type header file. The basic system data types are defined.
// <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/segment.h> Segment operation header file. An embedded assembly function is defined for
//   segment register operations.
// <fcntl.h> File control header file. The definition of the operation control constant symbol
//   used for the file and its descriptors.
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//   and constants.
7 #include <string.h>
8 #include <errno.h>
9 #include <linux/sched.h>
10 #include <linux/kernel.h>
11 #include <asm/segment.h>
12
13 #include <fcntl.h>
14 #include <sys/stat.h>
15
16 extern int sys\_close(int fd);          // close file system-call. (fs/open.c, 219)
17
//// Duplicate the file handle (file descriptor).
// The parameter fd is the file handle to be duplicated, and arg specifies the minimum number
// of the new file handle.
// Returns a new file handle or an error code.
18 static int dupfd(unsigned int fd, unsigned int arg)
19 {
// The function first checks the validity of the function parameters. If the file handle value
// is greater than the maximum number of open files NR_OPEN for a program, or if the file structure
// of the handle does not exist, an error code is returned; If the specified new handle value
// arg is greater than the maximum number of open files, an error code is also returned. Note
// that the file handle is actually the file structure pointer array item index number.
20     if (fd >= NR\_OPEN || !current->filp[fd])
21         return -EBADF;
22     if (arg >= NR\_OPEN)
23         return -EINVAL;
// Then we look in the process's file structure pointer array for an entry with an index number
// equal to or greater than arg and not yet used. If the new handle found is greater than the
// maximum number of open files NR_OPEN (ie, there are no free items), an error code is returned.
24     while (arg < NR\_OPEN)
25         if (current->filp[arg])
26             arg++;
27     else
28         break;

```



```

29         if (arg >= NR_OPEN)
30             return -EMFILE;
// Otherwise, for the found idle item (handle), the corresponding bit of the handle is reset
// in the flag close_on_exec. That is, when you run the exec() class function, the handle created
// with dup() will not be closed. The file structure pointer is then made equal to the pointer
// to the original handle fd, and the file reference count is incremented by one. Finally, the
// new file handle arg is returned.
31         current->close_on_exec &= ~(1<<arg);
32         (current->filp[arg] = current->filp[fd])->f_count++;
33         return arg;
34     }
35
// Duplicate the file handle system-call2.
// Duplicate the given file handle oldfd, the new file handle value is equal to newfd. If newfd
// is already open, close it first.
// Parameters: oldfd -- the original file handle; newfd -- the new file handle.
// Returns the new file handle.
36 int sys_dup2(unsigned int oldfd, unsigned int newfd)
37 {
38     sys_close(newfd);          // closed the newfd first if already opened.
39     return dupfd(oldfd, newfd); // duplicate and return the new handle.
40 }
41
// Duplicate the file handle system-call.
// Duplicate the given file handle oldfd, the value of the new handle will be the current smallest
// unused handle value.
// Parameters: fildes -- the file handle to be duplicated.
// Returns the new file handle.
42 int sys_dup(unsigned int fildes)
43 {
44     return dupfd(fildes, 0);
45 }
46
// The file control system-call.
// The parameter fd is the file handle; cmd is the control command (see include/fcntl.h, lines
// 23-30); arg has different meanings for different commands. For the duplicate handle command
// F_DUPFD, arg is the minimum value that can be taken for a new file handle; for the set file
// operation and the access flag command F_SETFL, arg is the new file operation and access mode.
// For the file lock commands F_GETLK, F_SETLK, and F_SETLKW, arg is a pointer to the flock
// structure. However, the file locking function is not implemented in this kernel.
// Returns: If an error occurs, all operations return -1. If successful, F_DUPFD returns a new
// file handle; F_GETFD returns the flag close_on_exec of the file handle; F_GETFL returns the
// file operation and access flags.
47 int sys_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)
48 {
49     struct file * filp;
50
// First check the validity of the given file handle, and then we will deal with them according
// to different commands cmd. If the file handle value is greater than the maximum number of
// open files NR_OPEN for a process, or if the file structure pointer for the handle is NULL,
// an error code is returned.
51     if (fd >= NR_OPEN || !(filp = current->filp[fd]))
52         return -EBADF;

```

```
53     switch (cmd) {
54         case F_DUPFD:           // duplicate file handle.
55             return dupfd(fd, arg);
56         case F_GETFD:           // get the close_on_exec flag bit for the file.
57             return (current->close_on_exec >> fd) & 1;
58         case F_SETFD:           // set/reset close_on_exec flag. set if arg is 1.
59             if (arg & 1)
60                 current->close_on_exec |= (1 << fd);
61             else
62                 current->close_on_exec &= ~(1 << fd);
63             return 0;
64         case F_GETFL:           // get the file status flag and access mode.
65             return filp->f_flags;
66         case F_SETFL:           // set status and access mode (append, non-block).
67             filp->f_flags &= ~(O_APPEND | O_NONBLOCK);
68             filp->f_flags |= arg & (O_APPEND | O_NONBLOCK);
69             return 0;
70         case F_GETLK:   case F_SETLK:   case F_SETLKW:       // not implemented.
71             return -1;
72         default:
73             return -1;
74     }
75 }
76
```

---

## 12.18 ioctl.c

### 12.18.1 Function

The `ioctl.c` program implements the input/output control system-call `ioctl()`. The `ioctl()` function can be thought of as an interface control function for each specific device driver. This function will call the IO control function in the driver of the device file specified by the file handle, mainly calling the `tty_ioctl()` function of the tty character device to control the I/O of the terminal. The tty device properties can usually be set in the user program when using the `termios` related function defined by the POSIX.1 standard, see the last part of the `include/termios.h` file. Those functions (such as `tcflow()`) are implemented in the compiled library `libc.a`, and the `ioctl()` function in the program is still executed through the system-call.

### 12.18.2 Code annotation

Program 12-17 `linux/fs/ioctl.c`

---

```
1  /*
2   *  linux/fs/ioctl.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  // <string.h> String header file. Defines some embedded functions about string operations.
8  // <errno.h> Error number header file. Contains various error numbers in the system.
```

```

// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//      and 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.
7 #include <string.h>
8 #include <errno.h>
9 #include <sys/stat.h>
10
11 #include <linux/sched.h>
12
13 extern int tty\_ioctl(int dev, int cmd, int arg);          // chr_drv/tty_ioctl.c, line 133.
14 extern int pipe\_ioctl(struct m\_inode *pino, int cmd, int arg); // fs/pipe.c, line 118.
15
// Define the input and output control (ioctl) function pointer type.
16 typedef int (*ioctl\_ptr)(int dev, int cmd, int arg);
17
// Get the number of device types in the system.
18 #define NRDEVS ((sizeof (ioctl\_table))/(sizeof (ioctl\_ptr)))
19
// Ioctl operation function pointer table.
20 static ioctl\_ptr ioctl\_table[]={
21     NULL,          /* nodev */
22     NULL,          /* /dev/mem */
23     NULL,          /* /dev/fd */
24     NULL,          /* /dev/hd */
25     tty\_ioctl,     /* /dev/ttyx */
26     tty\_ioctl,     /* /dev/tty */
27     NULL,          /* /dev/lp */
28     NULL};         /* named pipes */
29
30
//// The input and output control system-call.
// The function first determines whether the file descriptor given by the parameter is valid,
// then checks the file type according to the file mode in the i-node, and invokes the relevant
// io processing function according to the specific file type.
// Parameters: fd - file descriptor (handle); cmd - command code; arg - parameter.
// Returns: 0 if successful, otherwise returns an error code.
31 int sys\_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
32 {
33     struct file * filp;
34     int dev, mode;
35
// First determine the validity of the given file handle. If the file handle exceeds the number
// of openable files, or the file structure pointer of the corresponding handle is NULL, the
// error code is returned. If the file structure corresponds to the pipe i node, then we need
// to determine whether to execute the pipe IO according to whether the process has the right
// to operate the pipe. Pipe_ioctl() is called if there is permission to execute, otherwise
// an invalid file error code is returned.
36     if (fd >= NR\_OPEN || !(filp = current->filp[fd]))
37         return -EBADF;
38     if (filp->f_inode->i_pipe)
39         return (filp->f_mode&1)?pipe\_ioctl(filp->f_inode, cmd, arg):-EBADF;

```

```
// For other types of files, we take the mode of the corresponding file and determine the type
// of the file accordingly. If the file is neither a character device file nor a block device
// file, an error code is returned. If it is a character or block device file, the device number
// is taken from the i-node of the file. If the device number is greater than the number of
// devices in the system, an error number is returned.
40     mode=filp->f_inode->i_mode;
41     if (!S_ISCHR(mode) && !S_ISBLK(mode))
42         return -EINVAL;
43     dev = filp->f_inode->i_zone[0];           // device no. for device type file.
44     if (MAJOR(dev) >= NRDEVS)
45         return -ENODEV;
// Then, according to the IO control table ioctl_table, the ioctl function pointer of the
// corresponding device is found, and the function is called. If the device does not have a
// corresponding function in the table, an error code is returned.
46     if (!ioctl_table[MAJOR(dev)])
47         return -ENOTTY;
48     return ioctl_table[MAJOR(dev)](dev, cmd, arg);
49 }
50
```

---

## 12.19 select.c

### 12.19.1 Function

Linux programmers often find it necessary to use multiple file descriptors at the same time to access I/O devices where the data stream is intermittently transmitted. If we only use multiple read(), write() calls to handle this situation, then one of the calls may block and let the program wait on a file descriptor. At the same time, other file descriptors may be able to read/write, but they are not processed in time.

There are many ways to solve this problem. One way is to set up a process for each file descriptor that needs to be accessed at the same time. However, this method requires coordination of communication between these processes, so this method is more complicated. Another method is to set all file descriptors to a non-blocking form, and cyclically detect whether each file descriptor has data readable or writable in the program. However, because this loop detection method consumes a lot of processor time, this method is not recommended in multitasking operating systems. The third method is to use asynchronous I/O technology. The principle is to let the kernel use signals to notify the process when a descriptor can be accessed. Since there is only one such "notification" signal for each process, if multiple file descriptors are used, it is still necessary to set each file descriptor to a non-blocking state, and when receiving such a signal, it is necessary to test each descriptor to determine which one is ready.

Another good way to do this is to use the select()(sys\_select()) function in the selcet.c program to handle this situation. The select() function originally appeared in the BSD 4.2 operating system and can be used in operating systems that support the BSD Socket network programming. It is mainly used to handle situations where multiple file descriptors (or socket handles) need to be accessed efficiently at the same time. The main working principle of this function is to let the kernel monitor multiple file descriptors provided by the user at the same time. If the state of the file descriptor has not changed, let the calling process go to sleep state; if one of the descriptors is ready to be accessed, This function returns to the process and tells the process which

descriptor or descriptors are ready.

The `select()` function prototype is defined on line 277 of the `include/unistd.h` file, as shown below:

---

```
int select(int width, fd_set * readfds, fd_set * writefds, fd_set * exceptfds,
          struct timeval * timeout);
```

---

This function uses 5 parameters. The first parameter `width` is the value of the largest descriptor in the three descriptor sets given later plus one. This value is actually the range of values for the kernel code to check the number of descriptors. The next three parameters are pointers to the file descriptor set type `fd_set`, which point to the read operation descriptor set `readfds`, the write operation descriptor set `writefds`, and the descriptor set `exceptfds` where the exception condition occurs. Any of these 3 pointers can be `NULL`, indicating that we don't care about the corresponding set. If all three pointers are `NULL`, then the `select()` function can be used as a more accurate timer (the `sleep()` function can only provide second-level precision).

The file descriptor set type `fd_set` is defined in the `include/sys/types.h` file and is defined as an unsigned long word. Each bit of it represents a file descriptor, and the offset position value of the bit in the long word is the value of the file descriptor, as shown in the upper half of Figure 12-36.

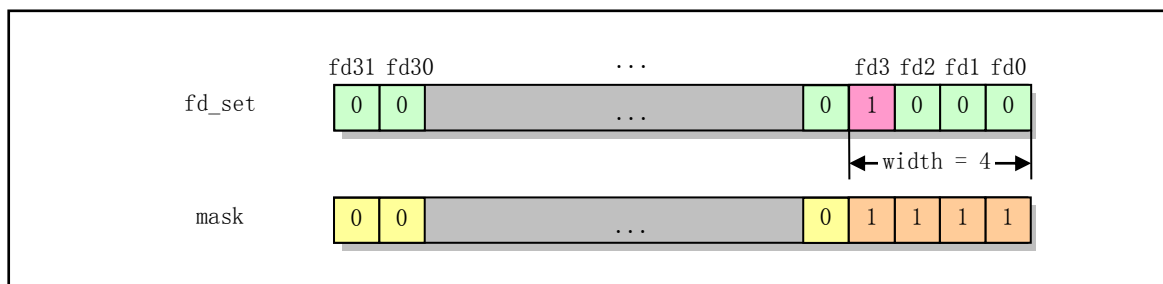


Figure 12-36 File descriptor set represents one descriptor per bit

If we are concerned with the descriptor `fd3` of the read operation, then we need to set `fd3` (bit 4) in the `readfds` set to 1; if we need to monitor the file descriptor `fd1` that performs the write operation, we need to set `fd1` in the `writefds` set to 1. If `fd3` is the largest descriptor value in all descriptor sets, then the first parameter `width` is equal to 4. To facilitate the operation of the `fd_set` type variable, the Linux system provides four macros as shown below, defined in the file `include/sys/time.h`. They are used to clear a descriptor set, set/reset/detect a bit of a given descriptor in a descriptor set.

---

```
#define FD_ZERO(fdsetp)      (*(fdsetp) = 0)           // zero all bits of descriptor set.
#define FD_SET(fd, fdsetp)  (*(fdsetp) |= (1 << (fd))) // set the bit of the descriptor.
#define FD_CLR(fd, fdsetp)  (*(fdsetp) &= ~(1 << (fd))) // reset the bit of the descriptor.
#define FD_ISSET(fd, fdsetp) ((*(fdsetp) >> fd) & 1) // test the bit of the descriptor.
```

---

After declaring a descriptor set variable, the user program should first clear it with `FD_ZERO()` and then use `FD_SET()` or `FD_CLR()` to set/reset the bit corresponding to the specified descriptor. `FD_ISSET` is used to test whether the specified bit of the descriptor set is still set when `select()` returns. When `select()` returns, the bits still set in the three descriptor sets indicate that the corresponding file descriptor is ready (read, write, or exception). Note that these macros need to use the descriptor set pointer as the second argument.

The last parameter of the `select()` function, `timeout`, is used to specify the maximum amount of time that a process will expect to wait for `select()` before any descriptor is ready. It is a pointer to the structure of type

timeval (defined in the include/sys/time.h file), as shown below.

---

```
struct timeval {
    long    tv_sec;        /* seconds */
    long    tv_usec;       /* microseconds */
};
```

---

When the parameter timeout pointer is NULL, it means that we will wait indefinitely until one of the descriptors specified in the descriptor set is ready to operate. However, if the process receives a signal, the wait process will be interrupted, and select() will return -1, and the global variable errno will be set to EINTR.

When the timeout pointer is not NULL, but the value of both fields in the structure is 0, it means no waiting. At this point the select() function can be used to test the state of all specified descriptors and return immediately. When at least one of the two time field values is not 0, the select() function waits for a while before returning. If a descriptor is ready during the wait period, it returns directly, and at this time the two time field values are modified to indicate the remaining wait time value. If no descriptor is ready within the set time, select() returns 0. In addition, it can be interrupted by the signal during the waiting period and returns -1.

In general, when select() returns -1, it indicates an error; when select() returns a value of 0, it means that no descriptor is ready under the specified conditions; when select() returns a positive value, it indicates the number of file descriptors that are ready for access in the descriptor set. At this time, the descriptor corresponding to the bit still set in the three descriptor sets is the ready descriptor.

Since the Linux 0.12 kernel only provides system calls with up to 3 parameters, and select() has 5 parameters, when the user program calls the select() function, the select() in the library file (for example, libc.a) will pass the address of the first parameter as a pointer to the system-call sys\_select() in the kernel. and the system-call will treat the pointer of the first argument as a "buffer" pointer holding all the arguments. It will first break the parameters in the "buffer" and then call the do\_select() function to handle them. Then, when do\_select() returns, the result is written to the user data "buffer". The following is the source code implementation of the select() function in the libc library of the Linux 0.1x system.

---

```
01 #define __LIBRARY__
02 #include <sys/time.h>
03 #include <unistd.h>
04
05 int select(int nd, fd_set * in, fd_set * out, fd_set * ex, struct timeval * tv)
06 {
    // First define the return result variable __res, and define the register variable __foebx
    // as a pointer to the first parameter. Then use the system-call inline assembly code, set
    // eax = select system-call function number; ebx is the first parameter nd pointer.
07     long __res;
08     register long __foebx __asm__ ("bx") = (long) &nd;
09     __asm__ volatile ("int $0x80"
10         : "=a" (__res)
11         : "0" (__NR_select), "r" (__foebx));
    // Finally, if the return value is greater than or equal to 0, the value is returned, otherwise
    // the global error number variable errno is set and then -1 is returned.
12     if (__res >= 0)
13         return (int) __res;
14     errno = -__res;
15     return -1;
```

In fact, the `select.c` program is more complicated. As Mr. Linus said in the 27th line of the program: "If you understand what I'm doing here, then you understand how the Linux sleep/wakeup mechanism works." Similar to the `kernel/sched.c` program, the main difficulty in this program is the understanding of the `add_wait()` and `free_wait()` functions. In order to understand the working principle of these two functions, we can refer to the `sleep_on()` function in the `sched.c` program, because these functions all involve the processing of the task waiting queue of a certain resource. Below we first explain the main working principle of the `sys_select()` system-call, and then detail how the `select()` handles the wait queues.

The code in the `sys_select()` function is mainly responsible for parameter copying and conversion before and after the `select()` function, and the main work of the `select()` operation is done in the `do_select()` function. `Do_select()` will first check the validity of each descriptor in the file descriptor set, then call the function `check_XX()` of the relevant descriptor set to check each descriptor, and also count the number of descriptors currently ready in the descriptor set. If any of the descriptors are ready, the function will return immediately, otherwise the process will invoke the `add_wait()` function to insert the current task into the corresponding wait queue and enter the sleep state in the `do_select()` function. If the process continues to run after a timeout has elapsed or because a process on the wait queue where a descriptor is located is awakened, the process will again check to see if a descriptor is ready. The `do_select()` function uses the `free_wait()` function to wake up the waiting tasks (if any) already on the queue before executing the repeat check operation.

The `select.c` program uses a wait table `wait_table` while processing the descriptor wait process, as shown in lines 37-45 in the program and Figure 12-37 below. The `wait_table` of type `select_table` contains a valid item count field `nr` and an array `entry[NR_OPEN * 3]`, each array item is a `wait_entry` structure. The valid entry field `nr` of `wait_table` records the number of `wait_entry` entries waiting for the descriptor in the descriptor set to wait on the associated wait queue. The `wait_entry` structure contains two fields, where the `wait_address` pointer field is used to point to the task wait queue header corresponding to the descriptor currently being processed, and the `old_task` field is used to point to the wait task that the wait queue header pointer originally points to.

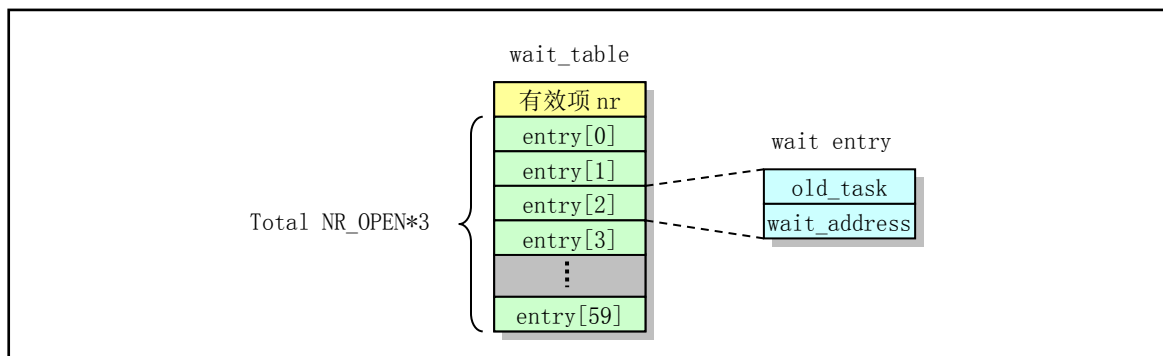


Figure 12-37 Wait table structure diagram

The wait table operates using the `add_wait()` and `free_wait()` functions. When a descriptor is not ready, `add_wait()` is used to add the current process to the task wait queue corresponding to the descriptor. Before adding an item to the wait list, it first searches for the wait queue header pointer field in the wait table that has the same wait item that you want to add. If it already exists, it will not be added to the wait table and will return directly (that is, only one waiting item will be inserted in the different waiting queue), otherwise the `wait_address` field of the wait table item points to the waiting queue head pointer, and the `old_task` field points

to the task that the queue head pointer originally pointed to. Then let the wait queue head pointer point to the current task. Finally, the valid item count value `nr` of the wait table is incremented by one.

For example, for a descriptor whose read buffer queue is empty and waiting for the terminal `tty` to input characters, the corresponding terminal's read buffer queue 'secondary' is provided with a task wait queue header pointer `proc_list` that waits for a readable character in the buffer queue (see `tty` queue structure in file `include/linux/tty.h`, line 26 ). When no characters in the buffer secondary can be read, the `select.c` program will add the current task to the wait table using the `add_wait()` function. It will make the `wait_entry` field `wait_address = proc_list` and let the field `old_task` point to the task that `proc_list` originally pointed to. If `proc_list` did not originally point to any task, then `old_task=NULL`. Then let `proc_list` point to the current task. This process is shown in Figure 12-38. In the figure, (a) shows the original task waiting for the queue head pointer before calling the `add_wait()` function, and (b) shows the form of waiting for the entry after executing `add_wait()`. Note that only one `wait_entry` entry in the wait table is shown in the figure.

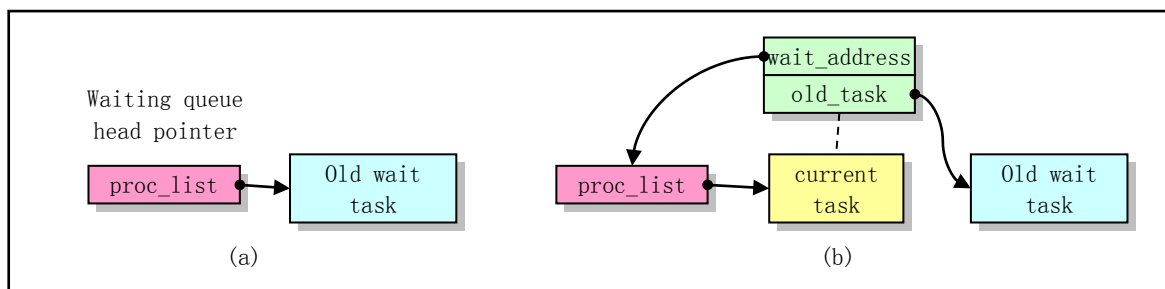


Figure 12-38 Add one wait item to the wait table

If the waiting tasks already in the waiting queue are inserted because the `sleep_on()` function is called, and after we insert the current task calling the `select` function into the waiting queue, another process inserts itself using the `sleep_on()` function. At this moment, the structure of the entire waiting queue is shown in Figure 12-39.

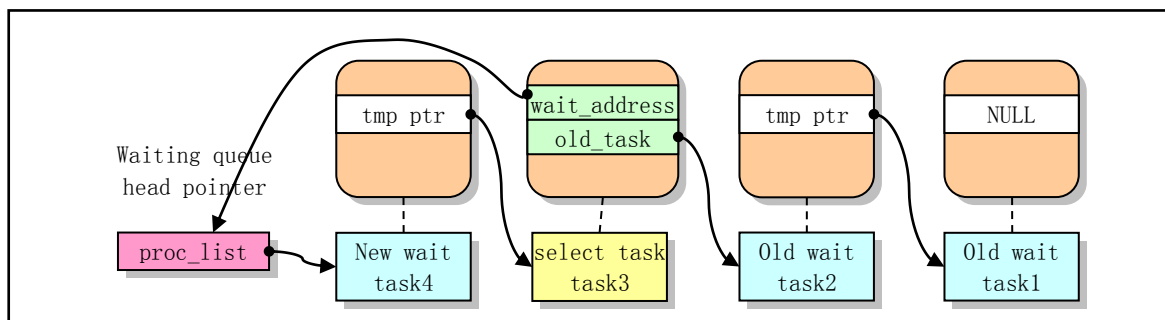


Figure 12-39 A new waiting task is inserted in the waiting queue

As can be seen from the figure, the wait table entry `old_task` pointer field is exactly the same as the `tmp` pointer in the `sleep_on()` function, and the `wait_address` field is only used for `select` to prevent the addition of an entry with the same wait queue pointer in the wait table `wait_table`. Therefore, when using the `free_wait()` function to clear items in the wait table, the algorithm used in `free_wait()` is exactly the same as in the `sleep_on()` function when the task is woken up.

When the waiting resource is available, for example, a character has been entered in the buffer secondary of the `tty` read buffer queue, the task pointed to by the head pointer in the waiting queue will be woken up. The



task will then wake up the task pointed to by its `tmp` pointer. When the task executing `select()` is woken up, it immediately executes the `free_wait()` function (see line 204 of the code). If the task is waking up and waiting queue head pointer is pointing to this task (`*wait_address == current`), then the `free_wait()` function will immediately wake up the subsequent tasks pointed to by `old_task`. It can be seen that the functionality of `free_wait()` is exactly the same as the code of `sleep_on()` waking up tasks. If the task executing the `select` function is awakened and another process calls the `sleep_on()` function and sleeps on the wait queue, then the wait queue head pointer does not point to the current process (`*wait_address != current`) Then we need to wake up these tasks first. The operation method is to set the task pointed to by the queue head to the ready state (`state = 0`) and set itself to the non-interruptible wait state. That is, the task itself has to wait for these subsequent queued tasks to be awakened to start execution, then wakes up itself. Then re-execute the scheduler.

Also note that since `select()` implemented in the Linux kernel will modify (decrement) the field values in the structure pointed to by `timeout` during the run to reflect the remaining wait time, and the `select()` implementation in many other operating systems does not do this, so this will cause Linux programs that access the `timeout` structure value during `select()` runs into problems. Similarly, programs that do not initialize the `timeout` in the loop and use the `select()` function multiple times will encounter problems if they are ported to a Linux system. So when `select()` returns, the structure pointed to by `timeout` should be considered to be in an uninitialized state.

## 12.19.2 Code annotation

Program 12-18 linux/fs/select.c

```

1 /*
2  * This file contains the procedures for the handling of select
3  *
4  * Created for Linux based loosely upon Mathius Lattner's minix
5  * patches by Peter MacDonald. Heavily edited by Linus.
6  */
7
8 // <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,
9 //     m_inode, etc.).
10 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
11 //     used functions of the kernel.
12 // <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
13 //     communication.
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 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
18 //     segment register operations.
19 // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
20 //     descriptors/interrupt gates, etc. is defined.
21 // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
22 //     and constants.
23 // <sys/types.h> Type header file. The basic system data types are defined.
24 // <string.h> String header file. Defines some embedded functions about string operations.
25 // <const.h> The constant file currently defines only the flags of i_mode field in the i-node.
26 // <errno.h> Error number header file. Contains various error numbers in the system.
27 // <sys/time.h> The timeval structure and the itimerval structure are defined.
28 // <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
29 //     manipulation function prototypes.

```

```

8 #include <linux/fs.h>
9 #include <linux/kernel.h>
10 #include <linux/tty.h>
11 #include <linux/sched.h>
12
13 #include <asm/segment.h>
14 #include <asm/system.h>
15
16 #include <sys/stat.h>
17 #include <sys/types.h>
18 #include <string.h>
19 #include <const.h>
20 #include <errno.h>
21 #include <sys/time.h>
22 #include <signal.h>
23
24 /*
25  * Ok, Peter made a complicated, but straightforward multiple_wait() function.
26  * I have rewritten this, taking some shortcuts: This code may not be easy to
27  * follow, but it should be free of race-conditions, and it's practical. If you
28  * understand what I'm doing here, then you understand how the linux sleep/wakeup
29  * mechanism works.
30  *
31  * Two very simple procedures, add_wait() and free_wait() make all the work. We
32  * have to have interrupts disabled throughout the select, but that's not really
33  * such a loss: sleeping automatically frees interrupts when we aren't in this
34  * task.
35  */
36 // Note that each process has its own EFLAGS flag register.
37 typedef struct {
38     struct task_struct * old_task;
39     struct task_struct ** wait_address;
40 } wait_entry;
41
42 typedef struct {
43     int nr;
44     wait_entry entry[NR_OPEN*3];
45 } select_table;
46
47 // Adding the wait queue to the wait table.
48 // Add the wait queue pointer of the not ready descriptor to the wait_table. The parameter
49 // *wait_address is the wait queue header pointer associated with the descriptor. For example,
50 // the wait queue head pointer of the tty read buffer queue is proc_list. The parameter p is
51 // a pointer to the wait table structure defined in do_select().
52 static void add_wait(struct task_struct ** wait_address, select_table * p)
53 {
54     int i;
55
56     // First check if the descriptor has a corresponding wait queue, and if not, return. Then in
57     // the wait table, search for the queue pointer given by the parameter to see if it has already
58     // been set in the wait table. If it was, the code will return immediately. This check is mainly
59     // for pipe file descriptors. For example, if a pipe is waiting to be read, it must be able

```

```

// to write immediately.
51     if (!wait_address)
52         return;
53     for (i = 0 ; i < p->nr ; i++)
54         if (p->entry[i].wait_address == wait_address)
55             return;
// Then we store the header pointer of the descriptor's wait queue in wait_table, and let the
// old_task field of the wait table entry points to the task pointed to by the wait queue header
// pointer (NULL if none), then let the wait queue header points to the current task. Finally,
// the wait table item count value nr is incremented by 1.
56     p->entry[p->nr].wait_address = wait_address;
57     p->entry[p->nr].old_task = * wait_address;
58     *wait_address = current;
59     p->nr++;
60 }
61
///// Free each wait queue in the wait table.
// The parameter p is a pointer to the wait table structure. This function (line 204, 207) is
// called when the current process is woken up in the do_select() function, and is used to wake
// up other tasks in the wait table that are on each wait queue. It is almost identical to the
// second half of the sleep_on() function in kernel/sched.c, see the description of it.
62 static void free\_wait(select\_table * p)
63 {
64     int i;
65     struct task\_struct ** tpp;
66
// If the wait queue header of the entries in the wait table (total nr entry) indicates that
// there are other wait tasks added later (for example, other processes call the sleep_on()
// function and sleep on the wait queue), then the queue header does not point to the current
// process, then we need to wake up these tasks first. The operation method is to set the task
// pointed to by the queue header to the ready state (state = 0), and set itself to the
// non-interruptible waiting state, that is, to wait for these subsequent queued tasks to be
// awakened and execute again to wake up this task. So re-execute the scheduler.
67     for (i = 0; i < p->nr ; i++) {
68         tpp = p->entry[i].wait_address;
69         while (*tpp && *tpp != current) {
70             (*tpp)->state = 0;
71             current->state = TASK\_UNINTERRUPTIBLE;
72             schedule();
73         }
// Execution here, indicating that the wait queue header field wait_address in the current entry
// of the wait table points to the current task. If it is NULL, it indicates that there is a
// problem with the schedule code, and a warning message is displayed. Then we let the wait
// queue header point to the task that entered the queue before us (line 76). If the head pointer
// does point to a task instead of NULL at this time, then there is still a task in the queue
// (*tpp is not empty), so the task is set to the ready state and wakes up. Finally, we set
// the item count field nr of the wait list to zero and clear the wait table.
74         if (!*tpp)
75             printk("free_wait: NULL");
76         if (*tpp = p->entry[i].old_task)
77             (**tpp).state = 0;
78     }
79     p->nr = 0;

```

```

80 }
81
82 // Get the tty according to the i-node.
83 // Check if the file is a character terminal device file according to the i-node. If it is,
84 // return its tty structure pointer, otherwise it returns NULL.
85 static struct tty\_struct * get\_tty(struct m\_inode * inode)
86 {
87     int major, minor;
88
89     // Returns NULL if it is not a character device file, or NULL if the major device number is
90     // not 5 (control terminal) or 4.
91     if (!S\_ISCHR(inode->i_mode))
92         return NULL;
93     if ((major = MAJOR(inode->i_zone[0])) != 5 && major != 4)
94         return NULL;
95
96     // If the major device number is 5, then the tty field of the process is its terminal device
97     // number, otherwise it is equal to the minor device number of the character device number.
98     // If the terminal device number is less than 0, it indicates that the process has no control
99     // terminal, or does not use the terminal, so it returns NULL, otherwise it returns the
100     // corresponding tty structure pointer.
101     if (major == 5)
102         minor = current->tty;
103     else
104         minor = MINOR(inode->i_zone[0]);
105     if (minor < 0)
106         return NULL;
107     return TTY\_TABLE(minor);
108 }
109
110 /*
111  * The check_XX functions check out a file. We know it's either
112  * a pipe, a character device or a fifo (fifo's not implemented)
113  */
114
115 // Check read in ready.
116 // Check if the read file operation is ready, that is, whether the terminal read buffer queue
117 // secondary has characters to read, or whether the pipe file is not empty.
118 // The parameter wait is the wait table pointer; the inode is the file i-node pointer.
119 // Returns 1 if the descriptor can be read, otherwise returns 0.
120 static int check\_in(select\_table * wait, struct m\_inode * inode)
121 {
122     struct tty\_struct * tty;
123
124     // First, according to the inode, call get_tty() to check if the file is a tty terminal (character)
125     // device file. If yes, check if there is any character in the secondary read buffer queue in
126     // the terminal, and return 1 if there is one. If secondary is empty at this time, the current
127     // task is added to the secondary wait queue proc\_list and returns 0. If it is a pipe file,
128     // check if there is any character in the current pipe, if it is, return 1; if not (pipe empty),
129     // add the current task to the waiting queue of the pipe i-node and return 0. Note that the
130     // PIPE_EMPTY() macro uses the current head and tail pointer position of the pipe to determine
131     // if the pipe is empty. The i_zone[0] and i_zone[1] fields of the pipe i-node store the current
132     // head and tail pointers of the pipe.
133     if (tty = get\_tty(inode))
134         if (!EMPTY(tty->secondary))

```

```

109         return 1;
110     else
111         add\_wait(&tty->secondary->proc_list, wait);
112     else if (inode->i_pipe)
113         if (!PIPE\_EMPTY(*inode))
114             return 1;
115     else
116         add\_wait(&inode->i_wait, wait);
117     return 0;
118 }
119
120 // Check write out ready.
121 // Check whether the file write operation is ready, that is, whether there is any free location
122 // in the terminal write buffer queue write_q, or whether the pipe file is not full.
123 // The parameter wait is the wait table pointer; the inode is the file i-node pointer.
124 // Returns 1 if the descriptor can be written, otherwise returns 0.
125 static int check\_out(select\_table * wait, struct m\_inode * inode)
126 {
127     struct tty\_struct * tty;
128
129     // First, according to the i-node, call get_tty() to check if the file is a tty terminal
130     // (character) device file. If yes, check if there is space in the write buffer queue write_q
131     // to write. If there is, return 1 if there is no empty space, then add the current task to
132     // the wait queue proc_list of write_q and return 0. If it is a pipe file, it is judged whether
133     // there is free space in the pipeline to write characters. If there is one, it returns 1; if
134     // not (the pipeline is full), the current task is added to the waiting queue of the pipeline
135     // i-node and returns 0.
136     if (tty = get\_tty(inode))
137         if (!FULL(tty->write_q))
138             return 1;
139         else
140             add\_wait(&tty->write_q->proc_list, wait);
141     else if (inode->i_pipe)
142         if (!PIPE\_FULL(*inode))
143             return 1;
144         else
145             add\_wait(&inode->i_wait, wait);
146     return 0;
147 }
148
149 // Check abnormal state.
150 // Check if the file is in an abnormal state. For terminal device files, the kernel always returns
151 // 0. For pipe files, return 1 if one or both of the two pipe descriptors have been closed at
152 // this time, otherwise add the current task to the wait queue of the i-node and return 0.
153 // The parameter wait is the wait table pointer; the inode is the file i-node pointer.
154 // Returns 1 if an exception condition occurs, otherwise returns 0.
155 static int check\_ex(select\_table * wait, struct m\_inode * inode)
156 {
157     struct tty\_struct * tty;
158
159     if (tty = get\_tty(inode))
160         if (!FULL(tty->write_q))
161             return 0;

```

```

144         else
145             return 0;
146     else if (inode->i_pipe)
147         if (inode->i_count < 2)
148             return 1;
149         else
150             add\_wait(&inode->i_wait, wait);
151     return 0;
152 }
153
154 // select() internal function.
155 // do_select() is the actual handler for the kernel to execute the select() system-call. The
156 // function first checks the validity of each descriptor in the descriptor set, and then calls
157 // the function check_XX() to check the descriptor of each descriptor set, and counts the number
158 // of descriptors currently ready in the descriptor set. If any of the descriptors are ready,
159 // the function will return immediately, otherwise the process will go to sleep, and the process
160 // will continue running after the timeout expires or because the process on the waiting queue
161 // where a descriptor is located is awakened.
162 int do\_select(fd\_set in, fd\_set out, fd\_set ex,
163             fd\_set *inp, fd\_set *outp, fd\_set *exp)
164 {
165     int count; // the number of ready descriptors
166     select\_table wait_table;
167     int i;
168     fd\_set mask;
169
170     // First, the three descriptor sets are ORed, and the effective descriptor bit mask in the
171     // descriptor set is obtained in the mask. It then loops through the current process to see
172     // if each descriptor is valid and included in the descriptor set. In the loop, each time a
173     // descriptor is judged, the mask is shifted to the right by 1 bit. Therefore, based on the
174     // least significant bit of the mask, we can determine whether the corresponding descriptor
175     // is in the descriptor set given by the user. A valid descriptor should be a pipe file descriptor,
176     // either a character device file descriptor or a FIFO descriptor, and the rest of the types
177     // return an EBADF error as an invalid descriptor.
178     mask = in | out | ex;
179     for (i = 0 ; i < NR\_OPEN ; i++, mask >>= 1) {
180         if (!(mask & 1)) // not in the descriptor set.
181             continue;
182         if (!current->filp[i]) // file not opened.
183             return -EBADF;
184         if (!current->filp[i]->f_inode) // file i-node is null.
185             return -EBADF;
186         if (current->filp[i]->f_inode->i_pipe) // valid: a pipe file.
187             continue;
188         if (S\_ISCHR(current->filp[i]->f_inode->i_mode)) // valid: a char dev file.
189             continue;
190         if (S\_ISFIFO(current->filp[i]->f_inode->i_mode)) // valid: a FIFO file.
191             continue;
192         return -EBADF; // all the rest are invalid.
193     }
194
195     // Let's start by looping to see if each descriptor in the three descriptor sets is ready
196     // (operational). At this point the 'mask' is used as the mask for the descriptor currently
197     // being processed. The three functions check_in(), check_out(), and check_ex() in the loop

```

```

// are used to determine whether the descriptor is ready for read in, write out or in abnormal
// condition. If a descriptor is ready to operate, the corresponding bit is set in the relevant
// descriptor set, and the count number of counts of the ready descriptor is incremented by
// one. On line 183, the statement "mask += mask" is equivalent to "mask << 1".
178 repeat:
179     wait_table.nr = 0;
180     *inp = *outp = *exp = 0;
181     count = 0;
182     mask = 1;
183     for (i = 0 ; i < NR_OPEN ; i++, mask += mask) {
// If the descriptor checked at this time is in the read operation descriptor set, and the
// descriptor is ready for a read operation, the corresponding bit in the descriptor set is
// set to 1, and the count of ready descriptors is incremented by one.
184         if (mask & in)
185             if (check_in(&wait_table, current->filp[i]->f_inode)) {
186                 *inp |= mask;           // set the bit in the set.
187                 count++;                // ready number.
188             }
// If the descriptor checked at this time is in the write operation descriptor set, and the
// descriptor is ready for a write operation, the corresponding bit in the descriptor set is
// set to 1, and the count of ready descriptors is incremented by one.
189         if (mask & out)
190             if (check_out(&wait_table, current->filp[i]->f_inode)) {
191                 *outp |= mask;
192                 count++;
193             }
// If the descriptor checked at this time is in the abnormal descriptor set, and the descriptor
// is in abnormal condition, the corresponding bit in the ex descriptor set is set to 1, and
// the count of abnormal descriptors is incremented by one.
194         if (mask & ex)
195             if (check_ex(&wait_table, current->filp[i]->f_inode)) {
196                 *exp |= mask;
197                 count++;
198             }
199     }
// After all the descriptors of the process have been checked, if there is no ready descriptor
// (count==0), and the process does not receive any non-blocking signal, and there are waiting
// descriptors at this time, or if the wait time has not expired, then we set the current process
// state to interruptible sleep state, and then execute the scheduler to perform other tasks.
// When the kernel schedules this task again, it will call free_wait() to wake up the tasks
// before and after the task on the relevant waiting queue, then jump to the repeat label (line
// 178) and re-detect whether there are descriptors we are concerned about.
200     if (!(current->signal & ~current->blocked) &&
201         (wait_table.nr || current->timeout) && !count) {
202         current->state = TASK_INTERRUPTIBLE;
203         schedule();
204         free_wait(&wait_table);        // the task is awakened and returned to here.
205         goto repeat;
206     }
// If count is not equal to 0 at this time, or if a signal is received, or the wait time is
// up and there is no descriptor to wait for, then we call free_wait() to wake up the task on
// the wait queue and then return the number of ready descriptors.
207     free_wait(&wait_table);

```

```

208         return count;
209     }
210
211     /*
212     * Note that we cannot return -ERESTARTSYS, as we change our input
213     * parameters. Sad, but there you are. We could do some tweaking in
214     * the library function ...
215     */
    // The parameter *timeout is changed during processing.
    /// select() system-call function.
    // The code in this function is mainly responsible for parameter copying and conversion before
    // and after the select() function operation. The main job of select() is done by the do_select()
    // function. Sys_select() will first decompose and copy the parameters of the select() function
    // from the user data space into the kernel space according to the buffer pointer given by the
    // parameter, then set the waiting timeout value, and then call do_select(). After returning,
    // the result will be copied back into the user space.
    // The parameter buffer points to the first parameter of the select() function in the user area.
    // If the return value is less than 0, an error occurs during execution; if the return value
    // is equal to 0, it means that no descriptor is ready for operation within the specified waiting
    // time; if the return value is greater than 0, it indicates the number of ready descriptors.
216 int sys_select( unsigned long *buffer )
217 {
218     /* Perform the select(nd, in, out, ex, tv) system call. */
    // First define several local variables that are used to decompose the select() function
    // arguments passed by the pointer arguments.
219     int i;
220     fd_set res_in, in = 0, *inp;           // read in fd set.
221     fd_set res_out, out = 0, *outp;        // write out fd set.
222     fd_set res_ex, ex = 0, *exp;          // abnormal fd set.
223     fd_set mask;                          // descriptor value range (nd) mask code.
224     struct timeval *tvp;                  // wait time structure pointer.
225     unsigned long timeout;
226
    // Then, the parameters are isolated and copied from the user data area into the local pointer
    // variables, and three descriptor sets in (read), out (write), and ex (exception or abnormal)
    // are respectively obtained according to whether the descriptor set pointer is valid. Where
    // mask is also a descriptor set variable. Based on the maximum descriptor value +1 in the three
    // descriptor sets (ie, the value of the first parameter nd), it is set to the mask of all
    // descriptors of interest to the user program. For example, if nd = 4, then mask = 0b00001111
    // (32 bits total).
227     mask = ~((~0) << get_fs_long(buffer++));
228     inp = (fd_set *) get_fs_long(buffer++);
229     outp = (fd_set *) get_fs_long(buffer++);
230     exp = (fd_set *) get_fs_long(buffer++);
231     tvp = (struct timeval *) get_fs_long(buffer);
232
233     if (inp)                                // read in set.
234         in = mask & get_fs_long(inp);
235     if (outp)                               // write out set.
236         out = mask & get_fs_long(outp);
237     if (exp)                               // abnormal set.
238         ex = mask & get_fs_long(exp);
    // Next we try to take the wait (sleep) time value 'timeout' from the time structure. First,
    // the 'timeout' is initialized to the maximum (infinite) value, and then the time value in

```



```

// the time structure is obtained from the user space, and the current tick value jiffies of
// the system is converted and added to it, and finally the time ticking value 'timeout' that
// needs to wait is obtained. We use this value to set the delay that the current process needs
// to wait. In addition, the tv_usec field on line 241 is a microsecond value. Dividing it by
// 1000000 gives the corresponding number of seconds, and multiplies it by the system ticks
// per second HZ, which converts tv_usec into a tick value.
239     timeout = 0xffffffff;
240     if (tvp) {
241         timeout = get\_fs\_long((unsigned long *)&tvp->tv_usec)/(1000000/HZ);
242         timeout += get\_fs\_long((unsigned long *)&tvp->tv_sec) * HZ;
243         timeout += jiffies;
244     }
245     current->timeout = timeout;    // set ticks that the current process need delay.
// The main work of the select() is done in function do_select().The code after calling this
// function is used to copy the processing result into the user data area and return it to the
// user. To avoid race conditions, you need to disable the interrupt before calling do_select()
// and then enable it after the function returns.
// If after the do_select() returns, the process's wait delay field timeout is still greater
// than the current system timing tick value jiffies, indicating that there is descriptors ready
// before the timeout. So here we will record the time value remaining until the timeout, and
// then we will return this value to the user. If the process's wait delay field timeout is
// already less than or equal to the current system jiffies, it means that do_select() may be
// returned due to a timeout, so the remaining time value is set to zero.
246     cli();    // disable int.
247     i = do\_select(in, out, ex, &res_in, &res_out, &res_ex);
248     if (current->timeout > jiffies)
249         timeout = current->timeout - jiffies;
250     else
251         timeout = 0;
252     sti();    // enable int.
// Next we clear the timeout field of the process. If the number of ready descriptors returned
// by do_select() is less than 0, it indicates an execution error, so the error number is returned.
// Then we write the processed descriptor set content and the delay time structure content back
// to the user buffer space. When writing the time structure content, it is also necessary to
// convert the remaining delay time represented by the tick time unit into seconds and microsecond
// values.
253     current->timeout = 0;
254     if (i < 0)
255         return i;
256     if (inp) {
257         verify\_area(inp, 4);
258         put\_fs\_long(res_in, inp);    // readable descriptor set.
259     }
260     if (outp) {
261         verify\_area(outp, 4);
262         put\_fs\_long(res_out, outp);    // writable descriptor set.
263     }
264     if (exp) {
265         verify\_area(exp, 4);
266         put\_fs\_long(res_ex, exp);    // abnormal descriptor set.
267     }
268     if (tvp) {
269         verify\_area(tvp, sizeof(*tvp));

```

```
270         put_fs_long(timeout/HZ, (unsigned long *) &tv->tv_sec); // seconds.
271         timeout %= HZ;
272         timeout *= (1000000/HZ);
273         put_fs_long(timeout, (unsigned long *) &tv->tv_usec); // microseconds.
274     }
    // If there is no ready descriptor available at this time and a non-blocking signal is received,
    // the interrupted error number is returned. Otherwise, the number of the ready descriptors
    // is returned.
275     if (!i && (current->signal & ~current->blocked))
276         return -EINTR;
277     return i;
278 }
279
```

---

## 12.20 Summary

In this chapter, we first give and explain the structure and composition of the MINIX file system, and describe the directory structure, directory entries, and file path name structures in the file system. After that, the structure and usage of the high-speed buffer (buffer cache) in Linux are explained in detail, and the way in which other programs in the kernel access the block device by using the buffer block is explained. Then from the implementation of the file system code, the program code is detailed in three aspects: the underlying general file system functions, the file access operation code, and the file access and control system-call interfaces.

The next chapter mainly describes the specific method and code implementation of Linux using the segmentation and paging functionalities provided by Intel 80X86 to manage memory. It also explains the principle and implementation of the copy-on-write mechanism and the demand loading mechanism.