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.

リスト 10-1 linux/kernel/chr_drv

Filename	Size	Last modified date(GMT)	Desc.
Makefile	3618 bytes	1992-01-12 19:49:17	
console.c	23327 bytes	1992-01-12 20:28:33	
keyboard.S	13020 bytes	1992-01-12 15:30:51	
pty.c	1186 bytes	1992-01-10 23:56:45	
rs_io.s	2733 bytes	1992-01-08 06:27:08	
serial.c	1412 bytes	1992-01-08 06:17:01	

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

端末ドライバは、端末装置の制御、端末装置とプロセス間のデータ転送、転送されたデータに対する一定の処理を行うために使用される。ユーザーがキーボードで入力した生データは、ターミナルプログラムで処理された後、受信側のプロセスに転送される。プロセスから端末に送られたデータは、端末プログラムで処理された後に端末の画面に表示されたり、シリアル通信回線を通じて遠隔地の端末に送信されたりする。端末の動作モードは、端末プログラムが入力データや出力データをどのように扱うかによって、2種類に分けられる。1つは正規のモードで、ターミナルプログラムを通過するデータは変換されてから送信される。例えば、TAB文字を8個のスペース文字に展開し、入力されたバ

ックスペースを使って

先に入力された文字を削除する。使用される処理機能は、一般にラインディシプリンまたはラインディシプリンモジュールと呼ばれている。もう1つは、非正規モードまたは生モードである。このモードでは、ライン・ディシプリン・プログラムは、データにカノニカル・モード変換を行うことなく、端末とプロセスの間でデータのみを転送する。

端末ドライバでは、コードはデバイスとの関係や実行フロー上の位置によって、文字デバイスの直接ドライバと上位層に直結するインタフェースプログラムに分けられる。この制御関係を図10-1を用いて説明することができる。

図10-1 端子ドライバ制御フロー

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
```

1. これらの端末機器のファイルは、以下の種類に分けられます。

2. Serial port terminal (/dev/ttySn)

シリアルポート端末とは、コンピュータのシリアルポートを利用して接続される端末機器のことである。コンピュータはシリアルポートを1つのキャラクターデバイスとして扱う。当時はターミナルに接続することが主な用途であったため、しばらくの間、これらのシリアルポートデバイスはターミナルデバイスと呼ばれることが多い。これらのシリアルポートに対応するデバイスファイル名は、/dev/ttyS0、/dev/ttyS1などで、デバイス番号はそれぞれ(4, 64)、(4, 65)などで、DOSシステム下のCOM1、COM2に対応しています。ポートにデータを送信するには、コマンドラインで標準出力をこれらの特別なファイル名にリダイレクトします。例えば、「echo test >」と入力すると
コマンドプロンプトで「/dev/ttyS1」と入力すると、ttyS1ポートに接続されたデバイスに「test」という単語が送信されます。

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

疑似端末（またはPseudo - TTY、PTYと略す）とは、一般的な端末のように機能するデバイスであるが、そのデバイスはいかなる端末ハードウェアとも関連していない。擬似端末装置は、他のプログラムに端末風のインターフェイスを提供するために使用され、主にネットワーク経由でホストにログインする際に、ネットワークサーバーやログインシェルプログラムに端末風のインターフェイスを提供したり、X Windowウィンドウ内で動作する端末プログラムに端末風のインターフェイスを提供するために使用されます。もちろん、ターミナル・インターフェースを使用する任意の2つのプログラム間でデータの読み書きチャンネルを確立するために、疑似ターミナルを使用することもできます。2つのアプリケーションやプロセスにターミナルスタイルのインターフェースを提供するために、擬似ターミナルはペアになっており、一方をマスターターミナルまたは擬似ターミナルマスターと呼び、他方をスレーブターミナルまたは擬似ターミナルスレーブと呼ぶ。ptyp1とttyp1のようにペアになった擬似端末の論理デバイスでは、ptyp1がマスターまたはコントロール端末、ttyp1がスレーブとなる。いずれかの疑似端末に書き込まれたデータは、カーネルを介してペアの疑似端末が直接受信します。例えば、マスターデバイスの/dev/ptyp3とスレーブデバイスの/dev/ttyp3の場合、プログラムがttyp3をシリアルポートデバイスとして扱うと、そのポートに対する読み書き操作が対応する論理端末デバイスのptyp3に反映され、ptyp3はプログラムが読み書き操作に使用する別の論理デバイスとなります。このようにして、2つのプログラムがこの論理デバイスを介して相互に通信することができ、スレーブデバイスttyp3を使用している一方のプログラムは、シリアルポートと通信しているとみなすことができます。これは、2つの論理デバイス間のパイプライン操作によく似ています。

擬似端子スレーブデバイスの場合は、シリアルポートデバイスを使用するように設計されたプロ

グラムであれば、ロジックデバイスを使用することができますが、マスターデバイスを使用するプログラムの場合は、擬似端子マスターデバイスを使用するように特別に設計されています。例えば、インターネット上で誰かがtelnetプログラムを使ってあなたのコンピュータに接続した場合、telnetプログラムは疑似端末マスターデバイスptyp2への接続を開始し、対応するttyp2ポート上でgettyプログラムが実行されるはずです。telnetが相手側から文字を取得すると、その文字はptyp2とttyp2を経由してgettyプログラムに渡され、gettyプログラムはttyp2、ptyp2、telnetプログラムを経由して「login:」の文字列情報をネットワークに送信します。このようにして、ログインプログラムとtelnetプログラムは、「疑似端末」を介して通信することができます。適切なソフトウェアを使用することで、2つ以上の疑似端末装置を同じ物理ポートに接続することができます。

以前のLinuxシステムでは、ttyp（ttyp0-ttypf）のデバイスファイル名は最大16組しかありませんでしたが、現在のLinuxシステムでは、/dev/ptm3のような「prm-ptyマスター」という命名法が一般的です。これに対応する末端は、/dev/pts/3 として自動的に作成され、必要に応じて pty 疑似端末を動的に提供できるようになっています。現在のLinuxシステムでは、/dev/ptsというディレクトリがdevpts型のファイルシステムになっています。

ファイル」の/dev/pts/3は、デバイスファイルシステムの1つのように見えますが、実際には異なるファイルシステムです。

4. Control terminal (/dev/tty)

キャラクター・デバイス・ファイルの/dev/ttyは、制御端末のエイリアスです。そのメジャーデバイス番号は5、マイナーデバイス番号は0です。現在のプロセスに制御端末がある場合、/dev/ttyは現在のプロセスの制御端末のデバイスファイルとなります。ps -ax」コマンドを使用して、プロセスがどの制御端末に接続されているかを確認できます。ログインシェルの場合、/dev/ttyは使用する端末で、そのデバイス番号は(5,0)です。tty」というコマンドを使えば、実際にどの端末デバイスに対応しているのかを確認することができます。実際には、/dev/ttyは実際の端末デバイスへのリンクに似ています。

5. 端末ユーザーがプログラムを実行しても制御端末を必要としない場合（バックグラウンドサーバープログラムなど）、プロセスはまず/dev/ttyファイルのオープンを試みることができます。オープンが成功すれば、プロセスは制御端末を持つことになります。この時点で、TIOCNOTTY（Terminal IO Control NO TTY）パラメータを指定したioctl()コールを使用して、制御端末を放棄することができます。

6. Console (/dev/ttyn, /dev/console)

Linuxシステムでは、コンピュータのモニタは、しばしばコンソールターミナルまたはコンソールと呼ばれます。VT200またはLinuxタイプの端末（TERM=Linux）をエミュレートしており、tty0、tty1、tty2などのキャラクターデバイスファイルが関連付けられています。コンソールでログインするときはtty1を使用しています。また、Alt+[F1-F6]で、tty2、tty3などに切り替えることができます。tty1～tty6を仮想端末と呼び、tty0は現在使用している仮想端末の別名である。Linuxシステムが生成する情報はtty0に送られるので、現在どの仮想端末を使っていても、システムの情報は私たちの画面に送られます。

7. 異なる仮想端末にログインできるので、同時に複数の異なるセッションを行うことができます。ただし、/dev/tty0に書き込めるのはシステムまたはスーパーユーザーのrootだけで、/dev/consoleもターミナルデバイスに接続されることがあります。ただし、Linux 0.12系では、/dev/consoleは通常、最初の仮想端末tty1に接続されます。

8. Other types of terminals

10.1.3 現在のLinuxシステムでは、他にも多くの文字デバイス用のターミナルデバイス特殊ファイルが用意されています。例えば、ISDN端末用の/dev/ttyIn端末デバイス。ここでは詳細を説明しません。

10.1.4 Terminal data structures

各端末機器はtty_structデータ構造を持っており、主に端末機器の現在のパラメータ設定、フォアグラウンドプロセスグループID、文字IOバッファキューなどの情報を格納するために使用されます。この構造体はinclude/linux/tty.hファイルで定義されており、その構造は以下の通りです。

```
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.

配列の各項目は，システム内の端末デバイスに対応するデータ構造 tty_struct である。Linux 0.12カーネルでは、コンソールデバイス用に1つ、システムの2つのシリアルポートを使用するシリアルターミナルデバイス用に残りの2つ、合計3つのターミナルデバイスをサポートしています。

termios構造体は，対応する端末機器のio属性を格納するために使用される。pgrpはプロセスグループIDであり，セッション内でフォアグラウンドにあるプロセスグループ，すなわち，現在端末装置を所有しているプロセスグループを示す。pgrpは主にプロセスのジョブ制御操作に使用される。stoppedは対応する端末装置が廃止されたかどうかを示すフラグである。関数ポインタ*write()は，端末装置の出力処理関数である．関数ポインタ*write()は、端末装置の出力処理関数である。コンソール端末の場合は、ディスプレイハードウェアを駆動し、画面に文字などを表示する役割を担い、システムのシリアルポートで接続されたシリアル端末の場合は、出力する役割を担う。 文字はシリアルポートに送られる。

端末で処理されたデータは、3つのtty_queue構造体の文字バッファキュー（または文字リスト）に格納される。その構造を以下に示す。

```
struct tty_queue {
        unsigned long data;                     // Current data statistics.
                                                // シリアルターミナル用のシリアルポートアドレス。
        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.

ユーザーが入力したデータを読み込む場合、割り込みハンドラは元の文字データを入力バッファのキューに入れるだけで、割り込み処理中に呼び出されたC関数（copy_to_cooked()）が文字変換を行うことになっています。例えば、プロセスが端末にデータを書き込む場合、端末ドライバは正規モードの関数copy_to_cooked()を呼び出し、ユーザバッファのデータを全てバッファキューに書き込み、端末に送信して表示させます。端末でキーが押されると、トリガされたキーボード割り込みハンドラは、キースキャンコードに対応する文字をリードキューread_qに入れ、カノニカルモードハンドラを呼び出してread_qの文字を処理し、補助キューsecondaryに入れます。 同時に、端末装置のエコーフラグ(L_ECHO)が設定されている場合は、その文字も書き込みキューwrite_qに入れ、端末の書き込み関

数を呼び出して画面に表示します。通常、パスワードの入力など特殊な条件を除き、エコーフラグは設定されています。これらのフラグの値を変更するには、端末のtermios構造体の情報を変更すればよい。

上記のtty_struct構造体には、include/termios.hヘッダーファイルで定義されているtermios構造体も含まれています。

その内容は以下の通りです。

```
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 は、出力モードフラグのセットです。ターミナル・デバイス・ドライバは、これらのフラグを使用して、主にtty_io.cのtty_write()関数で、ターミナルへの文字の出力を制御します。

c_flagは、コントロールモードフラグセットで、主にボーレート、キャラクタビット数、ストップビット数などのシリアル端末の伝送特性を定義するために使用されます。termios.hファイルの135～166行目を参照してください。

c_lflagは、ローカルモードのフラグセットで、主にドライバーとユーザーのやりとりを制御するために使用されます。例えば、文字をエコー(Echo)する必要があるかどうか、削除された文字を画面に直接表示する必要があるかどうか、端末に入力された制御文字に信号を生成させる必要があるかどうかなどです。これらの操作は、主にcopy_to_cooked()関数やtty_read()で使用される。例えば、ICANONフラグが設定されている場合、端末がカノニカルモードの入力状態であることを示し、そうでない場合は非カノニカルモードであることを示している。また、ISIGフラグが設定されている場合は、端末から発行される制御文字INTR、QUIT、SUSPを受信する際に、システムが対応する信号を生成する必要があることを意味する。termios.hファイルの169～183行目を参照。

上記の4種類のフラグセットはすべて符号なしの長さで、各ビットがフラグを表すことができるので、各フラグセットは最大32個の入力フラグを持つことができます。これらのフラグとその意味はすべて、termios.hヘッダーファイルに記載されています。

c_cc[NCCS]配列には、端末で変更可能なすべての特殊文字が含まれています。NCCS(=17)は配列の長さの値です。例えば、ブレーク文字(^C)を他のキーで生成するように修正することができます。端末のc_cc[]配列のデフォルト値は、include/linux/tty.hファイルで定義されています。配列項

目のシンボル名は、プログラムが配列の項目を参照する際に定義されます。これらの名前は，VINTR，VMINのようにVの文字で始まります。termios.hファイルの67～83行目を参照してください。

したがって、システムコールioctlを使ったり、相関関数（tcsetattr()）を使ったりして、termios構造体の情報を変更することで、端末の設定パラメータを変更することができる。正準モード関数は、これらの設定パラメータを操作する。例えば，制御端末は，入力された文字をエコーすること，シリアル端末送信のボーレートを設定すること，読み取りバッファキューをクリアすること，バッファキューを書き込むことなどである。

ユーザーが端末のパラメータを変更し、カノニカルモードフラグをリセットすると、端末がローモードで動作するように設定できます。このとき、カノニカル・モード・ハンドラは、ユーザが入力したデータをそのままユーザに転送し、キャリッジ・リターンは通常の文字として扱われる。したがって、ユーザーが

システム・コール・リードがいつ完了して返されるかを決定するために、何らかの決定方式を行う必要があります。これは、端末のtermios構造体のVTIMEとVMINの制御文字によって決定される。この2つは、読み出し動作のタイムアウトタイミングの値である。VMINは、読み取り操作を満足させるための最小文字数を示し、VTIMEは、読み取り操作の待ち時間のタイミング値である。sttyコマンドを使えば、現在の端末機器のtermios構造体のフラグの設定を見ることができる。Linux 0.1xシステムのコマンドラインプロンプトでsttyコマンドを入力すると、以下のように表示されます。の情報を提供します。

[/root]# stty

キャラクター

DISCARD（ディスカード）。'^o' reprint: '^r' rwerase: '^w'

コントロールフラグ

-cstopb cread -parenb -parodd hupcl -clocal -crtscts

ボーレート 9600 Bits: CS8

----------Input Flags----------

-IGNBRK -BRKINT -IGNPAR -PARMRK -INPCK -ISTRIP -INLCR -IGNCR
ICRNL -IUCLC IXON -IXANY IXOFF -IMAXBEL

出力フラグ

OPOST -OLCUC ONLCR -OCRNL -ONOCR -ONLRET -
OFILL -OFDEL 遅延モード。Cr0 nl0 tab0 bs0 ff0 vt0

ローカルフラグ

isig icanon -xcase echo -echoe -echok -echonl -noflsh
-TOSTOP ECHOCTL ECHOPRT ECHOKE -FLUSHO -PENDIN -IEXTEN

行数 0 コル数 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.

ターミナルプログラムが使用する上記の主なデータ構造と、それらの関係を図10-2に示します。

図10-2 ターミナルプログラムのデータ構造

10.1.5 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.

10.1.5.1 Canonical mode

c_lflagのICANONフラグがセットされていると、端末の入力データを正規のモードに合わせて処理します。これにより、入力された文字は一行にまとめられ、一行分の文字として読み込まれる処理が行われます。一行分の文字が入力されると、ターミナルドライバは直ちにリターンします。行のデリミタは、NL、EOL、EOL2、EOFです。最後のEOF(end of file)はハンドラで削除されるほか、残りの4文字は行の最後の文字として呼び出し側に返されます。

カノニカルモードでは、端末から入力された以下の文字が処理されます。ERASE, KILL, EOF, EOL, REPRINT, WERASE, EOL2。

ERASEは、バックスペースです。canonical modeでは、copy_to_cooked()関数がこの文字に遭遇すると、バッファのキューに入力されている最後の文字が削除される。もし、キューの最後の文字が前の行の文字（例えば、NL）であれば、処理は行われません。そして、この文字は無視され、バッファ・キューには入れられません。

KILL は、行削除文字です。これは、キューの中の最後の行の文字を削除します。この文字はその後無視されます。

EOFは、ファイルの終わりを意味します。copy_to_cooked()関数では、この文字と行末文字EOL、EOL2はキャリッジリターンとして扱われます。読み込み操作機能で遭遇した文字は

はすぐに戻ります。EOF文字はキューに入らず無視されます。

0.1.5.2 REPRINTとWERASEは、拡張カノニカルモードで認識される文字です。REPRINTは、未読の入力をすべて出力します。WERASEは、単語の削除（スキップホワイトスペース文字）に使用されます。Linux 0.12では、プログラムはこの2つの文字の認識と処理を無視します。

10.1.5.3 Non-Canonical mode

c_iflagセット内のICANONフラグがリセット状態の場合、端末プログラムは非正規モードで動作する。このとき、端末プログラムは上記の文字を処理せず、通常の文字として扱い、入力データはラインの概念を持たないものとしている。端末プログラムがいつ読み取り処理に戻るかは、MINとTIMEの値(VMIN, VTIME)によって決まる。この2つの変数は、c_cc[]配列の変数です。これらを変更することで、非正規モードでのプロセスの文字の読み方を変えることができる。

1. MINは読み出し動作の最小文字数、TIMEは文字の読み出しを待つタイムアウト値（単位は1/10秒）を指定します。これらの値によって、4つのケースに分けて説明することができます。
   2. MIN>0，TIME>0
3. この時、TIMEは文字間隔のタイムアウトのタイミング値で、最初の文字を受信した後に有効になります。タイムアウト前にMIN文字を受信した場合、読み出し動作はすぐに戻ります。MINキャラクターを受信する前にタイムアウトが発生した場合、リードオペレーションは受信したキャラクターの数を返します。このとき、少なくとも1つの文字を返すことができます。したがって、文字を受信する前にセカンダリが空になった場合、読み取り処理はブロックされます（スリープ）。
   4. MIN>0，TIME=0
5. このとき、MIN文字を受信した場合のみ、読み出し動作が戻ります。それ以外は無期限に待ちます（ブロッキング）。
   6. MIN=0，TIME>0
7. この時のTIMEは、読み出し動作のタイムアウトのタイミング値です。文字を受信した場合、またはタイムアウトした場合、読み出し動作はすぐに戻ります。タイムアウトした場合、読み出し動作は0文字を返します。
   8. MIN=0，TIME=0

この設定では、キューの中に読めるデータがあれば、必要な文字数だけ読み込みを行います。それ以外の場合は、直ちに0文字を返します。

10.1.6 上記の4つのケースでは、MINは読み込まれる文字数の最小値を示しているだけです。プロセスがMINよりも多くの文字を必要としている場合、キューに文字がある限り、プロセスの現在の要求を満たすことができます。端末デバイスの読み取り操作については、プログラムtty_io.cのtty_read()関数を参照してください。

10.1.7 Console Terminal and Serial Terminal Devices

Linux 0.12では、ホスト上のコンソール端末と、シリアルハードウェア端末の2種類の端末が使用できるようになっている。コンソール端末は，カーネル内のキーボード割り込みハンドラプログラムkeyboard.sと表示制御プログラムconsole.cによって操作・管理される．上位のtty_io.cプログラムから渡される表示文字や制御メッセージを受け取り、ホストの画面に文字を表示する制御を行う。同時に、コンソール（ホスト）は、キーボードで生成されたコードをkeyboard.sを介してtty_io.cプログラムに転送します。シリアル端末装置は、コンピュータのシリアルポートに回線を介して接続されており、カーネル内のシリアルプログラムrs_io.sを介してtty_io.cと直接情報を交換しています。

keyboard.sとconsole.cの2つのプログラムは、モニターとキーボードを使って、Linuxシステムのホストにあるハードウェア端末装置をエミュレートするプログラムによく似ています。ホスト上にあることから、このシミュレートされた端末環境をコンソール端末、あるいは直接コンソールと呼んでいます。 機能

この2つのプログラムが実装しているのは、シリアル端末機のROMに入っている端末処理プログラムの役割（通信部分を除く）に相当し、また、通常のPCの端末エミュレーションソフトのようなものでもあります。つまり、この2つのプログラムはカーネルの中にあるとはいえ、独立して見ることができるのです。この「シミュレート端末」と通常のハードウェア端末装置との大きな違いは、シリアルラインを介してドライバを通信する必要がないことです。したがって、keyboard.sおよびconsole.cプログラムは、実際の端末装置（DECのVT100端末など）のハードウェア処理機能、つまり端末装置のファームウェアにおける通信部分を除くすべての処理機能をシミュレートできる必要があります。コンソール端末とシリアル端末装置の処理構造の違いと共通点については、図10-3を参照のこと。したがって，一般的なハードウェアの端末装置や端末エミュレーションプログラムの動作原理をある程度理解していれば，この2つのプログラムを読むことに何の困難も生じないだろう。

図10-3 コンソール端末とシリアル端末のデバイス図

10.1.7.1 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.

例えば、ユーザがキーボードに文字を入力すると、キーボード割込み応答（割込み要求信号IRQ1、割込み番号INT33に対応）が発生し、ハンドラが実行されます。このとき、キーボード割り込みハンドラは、キーボードコントローラから対応するキーボードスキャンコードを読み取り、使用したキーボードスキャンコードマッピングテーブルに従って対応する文字をttyリードキューread_qに変換します。次に、割り込みハンドラのC関数do_tty_interrupt()を呼び出します。この関数は、canonical mode関数copy_to_cooked()を直接呼び出して文字をフィルタリングし、tty補助キュー「secondary」に入れ、tty書き込みキューwrite_qに文字を入れます。その後、コンソールの書き込み関数con_write()を呼び出し、コンソール出力(表示)処理を行います。このとき、端末のエコー属性が設定

されていれば、その文字は画面に表示されます。do_tty_interrupt()関数とcopy_to_cooked()関数はtty_io.cに実装されており、全体の動作プロセスは図10-4のようになります。

図10-4 コンソールキーボード割り込みサービス

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.

したがって、一般的には、コンソール端末のキーボード割り込みハンドラkeyboard.Sは、主にユーザが入力した文字をread_qバッファキューに入れるために使用されます。その画面表示ハンドラであるconsole.cは、write_qキューから文字を取り出して画面に表示するために使用されます。3つの文字バッファキューと上記の関数やファイルの関係は、図10-5にはっきりと示されています。

図10-5 コンソールキャラクターキューと機能の関係

10.1.7.2 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.

システムで発生するシリアル割込みは（1)モデムの状態変化によるもの、(2)回線状態の変化によるもの、(3)受信文字によるもの、(4)文字送信要求によるもの（送信保持レジスタ空のフラグがセットされている）。割り込みの原因となる最初の2つのケースの処理は、対応するステータスレジスタを読み出すことでリセットされます。キャラクタを受信するケースでは、プログラムはまず、キャラクタをリード・バッファ・キューread_qに入れ、copy_to_cooked()関数を呼び出して、文字行単位の正規モードのキャラクタに変換して、補助キュー「secondary」に入れる必要がある。文字の送信が必要な場合は、まず、書き込みバッファキューwrite_qのテールポインタが指す位置から文字を取り出して送信し、書き込みバッファキューが空になったかどうかを確認し、まだ文字がある場合は、送信動作を周期的に行う。

システムのシリアルポートからアクセスされる端末では、コンソールと同様の処理に加えて、シリアル通信のための入出力処理動作が必要となります。データの読み出しは、シリアル割り込みハンドラによってリードキューread_qに入れられた後、コンソール端末と同様の処理が行われる。

例えば、シリアルポート1に接続された端末の場合、入力された文字はまずシリアルラインを介してホストに送信され、ホストのシリアルポート1から割り込み要求が出されます。この時、シリアルポート割り込みハンドラは、シリアルターミナル1のttyリードキューread_qに文字を入れ、割り込みハンドラのC関数do_tty_interrupt()を呼び出します。この関数は、行規則関数copy_to_cooked()を直接呼び出して文字をフィルタリングし、tty補助キューsecondaryに入れます。同時に、文字はttyの書き込みキューwrite_qに入れられ、シリアルターミナル1の関数rs_write()が呼び出されます。この関数は、順番にシリアルターミナルに文字を送り返し、ターミナルのエコー属性が設定されていれば、シリアルターミナルの画面に文字が表示されます。

プロセスがシリアルターミナルにデータを書き込む必要がある場合、操作プロセスは、ターミナルのtty_structデータ構造の書き込み関数がシリアルターミナル書き込み関数rs_write()であることを除いて、書き込みターミナルと同様です。この関数は、「Transmit Holding Register Empty Enable Interrupt」のマスクを解除し、送信ホールディングレジスタが空のときにシリアル割り込みを発生させます。シリアル割込み処理では、割込みの原因に応じて、書込みバッファキューの write_q からキャラクタを取り出し、送信保持レジスタに入れてキャラクタ送信を行います。演算処理でも1割り込

み分のキャラクタを送信し、最後にwrite_qが空になると、送信保持レジスタエンプティイネーブル割り込みビットを再びマスクすることで、このような割り込みが再び発生しないようにしています。シリアル端子の書き込み関数rs_write()はserial.cプログラムに、シリアル割り込みプログラムはrs_io.sに実装されています。シリアル端子の3つのバッファキューと関数の関係は図10-6を参照してください。

図10-6 シリアルターミナル機器のキャラクターキューと機能の関係

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.

10.1.8 Terminal Driver Interface

通常、ユーザーはファイルシステムを介してデバイスとやり取りします。各デバイスはファイル名を持ち、それに応じてファイルシステム内にインデックスノード（iノード）を持つ。しかし、iノード内のファイルタイプは、他の通常のファイルと区別するためにデバイスタイプとなっています。そのため、ユーザーはファイルシステムコールを使ってデバイスに直接アクセスすることができる。端末ドライバは、このためにファイルシステムへのコール・インターフェース機能も提供する。端末ドライバとシステム内の他のプログラムとのインタフェースは、tty_io.cファイル内のジェネリック関数を用いて実装されており、このファイルには、読み取り端末関数tty_read()と書き込み端末関数tty_write()、および入力行規則関数copy_to_cooked()が実装されている。さらにtty_ioctl.cプログラムでは、端末のパラメータを変更する入出力制御関数（またはシステムコール）tty_ioctl()が実装されている。端末の設定パラメータは、端末データ構造の中のtermios構造に置かれています。多くのパラメータがあり複雑なので，include/termios.hファイルの記述を参照してください。

端末デバイスが異なると、一致するcanonical modeプログラムも異なる場合があります。しかし、Linux 0.12ではcanonical modeの機能は1つしかないため、termios構造体の行規則フィールド'c_line'は機能せず、0に設定されています。

10.2 keyboard.S

10.2.1 Function descriptions

keyboard.Sプログラムには、主にキーボードの割り込みハンドラが含まれており、関連するC言語の関数が呼び出されます。

do_tty_interrupt().プログラムではまず、キーボードの特殊キー（Alt、Shift、Ctrl、Capsなど）の状態に応じて、プログラムの後半で使用する状態フラグ変数「mode」を設定します。次に、キーボード割り込みの原因となったキーのスキャンコードに応じて、ジャンプテーブルに配置されている対応するスキャンコード処理サブルーチンを使用して、スキャンコードに関連する文字を読み取り文字キュー（read_q）に入れます。次に、C関数のdo_tty_interrupt()（tty_io.c、397行目）を呼び出します。この関数には、正規の関数であるcopy_to_cooked()の呼び出しが1つだけ含まれています。 この正規モード関数の主な目的は、リードバッファキューread_q内の文字を適切に処理した後、通常モードのキュー（二次キュー）に入れ、対応する端末装置がエコーフラグを設定した場合には、プログラムも文字を直接書き込みキュー（write_q）に入れ、書き込みtty関数を呼び出して端末画面に文字を表示することです。

ATキーボードのスキャンコードは、キーを押すとそのキーのスキャンコードが送信されますが、キーを離すと2バイト送信され、1バイト目は0xf0、2バイト目はキーを押した時と同じスキャンコードになります。後方互換性のために、PCの設計者は、ATキーボードのスキャンコードを古いPC/XT標準キーボードのスキャンコードに変換したので、プログラムはPC/XTのスキャンコードを処理するだけでよいのです。キーボードのスキャンコードの説明は、プログラムリストの10.3項の説明を参照してください。ここで、キーボード関連の用語で、「make」はキーが押されたことを意味し、「break」はキーが離されたことを意味することも覚えておいてください。

また、このプログラムのファイル名は、他のガスのアセンブリ言語プログラムとは異なり、サフィックスが大文字の「.S」になっています。このようなサフィックスを使用すると、アセンブラがGNU Cコンパイラのプリプロセッサ CPPを使用できるようになり、多くのC言語ディレクティブをアセンブリ言語プログラムで使用できるようになります。例えば、「#include」や「#if」などは、プログラム中の関連コードの具体的な使用方法を示しています。

10.2.2 Code Annotation

プログラム 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
```

```
 * KBD_GR for German keyboards
 * KBD_FR for Frech keyboard
 */
// キャラクタマッピングのコードテーブルを後から選択する際に使用するキーボードの種類を定義します。
#define KBD_FINNISH            // Linus is a Finnish man ☺

.text
// グローバル変数として宣言され、init時にキーボード割り込みディスクリプターを設定するために使用されます。
.globl _keyboard_interrupt     // same as keyboard_interrupt().
```

```
22
23 /*
24 * これらはキーボードの読み取り機能のためのもの
のです。
26 // Sizeは、キーボードバッファのキューの長さをバイト単位で表します。
27 size    = 1024          /* must be a power of two ! And MUST be the same
28                            as in tty_io.c !!!! */

   // キーボードバッファキューのデータ構造 tty_queue の各フィールドのオフセットを以下に示します。
   29                      // (include/linux/tty.h, line 22)。
30 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
   // このプログラムでは、3つのフラグバイト（mode, leds, e0）を使用しています。の各ビットの意味は以下の通りです。
   // 各バイトフラグは以下の通りです。
   // (1) 'mode'は、特殊キーの押下状態フラグである。の状態を示すのに使われます。
   // 大文字・小文字変換キー（caps）、代替キー（alt）、コントロールキー（ctrl）、シフトキー。
   //     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' は、キーボードの表示を示すステータスフラグ、つまり
   // num-lock, caps-lock, scroll-lockを示すLEDチューブ。
   //     bit 7-3 all zeros, not used.;   bit 1 num-lock (Initial setting 1);
   //     bit 2 caps-lock;                bit 0 scroll-lock.
   // (3) 'e0' このフラグは、スキャンコードが0xe0または0xe1の場合に設定されます。続いていることを示しています。
   // を1文字または2文字のスキャンコードで受信します。通常、スキャンコード0xe0を受信した場合、それは
   // スキャンコード0xe1を受信した場合、それは以下のことを意味します。
   // 続いて2文字であることを示します。プログラムリストの後の説明を参照してください。
   //     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
38 37 /*
39  * con_int is the real interrupt routine that reads the
40  * keyboard scan-code and converts it into the appropriate
41  * ascii character(s).
42  */                          // Note: The original comment above is obsolete.
   //// キーボード割り込みハンドラのエントリポイント。
   // キーボードコントローラは，ユーザのキー押下操作を受けると，インタラプト
   //要求信号IRQ1を割り込みコントローラに送信します。このキーボード割り込みハンドラが実行される
```

CPUが要求に応答したときに、//。割り込みハンドラは、キースキャンコードを
// キーボードコントローラポート(0x60)に対応するスキャンコードサブルーチンを呼び出して
// 処理を行います。
// まず、ポート0x60から現在のボタンのスキャンコードを読み込んで、次のように判断します。
// スキャンコードが0xe0または0xe1の場合。そうであれば、直ちにキーボードコントローラに応答する
// そして、割り込みコントローラにEOI（End of Interrupt）信号を送り、キーボードが
// コントローラーが割り込み信号を生成し続けることで、次の割り込み信号を受け取ることができます。
// 文字を表示します。2つの特殊なスキャンコードを受信しなかった場合は、対応するキーを起動して
スキャンコードの値に応じて、キージャンプテーブルkey_tableの中で、 // 処理サブルーチンを行います。

```
// スキャンコードに対応する文字を読み込み文字バッファのキューに入れる read_q.
// そして、キーボードコントローラに応答し、EOI信号を送信した後、関数
// do_tty_interrupt()を呼び出して（実際にはcopy_to_cooked()を呼び出して）文字を処理する
43 read_q の中の // を、セカンダリの補助キューに入れます。
44 _keyboard_interrupt:
45         pushl %eax
46         pushl %ebx
47         pushl %ecx
48         pushl %edx
49         push %ds
50         push %es
51         movl $0x10,%eax          // Set ds and es to kernel data segment.
52         mov %ax,%ds
53         mov %ax,%es
54         movl _blankinterval,%eax
55         movl %eax,_blankcount    // preset black screen time count (blankinterval).
56         xorl %eax,%eax           /* %eax is scan code */
57         inb $0x60,%al            // Read scan code to al.
58         cmpb $0xe0,%al           // jump to set_e0 if scan code is 0xe0
59         je set_e0
60         cmpb $0xe1,%al           // jump to set_e1 if scan code is 0xe1
61         je set_e1
62         call key_table(,%eax,4) // key handler: key_table + eax*4 (see line 513).
63         movb $0,e0               // Reset e0 flag after return.

// 次のコード（62-72行目）は、PC標準のハードウェアリセット操作を行います。
8255Aを使った//キーボード回路です。ポート0x61は、8255Aの出力ポートBのアドレスです。 ビット
このポートの//7（PB7）は、キーボードデータの処理を無効にしたり有効にしたりするために使用され
ます。このプログラム
//は、受信したスキャンコードに対応して、まずキーボードを無効にし、次に
64 // すぐにキーボードが使えるようにする。
65 e0_e1:  inb $0x61,%al            // Get port B state, PB7 is used to enable/disable keyboard.
66         jmp 1f                   // delay for awhile.
64 1:      jmp 1f
65 1:      orb $0x80,%al            // set bit7 of al.
66         jmp 1f
67 1:      jmp 1f
68 1:      outb %al,$0x61           // set port B bit7 to disable keyboard.
69         jmp 1f
70 1:      jmp 1f
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
91 90 /*
 92  * This routine fills the buffer with max 8 bytes, taken from
 93  * %ebx:%eax. (%ebx is high). The bytes are written in the
 94  * order %al,%ah,%eal,%eah,%bl,%bh ... until %eax is zero.
 94  */
    // まず、コンソールの読み込みバッファキューread_qのアドレスから
    // テーブル table_list (tty_io.c, line 81)では、alレジスタの文字を
    // キューのヘッダポインタを読み、ヘッドポインタを1バイト進めます。もし、ヘッドポインタが
    // 読み取りバッファの端から移動した場合は、バッファの先頭に回り込ませます。
    // そして、この時点でバッファキューが満杯になっているかどうかを確認する、つまり、キューヘッドポインタを比較する
    // をテールポインタで指定します（等しい場合は満杯を意味します）。満杯になった場合、残りの文字を捨てる
    // ebx:eaxにあるかもしれないもの。バッファが満杯でなければ、ebx:eaxのデータを右の
    // を8ビット分移動させて（つまり、ahの値をalに、blをahに、bhをblに）、上記の処理を繰り返します。アルのために//。すべての文字が処理されるまで、現在のヘッドポインタの値が保存されます。
 95 // そして、リードキューを待っているプロセスがあるかどうかをチェックし、あればウェイクアップします。
 96 put_queue:
 97         pushl %ecx
 98         pushl %edx                  // Get read buffer queue pointer.
 99         movl _table_list,%edx        # read-queue for console
100         movl head(%edx),%ecx        // Take the queue head pointer -> ecx.
101 1:      movb %al,buf(%edx,%ecx)     // put char in al into the head pointer position.
102         incl %ecx                   // moved forward by 1 byte.
103         andl $size-1,%ecx           // Adjust the head pointer.
104         cmpl tail(%edx),%ecx         # buffer full - discard everything
105         je 3f
106         shrdl $8,%ebx,%eax          // Move ebx right 8 bits into eax, ebx is unchanged.
107         je 2f                       // Are there still characters? if no then jump.
108         shrl $8,%ebx                // Move ebx value to the right by 8 bits
109         jmp 1b
110 2:      movl %ecx,head(%edx)        // If all chars in queue, the head pointer is saved.
111         movl proc_list(%edx),%ecx   // Waiting process pointer for this queue?
112         testl %ecx,%ecx             // Check if there are processes waiting for this queue.
113         je 3f                       // No, then jump (to line 114).
114         movl $0,(%ecx)              // Yes, the process is awake.
114 3:      popl %edx
115         popl %ecx
116         ret
117
    // ここから先は、ジャンプテーブルのポインタに対応する各キーのサブルーチンとなる
    // key_table で、上記60行目のステートメントから呼び出されます。ジャンプテーブルkey_tableの始まりは
```

// 513行目で
// 次のコードは、スキャンコードに応じてモードフラグの対応するビットを設定します。
ctrlやaltの//。スキャンコードの前に0xe0のスキャンコードを受信した場合（e0フラグがセットされている）。
// キーボードの右側にあるctrlキーやaltキーが押されていることを意味しており、ビット
118 // モードフラグがctrlまたはaltに対応して設定されます。

```
119 ctrl:   movb $0x04,%al              // 0x04 (bit2), the left ctrl key in mode.
120         jmp 1f
121 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
```

// このコードは、ctrlキーやaltキーが離されたときのスキャンコードを処理し、リセットして
// モードフラグの該当ビット。演算では、それを判断して
// は，e0フラグの有無に応じて，キーボードの右側にあるctrlキーまたはaltキーである。

```
126 // を設定します。
127 unctrl: movb $0x04,%al              // bit2 is for the left ctrl key in mode.
128         jmp 1f
129 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
```

// このコードは、左右のシフトキーを押したり離したりしたときのスキャンコードを処理します

```
136 モードの対応するビットを設定/リセットするための //。
137 lshift:
138         orb $0x01,mode              // left shift key is pressed, set mode bit0.
139         ret
140 unlshift:
141         andb $0xfe,mode             // left shift key is released, reset mode bit 0.
142         ret
143 rshift:
144         orb $0x02,mode              // right shift key is pressed, set mode bit1.
145         ret
146 unrshift:
147         andb $0xfd,mode             // right shift key is released, reset mode bit1.
148         ret
149
```

// このコードはcapsスキャンコードを処理 caps-lock bit (bit 2) of the leds flag, and set the caps key pressed flag bit (bit 7).

します。capsキーが現在押されている状態であるかどうかは、モードビット7でわかります。イエスならリターン、そうでなければモードフラグの

ビット6（capsキーが押されている）を反転させて

```
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  */
    // 次のコードは、まず右のテンキーでスキャンコードが発行されているかどうかを判断します。
    キーボードの//ボタンを押します。そうでない場合は、このサブロティンを終了します。キーボードの
    最後のキーDel(0x53)が押されたら
    // テンキーが押されたときに、「Ctrl-Alt-Del」キーが押されたかどうかを判断します。
    // コンビネーションが押されました。コンビネーションキーが押された場合、システム再起動コードに
    ジャンプします。
    // 10.2.3項に記載されているXTキーボードスキャンコード表を参照してください。
    // のリスト、または付録のキーボードスキャンコード表の最初のセットを参照してください。見ての通
    り
    // 表から、テンキーのキーのスキャンコード範囲は[0x47～0x53]となります。
179 // また、方向キーとして使用する場合は、プリアンブルコードe0となります。
180 cursor:
181         subb $0x47,%al          // If scan code not in range [0x47 - 0x53], it ret (198).
182         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 */
    // 12となった場合は、「del」キーが押されたことを示すので、続けて
185     // 'ctrl'と'alt'も同時に押されているかどうかをチェックします。
186         testb $0x0c,mode        // ctrl key pressed? jump if not.
187         je cur2
188         testb $0x30,mode        // alt key pressed?
189         jne reboot              // yes, then jump to reboot system (line 594).

    // 次に、テンキーを方向キーとして使うかどうかをコードで判断します（page up/dn，
    // 挿入、削除など）のキーを押します。押されたボタンが実際にキーパッド上にある場合、また、先頭
    の
    // このとき、スキャンコードe0を受信すると、キーパッドのボタンを
    //方向キーで処理します。その後、カーソル移動や挿入/削除ボタンの処理にジャンプします。もし、e0
    //が設定されていない場合、まず、num-lock LEDが点灯しているかどうかを確認します。点灯していな
    い場合は、同じくカーソル
    //の動作処理を行います。ただし、num-lockランプが点灯している場合（キーパッドが
    // テンキーとして使われる）で、シフトキーも押されている場合は、キーパッドキーを
```

190 カーソル移動の操作として、この時に//。

```
191 cur2:   cmpb $0x01,e0          /* e0 forces cursor movement */  // e0 flag set ?
192         je cur                 // jump to cur if e0 is set.
193         testb $0x02,leds       /* not num-lock forces cursor */  // test leds flag.
194         je cur                 // if num's LED is on, do cursor movement operation.
195         testb $0x03,mode       /* shift forces cursor */
196         jne cur                // if shift is pressed, also do cursor movement operation.
```

// 最後に、テンキーを数字キーとして使用します。続いて、スモールナンバーテーブルの
// num_tableは、スキャンコードに応じて照会され、対応する数値文字が

キーへの // が取り出され、バッファキューに置かれます。文字数が多いので
// 一度にキャラクターキューに入れるには<=8>が必要なので，その前にebxをクリアする必要があります。
// put_queueにジャンプして実行する。95行目のコメントをご覧ください。

```
        xorl %ebx,%ebx
        movb num_table(%eax),%al      // obtain the digital char with eax as index.
        jmp put_queue                 // parameter: 1-8 chars in ebx:eax.
1:      ret
```

// カーソルの移動や、挿入・削除ボタンを処理するコードです。ここでは、代表的な
カーソル文字表の対応するキーの // 文字が最初に取得されます。もし
// 文字が<='9'（5、6、2、3）の場合、Page Up、Page Dn、Insertのいずれかであることを意味します。
// またはDeleteキーを押すと、関数文字列に'~'という文字が追加されます。
// しかし、このカーネルはそれらを認識して処理することはありません。コードは次に、コンテンツをaxの//をeaxのハイワードに入れて、'esc['をaxに入れて、ハイワードの文字で
// eaxのワードで移動シーケンスを形成します。最後に，この文字列を
// キャラクターキュー。

```
cur:    movb cur_table(%eax),%al  // get cursor char to al.
        cmpb $'9,%al              // if <='9' (5,6,2 or 3) it's page up/dn or Ins/Del key
        ja ok_cur                 // it needs to add char '~'.
        movb $'~,%ah
ok_cur: shll $16,%eax             // move content of ax to high word of eax.
        movw $0x5b1b,%ax          // put 'esc [' in ax
        xorl %ebx,%ebx
        jmp put_queue

#if defined(KBD_FR)
num_table:
        .ascii "789 456 1230."     // ASCII code associated to keys on numeric keypad.
#else
num_table:
        .ascii "789 456 1230,"
#endif
cur_table:
        .ascii "HA5 DGC YB623"     // associated to the arrow, Ins, Del keys on keypad.

/*
* このルーチンはファンクション・キーを
処理する */。
```

// ファンクションキースキャンコードをエスケープ文字列に変換するサブルーチンであり
// が読み取りキューに格納されます。コードがチェックするファンクションキーの範囲はF1～F12です。
// ファンクションキーF1～F10のスキャンコードは0x3B～0x44、F11と
// F12は0x57, 0x58です。まず、F1--F12を、対応するシーケンス番号に変換します
// 0--11となり、関数キーテーブルfunc_tableを照会して、対応する
// エスケープ文字のシーケンスで、文字キューに入れられます。の間に Alt キーが押されると
// ファンクションキーを押すと、制御端子の切り替え動作のみが行われます。

```
ファンク        subb $0x3B,%al            // F1 scan code is 0x3B
```

です。

```
223
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
```

// 以下のコードは、alt + Fnキーの組み合わせでバーチャルコントロールの変更を処理します。
//端末です。この時、eaxはファンクションキーのインデックスまたはオーダー番号（F1 -- 0）であり、対応する

```
240 バーチャルコントロールの端末番号に //。
241 alt_func:
242         pushl %eax                  // push the index number as virtual console number.
243         call _change_console        // chr_dev/tty_io.c, line 87.
244         popl %eax                   // discard the parameter.
244 end_func:
245         ret
246
24  * function keys send F1:'esc [ [ A' F2:'esc [ [ B' etc.
7
7*
248
249  */
250 func_table:
251         .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
252         .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
253         .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b
254
```

// スキャンコード-ASCII文字対応表。
// type defined previously (FINNISH, US, GERMEN, FRANCH).

```
#if     defined(KBD_FINNISH)
key_map:
        .byte 0,27              // for scan code 0x00, 0x01;
        .ascii "1234567890+'"   // for scan code 0x02,...0x0c,0x0d. Bellow is similar.
        .byte 127,9
        .ascii "qwertyuiop}"
        .byte 0,13,0
        .ascii "asdfghjkl|{"
        .byte 0,0
        .ascii "'zxcvbnm,.-"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '<
        .fill 10,1,0

```

```
shift_map:                      // mapping table when shift is pressed at the same time.
        .byte 0,27
        .ascii "!\"#$%&/()=?`"
        .byte 127,9
        .ascii "QWERTYUIOP]^"
```

```
        .byte 13,0
        .ascii "ASDFGHJKL\\["
        .byte 0,0
        .ascii "*ZXCVBNM;:_"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '>
        .fill 10,1,0

alt_map:                        // mapping table when alt key is pressed at the same time.
        .byte 0,0
        .ascii "\0@\0$\0\0{[]}\\\0"
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte '~,13,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0,0,0           /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte 0,0,0,0,0         /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '|
        .fill 10,1,0

#elif defined(KBD_US)           // mapping table for us keyboard.

key_map:
        .byte 0,27
        .ascii "1234567890-="
        .byte 127,9
        .ascii "qwertyuiop[]"
        .byte 13,0
        .ascii "asdfghjkl;'"
        .byte '`,0
        .ascii "\\zxcvbnm,./"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '<
        .fill 10,1,0


shift_map:
        .byte 0,27
        .ascii "!@#$%^&*()_+"
        .byte 127,9
        .ascii "QWERTYUIOP{}"
        .byte 13,0
        .ascii "ASDFGHJKL:\""
```

```
        .byte '~,0
        .ascii "|ZXCVBNM<>?"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '>
        .fill 10,1,0

alt_map:
        .byte 0,0
        .ascii "\0@\0$\0\0{[]}\\\0"
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte '~,13,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0,0,0           /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte 0,0,0,0,0         /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '|
        .fill 10,1,0

#elif defined(KBD_GR)           // mapping table for german keyboard.

key_map:
        .byte 0,27
        .ascii "1234567890\\'"
        .byte 127,9
        .ascii "qwertzuiop@+"
        .byte 13,0
        .ascii "asdfghjkl[]^"
        .byte 0,'#
        .ascii "yxcvbnm,.-"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '<
        .fill 10,1,0


shift_map:
        .byte 0,27
        .ascii "!\"#$%&/()=?`"
        .byte 127,9
        .ascii "QWERTZUIOP\\*"
        .byte 13,0
        .ascii "ASDFGHJKL{}~"
        .byte 0,''
        .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$\0\0{[]}\\\0"
393         .byte 0,0
394         .byte '@,0,0,0,0,0,0,0,0,0,0
395         .byte '~,13,0
396         .byte 0,0,0,0,0,0,0,0,0,0,0
397         .byte 0,0
398         .byte 0,0,0,0,0,0,0,0,0,0,0
399         .byte 0,0,0,0            /* 36-39 */
400         .fill 16,1,0             /* 3A-49 */
401         .byte 0,0,0,0,0          /* 4A-4E */
402         .byte 0,0,0,0,0,0,0      /* 4F-55 */
403         .byte '|
404         .fill 10,1,0
405
406
407 #elif defined(KBD_FR)            // mapping table for France keyboard.
408
409 key_map:
410         .byte 0,27
411         .ascii "&{\"'(-}_/@)="
412         .byte 127,9
413         .ascii "azertyuiop^$"
414         .byte 13,0
415         .ascii "qsdfghjklm|"
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 "WXCVBN?./\\"
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  */
```

// コードはまず、対応するキャラクターマッピングテーブル（alt_map、shift_map、key_map）を選択します。

// モードフラグに応じてマッピングテーブルを検索し、スキャンコードに応じて

ボタンの//を押すと、対応する文字（ASCIIコード）が表示されます。そして、ボタンを押したときに

// 現在の文字がctrlボタンやaltボタンで同時に押され、その文字が

// ASCIIコードの値で、一定の変換を行います。最後に、変換された結果の文字

// は、読み取りバッファのキューに格納されます。

// 次のコードでは、まず、alt_map、shift_map、key_mapのいずれかのマッピングテーブルを選択します。

// モードフラグに基づいています。

```
464         lea alt_map,%ebx            // alt_map table address -> ebx (use alt_map table).
do_self
:
465
466         testb $0x20,mode            /* alt-gr */  // right alt key pressed ?
467         jne 1f                      // jump to label 1 if yes.
468         lea shift_map,%ebx          // otherwise, use shift_map table.
469         testb $0x03,mode            // any shift key pressed ?
470         jne 1f                      // jump to label 1 if yes.
471         lea key_map,%ebx            // otherwise, use key_map table.
```

// これで、使用するマッピングテーブルが決まりました。次は、マッピングの対応する文字

スキャンコードの値に応じて // テーブルを取得します。対応する文字がない場合は

```
472 // マッピングテーブルでは、（turns to none、495行目）と返されます。
473 1:      movb (%ebx,%eax),%al        // get the corresponding char from mapping table.
474          orb %al,%al                 // is it a non-zero ASCII code char ?
475          je none                     // jump to none and return if the code is zero.
```

// ctrlキーが押されているかcapsキーがロックされていて、文字が'a'--'}'の範囲にある場合。

```
    // [0x61--0x7D]の場合は、[0x41--0x5D]の範囲で対応する文字に変換されます。
476 // 0x20をデクリメントします。それ以外の場合は、ラベル2にジャンプして実行を続けます。
477         testb $0x4c,mode            /* ctrl or caps */   // ctrl pressed or caps lock ?
478         je 2f                       // jump to label 2 if not.
479         cmpb $'a,%al                // check to see if the char is in range of 'a'-- '}'
480         jb 2f                       // jum to label 2 if not.
481         cmpb $'},%al
482         ja 2f
483         subb $32,%al                // The char ASCII code value is decremented by 0x20.

    // ctrlキーが押されていて、文字が'@'--'_' [0x40--0x5F]の範囲にある場合。
    // 0x40から差し引かれ、[0x00--0x1F]の範囲の値に変換されます。の文字は
    // この範囲は制御文字です。つまり，ctrl + ['@'--'_'] の範囲にある文字は
    //は、[0x00-0x1F]の範囲で対応する制御文字を生成することができます。例えば
484 // ctrl + 'M'で対応するキャリッジリターン制御文字が生成されます。
485 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].

    // 左altキーが同時に押された場合、文字のビット7が設定されます。つまり
    // 0x7f以上の拡張文字セットの文字を生成することができます。
489 この時に///。
490 3:      testb $0x10,mode            /* left alt */  // left alt pressed ?
491         je 4f                       // jump to label 4 if not
492         orb $0x80,%al               // set bit 7 of the char.

493 // 最後に、alの文字を読み込みバッファのキューに入れます。
494 4:      andl $0xff,%eax             // only one character
495         xorl %ebx,%ebx
496         call put_queue
497 none:   ret
498
497 /*
498 * マイナスは、独自のルーチンを持っていて、「E0h」の前に
499 * マイナスのスキャンコードは、テンキーの
500 * スラッシュが押されま
した。501 */
    // なお、フィンランド語やドイツ語のキーボードでは、スキャンコード0x35が'-'キーに対応しています。
502 // 264行目と365行目をご覧ください。
503 minus:  cmpb $1,e0                 // e0 flag is set to 0x01 ? (e0 received ?)
504         jne do_self                 // jump to normal processing of char if no.
505         movl $'/,%eax               // otherwise replace '-' with '/'
506         xorl %ebx,%ebx
507         jmp put_queue
507
```

508 /*
509 * このテーブルは、スキャンコードが実行されたときに、どのルーチンを呼び出すかを決定します。
510 * ゲットしました。ほとんどのルーチンは do_self を呼び出すだけですが、次のような場合には何もしません。
511 * 彼らはメイク・オア・ブレイクです。

```
 */      // Note that 'make' means a key is pressed, and 'break' means released.
key_table:
        .long none,do_self,do_self,do_self      /* 00-03 s0 esc 1 2 */
        .long do_self,do_self,do_self,do_self   /* 04-07 3 4 5 6 */
        .long do_self,do_self,do_self,do_self   /* 08-0B 7 8 9 0 */
        .long do_self,do_self,do_self,do_self   /* 0C-0F + ' bs tab */
        .long do_self,do_self,do_self,do_self   /* 10-13 q w e r */
        .long do_self,do_self,do_self,do_self   /* 14-17 t y u i */
        .long do_self,do_self,do_self,do_self   /* 18-1B o p } ^ */
        .long do_self,ctrl,do_self,do_self      /* 1C-1F enter ctrl a s */
        .long do_self,do_self,do_self,do_self   /* 20-23 d f g h */
        .long do_self,do_self,do_self,do_self   /* 24-27 j k l | */
        .long do_self,do_self,lshift,do_self    /* 28-2B { para lshift , */
        .long do_self,do_self,do_self,do_self   /* 2C-2F z x c v */
        .long do_self,do_self,do_self,do_self   /* 30-33 b n m , */
        .long do_self,minus,rshift,do_self      /* 34-37 . - rshift * */
        .long alt,do_self,caps,func             /* 38-3B alt sp caps f1 */
        .long func,func,func,func               /* 3C-3F f2 f3 f4 f5 */
        .long func,func,func,func               /* 40-43 f6 f7 f8 f9 */
        .long func,num,scroll,cursor            /* 44-47 f10 num scr home */
        .long cursor,cursor,do_self,cursor      /* 48-4B up pgup - left */
        .long cursor,cursor,do_self,cursor      /* 4C-4F n5 right + end */
        .long cursor,cursor,cursor,cursor       /* 50-53 dn pgdn ins del */
        .long none,none,do_self,func            /* 54-57 sysreq ? < f11 */
        .long func,none,none,none               /* 58-5B f12 ? ? ? */
        .long none,none,none,none               /* 5C-5F ? ? ? ? */
        .long none,none,none,none               /* 60-63 ? ? ? ? */
        .long none,none,none,none               /* 64-67 ? ? ? ? */
        .long none,none,none,none               /* 68-6B ? ? ? ? */
        .long none,none,none,none               /* 6C-6F ? ? ? ? */
        .long none,none,none,none               /* 70-73 ? ? ? ? */
        .long none,none,none,none               /* 74-77 ? ? ? ? */
        .long none,none,none,none               /* 78-7B ? ? ? ? */
        .long none,none,none,none               /* 7C-7F ? ? ? ? */
        .long none,none,none,none               /* 80-83 ? br br br */
        .long none,none,none,none               /* 84-87 br br br br */
        .long none,none,none,none               /* 88-8B br br br br */
        .long none,none,none,none               /* 8C-8F br br br br */
        .long none,none,none,none               /* 90-93 br br br br */
        .long none,none,none,none               /* 94-97 br br br br */
        .long none,none,none,none               /* 98-9B br br br br */
        .long none,unctrl,none,none             /* 9C-9F br unctrl br br */
        .long none,none,none,none               /* A0-A3 br br br br */
        .long none,none,none,none               /* A4-A7 br br br br */
        .long none,none,unlshift,none           /* A8-AB br br unlshift br */
        .long none,none,none,none               /* AC-AF br br br br */
        .long none,none,none,none               /* B0-B3 br br br br */
        .long none,none,unrshift,none           /* B4-B7 br br unrshift br */
        .long unalt,none,uncaps,none            /* B8-BB unalt br uncaps br */
        .long none,none,none,none               /* BC-BF br br br br */
        .long none,none,none,none               /* C0-C3 br br br br */
        .long none,none,none,none               /* C4-C7 br br br br */
        .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 は、キーボードコントローラのバッファが空になるのを待ちます。
581 * タイムアウトはありません。バッファが空にならなければハングアッ
プします。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  */
    // 本サブルーチンでは、値0x1234を物理メモリアドレス0x472に書き込みます。この位置は
    // リブートモードフラグを設定します。ブートプロセス中、ROM BIOSはリブートモードフラグを読み
    込んで
    // の値に基づいて次の実行を指示します。値が0x1234の場合、BIOSは次の実行をスキップします。
    // メモリ検出処理を行い、ウォームブート処理を行います。この値が0の場合は、コールドブート
    //処理を行います。
594再起      call kb_wait                     // wait controller buffer to empty.
動しま
す。
595
596         movw $0x1234,0x472               /* don't do memory check */
597         movb $0xfc,%al                   /* pulse reset and A20 low */
598         outb %al,$0x64
599 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

出力ポートのビット1(P21端子)は，A20信号線をオンにするかどうかの制御に使用されます。出力ポートのビット1が1のとき，A20信号線がオン（ストローブ）になり，0のとき，A20信号線がディセーブルになります。

図10-7 キーボードコントローラ804Xの回路図

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.

表10-1 キーボードコントローラ804Xのポート 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.

表10-2 キーボードコマンド一覧 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.

表 10-3 キーボードコントローラコマンド一覧 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.

表10-4XTキーボードのスキャンコー	F2	`	1	2	3	4	5	6	7	8	9	0	-	=	\	BS	ESC	NUML	SCRL	SYSR

ド表 F1																			
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

はキーを押したときのスキャンコード、ビット7＝1はキーを離したときのスキャンコードを示す。例えば、ESCキーを押した場合、システムに送信されるスキャンコードは1（1はESCキーのスキャンコード）となり、キーを離すと1+0x80=129のスキャンコードが生成されます。

PC、PC/XTの標準83キーキーボードの場合、スキャンコードはキーナンバー（キーのポジションコード）と同じで、1バイトで表現されます。例えば、「A」キーの場合、キーポジション番号は30で、メイクコードとスキャンコードも30（0x1e）、ブレークコードはメイクコードに0x80を加えた0x9eとなります。

一部の「拡張」キーでは状況が多少異なります。拡張キーが押されると、割り込みが発生し、キーボードは拡張スキャンコードのプレフィックスである0xe0を出力しますが、次の割り込みでは「拡張」スキャンコードが与えられます。例えば、PC/XT標準キーボードの場合、左のコントロールキーctrlのスキャンコードは29(0x1d)であり、右の「拡張」コントロールキーctrlは拡張スキャンコードシーケンス0xe0, 0x1dとなります。このルールは、alt、方向キーにも適しています。

さらに、PrtScnキーとPause/Breakキーという非常に特殊なキーがあります。PrtScnボタンを押すと、42(0x2a)と55(0x37)の2つの拡張文字がキーボード割り込みルーチンに送られるので、実際のバイト列は0xe0, 0x2a, 0xe0, 0x37となり、ボタンを繰り返し発生させると、拡張文字0xaaも送られる、つまり0xe0, 0x2a, 0xe0, 0x37, 0xe0, 0xaaという配列が生成されることになります。キーを離すと，2つの拡張コードと0x80（0xe0, 0xb7, 0xe0, 0xaa）が再送される。PrtScnキーが押されたとき、shiftキーまたはctrlキーも押されていた場合は、0xe0, 0x37のみが送信され、離されたときは0xe0, 0xb7のみが送信される。altキーが同時に押された場合、PrtScnキーはスキャンコード0x54の通常のキーのようになります。

Pause/Breakキーの場合、キーを押しながらコントロールキーctrlのいずれかを押すと、拡張キー70（0x46）のような行になり、それ以外の場合は0xe1, 0x1d, 0x45, 0xe1, 0x9d, 0xc5という文字列が送信されます。ボタンをずっと押していても重複したスキャンコードは出てきませんし、ボタンを離してもスキャンコードは出てきません。したがって、スキャンコード0xe0はあと1文字続いていることを意味し、スキャンコード0xe1は2文字続いていることを意味する、という見方と対処方法ができます。

ATキーボードのスキャンコードは、PC/XTのものとは若干異なります。キーを押すと、対応するキーのスキャンコードが送信されるが、キーを離すと2バイトが送信され、1バイト目は0xf0で、2バイト目は同じキーのスキャンコードである。キーボード設計者は現在、ATキーボードの入力プロセッサとして8049を使用しており、下位互換性のために、ATキーボードからのスキャンコードは、システムに送信される前に古いPC/XT標準キーボードのスキャンコードに変換されます。

10.3 console.c

10.3.1 Function description

console.cは、カーネルの中でも最も長いプログラムの一つですが、その機能は比較的単純です。すべてのサブルーチンは、端末画面の書き込み関数con_write()と、端末画面表示の制御操作を実装するために使用されています。

con_write()関数は、コンソールデバイスへの書き込みが行われる際に呼び出されます。この関数は、アプリケーションの画面管理操作全体を提供するすべての制御およびエスケープ文字シーケンスを管理します。実装されているエスケープシーケンスは、vt102ターミナル仕様を使用しています。つまり、Linux以外のホストにtelnetプログラムを使用して接続する場合、環境変数にTERM=vt102を設定する必要があります。しかし、Linuxのコンソールはvt102の機能のスーパーセットを提供しているため、ローカルでの操作にはTERM=consoleを設定するのが最適です。
con_write()関数は、1文字ずつの有限長状態の自動エスケープシーケンスを解釈するための変換文を中心に構成されています。通常のモードでは、表示文字の書き込み

を現在の属性を使用して表示メモリに直接表示する。この関数は、端末のtty_structデータ構造の書き込みバッファ・キューwrite_qから文字または文字列を取り出し、文字の性質（通常の文字、制御文字、エスケープ・シーケンス、制御シーケンス）に応じて端末の画面に文字を表示したり、カーソルの移動や文字の消去などの画面制御操作を行います。

端末画面の初期化関数con_init()は、システムの初期化開始時に得られるシステム情報に従って、画面に関するいくつかの基本的なパラメータ値を設定し、con_write()関数の動作に使用される。

端末デバイスの文字バッファキューの説明は、include/linux/tty.hヘッダーファイルを参照してください。このヘッダーファイルには、文字バッファキューのデータ構造tty_queue、端末のデータ構造tty_struct、およびいくつかの制御文字の値が示されています。また，バッファキューを操作するいくつかのマクロ定義もあります。バッファキューとその動作の模式図は図10-14を参照してください。

10.3.2 Code annotation

プログラム 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
```

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_structや

```
//      data of the initial task 0, and some embedded assembly function macro statements
```

```
//      about the descriptor parameter settings and acquisition.
// <linux/tty.h> ttyヘッダーファイルは、tty_io、シリアルのパラメータと定数を定義しています。
//      communication.
// <linux/config.h> カーネル設定用のヘッダーファイルです。キーボードの言語とハードを定義する
//      disk type (HD_TYPE) options.
// <linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。
//      commonly used functions of the kernel.
// <asm/io.h> Io のヘッダーファイルです。の io ポートを操作する関数を定義します。
//      form of a macro's embedded assembler.
// <asm/system.h> システムのヘッダーファイルです。を定義する埋め込みアセンブリマクロです。
//      modifies descriptors/interrupt gates, etc. is defined.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
//      for segment register operations.
// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/config.h>
#include <linux/kernel.h>

#include <asm/io.h>
#include <asm/system.h>
#include <asm/segment.h>

#include <string.h>
#include <errno.h>

// このシンボリック定数は、ターミナルIO構造のデフォルトデータを定義します。のためのものです。
// シンボリックな定数は、include/termios.hファイルを参照してください。
#define DEF_TERMIOS (struct termios) {
        ICRNL, \
        OPOST | ONLCR, \
        0, \
        IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, \
        0, \
        INIT_C_CC \
}


/*
 * これらは起動時のsetup-routineによって設定されます。
 */
// boot/setup.sとシステムパラメータテーブルの読み込みと保存に関するコメントを参照してください。
```

```
//セットアッププログラムによる

#define ORIG_X            (*(unsigned char *)0x90000)        // original cursor colum no.
#define ORIG_Y            (*(unsigned char *)0x90001)        // original cursor row no.
#define ORIG_VIDEO_PAGE   (*(unsigned short *)0x90004)       // current video page.
#define ORIG_VIDEO_MODE   ((*(unsigned short *)0x90006) & 0xff)          // display mode.
#define ORIG_VIDEO_COLS   (((*(unsigned short *)0x90006) & 0xff00) >> 8) // screen colums.
#define ORIG_VIDEO_LINES  ((*(unsigned short *)0x9000e) & 0xff)          // screen rows.
#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
   // モノクローム／カラー表示モードタイプのシンボル定数を定義します。
   VIDEO_TYPE_MDA    0x10    /* Monochrome Text Display    */
70 #define
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
   // 以下のスタティック変数は、このファイルで使用されているグローバル変数の一部です。
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
   // バーチャルコンソールの構造は以下のように定義されています。の現在の情報をすべて含んでいます。
   // vc_origin と vc_scr_end は、対応するディスプレイメモリの位置です。
   // 現在処理されているバーチャルコンソールで使用されている開始行と最終行に
   // vc_video_mem_start と vc_video_mem_end は、ディスプレイの
92 // 現在の仮想コンソールで使用されているメモリ領域です。
93 static struct {
94         unsigned short  vc_video_erase_char;    // erase char attributes & char (0x0720)
95         unsigned char   vc_attr;                // char attribute.
96         unsigned char   vc_def_attr;            // default attribute.
97         int             vc_bold_attr;           // bold char attribute.
98         unsigned long   vc_ques;                // question mark char.
99         unsigned long   vc_state;               // state of the escape or control sequence.
```

```
100     unsigned long   vc_restate;             // next state of escape or control sequence.
101     unsigned long   vc_checkin;
102     unsigned long   vc_origin;              /* Used for EGA/VGA fast scroll*/
103     unsigned long   vc_scr_end;             /* Used for EGA/VGA fast scroll*/
104     unsigned long   vc_pos;                 // The mem location of the current cursor.
105     unsigned long   vc_x,vc_y;              // current cursor column, row value.
106     unsigned long   vc_top,vc_bottom;       // The top & bottom line nr when scrolling.
107     unsigned long   vc_npar,vc_par[NPAR];   // escape sequence param nr and array.
108     unsigned long   vc_video_mem_start;     /* Start of video RAM          */
109     unsigned long   vc_video_mem_end;       /* End of video RAM (sort of)  */
110     unsigned int    vc_saved_x;             // The saved cursor column number.
111     unsigned int    vc_saved_y;             // The saved cursor row number.
112     unsigned int    vc_iscolor;             // The color display flag.
113     char *          vc_translate;           // The character set used.
```

```
114 } vc_cons [MAX_CONSOLES];
115
    // 参考までに、現在使用しているコンソールのシンボルを以下のように定義します。
    // 処理されます。上記と同じ意味です。ここで、currconsは現在の仮想端末
    116                     // vc_cons[]構造体を使って、関数の引数に番号を入れます。
117 #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
144 143 /*
145  * this is what the terminal answers to a ESC-Z or csi0c
146  * query (= vt100 response).
147  */                                 // csi - Control Sequence Introducer.
    // ホストは、端末に対して、デバイス属性制御シーケンスの応答を
    // デバイスアトリビュート（DA）制御シーケンスで、パラメータなし、またはパラメータ0（'ESC [c'
    or
    // 'ESC [0c'）(ESC-Zも同じ働きをします)。端末は、以下のシーケンスを送信して応答します。
    // ホストです。このシーケンス（すなわち「ESC [?1; 2c」）は、端末がVT100互換であることを示しま
    す。
148 高度なビデオ機能を備えた//ターミナル。
149 #define RESPONSE "\033[?1;2c"
150
    // 使用する文字セットを定義します。上の部分は通常の7ビットのASCIIコードで、これは
    // US文字セットに対応しています。下の部分はVT100端末のライン文字に対応しています。
151 // デバイス、つまり、チャートラインを表示するキャラクタセットです。
152 static char * translations[] = {
150 /* normal 7-bit ascii */
```

```
151             " !\"#$%&'()*+,-./0123456789:;<=>?"
152             "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_"
153             "`abcdefghijklmnopqrstuvwxyz{|}~ ",
154 /* vt100 graphics */
```

```
155         " !\"#$%&'()*+,-./0123456789:;<=>?"
156         "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^ "
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
    //// カーソルの現在の位置をトラッキングします。
    // パラメータ: currcons - 現在の仮想端末; new_x - カーソルの列番号。
    // new_y - カーソルラインの番号です。
    // この関数は、現在のカーソル位置の変数 x, y を更新し、修正するために使用されます。
    // 表示メモリ内のカーソルの対応する位置pos。この関数は、まず
    // パラメータの有効性を確認します。与えられたカーソルの列数が最大値を超えた場合は終了します。
    // ディスプレイに表示されている列数、またはカーソルラインの番号が最大値よりも低くない場合
    // 表示される行の数。それ以外の場合は、現在のカーソル変数と新しいカーソル位置の
    // が更新され、ディスプレイメモリの位置posに対応します。なお、すべての変数
    関数内の // は、実際には vc_cons[currcons] 構造体の対応するフィールドです。
164 // と以下の機能があります。
165 /* NOTE! gotoxy thinks x==video_num_columns is ok */
166 static inline void gotoxy(int currcons, int new_x,unsigned int new_y)
166 {
167         if (new_x > video_num_columns || new_y >= video_num_lines)
168                 return;
169         x = new_x;
170         y = new_y;
171         pos = origin + y*video_size_row + (x<<1);       // One col needs 2 bytes, so x<<1.
172 }
173
    //// スクロールスタート表示のメモリアドレスを設定します。
    // まず、ディスプレイカードの種類がカラーカードであるかどうかを確認し、コンソールの
    パラメータで指定された // がフロントコンソールで、2つの条件が満たされない場合は終了します。
    // それ以外の場合は、スクロール開始アドレスをディスプレイカード上のレジスタに出力します。
174 static inline void set_origin(int currcons) 175 {。
    // まず、ディスプレイカードの種類を決定します。EGA/VGAカードの場合は、画面上の
    // MDAモノクロディスプレイカードではフルスクリーンでの表示しかできないが、スクロールできる範囲（エリア）は
    //スクロールします。そのため、EGA/VGAカードのみ、ディスプレイメモリのアドレスを設定する必要があります。
    // スクロールの開始ライン（開始ラインは原点に対応するライン）、それ以外は
    // の場合は直接終了します。さらに、フォアグラウンドのコンソールでのみ操作を行うため、以下の場合のみ
    // 現在のコンソールcurrconsはフォアグラウンドのコンソールなので、メモリスタートを設定する必要がある
176     // スクロール開始線に対応する位置です。
177         if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
```

```
178                 return;
179         if (currcons != fg_console)
180                 return;
```

// そして、12を「ディスプレイ・レジスタ・セレクト・ポート」のvideo_port_regに出力し、それによって

// 表示制御データレジスタr12に、スクロール開始アドレスの上位バイトを書き込みます。

// 9ビットを右にシフトするということは、実際には8ビットを右にシフトして除算することになります。

// 2（画面上の1文字を2バイトで表現）。次に、表示制御を選択します。

// データレジスタ r13 にスクロール開始アドレスの下位バイトを書き込みます。に1ビットシフトします。

// 右は2で割るということで、画面上の1文字を表現することにもなります。

// 2バイトのメモリで 出力値は、デフォルトのディスプレイメモリの開始位置を基準にして動作します。
// video_mem_base の位置、例えば EGA/VGA カラーモードの場合は viedo_mem_base = physical
181 // メモリアドレス0xb8000。

```
182         cli();
183         outb_p(12, video_port_reg);        // Select data register r12 to output high byte.
184         outb_p(0xff&((origin-video_mem_base)>>9), video_port_val);
185         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
```

//// コンテンツが1行分上にスクロールします。
// 画面のスクロールウィンドウを1行下に移動し、新しい行にスペース文字を追加して
画面のスクロール領域の下部に // が表示されます。スクロールエリアの大きさは
// 1行を表示します。スクロール操作の原理については、10.3.3.2項を参照してください。

188 static void scrup(int currcons) 189
{。

// スクロールエリアには少なくとも2行が必要です。スクロールエリアの先頭行番号が
// が最下段の行数以上であれば、ローリング動作の条件となる
//が満たされない。また、EGA/VGAカードの場合は、画面上の範囲（エリア）を指定して
// スクロールは、MDAモノクロディスプレイカードでは全画面スクロールしかできません。
// この関数は、ディスプレイタイプがEGAとMDAの場合、別々に処理します。ディスプレイタイプが
EGA の場合は
// また、フルスクリーン・ウィンドウ・ムーブメントとイントラ・エリア・ウィンドウ・ムーブメント
に分けられる。ここでは

```
//      if (bottom<=top)
```

このコードでは、まずディスプレイカードがEGA/VGAタイプ

の場合に対応しています。

```
190
191                 return;
192         if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
193         {
// 移動開始ラインtop=0、移動終了ライン=video_num_lines=25とすると、全体の
// 画面のウィンドウが1行下に移動します。したがって、開始メモリ位置の原点
画面全体のウィンドウの左上に対応する//メモリに調整されている
// の位置を1行分下方にシフトし、現在のメモリ位置に対応する
// カーソルと文字ポインタの位置 scr_end で画面の終わり end は
// を追跡します。最後に、新しいスクリーンウィンドウのメモリ開始位置の原点が
// ディスプレイコントローラー。
195                 if (!top && bottom == video_num_lines) {
196                         origin += video_size_row;
197                         pos += video_size_row;
198                         scr_end += video_size_row;
// スクリーンウィンドウの端に対応する表示メモリポインタscr_endが
// を除くすべてのラインに対応するメモリデータが、実際のディスプレイメモリの最後に
// 画面のコンテンツの最初の行を、開始位置 video_mem_start に移動します。
// の後に表示される新しい行にスペース文字を記入します。
// 画面のウィンドウが下に移動しました。続いて、画面メモリの状況に応じて
// データを移動すると、現在の画面に対応するスタートポインタ、カーソル位置
// ポインタと、それに対応する画面端のメモリポインタscr_endが再調整されます。
//
// この組み込みアセンブリコードは、まず、（画面の文字）に対応するメモリデータを移動させます。
// 行番号-1）行をディスプレイメモリの開始位置video_mem_startに追加してから
// 後続のメモリ位置に1行のスペース（消去）文字データを入れる。
// %0 - eax (消去文字 + 属性); %1 - ecx ((画面の行数 -1))
// 文字数に対応する / 2、長い言葉で動く）; %2 - edi（メモリを表示する
// 開始位置 video_mem_start); %3 - esi (スクリーン・ウィンドウ・メモリの開始位置の原点).
```

// 移動の方向。[edi] -> [esi], move ecx long words.

```
200                     if (scr_end > video_mem_end) {
201                             __asm__("cld\n\t"       // clear direction.
202                                     "rep\n\t"       // repeat move data.
203                                     "movsl\n\t"
204                                     "movl _video_num_columns,%1\n\t"
205                                     "rep\n\t"       // fill a line of spaces.
206                                     "stosw"
207                                     :: "a" (video_erase_char),
208                                     "c" ((video_num_lines-1)*video_num_columns>>1),
209                                     "D" (video_mem_start),
210                                     "S" (origin)
209                                     :"cx", "di", "si");
210                             scr_end -= origin-video_mem_start;
211                             pos -= origin-video_mem_start;
212                             origin = video_mem_start;
```

// 調整された画面の最後にあるメモリポインタscr_endが、調整された画面の最後を超えていない場合は

// 表示メモリ video_mem_end の消去行（スペース文字）を埋めればよい。

// %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                     }
```

// そして、新しいスクリーン・ウィンドウ・メモリの開始位置の原点をディスプレイ・コントローラに書き込みます。

```
222                     set_origin(currcons);
```

// それ以外の場合は、フルスクリーンでの移動ではないということです。つまり、からのすべての行は

// 指定した行の上端から下端までの領域が1行分上に移動し、指定した行の上端が

//が削除されます。このとき、画面の全行に対応する表示メモリデータ

指定された行頭から行末までの // がそのまま1行分上に移動して、消去

新たに表示される行には、 // の文字が入ります。

// %0 - eax (消去文字 + 属性); %1 - ecx (先頭からのロングワードの数 + 1)

// 行から下の行まで）; %2 - edi（上の行があるメモリの位置）; %3

// - esi

```
} else {
```

(最上位+1列目が配置されているメモリの位置)です。

```
223
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         }
    // ディスプレイの種類がEGAではなくMDAの場合、以下のような移動操作を行います。なぜなら
    // MDAディスプレイカードは、画面全体をスクロールするだけで、自動的に
    // 表示メモリの範囲を超えた（つまり、ポインタが自動的にスクロールする）ので
    // 画面の内容に対応したメモリを、表示とは別に処理しない。
        else            /* Not EGA/VGA */
237
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
    //// 表示内容が1行分下にスクロールします。
    // 画面スクロールウィンドウを1行上に移動し、対応する画面スクロール領域の内容を
    // が1行下に移動します。新しい行は、移動の開始行の上に表示されます。処理は
    // メソッドは scrup() と似ていますが、データのオーバーレイを行うのは
    //メモリデータで、コピー動作が逆になります。つまり、最後の文字からのコピーは
    // カウントダウンの2行目を最終行にして、3行目の文字をコピーします。
    カウントダウンの//行を、最後の2行目にする、といった具合です。
253 static void scrdown(int currcons) 254
{...
    // 同様に、スクロールエリアには少なくとも2行が必要です。の先頭行番号が2行以上であれば
    // スクロールエリアの最下段の行番号以上の条件である
    ローリング動作のために // を満たしていません。また、EGA/VGAカードに対しては、指定した
    // MDAモノクロディスプレイカードでは、画面上のスクロール可能な範囲（エリア）が
    // フルスクリーンスクロールを行います。ディスプレイの開始位置までウィンドウが移動するので
    // 最大でエリアメモリ、画面の終わりに対応する表示メモリポインタscr_end
    // ウィンドウが実際のディスプレイメモリの端を超えないように、通常のメモリデータの移動のみ
255     // をここで処理する必要があります。
256         if (bottom <= top)
257                 return;
258         if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
258         {
    // %0 - eax (消去文字 + 属性); %1 - ecx (対応する長い単語の数)
    // 上から下までの行数-1); %2 - edi (の最後の長い単語の位置)
    // ウィンドウの右下隅）; %3 - Esi（2行目の最後のロングワードの位置
    ウィンドウのカウントダウンの//）。
     // 移動          __asm__("std\n\t"          // set direction flag!!
     の方
     向。
     [esi] -
     >
     [edi],
     move
     ecx
     long
     words.
259
260                         "rep\n\t"
261                         "movsl\n\t"
262                         "addl $2,%%edi\n\t" /* %edi has been decremented by 4 */
```

のラインになります。

```
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
    //// カーソルは同じ列の位置で1行下に移動します。
    // カーソルが最終行にない場合、カーソルの現在行変数を直接変更する y++,
    // カーソルに対応する表示メモリ位置posを調整する（＋メモリ長を調整する
    // 1行分の文字数に対応）。それ以外の場合は、その内容を移動する必要があります。
    // 画面のウィンドウを1行分上げる。関数名のlf（ラインフィード）は、処理制御の
    // 文字 LF.
289         if (y+1<bottom) {
```

```
static void lf(int c)
```

u
r
r
c
o
n
s
)

2
9
0

{
。

```
291
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
```

//// カーソルは同じ列の1行上に移動します。

// カーソルが画面の最初の行にない場合は、カーソルのカレントを直接変更する

// 行変数y--を調整し、カーソル対応表示メモリ位置pos、マイナス

// 画面上の文字列1行に対応するメモリの長さのバイト数です。

// そうでなければ、スクリーンウィンドウの内容を1行下に移動させる必要があります。このとき、関数

// 名前のri（リバースインデックス）は、制御文字RIまたはエスケープシーケンス "ESC M "を指します。

```
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
```

// カーソルは0列目に戻ります。
// メモリロケーションposに対応するカーソルを調整する。カーソルの列番号
// *2は、0列が占めるメモリバイトの長さで、その列までの行では
// カーソルの位置を示します。関数名のcr（キャリッジリターン）は、制御文字である
// 処理中です。

```
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
```

// カーソルの前の文字を消去し、カーソルを1列前に移動させる。
// カーソルが0列目にない場合、カーソルは2バイトでバックされる。
// メモリポジションpos（画面上の1文字に対応）、次にカレントカーソル
// の列が1だけデクリメントされ、カーソル位置の文字が消去されます。

```
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
```

//// 画面上のコンテンツのうち、カーソル位置に関連する部分を削除する。
// この関数は、ANSI制御シーケンスを処理するために使用されます。文字の削除操作
指定された制御シーケンスに従って、カーソル位置に関連付けられた // を実行します。
// 表示されている文字の一部または全部が消去され、カーソル位置は変更されません。
文字や行が消去されたときに、//。
// この関数で処理されるANSI制御シーケンスは： 'ESC [Ps J' です。その中で、Ps = 0
// - カーソルを画面の一番下まで削除することを意味し、1 - カーソルまで画面を削除することを意味します。
// 2 - 画面全体を消す。
// 機能名csi_J（CSI - Control Sequence Introducer）は、制御の
// シーケンス「CSI Ps J」の処理を行います。引数'vpar'は、'Ps'の値に対応します。
// 上記の制御シーケンスを行います。端末制御コマンドの紹介については、セクション
プログラムリストの後に // 10.3.3.3. よく使われるエスケープシーケンスと制御シーケンス

```
326 // の付録3に記載されています。324 static void
csi_J(int currcons, int vpar) 325 {。
327         long count __asm__("cx");             // set to use register variable.
```

```
328         long start asm ("di");
328
    // まず、削除する文字数と、削除するディスプレイメモリの位置を設定します。
329     上記の3つのケースに応じて // 削除を開始します。
330         switch (vpar) {
331                 case 0: /* erase from cursor to end of display */
332                         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         }
    // その後、消去文字を使って、文字が消えた場所を埋める。
    // %0 -ecx (削除された文字数); %1 -edi (削除開始アドレス); %2 -eax (埋められた消去文字)。
```

352

```
374                         return;
375         }
    // その後、消去文字を使って、文字が消えた場所を埋める。
    //      __asm__("cld\n\t"
    %
    0 -
    ec
    x
    (d
    el
    et
    e
    co
    un
    t);
     %
    1 -
    ed
    i
    (d
    el
    et
    e
    ad
    dr
    es
    s
    sta
    rt)
    ;
    %
    2 -
    ea
    x
    (fi
    ll
    er
    as
    e
    ch
    ar
    ac
    ter
    ).
376
377                 "rep\n\t"
378                 "stosw\n\t"
379                 :: "c" (count),
380                 "D" (start), "a" (video_erase_char)
381                 : "cx", "di");
382 }
383
    //// 表示キャラクターの属性を設定します。
    // 制御シーケンスは、パラメータに応じて文字表示属性を設定します。すべての
    // 今後、端末に送られる文字には、ここで指定された属性が使用されます。
    // 文字表示の属性をリセットするために、再び制御シーケンスが使用されます。
    // ANSIのエスケープシーケンス：'ESC [ Ps; Ps m'. ここで Ps = 0 - デフォルトの属性、1 - 太くて明るい。
    // 4 - 下線、5 - 閃光、7 - 反転、22 - 非太字、24 - 下線なし、25 - 閃光なし。
    // 27 - 通常 ;30--38 - 前景色の設定 ;39 - デフォルトの前景色（白）。
```

```
// 40--48 - 背景色の設定; 49 - デフォルトの背景色（黒）。
384 void csi_m(int currcons )
385 {。
386         int i;
387
// 1つの制御シーケンスには、複数の異なるパラメータを設定することができます。そのパラメータは
// 配列par[]. 次のコードは，各パラメータPsを数に応じて循環的に処理します.
// 受信したパラメータnparの
// Ps = 0 の場合は、現在のバーチャルコンソールの横に表示される文字属性
// には，デフォルトの属性 def_attr が設定されています。初期化時、def_attr は 0x07 に設定されています。
//（黒地に白）。
// Ps = 1の場合、後に表示される文字属性が太字やハイライトに設定されます。
// カラーディスプレイの場合は、文字属性または上位0x08が強調表示されます。
// 文字。モノクロディスプレイの場合は、文字に下線を引く。
// Ps = 4の場合、カラーディスプレイとモノクロディスプレイの扱いが異なります。カラーディスプレイの場合
現時点では//モードは使用できません。文字には下線が引かれています。カラーディスプレイの場合
// 元の vc_bold_attr が -1 でない場合は背景色がリセットされ、そうでない場合は
// 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 forground == background */
398                             if (vc_cons[currcons].vc_bold_attr != -1)
399                               attr = (vc_cons[currcons].vc_bold_attr&0x0f)|(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;
    // Ps = 5の場合、現在のバーチャルコンソールに表示される文字は
    // ブリンクに設定されている、つまり、属性バイトのビット7が1に設定されている。
    // Ps = 7の場合は、その後に表示される文字が逆に設定され、つまり前景
    // と背景色が入れ替わります。
    // Ps = 22 の場合、それ以降の文字の強調表示を中止する（太字表示を中止する）。
    // Ps = 24 の場合、モノクロ表示時に後続の文字の下線がキャンセルされる
    //と緑はカラー表示のためにキャンセルされます。
    // Ps = 25の場合、それ以降の文字の点滅はキャンセルされます。
    // 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).
                        case 5: attr=attr|0x80;break; /* blinking */
```

```
408
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;
    // Ps(par[i])が別の値のとき、指定された前景色または背景色が設定される。
    // the instructions following the program list for color values.
                        default:
```

```
416
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
    //// ディスプレイカーソルを設定します。
    // メモリの位置に応じてコントローラのカーソルの表示位置を設定する
    // カーソルの位置。
430 static inline void set_cursor(int currcons) 431 {。
    // ディスプレイカーソルを設定する必要があるということは、キーボード操作があるということなので、
    次のようにします。
    //で黒画面操作の遅延カウント値を元に戻すことができます。また、コンソールの
432 // カーソルを表示するのはフォアグラウンドのコンソールでなければなりません。
433         blankcount = blankinterval;            // Resets the count of the black screen.
434         if (currcons != fg_console)
435                 return;
```

```
//次に、インデックスレジスタポートを使用して、表示制御データレジスタr14（現在の
```

// カーソルの表示位置（上位バイト）を表示し、カーソルの現在位置を書き込む
// 上位バイト（9ビット右に移動すると上位バイトが下位バイトに移動して分割される
//を2で割ったもの）。これは、デフォルトのディスプレイメモリの操作に相対するものです。次に、インデックスレジスタ
436 //でr15を選択し、カーソルの現在位置の下位バイトを書き込む。

```
437         cli();
438         outb_p(14, video_port_reg);
439         outb_p(0xff&((pos-video_mem_base)>>9), video_port_val);
440         outb_p(15, video_port_reg);
441         outb_p(0xff&((pos-video_mem_base)>>1), video_port_val);
442         sti();
441 }
442
```

// カーソルを隠す。
// カーソルを隠すために、現在のバーチャルコンソールウィンドウの端に設定します。

```
443 static inline void hide_cursor(int currcons) 444 {.
```

// 最初に表示制御データレジスタr14（カーソルの現在の表示位置）を選択する
// 上位バイト）を書き込んだ後、カーソル位置の上位バイトを（9ビット右に移動して
// は上位バイトを下位バイトに移動して2で割ることを意味します）。その後、r15を選択し、次のように書き込みます。
// カーソルの現在位置の下位バイト。

```
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
```

//// VT100 に応答シーケンスを送信します。
// つまり、ホストが端末を要求することに応答して、デバイス属性（DA）を送信する
//をホストに送信します。ホストは、端末にデバイスアトリビュート（DA）コントロールを送り返すよう要求します。
ESC[0c]」または「ESC Z」などのDA制御シーケンスをパラメータなしで送信することで、 // シーケンスを作成することができます。
// そして、端末は147行目で定義された応答シーケンス（例：「ESC [?1; 2c」）を送り返します。
ホストのシーケンスに対応して // を表示します。このシーケンスは、ホストに対して、端末がVT100
高度なビデオ機能を備えた//互換性のある端末です。処理は、応答を
// シーケンスをリードバッファのキューに入れ、copy_to_cooked()関数で処理して
// 補助（二次）キューに入れる。

```
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
```

//// カーソル位置にスペース文字を挿入します。

// カーソルの先頭のすべての文字を1スペース右に移動し、挿入する

// カーソルでの消去文字。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
    //// カーソル位置に行を挿入します。
    // 画面のウィンドウをカーソルのある行から下にスクロールする
    // ウィンドウに表示されます。カーソルは新しい、空の行になります。
478 static void insert_line(int currcons) 479 {。
480         int oldtop,oldbottom;
481
    // 最初に画面のスクロール開始行の上端と最終行の下端の値を保存し、その後スクロールする
    // 画面の内容をカーソルのある行から1行下にする。最後に
482     // スクロールの開始行'top'と最終行'bottom'の元の値です。
483         oldtop=top;
484         oldbottom=bottom;
485         top=y;                          // set start and end lines of the scroll screen.
486         bottom = video_num_lines;
487         scrdown(currcons);              // the screen content scrolls down one line
488         top=oldtop;
489         bottom=oldbottom;
489 }
490
    //// 文字を削除します。
    // カーソル位置の1文字を削除し，カーソルの右にあるすべての文字をシフトさせる
    // を1スペースだけ左に移動させます。
491 static void delete_char(int currcons) 492 {。
493         int i;
494         unsigned short * p = (unsigned short *) pos;
495
    // カーソルの現在の列の位置xが、画面の右端の列を超えているかどうかを返す。
    // それ以外の場合は、カーソルの右文字から行末までのすべての文字が
    //を1スペース分左にシフトします。そして、最後の文字で消去文字を埋める。
496         if (x>=video_num_columns)
497                 return;
498         i = x;
499         while (++i < video_num_columns) {        // shifted to the left by one space.
500                 *p = *(p+1);
501                 p++;
502         }
503         *p = video_erase_char;                   // finally fill in the erase character.
504 }
505
```

//// カーソルがある位置の1行を削除します。

// カーソルがある位置の1行を削除し、画面を1行上にスクロールします。

```
506 static void delete_line(int currcons) 507 {。
508         int oldtop,oldbottom;
509
// まず、画面のスクロール開始行「上」と最終行「下」を保存してから、スクロールの
// カーソルのある行から1行上の内容を表示する。最後に、元の
510     // 画面スクロールの開始行'上'と最終行'下'の値。
511         oldtop=top;
512         oldbottom=bottom;
513         top=y;                          // set start and end lines of the scroll screen.
514         bottom = video_num_lines;
515         scrup(currcons);                // the screen content scrolls up one line.
516         top=oldtop;
517         bottom=oldbottom;
517 }
518
//// カーソル位置に nr 文字を挿入します。
// ANSIのエスケープ文字列を扱う。'ESC [ Pn @'. 1つ以上のスペース文字の挿入
// 現在のカーソルの位置に Pnは挿入する文字数で，デフォルトは1です。
// カーソルは最初に挿入されたスペース文字の位置のままです。カーソル位置の文字と
// 右のボーダーが右にずれて、右のボーダーを超えた文字が消えてしまいます。
// エスケープシーケンスでのパラメータnr = Pn。
519 static void csi_at(int currcons, unsigned int nr) 520 {。
// 挿入された文字数が1行分の文字数を超える場合は、切り捨てられる
// 挿入文字数nrが0の場合、1文字が
// を挿入します。その後、周期的にnr個のスペース文字を挿入します。
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
//// カーソル位置に nr 行を挿入します。
// ANSIのエスケープシーケンスを扱う：'ESC [ Pn L'. 制御シーケンスは、1つまたは複数の空白を挿入
します。
// カーソルの位置にラインが表示されます。操作が完了しても、カーソル位置は変わりません。
// 空白行を挿入すると、カーソル下のスクロールエリアの行が下に移動します。る。
// 表示ページをスクロールして出てきた行がなくなる。エスケープシーケンスのパラメータ nr = Pn。
529 static void csi_L(int currcons, unsigned int nr) 530 { 。
// 挿入された行数が画面の最大行数よりも多い場合、その行数は
行の//が画面の行に切り取られます。挿入された行数nrが0の場合、1行の
// が挿入されます。
その後、Nrの空白行を周期的に挿入します。
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
//// カーソル位置の nr 文字を削除します。
// ANSIのエスケープシーケンスを扱う。'ESC [Pn P'. この制御シーケンスは Pn 個の文字を削除する
// をカーソルから削除します。文字が削除されると、カーソルの右側にあるすべての文字が
// を左にシフトすると、右のボーダーにスペース文字ができます。そのプロパティ
// は最後の左シフト文字と同じになるはずですが、ここでは簡略化されています。
// キャラクターのデフォルトのプロパティ（黒背景白前景空間0x0720）を
// スペース文字を設定します。パラメータ nr = エスケープシーケンスの Pn。
539 static void csi_P(int currcons, unsigned int nr) 540 {。
// 削除された文字数が1行分の文字数を超える場合は、切り捨てられます
// 削除された文字数nrが0の場合、1文字が
// 削除されます。そして、カーソルで指定された数の文字 nr を繰り返し削除します。
541 if (nr > video_num_columns)
542 nr = video_num_columns;
543 else if (!nr)
544 nr = 1;
545 while (nr--)
546 delete_char(currcons);
547 }
548
//// カーソルの先頭にある nr 行を削除します。
// ANSIのエスケープシーケンスを処理します。'ESC [Pn M' です。制御シーケンスは、1つまたは複数の行を削除する
スクロールエリアでカーソルが置かれている行から // を削除します。行が削除されると
スクロールエリアの削除された行の下にある//行が上に移動し、1つの空白行が
// 一番下の行です。Pnが表示ページに残っている行数よりも大きい場合には
// このシーケンスは、これらの残りの行を削除するだけで、スクロールエリア外では機能しません。
// エスケープシーケンスでのパラメータnr = Pn。
549 static void csi_M(int currcons, unsigned int nr) 550 {。
// 削除した行数が画面の最大行数よりも多い場合。
// 表示されている行数が切り捨てられます。削除する行数nrが0の場合、1
//行が削除されます。その後、指定された nr 行を繰り返し削除します。
551 if (nr > video_num_lines)
552 nr = video_num_lines;
553 else if (!nr)
554 nr=1;
555 while (nr--)
556 delete_line(currcons);
557 }
558
//// 現在のカーソル位置を保存します。
559 static void save_cur(int currcons) 560 {。
561 saved_x=x;
562 saved_y=y;
563 }
564

//// 保存されたカーソル位置を復元します。

565 static void restore_cur(int currcons) 566 {。

567 gotoxy(currcons, saved_x, saved_y);

```
}

// 以下の列挙定義は、以下のcon_write()関数で
// エスケープシーケンスやコントロールシーケンスを解決します。ESnormalは初期のエントリー状態であり
// また、エスケープシーケンスやコントロールシーケンスが処理されるときの状態。
// ESnormal - 初期の正常な状態であることを示す。このとき、もしノーマル
//     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 - エスケープシーケンスの先頭文字であるESC（0x1b = 033 = 27）が入力されたことを示します。
//     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 - 制御シーケンスプリアンブル（'ESC ['）を受信したことを示し、次のことを示します。
//     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 Esfunckey 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 - この状態は、パラメータの値を受け取ることを示しています。
//     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 - 完全な制御シーケンスを受信したことを示します。この時点で我々は
//     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.
// ESfunckey - キーボードのファンクションキーからシーケンスを受け取ったことを示す。
//     that we don't need to display. So we return to the normal state ESnormal.
// ESsetterm - DCS(Device Control String Sequence)状態であることを示す。このとき
//     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 - 受信した文字セットのエスケープシーケンス「ESC ('または 'ESC )'」を表します。
//     are used to specify the character set used by G0 and G1, respectively. At this time, if
//     the character '0' 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.
enum { ESnormal, ESesc, ESsquare, ESgetpars, ESgotpars, ESfunckey,
         ESsetterm, ESsetgraph };

//// コンソールは機能を書き込みます。
```

// 端末のtty書き込みバッファキューから文字を取り出し、それぞれの
//の文字です。制御文字やエスケープシーケンス、制御シーケンスの場合は、制御処理
カーソルの位置合わせや文字の削除などの//が行われ、通常の文字
カーソルの位置に // が直接表示されます。
// パラメータ tty は、現在のコンソールで使用されている tty 構造体ポインタです。

```
574 void con_write(struct tty_struct * tty) 575 {.
576         int nr;
577         char c;
578         int currcons;
579
```

// この関数は，まず tty の位置に応じてコンソール番号 currcons を取得します
現在のコンソールで使用されているttyテーブルの中で、 // の数を計算（CHARS()）します。
// 現在のttyの書き込みキューに含まれる文字数nrを、各文字を1つずつループアウトします。
// を1つずつ処理していきます。ただし、pauseコマンドによって現在のコンソールが停止している場合は
// キーボードやプログラム（Ctrl-Sボタンなど）から発行された場合は、機能が停止します。
// 書き込みキューの文字を処理して、関数を終了します。さらに、もし
// 制御文字 CAN（キャンセル、ASCIIコード24、Ctrl-Xで生成）またはSUB（Substitute、26。
// Ctrl-Z）が取られた場合は、エスケープシーケンスまたはコントロールシーケンス中にその文字を受信した場合。
// その場合、シーケンスは実行されず、直ちに終了し、後続の
// 表示された文字。なお、con_write()関数は、表示された文字を処理するだけです。
// 現在、書き込みキューに入っている文字数をフェッチする際に使用します。それは
// シーケンスが書き込みキューに入っている間の文字数を読み取ることができます。
// なので、「状態」は、以下のようなときに、エスケープシーケンスやコントロールシーケンスを処理する別の状態になっている可能性があります。
// 関数は最後に終了します。

```
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--) {
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) {
```

// 抽出された文字が通常の表示文字の場合、対応する表示文字
// は、現在マッピングされている文字セットから直接取り出され、ディスプレイメモリに配置されます。
// 現在のカーソルがある位置、つまり文字が直接表示されている位置。
// その後、カーソル位置を1文字分だけ右に移動させます。具体的には、もし
// の文字が制御文字でも拡張文字でもない場合、つまり（31<c<127）の場合は
// 現在のカーソルが行末にある場合、または行末以外の位置にある場合、カーソルが

//は次の行の最初の列に移動します。そして、メモリポインタのposを対応する
//をカーソル位置に設定します。その後、その文字はディスプレイメモリposに書き込まれ

//カーソルが1列分右に移動し、それに応じてposも2バイト移動しています。

```
592
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++;
```

// 'c'がエスケープ文字ESCの場合、状態はESescに変換されます（637行目）。

// cがLF(10)、またはVT(11)、またはFF(12)の場合、カーソルは次の行に移動します。

// cがキャリッジリターンCR(13)の場合、カーソルを先頭列(0列目)に移動します。

// cがDEL(127)の場合、カーソルの左側の文字が消去され、カーソルが移動します。

// を消去した文字位置に戻す。

// 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                                     }
    // cが水平タブHT(9)の場合、カーソルは8の倍数の位置に移動し、カーソルの数が
    // の列数が画面上の最大列数を超える場合は、カーソルを次の行に移動させてください。
    // cがベルBEL(7)の場合、システムのビープ機能が呼び出され、スピーカーが鳴ります。
    // cが制御文字SO(14)またはSI(15)の場合は、文字セットG1またはG0を
    // ディスプレイの文字セットを変更します。
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)                 // SO - Shift out, use G1.
633                                     translate = GRAF_TRANS;
634                             else if (c == 15)                 // SI - Shift in, use G0.
635                                     translate = NORM_TRANS;
636                             break;
    // ESnormal状態でエスケープ文字ESC(0x1b=27)を受信すると、次のようになります。
    //状態の処理を行います。この状態では、cの制御文字やエスケープ文字を処理します。
```

```
// 処理後、デフォルトの状態はESnormalになります。
```

```
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;
```

// ESescの状態で文字'['を受信した場合、それがCSIコントロールであることを示す
// の配列なので、ESsequareという状態になって処理されます。まず、保存に使われた配列par[]は
// ESCシーケンスのパラメータがクリアされると、インデックス変数nparは、最初の
//の項目を設定し、パラメータESgetparsを開始するように状態を設定します。ただし、受け取ったキャラクターが
この時点での//が「[]」であれば、キーボードのファンクションキーで送られたシーケンスが
// が受信されたので、次の状態はESfunckeyに設定されます。受信した文字が「？」でなければ

```
// 処理する状態ESgetparsに直接行く。受信した文字が'?'の場合、それは
// 端末機のプライベートシーケンスであること、その後に機能文字が続くこと。
```

// そのため、次の文字に進み、コードを処理するために状態ESgetparsに行きます。

```
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;
```

// この状態は、制御シーケンスのパラメータ値を受信することを示しています。

// この時のことです。パラメータは10進数で表現されており，受け取った

// 数値文字を値に変換して、par[]配列に保存します。セミコロンを受け取った場合

// ';'の場合は、この状態のままで、受信したパラメータ値を次の項目である

// have been obtained, then move to the state ESgotpars to process.

```
                        case ESgetpars:
```

```
693
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
    // で受信した終端文字に基づいて、制御シーケンスを処理することができます。
    //の状態になります。しかし、処理の前に、ESsquare状態で「？」を受信した場合、このシーケンスは
    //は、端末機器のプライベートシーケンスです。の処理をサポートしていません。
    // このシーケンスでは、直接ESnormal状態に復元します。それ以外の場合は、対応する
    // 制御シーケンスを処理します。シーケンスが処理された後、状態はESnormalに戻ります。
702                             case ESgotpars:
703                                     state = ESnormal;
704                                     if (ques)               // received '?'
705                                     { ques =0;
706                                       break;
706                                     }
707                                     switch(c) {
    // cが'G'または'`'の場合、par[]の最初の値は列番号を表し、それ以外の場合は
    // ゼロの場合、カーソルは1列分左に移動します。
    // cが'A'の場合、最初の値は、カーソルが上に移動した行数を表します。もし
    // パラメータが0の場合、1行上に移動します。
    // cが'B'または'e'の場合、最初のパラメータは、次のようにして下に移動した行数を表します。
    // カーソルを表示します。パラメータが 0 の場合は、1 行下に移動します。
    // cが'C'または'a'の場合、最初のパラメータは、その右にある列数を表します。
    // カーソルを表示します。パラメータが0の場合，1列分だけ右にシフトします.
    // cが'D'の場合、最初のパラメータは、カーソルの左にある列数を表します。
    // パラメータが0の場合は、1列分左にシフトされます。
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;
    // cが'E'の場合、最初の値はカーソルが下に移動した行数を表し、次の値を返します。
    // を0列目に移動します。値が0の場合は、1行下に移動します。
    // cが'F'の場合、最初の値はカーソルが上に移動している行数を表し、次のように返します。
    //を0列目に移動します。パラメータが0の場合は、1行上に移動します。
    // cが'd'の場合、最初の値はカーソルのある行番号（0から数えて）を表します。
    //が必要となります。
    // cが'H'または'f'の場合、最初の値はカーソルの移動先の行番号を表します。
                                    case 'E':       // CSI Pn E - move down, col 0.
```

```
728
729                                             if (!par[0]) par[0]++;
730                                             gotoxy(currcons,0,y+par[0]);
731                                             break;
732                                     case 'F':       // CSI Pn F - move up, col 0.
733                                             if (!par[0]) par[0]++;
734                                             gotoxy(currcons,0,y-par[0]);
735                                             break;
736                                     case 'd':       // CSI Pn d -  set cursor line no.
737                                             if (par[0]) par[0]--;
738                                             gotoxy(currcons,x,par[0]);
739                                             break;
740                                     case 'H': case 'f':  // CSI Pn H -set cusor postion.
741                                             if (par[0]) par[0]--;
742                                             if (par[1]) par[1]--;
743                                             gotoxy(currcons,par[1],par[0]);
744                                             break;
    // If the character c is 'J' (sequence 'ESC [ Ps J'), the first parameter represents the way
    // カーソルの位置に関連して、画面がクリアされます。
    // cが'K' ('ESC [ Ps K') の場合，最初のパラメータは，文字がどのように
    行内の // は、カーソル位置に関連して削除されます。
    // cが'L' ('ESC [ Pn L') の場合は、カーソル位置にn行が挿入されることを意味する。
    // cが'M' ('ESC [ Pn M') の場合は、カーソル位置でn行が削除されることを意味する。
        // c が 'P' ('ESC [ Pn P')の場合、カーソル位置で n 文字が削除されることを意味します。
    // If c is '@' ('ESC [ Pn @'), it means that n chars are inserted at the cursor position.
745                                     case 'J':       // CSI Pn J - erase chars on screen.
746                                             csi_J(currcons,par[0]);
747                                             break;
748                                     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;
```

// cが'm'（'ESC [Pn m'）の場合は、以下の文字の表示属性を変更することを意味します。
// 太字にするなど、カーソルに合わせて
// cが'r'（'ESC [Pn;Pn r'）の場合は、開始ライン番号と終了ラインの
// スクロールの数は2つのパラメータで設定します。
// cが's'('ESC [Pn s')の場合は、現在のカーソル位置が保存されることを意味します。
// cが'u'（'ESC [Pn u'）の場合は、カーソルを保存した位置に戻すことを意味します。

```
763                                 case 'm':       // CSI Ps m - set char attributes.
764                                         csi_m(currcons);
765                                         break;
766                                 case 'r':       // CSI Pn;Pn r - set scroll range.
767                                         if (par[0]) par[0]--;
768                                         if (!par[1]) par[1] = video_num_lines;
769                                         if (par[0] < par[1] &&
770                                             par[1] <= video_num_lines) {
771                                                 top=par[0];
772                                                 bottom=par[1];
773                                         }
774                                         break;
775                                 case 's':       // CSI s - saved cursor postion.
776                                         save_cur(currcons);
777                                         break;
778                                 case 'u':       // CSI u - restore cursor postion.
779                                         restore_cur(currcons);
780                                         break;
```

// 文字cが'l'または'b'の場合、スクリーンのブラックスクリーンのインターバル時間を設定することを意味します。
//、太字表示の設定をそれぞれ行います。このとき、par[1]とpar[2]は
// それぞれ、par[1]=par[0]+13、par[2]=par[0]+17となります。この場合
// 条件は、cが'l'の場合、par[0]は黒のスタート時に遅らせる分数

//画面；cが'b'の場合は、par[0]の値に太字属性が設定されます。

```
                                 case 'l': /* blank interval */
781
782                              case 'b': /* bold attribute */
783                                         if (!((npar >= 2) &&
784                                         ((par[1]-13) == par[0]) &&
785                                         ((par[2]-17) == par[0])))
```

```
786                 break;
787             if ((c=='I')&&(par[0]>=0)&&(par[0]<=60))
788             {
789               blankinterval = HZ*60*par[0];
790               blankcount = blankinterval;
```

```
791                                             }
792                                             if (c=='b')
793                                               vc_cons[currcons].vc_bold_attr
794                                                 = par[0];
795                                     }
796                                     break;
```

// ステータスESfunckeyは、以下のファンクションキーからのシーケンスを受信したことを示します。

// キーボードを表示します。表示する必要がないので、通常の状態であるESnormalに戻ります。

```
                        case ESfunckey:             // keyboard function key.
797
798                                     state = ESnormal;
799                                     break;
```

// 状態ESsettermは、DCS（Device Control String）シーケンス状態であることを示しています。

// この時、文字'S'を受信した場合、初期表示文字属性は

// 復元します。文字が'L'または'l'の場合は、折り畳み表示モードのON/OFFが行われます。

```
800                     case ESsetterm: /* Setterm functions. */
801                                     state = ESnormal;
802                                     if (c == 'S') {
803                                             def_attr = attr;
804                                             video_erase_char = (video_erase_char&0x0ff) |
                                                                   (def_attr<<8);
805                                     } else if (c == 'L')
806                                             ; /*linewrapon*/
807                                     else if (c == 'l')
808                                             ; /*linewrapoff*/
809                                     break;
```

// 状態ESsetgraphは、設定された文字セットのエスケープシーケンス'ESC（'または'ESC)'が

//を受信しました。これらは、それぞれG0とG1が使用する文字セットを指定するために使用されます。

// このとき、文字'0'を受信すると、グラフィック・キャラクタ・セットを

// G0とG1で、受信した文字が'B'の場合、通常のASCII文字セットが選択される

G0とG1のキャラクターセットとしての//。

```
810                     case ESsetgraph:        // 'CSI (0' or 'CSI (B' - select char set.
811                                     state = ESnormal;
812                                     if (c == '0')
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
```

```
/*
 *  void con_init(void);
 *
 * This routine initalizes console interrupts, and does nothing
 * else. If you want the screen to clear, call tty_write with
 * the appropriate escape-sequece.
 *
 * 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
```

// まず、setup.sプログラムで取得したシステムのハードウェアパラメーターに応じて、いくつかの
// この関数に固有のスタティックなグローバル変数が初期化されます（60～68行目を参照）。

```
843         video_num_columns = ORIG_VIDEO_COLS;
844         video_size_row = video_num_columns * 2;   // bytes used by a display row.
845         video_num_lines = ORIG_VIDEO_LINES;       // lines of the display.
846         video_page = ORIG_VIDEO_PAGE;             // display page.
847         video_erase_char = 0x0720;                // char: 0x20, attr: 0x07.
848         blankcount = blankinterval;               // unit: ticks.
849
```

// そして、表示モードがモノクロかカラーかに応じて、表示メモリのスタート
// 位置と表示レジスタのインデックスポート番号と表示レジスタのデータポート番号
//がそれぞれ設定されます。取得したBIOS表示モードが7に等しい場合、それは
// モノクロのディスプレイカードです。

```
850         if (ORIG_VIDEO_MODE == 7)               /* Is this a monochrome display? */
851         {
852                 video_mem_base = 0xb0000;       // start addr of monochrome display.
853                 video_port_reg = 0x3b4;         // index register port.
854                 video_port_val = 0x3b5;         // data register port.
```

// 次に、ディスプレイカードがモノクロカードかカラーカードかを判断します。
// BIOS割り込みint 0x10関数0x12で取得した表示モード情報に if
// 割り込みで得られたBXレジスタの戻り値が0x10になっていないことを意味します。
// EGAカードなので、初期表示タイプはEGAモノクロです。がありますが
// EGAカードの表示メモリを増やしても、32KBまでしか使用できないので、その分を
// モノクロモードでは0xb0000--0xb8000のアドレス範囲になります。このコードでは、ディスプレイに
// 説明文字列を「EGAm」にすると、画面右上に表示されます。
// システムの初期化時に
// なお、BXレジスタは、カードの種類が変更される前や
// 割り込み後にint 0x10が呼び出されます。割り込みの後にBLの値が変更された場合
// が呼び出された場合、ディスプレイカードがAh=12hのファンクションコールをサポートしていること
とを意味し、これはタイプ
以降のVGAの//。割り込みコールの戻り値が変化していない場合、示す
// ディスプレイカードがこの機能をサポートしていないということは、一般的な
// モノクロのディスプレイカードです。

```
855                 if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
856                 {
857                         video_type = VIDEO_TYPE_EGAM;  // video type (EGA mono).
858                         video_mem_term = 0xb8000;      // video memory end address.
```

```
859                         display_desc = "EGAm";
860                 }
    // BXレジスタの値が0x10に等しい場合、モノクロディスプレイカードMDAを意味する
    // と、8KBのディスプレイメモリしかありません。
861                 else
```

```
862                 {
863                         video_type = VIDEO_TYPE_MDA;    // MDA mono.
864                         video_mem_term = 0xb2000;       // memory end address.
865                         display_desc = "*MDA";
866                 }
867         }
```

//表示モードが7でない場合は、カラーディスプレイカードであることを示しています。このとき、ディスプレイの
//テキストモードで使用するメモリの開始アドレスは0xb8000です。

```
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.
```

//続いて、ディスプレイカードのカテゴリーを判定します。もしBXが0x10に等しくない場合は、それが
// 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         }
```

// では、現在のディスプレイで開くことのできるバーチャルコンソールの数を計算してみましょう

//カードです。ハードウェアで許可されているバーチャルコンソールの数は、合計で
// メモリの video_memory を、各仮想コンソールが占有するバイト数で割ったもの。のは
// 各バーチャルコンソールが占有するディスプレイメモリの数は、ライン数と同じです。
// 文字の1行あたりのバイト数を掛けたもの： video_num_lines * video_size_row.
// 許可されたバーチャルコンソールの数が、システムで定義された最大数よりも多い場合は
// MAX_CONSOLESに、バーチャルコンソールの数を設定します。もし、仮想コンソールの数が
このようにして算出された//コンソールは、0であれば1に設定されます。最後に、ビデオメモリの合計数を分割した
// バーチャルコンソールの数によって、各バーチャルコンソールが占有するメモリのバイト数が決まります。
// コンソール（VC）です。

```
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
```

// そして、説明文字列を画面の右上に表示します。メソッドは
// 文字列をディスプレイメモリの対応する場所に直接書き込むことができます。

// まず、ディスプレイポインタ display_ptr が右端の最後の4文字を指します。
// の最初の行をコピーし（各文字は2バイト必要なので、マイナス8）、その後、循環的に
// 文字列の文字を、各文字ごとに1つの属性バイトを確保します。

```
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
```

// この時、現在の仮想コンソール番号 currcons は 0 に初期化されていることに注意してください。
// つまり、以下は実際に構造体vc_cons[0]のすべてのフィールドを初期化しています。
// 例えば、ここでのシンボル'origin'は、行の'vc_cons[0].vc_origin'として定義されています。
// 上記の115。以下ではまず、デフォルトのスクロール開始メモリ位置 video_mem_start を設定します。
// とデフォルトのスクロールする最終行のメモリー位置（実際には、これらはディスプレイの一部です
// 906 base = origin = video_mem_start = video_mem_base; // デフォルトのスクロール開始アドレス 907 term = video_mem_end = base + video_memory; // #0 vcのend mem。

```
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
```

// コンソール0の現在のカーソル位置とメモリ位置posを設定した後に
カーソルに対応する // 残りのバーチャルコンソールの構造をループします。
// それぞれが占有しているディスプレイメモリの開始位置と終了位置を除いて、残りの
// 初期値は基本的にコンソール0と同じです。

```
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         }
```

// 最後に、現在のフォアグラウンド・コンソールのスクリーン・オリジン（左上隅）の位置を設定する
// ディスプレイコントローラのカーソル位置に、キーボード割り込み0x21トラップを設定する。
// ゲートディスクリプターになります。その後、キーボード割り込みのマスキングが無効になり、IRQ1の

//応答すべきキーボードからの要求信号。最後にキーボードコントローラをリセットする
キーボードが正常に動作を開始できるようにするためには、//を使用します。

```
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);                         // enabe it again to reset keyboard.
934 }
935
```

// 現在のフォアグラウンドのコンソールを更新します。
// フォアグラウンドコンソールをfg_consoleで指定したバーチャルコンソールに切り替える fg_console
//はフォアグラウンドのバーチャルコンソールの番号セットです。

```
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
```

//// ビープ音を止める。
// 8255AのPBポートのビット1とビット0をリセットします。の後のタイマープログラミングの説明を参照してください。

```
void sysbeepstop(void)
944
945 {
946         /* disable counter 2 */
947         outb(inb_p(0x61)&0xFC, 0x61);
```

// ビープ音のカウント（ティック）。

// ビープ音を鳴らす。
// 8255AチップのPBポートのビット1がスピーカーのドアオープン信号として使用され、ビット0は
// 8253タイマー2のドア信号として使用し、タイマーの出力パルスを
// スピーカーが音を出すときの周波数として、スピーカーを したがって、スピーカーを

// ビープ音を鳴らすためには、2つのステップが必要です。まず、PBポート（0x61）のビット1とビット0をオン（セット）にし、次に
// タイマー2チャンネルに特定のタイミング周波数を送るように設定します。8259Aの割り込みコントローラを参照
boot/setup.sプログラムの後に、//チップのプログラミング方法を参照してください。
// kernel/sched.cプログラムの後にタイマーの命令を出す。

```
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
```

//// 画面をコピーする。
// 画面の内容を、パラメータで指定されたユーザーバッファargにコピーする。
// パラメータargには2つの目的があり、1つはコンソール番号を渡すこと、もう1つは
// ユーザーバッファのポインタとして機能します。

```
965 int do_screendump(int arg)
966 {
```

```
967         char *sptr, *buf = (char *)arg;
968         int currcons, l;
969
    // この関数は，まずユーザが提供したバッファ容量を確認し，それが十分でない場合は
    // が適切に展開されます。そして、その先頭からコンソール番号のカーコンを取ります。その後
    // コンソール番号が有効であると判断された場合、コンソール画面のすべてのメモリコンテンツが表示
    されます。
    // がユーザーバッファにコピーされます。
970         verify_area(buf,video_num_columns*video_num_lines);
971         currcons = get_fs_byte(buf);
972         if ((currcons<1) || (currcons>NR_CONSOLES))
973                 return -EIO;
974         currcons--;
975         sptr = (char *) origin;
976         for (l=video_num_lines*video_num_columns; l>0 ; l--)
977                 put_fs_byte(*sptr++,buf++);
978         return(0);
979 }
980
    // ブラックスクリーン処理。
    // blankIntervalの間にユーザーが何もボタンを押さなかった場合、画面は
    void blank_screen()
```

```
981
982 {
983         if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
984                 return;
985 /* blank here. I can't find out how to do it, though */
986 }
987
    // Restore the black screen.
    void unblank_screen()
```

```
988
989 {
990         if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
991                 return;
992 /* unblank here */
993 }
994
```

//// コンソールプリント機能です。

// この関数は、カーネルの表示関数であるprintk() (kernel/printk.c)によってのみ使用され、印刷を行います。

// 現在のフォアグラウンド・コンソールのカーネル情報です。処理方法は、ループアウトして

// バッファ内の文字をカーソルの動きに合わせて制御したり、画面に直接表示する

文字の特性に応じて、//。引数bは，ヌル終端の文字列

void console_print(const char * b)

```
995
996 {
```

```
997         int currcons = fg_console;
998         char c;
999
    // バッファ内の文字を循環させます。現在の文字が改行文字の場合。
    // カーソルでキャリッジリターン・ラインフィード操作が行われ、次に次の文字
        // が処理されます。それがキャリッジリターンであれば、直接キャリッジリターンを実行します。
    その後
    // go to the next character.
1000         while (c = *(b++)) {
```

```
                if (c == 10) {
                        cr(currcons);
                        lf(currcons);
                        continue;
                }
                if (c == 13) {
                        cr(currcons);
                        continue;
                }
    // キャリッジリターンや改行ではない文字を読み込んだ後に、もし
    // 現在のカーソル列の位置が画面の右端に達したので、カーソルをフォールドさせる
    // 次の行の先頭に戻ります。そして、その文字を
    // 画面に表示されている、カーソルの位置です。カーソルを右に1列移動
    // 次の文字の準備をします。
                if (x>=video_num_columns)
{                       x -= video_num_columns;
                        pos -= video_size_row;
                        lf(currcons);
                }
    // レジスタALには、表示する文字が入ります。ここではAHに属性バイトが入ります。
    //となり、AXの内容がカーソルメモリの位置posに格納される、つまり、文字
    カーソルの位置に // が表示されます。
                _asm_("movb %2,%%ah\n\t"        // attribute byte -> ah.
                        "movw %%ax,%1\n\t"      // put value of ax to pos.
                        ::"a"(c),
                        "m"(*(short *)pos),
                        "m"(attr)
                        :"ax");
                pos += 2;
                x++;
        }
    //  Finally, set the cursor position in the display controller with the value set above.
        set_cursor(currcons);
}
```

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.

ディスプレイ・カードのプログラミングの基本的な手順は、まず、ディスプレイ・カードのインデックス・レジスタ（ポート0x3d4）に0--17の値を書き込み、ディスプレイ・コントロールの内部レジスタ（r0--r17）の1つを選択し、データ・レジスタ・ポート（0x3d5）が内部レジスタに対応します。そして、データ・レジスタ・ポートにパラメータを書き込みます。つまり、ディスプレイ・カードのデータ・レジスタ・ポートは、一度に1つのディスプレイ・カードの内部レジスタに対してしか操作

できません。ディスプレイ・カードの主な内部レジスターを表10-6に示します。

表10-5 CGAポートレジスタ名と機能

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).

表10-6 MC6845内部データレジスタと初期値 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 r11	Cursor Start Cursor End	Scan line Scan line	W W	0x06 0x07	0x06 0x07	0x06 0x07
r12 r13	Start Mem Address (High) Start Mem Address (Low)		W W	0x00 0x00	0x00 0x00	0x00 0x00
r14 r15	Cursor position (High) Cursor position (Low)		R/W R/W	Vary		
r16 r17	Light Pen (High) Light Pen (Low)		R R	Vary		

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.

スクロールアップについては、画面の対応する表示メモリウィンドウが下方向に移動した後も表示メモリの範囲内にある場合、つまり現在の画面に対応するメモリブロックの位置が常にメモリの開始位置（video_mem_start）と終了位置（video_mem_end）の間にある場合には、ディスプレイコントローラの開始表示メモリ位置を調整するだけでよい。しかし、画面に対応するメモリブロックの位置が実際のメモリの終了位置(video_mem_end)を超えて移動した場合、現在の画面データがすべて表示メモリの範囲内に収まるように、対応するメモリのデータを移動させる必要がある。この2つ目のケースでは、プログラムは画面に対応するメモリデータを実際の表示メモリの先頭（video_mem_start）に移動させる。

プログラムでの実際の処理は、3つのステップで行われる。まず、画面の表示開始位置の原点を調整する。次に、対応する画面のメモリデータが表示メモリの下限（video_mem_end）を超えているかどうかを判断し、超えている場合は、画面に対応するメモリデータを実際の表示メモリの先頭（video_mem_start）に移動させる。画面に表示される新しい行には、スペース文字が入ることになる。その動作図を図10-8に示す。図(a)は最初の単純な場合、図(b)はメモリデータの移動が必要な場合に対応する。

図10-8 スクロールアップ操作の模式図

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

1. 端末機は通常、コンピュータの情報入力装置（キーボード）と出力装置（ディスプレイ）の2つの機能を持っている。端末には、単に画面に文字を表示するだけではなく、端末に特定の操作を行わせる制御コマンドをいくつか持たせることができる。このようにして、コンピュータは、カーソルの移動、表示モードの切り替え、呼び出し音などの操作を端末に命令することができる。端末の制御コマンドは、さらに「制御文字コマンド」と「ANSIエスケープ制御シーケンス」の2種類に分けられる。前述したように、Linuxカーネルに含まれるconsole.c（上記のkeyboard.sを含む）プログラムは、実際には端末エミュレータを模擬したものと見ることができる。そこで、console.cプログラムの処理を理解するために、端末機器のROM内のプログラムがホストから受け取ったコードデータをどのように処理するかを概説する。まず、ASCIIコード表の構造を簡単に説明し、次に端末機器が受信した制御文字や制御シーケンスの文字列コードをどのように扱うかを説明する。

2. Character encoding method

従来の文字端末では、ANSI（米国規格協会）やISO（国際標準化機構）規格の8ビット符号化方式と、7ビットの符号拡張技術を採用していた。ANSIとISOは、コンピュータや通信の分野における文字符号化規格を定めている。ANSI X3.4-1977とISO 646-1977では、American Standard Code for Information Interchange、つまりASCIIコードセットを定義している。ANSI X3.41-1974とISO 2022.2は、7ビットと8ビットのコードセットのコード拡張技術を規定しています。ANSI X3.32、ANSI X3.64-1979では、ASCIIコードのテキスト文字を使って端末制御文字を表現する方法が開発さ

れています。Linux 0.1xのカーネルでは、Digital Equipment Corporation DEC（現在はCompaqとHPに組み込まれている）のVT100とVT102のターミナルデバイスとの互換性しか実装されていませんが。

3. であり、これら2つのデファクトスタンダードな端末機器は7ビットの符号化方式しかサポートしていない。しかし、ここでは説明の完全性と利便性のために、8ビットの符号化方式についても紹介する。

4. ASCII code table

ASCIIコードには7ビットと8ビットのコード表現があります。7ビットコード表には、表10-7の左半分に示すように、合計128の文字コードがある。各行は7ビットのうち4ビット下位の値を表し、各列は3ビット上位の値を表しています。例えば、4列目1行目のコード「A」の2進法の値は0b0100, 0001（0x41）で、10進法の値は65である。

表の中の文字は2種類に分けられます。一つは1列目と2列目で構成される制御文字で、残りは図形文字や表示文字、テキスト文字です。端末はこの2種類の文字を別々に処理します。グラフィック文字は画面に表示できる文字であり、制御文字は通常画面に表示されない文字である。制御文字は、データ通信やテキスト処理時の特別な制御に使用されます。なお、DEL文字（0x7F）は制御文字でもあり、スペース文字（0x20）は通常のテキスト文字でも制御文字でもある。制御文字とその機能は、ANSIによって標準化されており、その名称はANSI標準のニーモニックである。例えば、以下のようになります。CR(キャリッジリターン)、FF(フォームフィード)、CAN(キャンセル)などです。通常、7ビットの符号化方式は、8ビットの符号化にも適用できる。表10-7は、8ビットコード表（左半分は7ビットコード表と同じ）で、右半分の拡張コードは記載されていない。

表10-7 8ビットASCIIコード表	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	VTS							
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

テーブルには2つのコントロールコードセットがあります。C0とC1です。また、グラフィックキャラクタセットは、左グラフィックキャラクタセットGL（Graphic Left）と右グラフィックキャラクタセットGR（Graphic Right）の2種類があります。

5. C0とC1の制御文字の機能は変更できないが、GLやGRの領域に異なる表示文字をマッピングすることができる。使用できる（マッピングされた）様々なテキスト文字セットは、通常、端末機器に格納されている。これらを使用するためには、まずマッピングを行う必要がある。デファクトスタンダードとなっているDECの端末機器では、通常、DEC多国語文字セット（ASCII文字セットとDEC補助文字セット）、DEC特殊文字セット、NCR（National Replacement Character Set）が格納されている。端末装置の電源を入れると、デフォルトでDEC多国語文字セットが使用される。

6. Control functions

受信したデータをどのように処理するかを端末装置に指示するためには、端末装置の制御機能を利用する必要がある。ホストは、制御コードまたは制御コード列を送信することにより、端末装置による文字の表示処理を制御することができるが、これはあくまでもテキスト文字の表示、処理、送信を制御するためのものであり、それ自体が画面に表示されるわけではない。制御機能は、ディスプレイ上のカーソル位置の移動、テキストの一行削除、文字の変更、文字セットの変更、端末の動作モードの設定など、多くの用途がある。テキストモードでは、すべての制御機能を使用することができ、制御機能を表すために1バイトまたは複数バイトを使用することができる。

画面への表示に使用されない制御文字や制御文字列はすべて制御機能であると考えることができる。ANSIに準拠した各端末機器において、すべての制御機能がその制御動作を行うわけではないが、端末機器はすべての制御機能を認識し、動作しない制御機能を無視することができるはずである。そのため、通常、端末機器はANSI制御関数のサブセットしか実装していない。機器によって使用する制御機能のサブセットが異なるため、ANSI規格との互換性は、これらの機器が相互に互換性を持つことを意味しない。互換性は、様々なデバイスが同じ制御関数を使用しているという事実にのみ反映される。

7. シングルバイトの制御機能は，表10-7に示すC0およびC1の制御文字である。C0の制御文字では、限られた制御機能しか使用できません。C1 の制御文字は、いくつかの追加制御機能を提供できるが、8 ビット環境でしか使用できない。したがって、ここでLinuxカーネルにエミュレートされているVT100タイプの端末では、C0の制御文字しか使用できない。マルチバイト制御コードは、多くの制御機能を提供することができます。これらのマルチバイト制御コードは、一般にエスケープシーケンス、制御シーケンス、デバイス制御文字列などと呼ばれています。これらの制御シーケンスには、業界のANSI規格の共通シーケンスもあれば、メーカーが自社製品のために設計した独自の制御シーケンスもあります。ANSI規格のシーケンスと同様に、独自の制御シーケンス文字もANSI文字コードの複合規格に準拠しています。

8. Escape Sequences

ホストがエスケープシーケンスを送信することで、端末画面上のテキスト文字の表示位置や属性を制御することができます。エスケープ・シーケンスは、C0の制御文字ESC（0x1b）で始まり、その後

に1つ以上のASCII表示文字が続く。エスケープシーケンスのANSI標準フォーマットは以下の通りです。

ESC	I.....I	F
0x1b	0x20--0x2f	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.

中間文字とは、ESCの後に受信する0x20～0x2f（上記ASCII表の2列目）の文字のことです。端末は制御機能の一部としてこれらを保存する必要がある。

最終文字は、0x30 -- 0x7e（ASCII表の3列目 -- 7列目）の範囲でESCの後に受信する文字で、最終文字はエスケープシーケンスの終わりを示します。中間文字と終了文字を合わせて、シーケンスの機能を定義します。この時点で、端末はエスケープシーケンスで指定された機能を実行し、その後に受信した文字を表示し続けることができます。ANSI標準のエスケープシーケンスの最終文字は、0x40から0x7e（ASCII表の4列目から7列目）の範囲である。また、各端末機器メーカーが独自に定義したエスケープシーケンスの最終文字は、0x30～0x3f（ASCII表の3列目）となっています。例えば、ASCII文字セットとしてG0を指定するエスケープシーケンスを以下に示します。

```
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.

9. Control sequences

コントロール・シーケンスは、コントロール文字CSI（0x9b）の後に、1つまたは複数のASCIIグラフィック文字が続きます。コントロール・シーケンスのANSI標準フォーマットは以下の通りです。

```
CSI          P.....P            I.....I            F
0x9b         0x30--0x3f         0x20--0x2f         0x40--0x7e

Introducer   Parameter chars    Intermediate chars Final char
             (0 or more chars)  (0 or multi chars)  (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

2 番目の文字が左括弧「[]」であるエスケープ・シーケンス。イントロデューサーCSIを受信した後、端末はそれに続くすべての制御文字を一定の順序で保存（表示ではなく）する必要があります。

パラメータ文字とは、CSIの後に0x30--0x3f（ASCII表の3列目）の範囲で受信する文字で、制御シーケンスの役割や意味を変更するために使用されます。パラメーター・キャラクターが任意の「<=>?」で始まる場合（0x3c～0x3f）で始まる場合、端末はこの制御シーケンスを専有（プライベート）制御シーケンスとして使用する。端末で使用できるパラメータ文字は、数字文字とセレクト文字の2種類です。数値文字のパラメータは10進数を表し、Pnで表される。セレクト文字のパラメータは、Psで示される指定されたパラメータリストから得られます。 制御シーケンスに複数のパラメータが含まれる場合は、セミコロン「;」（0x3b）で区切られます。

中間文字とは、CSI の後に受信する文字で、0x20 -- 0x2f（ASCII 表の 2 列目）の範囲のものをいう。端末は、制御機能の一部としてこれらを保存する必要がある。なお、端末機では中間文字を使用しない。

最終文字は、CSIの後に受信した0x40 -- 0x7e（ASCII表の4列目 -- 7列目）の範囲の文字です。最終文字は、制御シーケンスの終了を示す。中間文字と最終文字の組み合わせにより、シーケンスの機能が定義される。この時点で、端末は指定された機能を実行し、その後に受信した文字を表示し続けることができる。ANSI標準エスケープシーケンスの最終文字は、0x40～0x6f（ASCII表の4～6列目）です。各端末機器メーカーが独自に定義したエスケープシーケンスの最終文字は、0x70～0x7e（ASCII表の7列目）となります。例えば、以下のシーケンスは、画面のカーソルを指定された位置に移動させる制御シーケンスを定義しています（5行目、9列目）。

```
    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

図10-9 制御シーケンスの例

10. Terminal handling of received characters

ここでは、端末が受信した文字をどのように処理するか、つまり、レスポンスの

アプリケーションやホストシステムから送られてきたコードを端末に入力します。端末が受信する文字は、グラフィック（表示またはテキスト）文字と制御文字の2つに分けられる。グラフィック文字とは、受信して画面に表示される文字のことです。実際に画面に表示される文字は、選択された文字セットによって異なります。文字セットは制御機能で選択できる。

端末が受信するすべてのデータは、1つまたは複数の文字コードで構成される。これらのデータには、グラフィック文字、制御文字、エスケープシーケンス、制御シーケンス、デバイス制御文字列が含まれる。データのほとんどは、画面に表示されるだけで他の効果を持たないグラフィック文字で構成されている。制御文字、エスケープシーケンス、制御シーケンス、デバイス制御文字列はすべて「制御関数」であり、私たちが独自のプログラムやオペレーティングシステムで、端末がどのように文字を処理し、送信し、表示するかを示すために使用することができる。各制御関数には固有の名前と略記法があります。これらの名前とニーモニックは標準的なものです。デフォルトでは、制御文字や表示文字に対するターミナルの解釈は、使用されているASCII文字セットに依存する。サポートされていない制御コードの場合、ターミナルが取る通常のアクションはそのコードを無視することであることに注意してください。ここに記載されていない文字がターミナルに送信されると、予想外の結果になる可能性があります。

本書の付録では、C0表とC1表の中でよく使われる制御文字の説明と、端末がそれを受け取ったときの動作の概要を示しています。特定の端末では、通常、C0およびC1のすべての制御文字を認識するわけではありません。また、付録には、Linux 0.1xカーネルのconsole.cプログラムで使用されるエスケープシーケンスと制御シーケンスも記載されています。すべてのシーケンスは、特に断らない限り、ホストから送られる制御関数のシーケンスを表しています。

10.4 serial.c

10.4.1 Function

プログラムserial.cは、システムのシリアルポートの初期化を実装し、シリアルターミナルデバイスを使用できる状態にします。rs_init()初期化関数では、デフォルトのシリアル通信パラメータが設定され、シリアルポートの割り込みトラップゲート(割り込みベクトル)も設定されます。rs_write()関数は、シリアル・ターミナル・デバイスのライト・バッファ・キューにある文字を、シリアル・ラインを通してリモート・ターミナル・デバイスに送信するために使用されます。

関数 rs_write()は、ファイルシステムでキャラクタデバイスファイルを使用する際に呼び出される。プログラムがシリアルデバイスの /dev/tty64 ファイルに書き込む際には、システムコールの sys_write() が実行される（fs/read_write.c 内)。システムコールで、読み込まれているファイルがキャラクタデバイスファイルであると判断されると、 rw_char() 関数（fs/char_dev.c）が呼び出される。

この関数は、読み込んでいるデバイスのサブデバイス番号などの情報に応じて、キャラクタデバイスの読み書き機能テーブル(デバイススイッチテーブル)を用いて、rw_tty()を呼び出す。これは最終的に、ここでシリアル端末の書き込み関数rs_write()を呼び出すことになります。

rs_write()関数は、実際にシリアル送信ホールドレジスタエンプティ割り込みフラグをオンにし、UARTがデータを送出した後に割り込み信号を送出できるようにします。具体的な送信操作はrs_io.sプログラムで行います。

10.4.2 Code annotation

プログラム 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
```

// <linux/tty.h> ttyヘッダーファイルは、tty_io、シリアルのパラメータと定数を定義しています。
// communication.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_structや
// data of the initial task 0, and some embedded assembly function macro statements
// about the descriptor parameter settings and acquisition.
// <asm/system.h> システムのヘッダーファイルです。を定義する埋め込みアセンブリマクロです。
// modifies descriptors/interrupt gates, etc. is defined.
// <asm/io.h> Io のヘッダーファイルです。の io ポートを操作する関数を定義します。
// form of a macro's embedded assembler.

```
16 #include <linux/tty.h>
17 #include <linux/sched.h>
18 #include <asm/system.h>
19 #include <asm/io.h>
20
```

// 書き込みキューにWAKEUP_CHARS文字が含まれているときに送信を開始する。

```
21 #define WAKEUP_CHARS (TTY_BUF_SIZE/4)
22
23 extern void rs1_interrupt(void); // シリアル1割り込みハンドラ(rs_io.s, 34) 24 extern void
rs2_interrupt(void); // シリアル2割り込みハンドラ(rs_io.s, 38) 25
```

//// シリアルポートの初期化
// 指定したシリアルポートの送信ボーレート（2400bps）を設定し、すべての割り込みを許可する
// のソースは、ライトホールドレジスタ以外は空です。また、2バイトボーの出力時には
// また、ラインコントロールレジスタのDLABビット（ビット7）を設定する必要があります。
// パラメータ： port はシリアルポートのベースアドレスで、port 1 - 0x3F8; port 2 - 0x2F8 です。

```
26 static void init(int port) 27 {。
28          outb_p(0x80,port+3);    /* set DLAB of line control reg */
29          outb_p(0x30,port);      /* LS of divisor (48 -> 2400 bps */
30          outb_p(0x00,port+1);    /* MS of divisor */
31          outb_p(0x03,port+3);    /* reset DLAB */
32          outb_p(0x0b,port+4);    /* set DTR,RTS, OUT_2 */
33          outb_p(0x0d,port+1);    /* enable all intrs but writes */
34          (void)inb(port);        /* read data port to reset things (?) */
35 }
36
```

//// シリアルインタラプトルーチンとシリアルインターフェースを初期化します。
// 割り込みディスクリプターテーブルIDTのゲートディスクリプター設定マクロset_intr_gate()は
// include/asm/system.hで実装されています。

```
37 void rs_init(void)
38 {
```

// 次の2つの文は、2つのシリアルの割込みゲート記述子を設定するために使用されます。
// rs1_interruptは、シリアルポート1の割込みハンドラポインタです。使用される割込みは
39 シリアルポート1による//はint 0x24、ポート2はint 0x23です。表5-2およびsystem.hファイルを参照してください。

```
40          set_intr_gate(0x24,rs1_interrupt);   // set int gate descriptor of port 1 (IRQ4).
41          set_intr_gate(0x23,rs2_interrupt);   // set int gate descriptor of port 2 (IRQ3).
42          init(tty_table[64].read_q->data);    // Initialize port 1 (.data is base address).
43          init(tty_table[65].read_q->data);    // Initialize port 2
44          outb(inb_p(0x21)&0xE7,0x21);         // enable 8259A to respond to IRQ3, IRQ4.
44 }
45
```

47 46 /*

```
48  * This routine gets called when tty_write has put something into
49  * the write_queue. It must check wheter the queue is empty, and
50  * set the interrupt register accordingly
50  *
51  *      void _rs_write(struct tty_struct * tty);
52 */
```

//// シリアルデータ送信書き込み機能。
// この関数は、実際には送信ホールドレジスタエンプティ割り込みフラグをオンにするだけです。
// その後、送信保持レジスタが空になると、UARTは割り込みを発生させます。
//のリクエストを行います。シリアルインタラプトハンドラでは、プログラムは最後に文字をフェッチして
// を書き込みキューに入れ、送信ホールディングレジスタに出力します。一旦、UARTが
// キャラクタを送信すると、送信保持レジスタが空になり、割り込み要求が発生します。
//を再び行う。つまり、書き込みキューに文字がある限り、システムはこのプロセスを繰り返します。
//と文字を1つずつ送信していきます。書き込みキューの中のすべての文字が送信されると
// アウト、書き込みキューが空になり、割り込みハンドラが送信ホールドレジスタをリセット
割り込みイネーブルレジスタの // 割り込みイネーブルフラグで、再び送信
// ホールドレジスタを空にすると、割り込み要求が発生します。この時点で、"ループ "送信動作の
//終了します。

53 void rs_write(struct tty_struct * tty) 54 {。

// 書き込みキューが空でない場合、まず0x3f9から割り込みイネーブルレジスタの内容を読み出す
//（または0x2f9）、送信ホールドレジスタの割り込み許可フラグ（ビット1）を追加してから、次のように書き込みます。
//をレジスターに戻すことができます。このようにして、UARTは以下を取得したいときに割り込みを開始することができます。
// 送信保持レジスタが空のときに送信する文字。ただ、次のことに注意してください。
55 // ここでのwrite_q.dataは、シリアルポートのベースアドレスです。

```
56          cli();
57          if (!EMPTY(tty->write_q))
58                  outb(inb_p(tty->write_q->data+1)|0x02,tty->write_q->data+1);
59          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

をゼロにしています。データビットは、実際に送信されるデータ、つまり1文字分のコードである。その長さは5～8ビットになります。パリティビットはオプションで、プログラムで設定することができます。ストップビットは常に1で、プログラムによって1、1.5、2ビットに設定することができます。通信で情報を送り始める前に、双方が同じフォーマットに設定し、同じ伝送速度を使用する必要があります。例えば、データビットとストップビットの数を同じにする必要があります。非同期通信の仕様では、送信1をMARK、送信0をSPACEと呼んでいます。そのため、以下の説明でもこの2つの用語を使用します。

図10-10 非同期シリアル伝送のフレームフォーマット

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.

1. キャラクタフレームを受信する際、受信機は次の3種類のエラーのいずれかを検出することがあります。(1)パリティエラー。(1)パリティエラー　この時点で、プログラムは相手に文字の再送を依頼しなければなりません。このエラーは、プログラムが受信速度よりも遅い速度で文字を取り込んだために発生します。ここでは、プログラムを修正して、文字の周波数を速くする必要があります。このエラーは、受信要求されたフォーマット情報が正しくない場合に発生します。例えば、ストップビットを受信すべきところを、SPACEビットを受信してしまった場合などです。一般的には、回線干渉の他に、通信相手のフレームフォーマットの設定が異なることが考えられます。

2. Serial communication interface and UART structure

シリアル通信を実現するために、PCは通常、RS-232C規格に準拠した2つのシリアルインターフェースを備えており、シリアルデータの送受信を処理するためにUART（Universal Asynchronous Receiver/Transmitter）制御チップを使用しています。PCのシリアルインターフェースには、通常、25ピンのDB-25または9ピンのDB-9コネクタが使用されており、主にMODEM機器を接続して動作させるために使用されます。そのため、RS-232C規格では、多くのMODEM専用インターフェースピンが規定されています。RS-232C規格の詳細やMODEMデバイスの動作原理については、他の資料をご参照ください。ここでは主に、UART制御チップの構造を説明し、プログラミングの準備をします。

以前のPCでは、ナショナル・セミコンダクター社のUARTチップ「NS8250」または「NS16450」を使用していました。現在のPCでは、16650Aまたはその互換チップを使用していますが、いずれもNS8250/16450チップと互換性があります。これらのチップの主な違いは、16650Aチップがさらに FIFO送信をサポートしていることです。これにより、UARTは最大16文字までの送受信後に割り込みを発生させることができ、システムやCPUの負担を軽減することができます。ただし、今回取り上げたLinux 0.12ではNS8250/16450の属性のみを使用しているため、FIFOモードについてはここでは詳しく説明しません。

PCで使用されているUART非同期シリアルポートのハードウェアロジックを図10-11に示します。これは3つの部分に分けられます。第1部には主に、データバスバッファD7～D0、内部レジスタセレクトピンA0～A2、CPUリード／ライトデータセレクトピンDISTRおよびDOSTR、チップリセットピンMR、割り込み要求出力ピンINTRPT、および割り込みの禁止／許可を定義したユーザピンOUT2が含まれます。OUT2が1のとき、UARTは割り込み要求信号の発行を禁止することができます。

図10-11 NS8250/16450の基本的なハードウェア構成図

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.

第3部は、UARTチップクロック入力回路部です。UARTの動作クロックは、XTAL1端子とXTAL2端子の間に水晶振動子を接続して生成する方法と、XTAL1端子を介して外部から直接入力する方法があります。PCでは後者の方法で、XTAL1端子に1.8432MHzのクロック信号を直接入力しています。端子BAUDOUTからはUARTの送信ボーレートの16倍信号が出力され、端子RCLKからは受信データのボーレートが出力されます。この2つが接続されているので、PCでデータを送信するときと受信するときのボーレートは同じになります。

UARTは、割り込みコントローラチップ8259Aと同様に、プログラマブルコントロールチップでもあります。その内部レジスタを設定することで、シリアル通信の動作パラメータや、UARTの動作方法を設定することができます。UARTの内部ブロック図を図10-12に示します。

図 10-12 NS8250 UART 内部コンポーネントの基本ブロック図

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 inTable 10-8.

表10-8 UART内部レジスタのポートと目的 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		Write the line control register LCR. Bit 7 = 1 Divisor Latch Access Bit (DLAB). = 0 Receiver, transmit hold or interrupt enable register access; Bit 6=1 allows for a break; Bit 5=1 holds the parity bit; Bit 4=1 even parity; =0 odd parity; Bit 3=1 allows parity; =0 no parity; 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; Bit 1-0 Data Bit Length: = 00 5 data bits; = 01 6-bit data bits; = 10 7-bit data bits; = 11 8-bit data bits.
0x3fc (0x2fc)	W		Write MODEM control register MCR. Bits 7-5 all 0, reserved; Bit 4=1 chip is in cyclic feedback diagnostic mode of operation; Bit 3=1, auxiliary user specifies output 2, enable INTRIPT to system; Bit 2=1, auxiliary user specifies output 1, and the PC is not used; Bit 1=1 makes the request to send RTS valid; Bit 0 = 1 makes the data terminal ready DTR active.
0x3fd (0x2fd)	R		Read the line status register LSR. Bit 7=0 reserved; Bit 6=1 The transmit shift register is empty; Bit 5=1 Transmit holding register is empty and can get character to send; Bit 4=1 receives a sequence of bits that satisfy the break condition; Bit 3=1 frame format error; Bit 2=1 parity error; Bit 1=1 exceeds the overlay error; Bit 0=1 Receiver data is ready and the system is readable.
0x3fe (0x2fe)	R		Read the MODEM status register MSR. δ indicates a change in the signal or condition. Bit 7=1 carrier detect (CD) is valid; Bit 6=1 ring indication (RI) is valid; Bit 5=1 Data Device Ready (DSR) is valid; Bit 4=1 Clear Transmit (CTS) is valid; Bit 3=1 detects the δ carrier; Bit 2=1 detects the edge of the ring signal; Bit 1 = 1 δ Data Device Ready (DSR); Bit 0 = 1 δ Clear Transmit (CTS).

3. UART initialization programming method

a) PCの電源を入れると、システムRESET信号がNS8250チップのMRピンを介して、UARTの内部レジスタと制御ロジックをリセットします。その後、UARTを使用する場合は、UARTの動作ボーレート、データビット数、動作モードを設定するために初期化する必要があります。以下、PCのシリアルポート1を例にとり、初期化の手順を説明します。シリアルポートのポートベースアドレスはport ＝ 0x3f8で、UARTチップの割り込みピンINTRPTは割り込み制御チップのピンIRQ4に接続されています。もちろん、初期化の前にIDTテーブルでシリアル割り込みハンドラの割り込みディスクリプタエントリを最初に設定する必要があります。

b) Set the transmission baud rate.

通信伝送ボーレートの設定は、2つの除算器ラッチレジスタのLSBとMSBの値、つまり16ビットボーレート係数を設定することです。表10-8からわかるように、2つの除算器ラッチレジスタにアクセスするには、まずラインコントロールレジスタLCRのビット7 DLAB=1を設定する必要があります。つまり、ポート+3（0x3fb）に0x80を書き込みます。その後、ポート(0x3f8)とポート+1(0x3f9)の出力操作を行い、ボーレートファクターをLSBとMSBにそれぞれ書き込むことができます。指定されたボーレート（例：2400bps）の場合、ボーレートファクターは次のように計算されます。

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

c) 従って、ボーレートを2400pbsに設定するには、LSBに48（0x30）、MSBに0を書き込む必要があります。ボーレートを設定した後、ラインコントロールレジスタのDLABビットをリセットする必要があります。

d) Set the transfer format.

e) シリアル通信の伝送フォーマットは、ラインコントロールレジスタLCRのビットによって定義されます。各ビットの意味を表10-8に示します。伝送フォーマットをパリティなし、8データビット、1ストップビットに設定する必要がある場合は、LCRに値0x03を出力する必要があります。LCRの下位2ビットはデータビット長を示しており、0b11の場合はデータ長が8ビットであることを示しています。

f) Set the MODEM control register.

このレジスタに書き込むことで、UARTの動作モードを設定し、MODEMを制御します。UARTの動作モードには、割り込みモードと問い合わせモードがあります。ループフィードバック方式もありますが、この方式はあくまでもUARTチップの品質を診断・テストするためのものであり、実際の通信方式としては使用できません。また、PC ROMのBIOSではクエリー方式が採用されていますが、本書で取り上げるLinuxシステムでは、効率の良い割り込み方式が採用されています。そのため、ここでは割込みモードでのUARTの動作プログラミング方法のみを紹介します。

MCR のビット 4 を設定すると、UART をループフィードバック診断モードで動作させることができます。 このモードでは、UARTチップは入力（SIN）および出力（SOUT）ピンを自動的に「スナップ」します。したがって、この時点で送信されたデータシーケンスが受信したシー

ケンスと同じであれば、UART チップは正常に動作していることになります。

割り込みモードとは、MODEMの状態が変化したとき、エラーが発生したとき、送信保持レジスタが空になったとき、キャラクターを受信したときに、UARTがINTRPT端子を通じてCPUに割り込み要求信号を送ることを許可することです。これらの条件で割り込み要求を出すことを許可するかどうかについては、割り込みイネーブルレジスタIERによって決定されます。ただし、UARTの割り込み要求信号を8259Aの割り込みコントローラに送れるようにするには、MODEMコントロールレジスタMCRのビット3（OUT2）を設定する必要があります。なぜなら、PCではこのビットが8259A回路へのINTRPTピンを制御するからです（図10-11参照）。

問い合わせモードとは、MODEM制御レジスタMCRのビット3（OUT2）が「0」の状態で、UARTレジスタの内容をサイクリックにポーリングしてシリアルデータを送受信するモードです。

をリセットすることができます。MCRビット3＝0の場合でも、MODEMの状態が変化した条件下でUARTはINTRPT端子に割り込み要求信号を発生させることができ、割り込みフラグレジスタIIRは割り込みが発生した条件に応じて設定することができますが、割り込み要求信号を8259Aに送信することはできません。そのため、プログラムは、ラインステータスレジスタLSRと割り込み識別レジスタIIRの内容を問い合わせることで、UARTの現在の動作状態を判断し、データの送受信動作を行うことしかできません。

g) MCRのビット1およびビット0は、MODEMの制御に使用されます。この2ビットがセットされていると、UARTのデータ端子レディーDTRピンと送信要求RTSピンの出力が有効になります。UARTを割込みモードに設定し、DTRとRTSを有効にするには、MODEMコントロールレジスタに0x0b（2進数の01011）を書き込む必要があります。

h) Initialize the interrupt enable register IER

割り込みイネーブルレジスタIERは、割り込みを発生させるための条件、つまり割り込みソースの種類を設定するために使用します。割り込みソースは、表10-8に示すように4種類から選択できます。対応するビットが1であれば、その条件で割り込みを発生させることができることを意味し、そうでなければ無効となります。割り込みソースタイプが割り込みを発生させた場合、どの割り込みソースで発生した割り込みかは、割り込みフラグレジスタIIRのビット2～ビット1で指定され、特定のレジスタの内容を読み書きすることで、UART割り込みをリセットすることができます。IERのビット0は現在割り込みがあるかどうかを判断するために使用され、ビット0＝0は処理すべき割り込みがあることを示します。

4. Linux 0.12のシリアルポート初期化機能では、3種類の割り込みソースによる割り込み発生を許可する設定(write 0x0d)、つまり、MODEMの状態が変化したとき、受信エラーが発生したとき、受信側が文字を受信したときに割り込みを発生させることを許可しています。しかし、送信ホールドレジスタが空になっても、この時点では送信するデータがないので、割り込みを発生させることはできません。対応するシリアル端末の書き込みキューに送出すべきデータがある場合、tty_write()関数はrs_write()関数を呼び出して送信ホールドレジスタエンプティのイネーブル割り込みフラグを設定することで、割り込みソースが開始するシリアル割り込み処理中に、カーネルプログラムは書き込みキュー内のキャラクタのフェッチを開始し、その出力を送信ホールドレジスタに送ってUARTが送出することができます。UARTが文字を送出すると、送信保持レジスタは空になり、再び割り込み要求が発生します。このように、書き込みキューに文字がある限り、システムはこのプロセスを繰り返し、文字を1つずつ送信していきます。書き込みキューの文字がすべて送出されると、書き込みキューが空になり、割り込みハンドラは割り込みイネーブルレジスタの送信ホールドレジスタ割り込みイネーブルフラグをリセットすることで、再び送信ホールドレジスタが空になって割り込み要求が発生しないようにします。この「ループ」送信動作も終了します。

5. UART interrupt handler programming method

Linuxカーネルでは、シリアルターミナルは、リード/ライトキューを使用してターミナルデータ

を送受信します。シリアルポートから受信したデータは、tty_io.cプログラムが読み取るためのリードキューヘッダに置かれ、シリアルターミナルに送信する必要のあるデータはライトキューポインタに置かれます。したがって、シリアル割り込みハンドラの主なタスクは、UARTが受信した受信バッファレジスタRBR内の文字を読み取りキューの最後のポインタに入れ、書き込みキューの最後のポインタから取り出した文字をUARTの送信ホールドレジスタTHRに入れて送出することです。同時に、シリアル割り込みハンドラは、いくつかの他のエラー状態を処理する必要があります。

上記の説明からわかるように、UARTは4種類の異なる割り込みソースタイプで割り込みを発生させることができます。そのため、シリアル割り込みハンドラが最初に実行を開始したとき、割り込みが発生したことだけはわかりますが、どのケースで割り込みが発生したかはわかりません。そのため、シリアル割り込みハンドラの最初のタスクは、割り込みが発生する特定の条件を決定することです。これには

割り込みフラグレジスタIIRは、現在の割り込みがどのソースタイプから発生したものかを判断します。したがって、シリアル割り込みハンドラは、サブルーチンアドレスジャンプテーブルjmp_table[]を使用して、割り込みを発生させるソースタイプに応じて別々に処理することができます。そのブロック図を図10-13に示します。rs_io.sのプログラムの構成は、基本的にこのブロック図と同じです。

図10-13 シリアル通信割り込み処理ブロック図

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

rs_io.sアセンブリファイルは、rs232シリアル通信の割り込み処理を実装しています。文字の送信や保存の際、割り込み処理は主にターミナルのリードバッファキューとライトバッファキューに対して動作します。シリアルラインから受信した文字をシリアルターミナルのリードバッファキューread_qに格納したり、ライトバッファキューwrite_qの文字をシリアルラインを通じてリモートのシリアルターミナルデバイスに送出したりします。

システムがシリアル割り込みを発生させる状況には、次の4種類があります。

MODEMのステータスが変化したため、b. ラインのステータスが変化したため、c. キャラクタを受信したため、d. 送信ホールドレジスタの空割り込みイネーブルフラグが設定されているため、送信するキャラクタがあるため。最初の2つの割り込みが発生した場合は、対応するステータスレジスタの値を読み出すことでリセットされます。キャラクタを受信するケースでは、プログラムはまずキャラクタをリード・バッファ・キューread_qに入れ、次にcopy_to_cooked()関数を呼び出して、キャラクタ・ライン・ユニットのカノニカル・モードのキャラクタに変換して補助キューsecondaryに入れます。また、文字を送信する必要がある場合には、まず、書き込みバッファキューwrite_qから文字を取り出して送信し、その後、書き込みバッファキューが空になったかどうかを判断し、まだ文字がある場合には、送信動作を周期的に行います。

そのため，このプログラムを読む前に，include/linux/tty.hというヘッダーファイルを見ておくとよいでしょう。このファイルには、文字バッファキューのデータ構造tty_queue、端末のデータ構造tty_struct、いくつかの制御文字の値が記載されています。また、バッファキューを操作するマクロの定義もあります。バッファキューとその動作図を図10-14に示す。

10.5.2 Code annotation

プログラム 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 * RS232 の io 割り込みを実装するモジュールです。11
*/
12
13 .text
14 .globl _rs1_interrupt,_rs2_interrupt
15
   // size is the length in bytes of the read and write queue buff
16 size    = 1024                          /* must be power of two !
17                                            and must match the value
18                                            in tty_io.c!!! */
19
20 /* these are the offsets into the read/write buffer structures */
   // include/linux/tty.hのtty_queue構造体の各フィールドのオフセットに対応します。
   //ファイルで、rs_addrはtty_queue構造体のデータフィールドに対応しています。しかし、シリアルの場合
21 // 端末のバッファキューは、このフィールドにシリアルポートのベースアドレス（0x3f8または0x2f8）を保持します。
22 rs_addr = 0                      // field offset of the serial ports (0x3f8 or 0x2f8).
```

```
head = 4                        // the head pointer field offset in the buffer.
tail = 8                        // the tail pointer field offset in the buffer.
proc_list = 12                  // wait for the buffered process field offset.
buf = 16                        // buffer field offset.

// 書き込みバッファのキューがいっぱいになると、カーネルは（キューを埋めた）プロセスを
// 待機状態になります。書き込みバッファのキューに256文字以上の文字が残っていないとき。
// 割り込みハンドラは、待機中のプロセスを起こして、文字の入力を続けることができます。
// 書込みキューです。
```

```
27 startup = 256                    /* chars left in write queue when we restart it */
28
30 29 /*
31  * These are the actual interrupt routines. They look where
32  * the interrupt is coming from, and take appropriate action.
32  */
    //// シリアルポート1の割り込みハンドラのエントリポイントです。
    // 初期化時、rs1_interruptのアドレスは割り込みディスクリプター0x24に配置されます。
    8259Aの割り込み要求IRQ4端子に対応する//。まず、シリアルのアドレスである
    // ttyテーブルのターミナル1（シリアルポート1）のリード／ライトバッファキューのポインタが最初に
    プッシュされる
    // をスタックに格納し（tty_io.c, 81）、rs_int にジャンプして処理を続けます。これにより
    // シリアルポート1とシリアルポート2の処理コードを共有することができます。文字の
33 // バッファキュー構造 tty_queue は、include/linux/tty.h の 22 行目にあります。
34 .align 2
35 _rs1_interrupt:
36         pushl $_table_list+8    // serial port 1 r/w queues pointer is pushed onto stack.
37         jmp rs_int
37 .align 2
38 //// シリアルポート2の割り込みハンドラのエントリポイントです。
39 _rs2_interrupt:
40         pushl $_table_list+16   // serial port 2 r/w queues pointer is pushed onto stack.

    // このコードはまず、セグメントレジスタds, esがカーネルデータセグメントを指すようにして
    // そして、対応するリード/ライトのデータフィールドからシリアルポートのベースアドレスを取得しま
    す。
    // バッファキューです。このアドレスに2を加えたものが、割り込み識別レジスタのポートアドレスとな
    る
    // IIRです。そのビット0＝0であれば、処理すべき割り込みがあることを示します。その後、に従って
    ビット2とビット1に // を入力すると、対応する割込みソースタイプ処理サブルーチンが呼び出されます。
    // ポインタジャンプテーブルを使用します。各サブルーチンは、対応する割り込みをリセットします。
    // 処理後のUARTのソースです。
    // サブルーチンが戻った後、このコードは他の割り込みソースがあるかどうかを判断するためにループ
    します。
    // (ビット0=0?)となります。この割込みに対して他の割込みソースがある場合、IIRのビット0は
    // 割り込みハンドラは、対応する割り込みソースサブルーチンを再度呼び出します。
    //で処理を継続します。この割り込みの原因となるすべての割り込みソースが処理されるまで
    // のないことを示す IIR ビット 0 =1 が自動的に設定されます。
41 // 保留中の割り込み、つまり割り込みハンドラが終了できるように。
42 rs_int:
43         pushl %edx
44         pushl %ecx
45         pushl %ebx
46         pushl %eax
47         push %es
48         push %ds                 /* as this is an interrupt, we cannot */
49         pushl $0x10              /* know that bs is ok. Load it */
```

```
50          pop %ds                 // Let ds, es point to the kernel data segment.
51          pushl $0x10
52          pop %es
53          movl 24(%esp),%edx      // get serial buffer queue address.
54          movl (%edx),%edx        // get read queue struct address -> edx.
55          movl rs_addr(%edx),%edx // get serial port 1 (or port 2) base address -> edx.
56          addl $2,%edx            /* interrupt ident. reg */
                                    // IIRポートアドレスは0x3fa(0x2fa)です。
```

// IIR（割り込み識別バイト）の内容を取得して、割り込みの原因を特定します。

// 割り込みです。ビット2とビット1で決まる割り込み条件は4種類あります。モデム

```
   // ステータスの変化、書き込まれる文字（送信）、読み込まれる文字（受信）、ラインの状態
   // を変更しています。このコードでは、まず処理すべき割り込みがあるかどうかを判断し、次に
57 // それに応じて、対応するインタラプトを
58 rep_int:
59         xorl %eax,%eax
60         inb %dx,%al              // get interrupt identification byte.
61         testb $1,%al             // bit0 = 0 ? (any interrupts ?)
62         jne end                  // no, jump to end.
63         cmpb $6,%al              /* this shouldn't happen, but ... */
64         ja end                   // if al > 6, jump to end.
65         movl 24(%esp),%ecx       // get serial buffer queue address -> ecx.
66         pushl %edx               // save the IIR port address temperoly.
67         subl $2,%edx             // edx restore to port base address 0x3f8 (0x2f8).
68         call jmp_table(,%eax,2) /* NOTE! not *4, bit0 is 0 already */
   // 上記の記述は、処理すべき割り込みがあるときに、ビット0＝0の
   // AL、ビット2、ビット1が割込みタイプなので、割込みを掛けるのと同じです。
   // 2で入力します。 そして、アドレスは4バイトなので、ここでは2を掛けて、ジャンプを得ます。
   各割込みタイプのアドレスに対応する // テーブル（79行目）を作成し、そこにジャンプして
   // 対応する処理
   // キャラクター割り込みの送信を許可するには、送信
   // ホールディングレジスターフラグ。serial.cプログラムでは、書き込みキューにデータがあるときは
   // rs_write()関数は、割り込みイネーブルレジスタの内容を変更して
   // 送信ホールドレジスタ割り込みイネーブルフラグ。
69 // システムが文字を送信する必要があります。
70         popl %edx                // restore IIR port address 0x3fa (or 0x2fa).
71         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
   // 各割込み種別処理サブルーチンのアドレスジャンプテーブルです。の4種類があります。
   // 割り込み条件。MODEMの状態変化、書き込み（送信）される文字、送信される文字
79 // 読まれる（受信される）；ラインの状態が変化する。
80 jmp_table:
81         .long modem_status,write_char,read_char,line_status
81
   // この割り込みはMODEMの状態が変化したときに発生します。にリセット操作が行われます。
82 // MODEMステータスレジスタMSRを読み出して
83 .align 2
84 modem_status:
85         addl $6,%edx             /* clear intr by reading modem status reg */
```

```
86          inb %dx,%al             // MODEM status register port: 0x3fe
87          ret
87
```

// このシリアルインタラプトは、ラインの状態が変化したときに発生します。リセット操作が行われる

```
ライン・ステータス・レジスタLSRを読み込むことで、//。
.align 2
line_status:
        addl $5,%edx            /* clear intr by reading line status reg. */
        inb %dx,%al             // LSR port: 0x3fd
        ret

// UARTがキャラクターを受信した際に発生する割り込みの処理サブルーチンです。からの読み取りは
// 受信バッファレジスタを使用することで、この割り込みソースをリセットすることができます。本サブルーチンでは、受信した
// 文字をリードバッファキューread_qの先頭に入れ，ポインタを
// 1文字分の位置。もしヘッドポインタがバッファの終端に達していたら、フォールドさせます。
// バッファの先頭に戻ります。最後に、C関数のdo_tty_interrupt()（つまり。
// copy_to_cooked())が呼び出され、読み込まれた文字がカノニカルモードに処理されます。
// バッファキュー（セカンダリバッファ）。
// 現在のシリアルポートの番号を知るには、現在のシリアルポートのキューアドレスから、その番号を差し引いてください。
// キュー・タルベ・アドレス・テーブル・リストを8で割った結果、1がシリアル1、2が
// シリアル2.
.align 2
read_char:
        inb %dx,%al             // read char in receive buffer register RBR into AL.
        movl %ecx,%edx          // serial buffer queue address -> edx
        subl $_table_list,%edx  // current serial no = (queue address - table_list) / 8
        shrl $3,%edx
        movl (%ecx),%ecx         # read-queue
        movl head(%ecx),%ebx    // get the head pointer in the read queue -> ebx.
        movb %al,buf(%ecx,%ebx) // put the char at the head position.
        incl %ebx               // and move the head pointer forward by one.
        andl $size-1,%ebx       // modulate the head pointer by the buffer length.
        cmpl tail(%ecx),%ebx    // compare the head with the tail pointer.
        je 1f                   // queue is full if equal, then jump forward.
        movl %ebx,head(%ecx)    // otherwise save the new head pointer.
1:      addl $63,%edx           // convert serial no to tty no (63 or 64) & push into stack.
        pushl %edx
        call _do_tty_interrupt  // Call tty int handler C function (tty_io.c, line 397).
        addl $4,%esp            // discard paras and ret.
        ret

// 文字の送信のサブルーチンです。この割り込みは、送信ホールディングの設定により
// レジスタイネーブルインタラプトフラグ。書き込みキューに文字があることを示す
送信する必要のあるシリアル端末に対応する//。そして、文字数
書込みキューに現在含まれている//を計算し、それが256未満であれば、プロセスは
書き込み操作を待っている // が起こされます。その後、書き込みの最後から1文字を取り出して
// バッファのキューを調整し、テールポインタを調整して保存します。書き込みバッファのキューが空の場合は、ジャンプ
// に write_buffer_empty のラベルを付けて対応します。
.align 2
```

```
write_char:
        movl 4(%ecx),%ecx            # write-queue  // address -> ecx
        movl head(%ecx),%ebx        // get head pointer of write queue to ebx
        subl tail(%ecx),%ebx        // number of chars = head - tail.
        andl $size-1,%ebx            # nr chars in queue
        je write_buffer_empty       // if head == tail, the queue is empty, then jump.
        cmpl $startup,%ebx          // check the number of chars in write queue.
        ja 1f                       // jump if more than 256 chars.
```

```
        movl proc_list(%ecx),%ebx     # wake up sleeping process  // get wait proc list.
        testl %ebx,%ebx               # is there any?    # any process waiting to write ?
        je 1f                        // none, then jump forward to label 1.
        movl $0,(%ebx)               // otherwise wake process (change state to runnable).
1:      movl tail(%ecx),%ebx         // get a char from the tail position to AL.
        movb buf(%ecx,%ebx),%al
        outb %al,%dx                 // output to THR register (port 0x3f8 or 0x2f8).
        incl %ebx                    // move the tail forward by one.
        andl $size-1,%ebx            // modulate and save the tail pointer.
        movl %ebx,tail(%ecx)
        cmpl head(%ecx),%ebx         // tail == head ?
        je write_buffer_empty        // if true, it means queue is empty, then jump.
        ret
// 次のコードは，書き込みバッファのキュー write_q が空である場合を処理します。もし
//がシリアル端子への書き込みを待っているプロセスである場合、ウェイクアップして、送信をマスクする
// ホールディングレジスターエンプティ割り込み、および送信ホールディングレジスターを開始するためのUARTのディセーブル
// 空の割り込みです。この時、書き込みバッファのキュー write_q が空であれば、それは、何の
// 文字を送信する必要があるのが現状です。そこで、次の2つのことを行う必要があります。1つ目は
// 書き込みキューが空くのを待っているプロセスがあるかどうかを判断し、ある場合は、ウェイク
// 立てます。さらに、現在システムには送信する文字がないため、一時的に
// トランスミッションホールドレジスタTHRが空の時に発生する割り込みを無効にします。文字が入力されると
// が再び書き込みキューに入ると、serial.cのrs_write()関数は、再び
// 送信保持レジスタが空になったときに割り込みが発生するようにして、UARTが
// "自動的に "書き込みキューの中の文字を取り込んで、送信します。
.align 2
write_buffer_empty:
        movl proc_list(%ecx),%ebx     # wake up sleeping process
        testl %ebx,%ebx               # is there any?
        je 1f                        // no process waiting, jump forward to label 1.
        movl $0,(%ebx)               // otherwise wake up the process.
1:      incl %edx                    // read int enable register IER (0x3f9 or 0x2f9).
        inb %dx,%al
        jmp 1f                       // delay for a while.
1:      jmp 1f                       // masks transmit hold register empty int (bit 1).
1:      andb $0xd,%al                /* disable transmit interrupt */
        outb %al,%dx
        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.

図10-14 ttyキャラクターバッファキューの動作

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()およびtty_write()は、ファイルシステムでキャラクタデバイスファイルが使用されたときに 呼び出される。例えば、プログラムが /dev/tty ファイルを読み込む場合、システムコールのsys_read()（fs/read_write.c）を実行し、読み込んだファイルがキャラクタデバイスファイルであると判断した場合には、キャラクタデバイスの読み書き機能テーブル（デバイススイッチテーブル）から、読み込んだデバイスのサブデバイス番号などに応じて rw_tty() を呼び出します。を呼び出し、最後に端末読み取り操作関数 tty_read()を呼び出す。

1. copy_to_cooked()関数は、キーボード割り込みハンドラ（do_tty_interrupt()経由）によって呼び出され、端末のtermios構造体に設定されている文字入出力フラグ（INLCR、OUCLCなど）に従って、read_qキューの文字を処理し、正規モードの文字列に変換して、上記のtty_read()による読み込み用の補助キュー（正規モードキュー）に格納する。変換処理中に、端末のエコーフラグL_ECHOが設定されていれば、その文字も書き込みキューwrite_qに入れられ、端末の書き込み関数が呼び出されて画面に表示されます。シリアルターミナルであれば、書き込み関数はrs_write()（serial.cの53行目）となります。rs_write()は、シリアルターミナルの書き込みキューにある文字をシリアルライン経由でシリアルターミナルに送信し、リモートのシリアルターミナルの画面に表示します。また、copy_to_cooked()関数は、補助キューを待っているプロセスを起こします。この関数を実装する手順は以下の通りです。

    2. If the read queue is empty or the auxiliary queue is full, skip to the last step (step 10), otherwise perform the following operations;
    3. Get a character from the tail pointer of the read queue read_q, and move the tail pointer forward by one character position;
    4. 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;
    5. If the uppercase to lowercase flag IUCLC is set, replace the character with the corresponding lowercase character;

6. 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.

7. If the receive keyboard signal flag ISIG is set, generate a signal corresponding to the typed control character for the process;
8. 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;
9. 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;
10. 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;
11. Finally wake up the processes that sleep on the secondary queue.

以下のプログラムを読むと、include/linux/tty.hというヘッダーファイルを確認することができます。このヘッダーファイルでは、ttyの文字バッファキューのデータ構造や、いくつかのマクロ操作の定義がなされています。また、制御文字のASCIIコード値も定義されています。

10.6.2 Code annotation

プログラム 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
   // <ctype.h> 文字型ファイルです。文字型変換のためのマクロを定義しています。
   // <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
   // <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、および
   //     signal manipulation function prototypes.
   // <unistd.h> Linux標準のヘッダーファイルです。様々なシンボル定数や型が定義されている
   //     and various functions are declared. If_LIBRARY_is defined, it also includes the
   //     system call number and the inline assembly _syscall0().
14 #include <ctype.h>
15 #include <errno.h>
16 #include <signal.h>
17 #include <unistd.h>
18
   // アラーム信号の対応するビットマスクを信号ビットマップに与える。
19 #define ALRMMASK (1<<(SIGALRM-1))
20
   // <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_structや
   //     data of the initial task 0, and some embedded assembly function macro statements
```

```
//      about the descriptor parameter settings and acquisition.
// <linux/tty.h> ttyヘッダーファイルは、tty_io、シリアルのパラメータと定数を定義しています。
//      communication.
// <asm/system.h> システムのヘッダーファイルです。を定義する埋め込みアセンブリマクロです。
//      modifies descriptors/interrupt gates, etc. is defined.
  // <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
//      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
// プロセスグループ関数をキルする（プロセスグループにシグナルを送る）。パラメータ pgrp
//はプロセスグループ番号、sigはシグナル、privは優先度を指定します。
// この関数の主な目的は、指定されたシグナル・シグを各プロセスに
// 指定されたプロセスグループ pgrp. プロセスへの送信が成功している間は
// 0を返します。それ以外の場合は、指定されたプロセスグループ番号pgrpに対応するプロセスが見つからなかった場合。
// エラー番号 -ESRCH が返されます。プロセスグループ番号を持つプロセスが pgrp
26 // が見つかっても、シグナルの送信操作に失敗した場合は、送信失敗のエラーコードが返されます。
27 int kill_pg(int pgrp, int sig, int priv);              // kernel/exit.c, line 171.
28                                                        // プロセスグループがオーファンプロセスであるかどうかを判定します。そうでない場合は0を、そうでない場合は1を返します。
29 int is_orphaned_pgrp(int pgrp);                        // kernel/exit.c, line 232.
28
// termios構造体の3つのモードフラグセットのうちの1つを取得するか、またはフラグが
// setには、セットフラグが含まれています。
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
33 // termios構造の特別な（ローカルな）モードフラグセットにフラグを取る。
34 #define L_CANON(tty)     _L_FLAG((tty),ICANON)     // canonical mode flag.
35 #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_ECHOCTL(tty)   _L_FLAG((tty),ECHOCTL)    // echo control char flag.
39 #define L_ECHOKE(tty)  _L_FLAG((tty),ECHOKE) // キャノンモードで消去行とエコーフラグをKILLする 40 #define L_TOSTOP(tty) _L_FLAG((tty),TOSTOP) // SIGTTOU信号を送信してバックグランド出力する。41
42 // termios構造の入力モードフラグセットからフラグを取得します。
43 #define I_UCLC(tty)      _I_FLAG((tty),IUCLC)      // get uppercase to lowercase flag.
44 #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
#define O_POST(tty)      _O_FLAG((tty),OPOST)      // Post-process output.
```

```
48
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
```

```
// termios構造の制御フラグでボーレートを取得します。CBAUDはボーレートマスク(0000017)です。
#define C_SPEED(tty)      ((tty)->termios.c_cflag & CBAUD)
// tty端末がハングアップしたかどうかを判断する、つまり送信
                          // ボーレートはB0（0）です。
#define C_HUP(tty)        (C_SPEED((tty)) == B0)

#ifndef MIN
#define MIN(a,b) ((a) < (b) ? (a) : (b))
#endif

// tty端末が使用するバッファキュー構造の配列tty_queuesを以下のように定義します。
// そして、ttyターミナルテーブル構造の配列tty_table. QUEUESは、バッファの最大数である
// tty端末で使用されるキュー。疑似端末は2種類（マスターとスレーブ）に分かれています。
// 各tty端末は、3つのttyバッファキューを使用しており、これはバッファリングされた読み取りキューである
// キーボードまたはシリアル入力 read_queue, バッファリングされた画面またはシリアル出力のための書き込みキュー。
// write_queueと、canonical modeの文字を保存するための補助的なqueue secondary。
#define QUEUES  (3*(MAX_CONSOLES+NR_SERIALS+2*NR_PTYS))     // total 54 queues.
static struct tty_queue tty_queues[QUEUES];
struct tty_struct tty_table[256];

// 以下は、様々なタイプで使用されるバッファキュー構造の開始位置を設定します。
// tty_queues[]配列のtty端末の。8つの仮想コンソール端末は，最初の
// tty_queues[]配列の24項目(3 X MAX_CONSOLES) (0 -- 23)です。
// 2つのシリアル端子は次の6項目（3 X NR_SERIALS）を占めています（24 -- 29）。
// 4つの主要な疑似端末は次の12項目（3 X NR_PTYS）（30 -- 41）を占めています。
// 4つのスレーブ疑似端末は、次の12項目（3×NR_PTYS）（42 -- 53）を占めています。
#define con_queues tty_queues
#define rs_queues ((3*MAX_CONSOLES) + tty_queues)
#define mpty_queues ((3*(MAX_CONSOLES+NR_SERIALS)) + tty_queues)
#define spty_queues ((3*(MAX_CONSOLES+NR_SERIALS+NR_PTYS)) + tty_queues)

// 以下は、さまざまなタイプのttyで使用されるtty構造の開始位置を設定します。
// tty_table[]配列の中にある端末。8つの仮想コンソール端末では、64項目（0～63）を使用できます。
// tty_table[]配列の先頭にある。
// 2つのシリアル端末は、次の2つの項目（64 -- 65）を使用します。
    // 4つのメイン擬似端末は、128から始まるアイテムを64アイテム（128 -- 191）まで使用します。
// The four slave pseudo-terminals use items starting from 192, up to 64 items (192 -- 255).
#define con_table tty_table                 // Define the console terminal tty table symbol.
#define rs_table (64+tty_table)             // Serial terminal tty table.
#define mpty_table (128+tty_table)          // The main pseudo terminal tty table.
#define spty_table (192+tty_table)          // The slave pseudo terminal tty table.

int fg_console = 0;               // Current foreground console number (range 0--7).

/*
 * these are the tables used by the machine code handlers.
```

```
 * you can implement virtual consoles.
*/
```

// tty read and write buffer queue structure address table. rs_io.sプログラムで使用され、以下を取得します。

```
// 読み書き可能なバッファキュー構造のアドレス。
struct tty_queue * table_list[]={
        con_queues + 0, con_queues + 1,     // foreground console read/write queue address.
        rs_queues + 0, rs_queues + 1,       // serial terminal 1 read/write queue address.
```

```
85          rs_queues + 3, rs_queues + 4        // serial terminal 2 read/write queue address.
85          };
86
    //// フォアグラウンドのコンソール機能を変更します。
    // フォアグラウンドコンソールを指定されたバーチャルコンソールに設定します。
    // Parameters: new_console - 指定された新しいコンソール番号です。
87 void change_console(unsigned int new_console) 88
{...
    // パラメータで指定されたコンソールが既にフォアグラウンドにある場合や、パラメータの
    // は無効です。それ以外の場合は、現在のフォアグラウンドのコンソール番号が設定され、フォアグラウンドの
    // console read and write queue 構造体の table_list[] のアドレスが更新されます。最後に更新
89  // 現在のフロントコンソールの画面です。
90          if (new_console == fg_console || new_console >= NR_CONSOLES)
91                  return;
92          fg_console = new_console;
93          table_list[0] = con_queues + 0 + fg_console*3;
94          table_list[1] = con_queues + 1 + fg_console*3;
95          update_screen();                    // kernel/chr_drv/console.c, line 936.
95 }
96
    //// キューバッファが空になると、プロセスは割り込み可能なスリープ状態になります。
    // Parameters: queue - 指定されたキューへのポインタ。プロセスは、この関数を呼び出す必要があります。
    // キューバッファ内の文字を取得する前に検証するためのものです。もし現在のプロセスが
    // プロセスへのシグナルがなく、指定されたキュー・バッファが空であれば、プロセスに
    // 割り込み可能なスリープ状態にして、キューのプロセス待ちポインタがプロセスを指すようにする。
99  97 static void sleep_if_empty(struct tty_queue * queue) 98 {。
100         cli();
101         while (!(current->signal & ~current->blocked) && EMPTY(queue))
102                 interruptible_sleep_on(&queue->proc_list);
103         sti();
103 }
104
    //// キュー・バッファがいっぱいになると、プロセスは割り込み可能なスリープ状態になります。
    // Parameters: queue - 指定したキューへのポインタ。
    // プロセスがキューバッファに文字を書き込む前に、キューの状態を決定します。
    // キューのバッファが一杯になっていない場合に返します。そうでない場合は、プロセスに処理するシグナルがない場合。
    //で、キューバッファに残っている空き領域が<128の場合、プロセスを
    // 割り込み可能なスリープ状態で、キューのプロセス待ちポインタがプロセスを指します。
07  105 static void sleep_if_full(struct tty_queue * queue) 106 {。
108         if (!FULL(queue))
109                 return;
110         cli();
111         while (!(current->signal & ~current->blocked) && LEFT(queue)<128)
112                 interruptible_sleep_on(&queue->proc_list);
```

```
113         sti();
113 }
114
```

//// キーが押されるのを待ちます。

// フォアグラウンドのコンソールの読み取りキューが空の場合、プロセスは割り込み可能なスリープ状態に入ります。

```
115 void wait_for_keypress(void)
116 {
```

```
117         sleep_if_empty(tty_table[fg_console].secondary);
118 }
119
```

//// カノニカルモードの文字列をコピーして変換します。
// 指定した端末の読み取りキューにある文字をコピーして、正規の
//モード（クックドモード）のキャラクターと、様々なフラグに応じてAUXキューに格納される
// 末端のtermios構造に設定されています。

```
120 void copy_to_cooked(struct tty_struct * tty) 121 {
122         signed char c;
123
```

// まず、現在の端末のtty構造体にあるバッファキューのポインタが有効かどうかをチェックします。

```
//      if (!(tty->read_q || tty->write_q || tty->secondary)) {
```

3つのキューのポインタがすべてNULLの場合、ttyの初期化関数に問題がある。

```
124
125                 printk("copy_to_cooked: missing queues\n\r");
126                 return;
127         }
```

// そうでなければ、ttyのリードキューバッファから取得した各文字を適切に処理します
// 端末のtermios構造の入力フラグとローカルフラグに応じて、それを
// auxiliary queueに入る secondary.
// 以下のループ本体では、読み取りキューがすでに空であるか、または補助キューが満杯の場合は
文字の//であれば、ループ本体を終了します。そうでなければ，プログラムは1文字を
// 読み込みキューの最後にあるポインターを、テールポインターを1つ前に移動させます。

// その後、文字コードに応じた処理が行われます。また、_POSIX_VDISABLE(゛゜-゛゜)の場合は
//が定義されている場合、文字コードの値が_POSIX_VDISABLEの値と同じであれば、関数

対応する特殊制御文字の//が禁止されています。

```
        while (1) {
128
129                 if (EMPTY(tty->read_q))
130                         break;
131                 if (FULL(tty->secondary))
132                         break;
133                 GETCH(tty->read_q,c);           // get a char and the tail moves foreward.
```

// 文字がキャリッジリターンCR（13）の場合は、キャリッジリターンからニューライン（ライン
// 送り）フラグCRNLが設定されている場合、その文字は改行NL（10）に変換されます。それ以外の場合は
// キャリッジリターンフラグを無視する NOCRが設定されている場合は、文字を無視して処理を続行する
// その他の文字。文字が改行文字の場合 NL(10)とラインフィードからキャリッジ
// flag UCLC is set, the character is converted to lowercase.

```
134                 if (c==13) {
```

```
135                         if (I_CRNL(tty))
136                                 c=10;
137                         else if (I_NOCR(tty))
138                                 continue;
139                 } else if (c==10 && I_NLCR(tty))
140                         c=13;
141                 if (I_UCLC(tty))
142                         c=tolower(c);
```

// ローカルモードフラグセットのカノニカルモードフラグCANONが設定されている場合、読み込まれる文字は

// 以下のように処理されます。

// まず、その文字がキーボードの終了制御文字であるKILL（^U）の場合、削除の

// 処理は、入力された現在の行に対して行われます。を削除する処理を行います。

// 一行分の文字が ttyの補助キューが空でなく、最後の文字が

補助キューでフェッチされた//は改行文字ではない NL(10)であり
// ファイル終端文字(^D)またはファイル終端文字が_POSIX_VDISABLEになっていない。
// そして、その行の他の文字をループして削除処理を行います。この間
// 消去処理では、ローカルエコーフラグECHOが設定されている場合、消去制御文字の
// ERASE (^H) を書き込みキューに入れる必要があります。その文字が制御文字の場合
//（値<32）で、2バイト表現（例：^V）の場合は、追加のイレーズ
// 文字 ERASE を配置する必要があります。その後、tty write関数が呼び出されて、すべての
// 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                         }
```

// 文字が削除制御文字ERASE(^H)であり、かつ削除文字が
// _POSIX_VDISABLE と等しい場合： tty の補助キューが空であるか、その最後の文字が
// 改行である NL(10)、またはファイルの終わりの文字であるが、_POSIX_VDISABLE と一致しない場合は続行する
//で他の文字を処理します。削除処理の途中で、ローカルエコーフラグ ECHO が
// がセットされている場合は、消去制御文字ERASE（^H）を書き込みキューに入れる必要があります。もし
// その文字が制御文字（値＜32）であり、2バイトで表現された
// 表現（例：^V）では、追加の消去文字ERASEを配置する必要があります。ttyの書き込み
その後、 // 関数が呼び出され、書き込みキューに入っているすべての文字が端末機器に出力されます。
// 最後に、tty補助キューのヘッドポインタを1バイト後退させて、コードを続行します。
// 他のチャラチャラしたものを処理するために

```
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                 }
// IXONフラグが設定されている場合は、端子の停止/開始出力制御文字が有効になります。もし
// このフラグが設定されていない場合、ストップとスタートの文字は、通常の文字として読み込まれ
// 処理を行います。このコードでは、読み込まれた文字が停止文字STOP(^S)の場合は、その文字を
// tty停止フラグ、ttyに出力を一時停止させ、特別な制御文字を破棄する（配置されていない
// を補助キューに入れる）、他の文字の処理を続けます。もし、その文字が
// 開始文字START(^Q)を入力すると，ttyの停止フラグがリセットされ，ttyの出力が回復して
// 制御文字は破棄され、他の文字は引き続き処理されます。
// コンソールでは、tty->write()は、console.cのcon_write()関数と同じです。
// 新しい文字を画面に表示するのを直ちに中止する (chr_drv/console.c, line 586) due
ストップド=1の発見に//。疑似端末の場合、書き込み操作が中断される
// (chr_drv/pty.c, line 24)のように、ターミナルストップフラグが設定されているからです。シリアルタ
ーミナルの場合
// 送信中の端末の停止フラグに応じて、送信を中断する必要がある
// 端末を使用していますが、このバージョンは実装されていません。
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                 }
// 入力モードフラグセットにISIGフラグが設定されている場合は、端末のキーボードを示す
//は、信号を生成することができ、制御文字INTR、QUIT、SUSP、DSUSPを受信したときに
// プロセスに対応する信号を生成する必要があります。その文字がキーボードの
// 割り込み(^C)、キーボード割り込み信号SIGINTをプロセス内の全てのプロセスに
現在の処理の//グループで、次の文字の処理を続けます。もし、そのような
// 文字が終了文字(^^)の場合、キーボードの終了信号SIGQUITが全プロセスに送信されます。
// を現在のプロセスのプロセスグループに追加し、次の文字の処理を続けます。
// 文字がサスペンド文字(^Z)の場合、ストップ信号SIGTSTPを現在の
//処理を行います。同様に、_POSIX_VDISABLE(0)が定義されている場合は、文字コードの値が
// 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                     }
```

// 文字が改行文字 NL(10)またはファイル終端文字 EOF(4,^D)の場合は、次のことを示します。
// 一行分の文字が処理されたことを示す、行番号「secondary.data」現在
補助キューに格納されている // を1つずつ増やしていきます。文字の行を取ると
関数 tty_read() で補助キューから // 受け取った場合、行番号は 1 つずつ減算されます。

```
// 315 行目をご覧ください。
                    if (c==10 || (EOF_CHAR(tty) != _POSIX_VDISABLE &&
210
211                                     c==EOF_CHAR(tty)))
212                             tty->secondary->data++;
```

// ローカルモードフラグセットのエコーフラグECHOが設定されている場合、文字が改行の場合は
// NL(10)、改行NL(10)、CR(13)もttyの書き込みキューに入れなければならないし
// 文字が制御文字(値<32)でエコー制御文字フラグECHOCTLの場合
//が設定されると，文字'^'と文字c+64がttyの書き込みキューに入ります（^C, ^H,
// などが表示されます）、そうでない場合は、文字が直接、tty write

```
// のキューに入ります。最後に、ttyの書き込み操作関数が呼び出されます。
                    if (L_ECHO(tty)) {
213
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                     }
```

// 処理された文字は，各ループの最後にAUXキューに入れられます.

```
                    PUTCH(c,tty->secondary);
```

```
226
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 */

//// 端末を使用するプロセスグループのすべてのプロセスに信号を送ります。

// この関数は、バックグラウンドのすべてのプロセスにSIGTTINまたはSIGTTOUシグナルを送信するために使用されます。

グループ内のプロセスが制御端末にアクセスすると、//プロセスグループ かどうかに関わらず

// バックグラウンドプロセスグループのプロセスがこの2つの信号をブロックしたり、無視したりすると

// 現在のプロセスは、直ちに読み取りと書き込みの操作を終了して戻ります。

246 int tty_signal(int sig, struct tty_struct *tty) 247 {。

// 孤児のプロセスグループのプロセスを停止したくない（行の説明を参照

ファイルkernel/exit.cの//232）を使用します。そのため、現在のプロセスグループがオーファンプロセスグループの場合は

// エラーが返されます。それ以外の場合は、指定されたシグナル・シグが

248 // 現在のプロセスグループ。

```
249         if (is_orphaned_pgrp(current->pgrp))
250                 return -EIO;            /* don't stop an orphaned pgrp */
251         (void) kill_pg(current->pgrp,sig,1);    // send signal sig.
```

// このシグナルが現在のプロセスによってブロック（マスク）されているか、無視されている場合は、エラーが返されます。

// そうでなければ、現在のプロセスがシグナルsigに新しいハンドラーを設定している場合は、次のように返します。

// 割り込まれる可能性のある情報を返します。それ以外の場合は

```
        if ((current->blocked & (1<<(sig-1))) ||
```

```
251
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機能。
// 端末の補助キューから指定された数の文字を読み込んで配置する
// ユーザが指定したバッファに格納されます。パラメータ: channel - サブデバイスの番号, buf - ユーザーが指定した
// バッファポインタ; nr - 読み込むバイト数。読み込んだバイト数を返します。

```
int tty_read(unsigned channel, char * buf, int nr) {。
        struct tty_struct * tty;
        struct tty_struct * other_tty = NULL;
        char c, * b=buf;
        int minimum,time;

```

// まず、関数のパラメータの有効性を判断し、tty構造体のポインタを取る
端末の //。tty端末の3つのキューポインタがともにNULLの場合，EIOエラー
// のメッセージが返されます。tty端末が擬似端末の場合は，別のtty構造体 other_tty
疑似端末に対応する//を取得する必要があります。

```
        if (channel > 255)
                return -EIO;
        tty = TTY_TABLE(channel);
        if (!(tty->write_q || tty->read_q || tty->secondary))
                return -EIO;
```

// 現在のプロセスが、ここで処理されているtty端末を使用しているが、プロセスグループが
端末の//番号が現在のプロセスグループ番号と異なっていることを示しています。
// 現在のプロセスがバックグラウンドプロセスグループのプロセスである場合、つまり
// フォアグラウンドではありません。そのため、現在のプロセスグループのすべてのプロセスを停止する必要があります。そのため

// 現在のプロセスグループにSIGTTINシグナルを送信してリターンする必要があり、待機している
//でフォアグラウンドのプロセスグループになってから、読み取り操作を行います。
// さらに、現在の端末が疑似端末の場合、対応する他の疑似
// 端末はother_ttyです。ここでの tty が主要な擬似端末である場合、other_tty は
274 // 対応するスレーブの疑似端末、またはその逆です。

```
275         if ((current->tty == channel) && (tty->pgrp != current->pgrp))
276                 return(tty_signal(SIGTTIN, tty));
277         if (channel & 0x80)
278                 other_tty = tty_table + (channel ^ 0x40);
```

// 続いて、コードは読み取り文字操作のタイムアウトのタイミング値と最小数を設定します。
VTIMEに対応する制御文字配列の値に基づいて読み込まれる文字の//。
//とVMINです。非正規モードでは、この2つはタイムアウトのタイミング値です。VMINは
// 読み取り操作を満足させるために必要な最小文字数。
279 // VTIMEは、1/10秒カウントのタイミング値です。

```
280         time = 10L*tty->termios.c_cc[VTIME];        // set read timeout timing value.
281         minimum = tty->termios.c_cc[VMIN];          // the minimum nr of chars to read.
```

// tty端末がカノニカルモードの場合、最小文字数を設定して
// 読み込んだ文字数nrと同じになるように、タイムアウト値
// nr文字を読むプロセスを最大値にする（タイムアウトなし）。そうでなければ、端末の
// が非正規のモードになっています。このとき、最小の読み取り文字数が設定されていれば
// 一時的なリードタイムアウトのタイミングの値を一時的に無限大にして、プロセスの読み取りを
// 補助キューにある既存の文字を先に読み出します。読んだ文字の数が
// 以下のコードでは、プロセスの読み取りタイムアウト値を
// を指定されたタイムアウト値時間に設定し、残りの文字が読み込まれるのを待ちます。
// 328行目をご覧ください。
// この時点で読み込み文字数の最小値が設定されていない場合は
// 読み込む文字数nrで、タイムアウトのタイミング値が設定されている場合は、プロセスが読み込んだ
// タイムアウトのタイミング値 タイムアウトは、現在のシステム時刻＋指定したタイムアウト時間に設定されます。
//となり、時間がリセットされます。また、上記で設定された最小読み取り文字数が
// がプロセスで読み取る文字数nrよりも大きい場合、minimum=nrとする。 その
//は、カノニカルモードでの読み出し操作に対して、制約を受けずに
// VTIMEとVMINの対応する制御文字の値を制御し、それらが唯一機能する
282 // 非正規のモード（ローモード）での操作では

```
283         if (L_CANON(tty)) {
284                 minimum = nr;
285                 current->timeout = 0xffffffff;
286                 time = 0;
287         } else if (minimum)
288                 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.
```

// ここで、AUXキューからキャラクターをループアウトさせて、ユーザ
// バッファ buf. 読み込むバイト数が0より大きい場合、次のようなループ処理を行います。
//が実行されます。ループの間、もし現在の端末が擬似端末であれば、次のことを実行します。
// 対応する他の疑似端末の書き込み操作機能を利用して、別の

```
// 擬似端末で、現在の擬似端末の補助キューに文字を書き込む
//バッファにコピーします。つまり，他の端末は，書き込みキュー・バッファの文字を
// 現在の疑似端末のリード・キュー・バッファを変換して
// カノニカルモード機能で変換された後のAUXキュー。
        while (nr>0) {
                if (other_tty)
                        other_tty->write(other_tty);

// ttyの補助キューが空の場合、または、canonical modeフラグが設定されていて、ttyのリードキューが
//が満杯ではなく、補助キューの文字列数が0であれば、プロセスの
// 読み取り文字のタイムアウト値(0)が設定されていない、または現在のプロセスが信号を受信している。
// であれば、まずループ本体を終了します。そうでない場合は、端末がスレーブ疑似端末であり
// 対応するマスター疑似端末がハングアップしたならば、ループ体も終了します。
// 上記2つの状況のいずれでもない場合は、現在のプロセスを
// 割り込み可能なスリープ状態にし、復帰後に処理を継続する。カーネルは
// カノニカルモードでは、ユーザーにデータを文字列単位で提供するために、少なくとも1つの
// このモードでの補助キューの文字のライン、つまりセカンダリ.データの最小値である
 //は1            cli();
 です。

                if (EMPTY(tty->secondary) || (L_CANON(tty) &&
                    !FULL(tty->read_q) && !tty->secondary->data)) {
                        if (!current->timeout ||
                          (current->signal & ~current->blocked)) {
                                sti();
                                break;
                        }
                        if (IS_A_PTY_SLAVE(channel) && C_HUP(other_tty))
                                break;
                        interruptible_sleep_on(&tty->secondary->proc_list);
                        sti();
                        continue;
                }
                sti();
// 以下は、文字のフェッチ操作の正式な実行を開始します。数字の
// nr=0になるか、補助バッファが終了するまで、連続してデクリメントされます。
// キューは空です。
// このループでは、補助キューの文字cが最初に取り込まれ、キューのテールポインタの
// テールは1文字分だけ右にシフトします。取得した文字がファイル
// ターミネーター(^D)または改行文字 NL(10)は、以下に含まれる文字列の行数です。
// 補助キューが1つデクリメントされます。その文字がファイル終了文字 (^D) の場合は
// で，canonical mode フラグが設定されていれば，ループは中断され，そうでなければ，ファイル終了
文字
// に遭遇していないか、または生（非正規）モードである。このモードでは、ユーザーは
// 文字ストリームを読み取り対象とし、制御文字を認識しません（例えば
// をファイルターミネーターとして使用します）。その後、コードはその文字をユーザーデータバッフ
ァに直接入れます。
```

// bufを読み込んで、読むべき文字数を1つデクリメントします。このとき、もし
読むべき文字の//がすでに0であれば、ループは中断されます。また、端末が
// がカノニカルモードで、読み込まれた文字が改行NL(10)の場合は、ループも終了します。
// 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));
```

// この時点で、ttyターミナルがcanonicalモードであれば、改行や
// end of file 文字に遭遇しました。非正規モードであれば、読み込んだnr
//の文字が入っていたり、補助キューが取り出されていたりします。そこで、まず待機中のプロセスを起こして
// をリードキューに入れて、タイムアウトのタイミング値が設定されているかどうかを確認します。もしタイムアウトのタイミングが
// 値が0でない場合は、他のプロセスが文字を書き込むのを一定時間待ちます
//を読み込みキューに入れます。そこで、プロセスリードタイムアウトのタイミング値をシステムの現在の
// 時間のジフティ＋読み取りタイムアウト時間。もちろん、端末がカノニカルモードの場合や

```
//              wake_up(&tty->read_q->proc_list);
```

// nr文字が読み込まれたので、この大きなループを直接終了することができます

```
。
327
328                 if (time)
329                         current->timeout = time+jiffies;
330                 if (L_CANON(tty) || b-buf >= minimum)
331                         break;
332         }
    // この時点でtty文字のループ操作が終了するので、プロセスの読み取りタイムアウトをリセットする
    // タイミング値のタイムアウト。現在のプロセスがシグナルを受信し、かつ、どのキャラクターも
    // がまだ読まれていない場合は、システムコール番号を再起動して返され、そうでない場合は
333 // 読んだ文字数(b-buf)です。
334         current->timeout = 0;
335         if ((current->signal & ~current->blocked) && !(b-buf))
336                 return -ERESTARTSYS;
337         return (b-buf);
337 }
338
    //// ttyの書き込み機能。
    // ユーザーバッファの文字をttyの書き込みキューバッファに入れる。
    // パラメータ： channel - サブデバイスの番号、 buf - バッファポインタ、 nr - バッファの数
    // 書き込まれたバイト数 書き込まれたバイト数を返します。
341 339 int tty_write(unsigned channel, char * buf, int nr) 340 {。
342         static cr_flag=0;
343         struct tty_struct * tty;
344         char c, *b=buf;
344
    // まずパラメータの有効性を判断し、tty構造のポインタを
    // 端末になります。tty端末の3つのキューポインターがともにNULLの場合、EIOエラーメッセージ
345 // を返します。
346         if (channel > 255)
347                 return -EIO;
348         tty = TTY_TABLE(channel);
```

```
349         if (!(tty->write_q || tty->read_q || tty->secondary))
350                 return -EIO;
    // 端末のローカルモードフラグセットでTOSTOPが設定されている場合、バックグラウンドプロセスの
    // 出力はSIGTTOUという信号を送る必要があります。このとき、現在のプロセスがttyを使用している
    場合は
    // ここで処理されている端末ですが、端末のプロセスグループ番号が異なります
    現在のプロセスグループ番号から // を選択すると、現在のプロセスが
    // バックグラウンドのプロセスグループ、つまり、そのプロセスはフォアグラウンドではないということです。そのため、私たちは
    // 現在のプロセスグループのすべてのプロセスを停止します。そのため、SIGTTOUを送る必要があります。
    // 現在のプロセスグループにシグナルを送り、フォアグラウンドプロセスになるのを待って戻る
351 //グループを作成してから、書き込み操作を行います。
352         if (L_TOSTOP(tty) &&
353             (current->tty == channel) && (tty->pgrp != current->pgrp))
354                 return(tty_signal(SIGTTOU, tty));

    // ここで、ユーザバッファbufから文字をループさせて、それを
    // 書き込みキューのバッファ。書き込むバイト数が0より大きいときは、次のように
    // ループ動作を行います。ループ中、この時点でttyの書き込みキューが満杯になっていたら
    // カレントプロセスが割込み可能なスリープ状態になる。もし現在のプロセスがシグナル
355 // 処理するために、ループ本体を終了します。
356         while (nr>0) {
357                 sleep_if_full(tty->write_q);
358                 if (current->signal & ~current->blocked)
359                         break;
    // 書き込まれる文字数nrがまだ0より大きく、tty write
    // キューバッファが満杯にならない場合、以下の操作がサイクリックに実行されます。最初に1を取る
    // バイトをユーザーバッファから取り出します。端末の実行出力処理フラグがOPOSTの場合
    // 出力モードフラグセットが設定されていると、キャラクターの後処理動作を行います。
361                 while (nr>0 && !FULL(tty->write_q)) {
362                         c=get_fs_byte(b);
363                         if (O_POST(tty)) {
    // 文字がキャリッジリターン「¥r」（CR, 13）とキャリッジリターン変換行の場合
    // 文字 OCRNL が設定されると、その文字は改行文字 '¥n' (NL, 10) に置き換えられます。
    // それ以外の場合は、文字が改行文字 '¥n' であり、かつラインフィードがリターン関数
    // フラグONLRETがセットされていると、その文字はキャリッジリターン文字'r'に置き換えられます。
365                                 if (c=='\r' && O_CRNL(tty))
361                                         c='\n';
362                                 else if (c=='\n' && O_NLRET(tty))
363                                         c='\r';
    // 文字が改行文字 '¥n' であり、キャリッジリターンフラグ cr_flag が
    // セットされていますが、カーニューラインフラグONLCRへの改行がセットされ、cr_flagフラグがセットされ
    // CRが書き込みキューに入ります。その後、次の文字の処理を続けます。
    // 小文字→大文字フラグ OLCUC が設定されている場合、その文字は大文字に変換されます。
    // 文字を表示します。
```

```
365                                 if (c=='\n' && !cr_flag && O_NLCR(tty)) {
366                                         cr_flag = 1;
367                                         PUTCH(13,tty->write_q);
368                                         continue;
368                                 }
369                                 if (O_LCUC(tty))
370                                         c=toupper(c);
371                         }
```

// 次に、ユーザデータバッファのポインタbを1バイト進め、書き込むバイト数を指定します。
// は1バイト減り、cr_flagフラグがリセットされ、そのバイトがttyのライト

// queue.

```
372                         b++; nr--;
373                         cr_flag = 0;
374                         PUTCH(c,tty->write_q);
375                 }
```

// 必要な文字がすべて書き込まれるか、書き込みキューがいっぱいになると、プログラムは終了します。

// のループに入ります。この時点で、対応する tty 書き込み関数が呼び出され、表示されます。

// 書き込みキューの文字をコンソール画面に表示したり、シリアルポートから送信したりすることができます。

// 現在処理されている tty がコンソール端末の場合、tty->write() は con_write(); if

// ttyがシリアル端末の場合、tty->write()はrs_write()関数です。まだバイトがある場合

// を書き込むためには、書き込みキューの中の文字が取り込まれるのを待つ必要があります。そこで、コール

// ここでスケジューラーを使って、他のタスクを先に進めてください。

```
377                 tty->write(tty);
378                 if (nr>0)
379                         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  */
```

//// 割り込みハンドラで呼び出される関数 - キャラクタのカノニカルモード処理。

// パラメーター： tty - 指定されたttyターミナル番号。

// 指定されたtty端末のキューの文字をカノニカル（調理済み）にコピーまたは変換する

//モードの文字を、AUXキューに格納します。この関数は、シリアル

// ポート読み取り文字割り込み（rs_io.s、110）とキーボード割り込み（kerboard.S、76）です。

```
397 void do_tty_interrupt(int tty) 398
{。
399         copy_to_cooked(TTY_TABLE(tty));
400 }
401
```

//// 文字デバイスの初期化機能。空の状態で、将来の拡張に備えます。

```
402 void chr_dev_init(void)
403 {
```

```
}

//// tty端末の初期化機能。
// すべての端末のバッファキューを初期化し、シリアル端末とコンソール端末を初期化する。
void tty_init(void)
{
        int i;
```

```
409
    // まず，全端末のバッファキュー構造を初期化し，初期値を設定する。のために
    // シリアルターミナルのリード/ライトバッファのキューは、そのデータフィールドをシリアルポートの
    ベースアドレスに設定します。
    // シリアルポート1は0x3f8、シリアルポート2は0x2f8です。次に、初期にtty構造を設定します。
    すべての端末の //。特殊文字の配列c_cc[]の初期値を定義するのは
    // include/linux/tty.hファイルです。
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         }
    // そして、コンソール端末を初期化します（console.c、834行目）。con_init()をここに置いたのは
    // に基づいて、システム内のバーチャルコンソールの数NR_CONSOLESを決定する必要があります。
    // ディスプレイカードの種類とビデオメモリの量を表します。この値は、その後の
    // コンソールのtty構造の初期化ループ。コンソールのtty構造の場合、425--430の
    //行は、tty構造体に含まれるtermios構造体のフィールドです。入力モードフラグ
    // セットはICRNLフラグに初期化され、出力モードフラグには
    // 後処理フラグOPOSTと、NLをCRNLに変換するフラグONLCR、ローカルモードフラグ
    IXON, ICANON, ECHO, ECHOCTL, ECHOKE フラグを含むように // セットが初期化されます。
    // 文字配列c_cc[]には，初期値INIT_C_CCが設定される。読み込みバッファ、書き込み
    コンソール端末のtty構造にある // バッファ、補助バッファキュー構造
    // は435行目で初期化され、それぞれ対応する構造体アイテムを指しています。
422 // ttyキュー構造の配列tty_table[]の中で、61--73行目の記述を参照してください。
423         con_init();
424         for (i = 0 ; i<NR_CONSOLES ; i++) {
425                 con_table[i] = (struct tty_struct) {
426                         {ICRNL,         /* change incoming CR to NL */
427                         OPOST|ONLCR,  /* change outgoing NL to CRNL */
428                         0,                                  // control mode flag set.
429                         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,
    // それらを初期化する必要があります。以下のループでは，まず，tty構造体を初期化します。
    // 各マスター疑似端末の対応する tty 構造を初期化します。
453     // スレーブ疑似端子。
454         for (i = 0 ; i<NR_PTYS ; i++) {
455                 mpty_table[i] = (struct tty_struct) {
456                         {0, /* no translation */   // input mode flag set.
457                         0,  /* no translation */   // output mode flag set.
458                         B9600 | CS8,             // control mode flag set. 9600bps, 8bits.
459                         0,                         // local mode flag set.
460                         0,                         // line procedure, 0--TTY.
461                         INIT_C_CC},                // control character array.
462                         0,                         // initial process group.
463                         0,                         // initial session.
464                         0,                         // initial stopped flag.
465                         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         }
    // 最後に、シリアル割り込みハンドラとシリアルインタフェース1、2（serial.c、37行目）が
    // 初期化され、仮想コンソールの数 NR_CONSOLES と疑似端末の数
            rs_init();
```

```
481
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

本プログラムは，キャラクタデバイスの制御操作に使用されるもので，関数tty_ioctl()を実装している。この関数を使うことで、ユーザプログラムは指定された端末のtermios構造体のフラグ設定などの情報を変更することができます。tty_ioctl()関数は、fs/ioctl.cのsys_ioctl()で入出力制御システムから呼び出され、 ファイルシステムベースの統合デバイスアクセスインタフェースを実現する。

一般ユーザプログラムは，システムコールであるsys_ioctl()を直接使用するのではなく，ライブラリファイルに実装されている関連機能を使用します。例えば，端末のプロセスグループ番号（フォアグラウンドプロセスグループ番号）を取得する端末IO制御コマンドTIOCGPGRPの場合，ライブラリファイルlibcは，sys_ioctl()システムコールを呼び出すコマンドを使用して，関数tcgetpgrp()を実装している。したがって、一般ユーザはtcgetpgrp()を使うだけで同じ目的を達成することができます。もちろん、同じ機能を実現するために、ライブラリ関数ioctl()を使用することもできます。

10.7.2 Code annotation

プログラム 10-6 linux/kernel/chr_drv/tty_ioctl.c

```
/*
 *  linux/kernel/chr_drv/tty_ioctl.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <termios.h> 端末入出力機能のヘッダファイル。主に端末の入出力機能を定義しています。
//     interface that controls the asynchronous communication port.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
```

// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
//　　used functions of the kernel.

// <linux/tty.h> ttyヘッダーファイルは、tty_io、シリアルのパラメータと定数を定義しています。
//　　communication.
// <asm/io.h> Ioのヘッダーファイルです。の形で、ioポートを操作する関数を定義します。
//　　a macro's embedded assembler.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
//　　for segment register operations.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
//　　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
```

// 次の行では、2つの外部関数プロトタイプとボーレート係数の配列を与えています。
シリアルポートのための //。最初の関数は、セッション番号を取得するために使用されます。
// プロセスグループは、プロセスグループ番号pgrp（kernel/exit.cで定義されています）に従って所属します。
// 161行目）。2つ目の関数tty_signal()は、プロセス内のすべてのプロセスにシグナルを送るために使用されます。
chr_drv/tty_io.c の 246 行目で定義されている、指定された tty ターミナルを使用する // グループです。
// ボーレート係数の配列（または除数の配列）は、ボーレートとの対応を示す。
//レートとボーレート係数を示しています。例えば、ボーレートが2400bpsの場合、対応する
//係数は48(0x30)、9600bpsの係数は12(0x1c)です。の後の命令をご覧ください。

```
18 // プログラムリスト。
19 extern int session_of_pgrp(int pgrp);
20 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
```

//// 送信ボーレートの変更。
// Parameters: tty - 端末に対応する tty データ構造体。割り算が
// ラッチフラグDLABがセットされると、ボーレートファクターの下位バイトと上位バイトがそれぞれ
// シリアルポート1のポート0x3f8および0x3f9を介してUARTに書き込まれた、DLABの
// ビットは書き込み後にリセットされます。シリアルポート2の場合、0x2f8と0x2f9の2つのポートがあります。

```
27 static void change_speed(struct tty_struct * tty) 28 {。
```

```
29          unsigned short port,quot;
30
```

// この関数はまず、パラメータ tty で指定された端末がシリアル端末であるかどうかをチェックします。
// し、そうでない場合は終了します。シリアルターミナルのtty構造の場合、readのデータフィールドは
// キューには、シリアルポートのベースアドレス（0x3f8または0x2f8）と、read_q.dataフィールドが格納される。
一般的なコンソール端末の//値は0です。 そして、ボーレートのインデックス番号を取得すると
//が端末のtermios構造の制御モードフラグセットから設定されていることを確認し、取得する。
// ボーレートファクター配列quotient[]から対応するボーレートファクターの値を指定します。CBAUD
//は、コントロールモードフラグが設定されている場合のボーレートビットマスクです。

```
31          if (!(port = tty->read_q->data))
32                  return;
33          quot = quotient[tty->termios.c_cflag & CBAUD];
    // その後、ボーレート係数がUARTチップのボーレート係数ラッチに書き込まれます。
    // シリアルポートに対応しています。書き込みの前に，まず，除算器のラッチアクセスを
    // ラインコントロールレジスタLCRのビットDLAB（ビット7）を書き込み、16ビットボーレートファクタの
    // ローバイトとハイバイトをそれぞれポート0x3f8, 0x3f9に送る。のDLABフラグビットをリセットします。
      //      cli();
      LCRです。
34
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
    //// tty バッファキューをクリアします。
    // Parameter: queue は、指定されたバッファキューのポインタです。
    // バッファをクリアするために、バッファのヘッドポインタをテールポインタと同じにします。
42 static void flush(struct tty_queue * queue) 43 {。
44
45          cli();
46          queue->head = queue->tail;
47          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
    //// 端末のtermios構造情報を取得します。
    // パラメータ: tty - 端末のtty構造体ポインタを指定する; termios - ユーザーバッファ
    // このようにして、Termiosの構造を保持していた。
59 static int get_termios(struct tty_struct * tty, struct termios * termios) 60 {。
61          int i;
62
    // まず，ユーザバッファポインタが示すメモリ領域が十分であることを確認する。もし
    //ない場合は、メモリを確保します。次に、指定された端末のtermios構造情報をコピーする
63  // をユーザバッファに送信します。最後に0を返します。
64          verify_area(termios, sizeof (*termios));        // kernel/fork.c, line 24.
```

```
65          for (i=0 ; i< (sizeof (*termios)) ; i++)
66                  put_fs_byte( ((char *)&tty->termios)[i] , i+(char *)termios );
67          return 0;
67 }
68
```

//// 端末のtermios構造に情報を設定します。

// パラメータ: tty - 端末のtty構造体ポインタを指定; termios - ユーザーデータ領域
69 // termios構造のポインタ。
70 static int set_termios(struct tty_struct * tty, struct termios * termios,
71 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) */
// 現在のプロセスのtty端末のプロセスグループ番号が
// プロセスのプロセスグループ番号、つまり現在のプロセス端末が
現在のプロセスがtermios構造を変更しようとしていることを示す // フォアグラウンド
制御されていない端末の //。そのため、POSIX規格の要件にしたがって
// ここでSIGTTOUシグナルを送る必要があり、このターミナルを使用しているプロセスが一時的に
// の実行を停止して、先にtermios構造を修正できるようにします。しかし、もしsend
// 関数 tty_signal() が ERESTARTSYS または EINTR の値を返した場合、操作は
77 // 後日、再度実行されます。
78 if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
79 retsig = tty_signal(SIGTTOU, tty); // chr_drv/tty_io.c, line 246.
80 if (retsig == -ERESTARTSYS || retsig == -EINTR)
81 return retsig;
81 }
// ユーザーデータエリアにあるtermios構造の情報は、その後、termios構造にコピーされます。
指定された端末のtty構造の//。ユーザーが端末の構造を変更している可能性があるため
//シリアルポート送信ボーレート、シリアルUARTチップ内のボーレートを再度修正
// コントロールモードフラグc_cflagのボーレート情報に基づく
82 // 構造体を作成し、最後に0.を返します。
83 for (i=0 ; i< (sizeof (*termios)) ; i++)
84 ((char *)&tty->termios)[i]=get_fs_byte(i+(char *)termios);
85 change_speed(tty);
86 return 0;
86 }
87
//// Termio構造の情報を取得します。
// パラメータ: tty - 端末のtty構造体ポインタを指定する; termio - ユーザーバッファ
termioの構造情報を保持する//。
90 88 static int get_termio(struct tty_struct * tty, struct termio * termio) 89 {。
91 int i;
92 struct termio tmp_termio;
92
// まず、ユーザーのバッファ容量が十分かどうかを確認し、十分でない場合は、割り当て
//メモリ。その後、termios構造の情報は、一時的なtermioにコピーされます。
//構造になっています。この2つの構造体は基本的には同じですが、入力のデータタイプが異なります。
// 出力、制御、ローカルのフラグセットが異なります。前者のデータ型はlong、後者は
// 後者は短い。したがって、一時的なtermio構造へのコピーの目的である
//はデータ型変換のためのものです。最後に、一時的なtermio構造の情報は

```
93      //バイトごとにユーザーバッファにコピーして0を返す。
94          verify_area(termio, sizeof (*termio));
95          tmp_termio.c_iflag = tty->termios.c_iflag;
96          tmp_termio.c_oflag = tty->termios.c_oflag;
97          tmp_termio.c_cflag = tty->termios.c_cflag;
```

```
98          tmp_termio.c_lflag = tty->termios.c_lflag;
99          tmp_termio.c_line = tty->termios.c_line;
100         for(i=0 ; i < NCC ; i++)
101                 tmp_termio.c_cc[i] = tty->termios.c_cc[i];
102         for (i=0 ; i< (sizeof (*termio)) ; i++)
103                 put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio );
104         return 0;
104 }
105
106 /*
107  * This only works as the 386 is low-byt-first
108  */
    //// 端末のtermio構造情報を設定します。
    // Parameters: tty - 端末のtty構造のポインタを指定する; termio - termioの
    // 構造体をユーザーデータ領域に配置します。この関数は、ユーザーの情報をコピーするために使用されます。
109 // 端末のtermios構造体にtermioをバッファリングして0を返す。
110 static int set_termio(struct tty_struct * tty, struct termio * termio,
111                         int channel)
12  111 {
113         int i, retsig;
114         struct termio tmp_termio;
114
    // set_termios()と同様に、プロセスが使用している端末のプロセスグループ番号が
    // プロセスのプロセスグループ番号とは異なる、つまり、現在のプロセス端末の
    // がフォアグラウンドにない場合は、現在のプロセスがtermiosを修正しようとしていることを示します。
    // 制御されていない端末の構造です。そのため、POSIXの要件によると
    // 標準では、この端末を使用するプロセスを許可するために、ここでSIGTTOUシグナルを送信する必要があります。
    // を使って実行を一時的に停止し、termiosの構造を先に修正できるようにしました。ただし。
    // 送信関数 tty_signal() が ERESTARTSYS または EINTR の値を返す場合は、操作
115     // は後で再度実行されます。
116         if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
117                 retsig = tty_signal(SIGTTOU, tty);
118                 if (retsig == -ERESTARTSYS || retsig == -EINTR)
119                         return retsig;
119         }
    // 次に、ユーザーデータ領域のtermio構造情報を一時的なtermioにコピーします。
    //の構造体を作成し、その情報をttyのtermios構造体にコピーします。の目的は
    // これは、モードフラグが設定されているタイプを変換するためのもので、つまり、短整数型の
    // termioを長整数型にしたものです。しかし、2つのフィールドのc_lineとc_cc[]は
120     // の構造は同じです。
121         for (i=0 ; i< (sizeof (*termio)) ; i++)
122                 ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
123         *(unsigned short *)&tty->termios.c_iflag = tmp_termio.c_iflag;
124         *(unsigned short *)&tty->termios.c_oflag = tmp_termio.c_oflag;
125         *(unsigned short *)&tty->termios.c_cflag = tmp_termio.c_cflag;
126         *(unsigned short *)&tty->termios.c_lflag = tmp_termio.c_lflag;
```

```
127         tty->termios.c_line = tmp_termio.c_line;
128         for(i=0 ; i < NCC ; i++)
129                 tty->termios.c_cc[i] = tmp_termio.c_cc[i];
    // 最後に、ユーザーが端末のシリアルポートの速度を変更した可能性があるため、ボーレートの
    // のボーレート情報に応じて変更され、シリアルUARTチップの
130     // termios構造体のモードフラグc_flagを設定し、0を返します。
131         change_speed(tty);
```

```
130         return 0;
131 }
132
    //// tty terminal device input and output control function.
    // Parameters: dev - device number; cmd - ioctl command; arg - operation parameter pointer.
    // The function first finds the tty structure of the corresponding terminal according to the
    // device number, and then processes it according to the control command cmd.
133 int tty_ioctl(int dev, int cmd, int arg)
134 {
135         struct tty_struct * tty;
136         int     pgrp;
137
    // まず、デバイス番号に応じてttyのサブデバイス番号を取得し、それにより
    // 端末のtty構造を示す。メジャーデバイス番号が5(制御端末)の場合は
    プロセスの // tty フィールドは tty サブデバイス番号です。このとき、もしプロセスの tty
    // サブデバイス番号が負の数の場合は、プロセスがコントロールを持っていないことを示す
    // 端末では、すなわちioctlコールが発行できず、エラーメッセージが表示されて
    // マシンを停止します。メジャーデバイスの番号が5ではなく4の場合は、サブデバイスの
    デバイス番号からの // 番号です。サブデバイスの番号は、0（コンソール端末）、1（シリアル
138 // 1端子）、または2（シリアル2端子）を使用します。
139         if (MAJOR(dev) == 5) {
140                 dev=current->tty;
141                 if (dev<0)
142                         panic("tty_ioctl: dev<0");
143         } else
144                 dev=MINOR(dev);
    // そして、サブデバイスの番号とttyテーブルにしたがって、ttyの構造を
    // 対応する端末を指します。に対応する tty 構造体を tty に指定します.
    // サブデバイスの番号を指定し、指定されたioctlコマンドcmdに従って個別に処理を行う
    // をパラメータとして使用しています。144行目の後半では、対応するtty構造を選択するために
    サブデバイス番号devに基づいて、tty_table[]テーブルの中の//。dev = 0の場合、それは
    // フォアグラウンドのターミナルが使用されているので、ターミナル番号を直接使用できる fg_console
    // tty_table[]のエントリーインデックスとして、tty構造を取得します。devが0より大きい場合は
    //は2つのケースで考える必要があります。(1) dev は仮想端末の番号、(2) dev は
    // シリアルターミナル番号または疑似ターミナル番号。仮想端末の場合は、tty構造の
    tty_table[]内の // は dev-1(0 -- 63)でインデックス化されており、他のタイプの端末の場合はその tty
    // 構造体のインデックスエントリはdevです。例えば、シリアルターミナル1を示すdev=64の場合、その
    // tty構造はtty_table[dev]です。dev = 1 の場合、対応する端末の tty 構造は
145 // はtty_table[0]です。tty_io.cプログラムの70～73行目をご覧ください。
146         tty = tty_table + (dev ? ((dev < 64)? dev-1:dev) : fg_console);

    // TCGETS: 対応する端末のtermios構造情報を取得します。この時点で
    // パラメータargはユーザバッファのポインタです。
    // TCSETSF：ターミオスを設定する前に、出力キュー内のすべてのデータを待って
    // を処理し、入力キューをフラッシュします。その後、termiosを設定する操作を行います。
    // TCSETSW:termiosを設定する前に、出力キューのすべてのデータを待って
    // を処理します。この形式は、パラメータを変更すると出力に影響する場合に必要です。
```

// TCSETS: 対応する端末のtermios構造情報を設定します。この時点では
//パラメータargは、termios構造体を保持するユーザバッファポインタです。
// TCGETA: 対応する端末のtermio構造の情報を取得します。このとき
// パラメータ arg がユーザバッファポインタであることを示す。
// TCSETAF: termioを設定する前に、出力キューのすべてのデータが処理されるのを待つ必要がある
//となり、入力キューがフラッシュされます。その後、設定端末のtermio操作を行います。

```
// TCSETAW：ターミオスを設定する前に、出力キューのすべてのデータを待って
// を処理します。この形式は、パラメータを変更すると出力に影響するような場合に必要です。
// TCSETA：対応する端末のtermio構造情報を設定します。この時点で
// パラメータargは、termio構造体を保持するユーザバッファポインタです。
// TCSBRK：パラメータargの値が0の場合、出力キューの処理を待つ
// して、ブレークを送ります。
        switch (cmd) {
                case TCGETS:
                        return get_termios(tty,(struct termios *) arg);
                case TCSETSF:
                        flush(tty->read_q);     /* fallthrough */
                case TCSETSW:
                        wait_until_sent(tty);   /* fallthrough */
                case TCSETS:
                        return set_termios(tty,(struct termios *) arg, dev);
                case TCGETA:
                        return get_termio(tty,(struct termio *) arg);
                case TCSETAF:
                        flush(tty->read_q);     /* fallthrough */
                case TCSETAW:
                        wait_until_sent(tty);   /* fallthrough */
                case TCSETA:
                        return set_termio(tty,(struct termio *) arg, dev);
                case TCSBRK:
                        if (!arg) {
                                wait_until_sent(tty);
                                send_break(tty);
                        }
                        return 0;
// フロー制御を開始／停止する。パラメータargがTCOOFF（端子制御出力OFF）の場合は
//の出力が保留され、TCOONであれば、保留されていた出力が復元されます。と同じタイミングで
// 出力を中断または再開する際には、書き込みキューにある文字を
// ユーザーインタラクションのレスポンスを高速化する。argがTCIOFF(Terminal Control Input ON)の
場合は
// TCIONであれば、保留されていた入力が再び行われます。
                case TCXONC:
                        switch (arg) {
                        case TCOOFF:
                                tty->stopped = 1;       // stop the terminal output.
                                tty->write(tty);        // write queue output.
                                return 0;
                        case TCOON:
                                tty->stopped = 0;       // restore the terminal output.
                                tty->write(tty);
                                return 0;
// パラメータargがTCIOFFの場合は、端末の入力停止が要求されていることを意味しますので
// STOP文字を端末の書き込みキューに入れて、入力を中断する
// 端末がその文字を受信したとき。パラメータがTCIONの場合は、START
端末に送信を再開させるために // 文字を送信します。
```

// STOP_CHAR(tty)は、((tty)->termios.c_cc[VSTOP])と定義されており、これを使って
// 端末のtermios構造体の制御文字配列の対応する値。もし
// カーネルで_POSIX_VDISABLE(0)が定義されている場合は、アイテムの値が
// _POSIX_VDISABLE は、対応する特殊文字が禁止されていることを意味します。つまり、ここでは
// 値が0かどうかを直接チェックして、停止制御文字を入れるかどうかを判断する

```
// 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).
// TIOCGPGRP: 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 TIOCGPGRP:                         // implement function tcgetpgrp().
206                     verify_area((void *) arg,4);
207                     put_fs_long(tty->pgrp,(unsigned long *) arg);
208                     return 0;
```

//端末のプロセスグループ番号pgrpを設定する（つまり、フォアグラウンドのプロセスグループ番号を設定する）。

//パラメータargは、ユーザバッファ内のプロセスグループ番号pgrpへのポインタになりました。このようにして

//このコマンドを実行するには、プロセスに制御端末があることが前提となります。

//現在のプロセスが制御端末を持っていないか、devがその制御端末でない場合。

//または制御端末が処理中の端末devになりましたが、セッション番号の

//プロセスがターミナルデブのセッション番号と異なる場合は、ターミナレスの

//エラーメッセージが返されます。

//そして、ユーザーバッファからプロセスグループ番号を取得し、その有効性を検証します。

// グループ番号です。グループ番号pgrpが0より小さい場合は、無効なグループ番号のエラーメッセージ
// pgrpのセッション番号が現在のプロセスと異なる場合は、パーミッション
// エラーメッセージが返されます。それ以外の場合は、端末のプロセスグループ番号を設定します。

//をprgpに変更します。この時点で、prgpはフォアグラウンドのプロセスグループになります。

```
210                 case TIOCSPGRP:                     // implement function tcsetpgrp().
211                         if ((current->tty < 0) ||
212                             (current->tty != dev) ||
213                             (tty->session != current->session))
214                                 return -ENOTTY;
215                         pgrp=get_fs_long((unsigned long *) arg); // get from user buffer.
216                         if (pgrp < 0)
217                                 return -EINVAL;
218                         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

// ユーザーのバッファの長さに合わせて、キューの中の文字数をユーザーにコピーします。このとき

// パラメータ arg がユーザバッファポインタであることを示す。

// TIOCINQ: 入力補助キューでまだ読み込まれていない文字の数を返す。のは

// ユーザーバッファの長さが最初に検証され、AUXの文字数が

// キューがユーザーにコピーされます。この時点でのパラメータargは、ユーザバッファのポインタです。

```
222                 case TIOCOUTQ:
223                         verify_area((void *) arg,4);
224                         put_fs_long(CHARS(tty->write_q),(unsigned long *) arg);
225                         return 0;
226                 case TIOCINQ:
227                         verify_area((void *) arg,4);
228                         put_fs_long(CHARS(tty->secondary),
229                                 (unsigned long *) arg);
230                         return 0;
```

// TIOCSTI: 端末の入力操作をシミュレートします。このコマンドは，文字へのポインタを

// をパラメータとし、その文字が端末で入力されたものと仮定します。ユーザーには

// スーパーユーザー権限、または制御端末の読み取り権限を持っていること。

// TIOCGWINSZ: ターミナルデバイスのウィンドウサイズを読み取る（termios.hのwinsize構造体を参照）。

// TIOCSWINSZ: ターミナルデバイスのウィンドウサイズ情報を設定する（winsize構造体を参照）。

```
232                 case TIOCSTI:
233                         return -EINVAL; /* not implemented */
234                 case TIOCGWINSZ:
235                         return -EINVAL; /* not implemented */
236                 case TIOCSWINSZ:
237                         return -EINVAL; /* not implemented */
```

// TIOCMGET: MODEMのステータスコントロールピンに設定されている現在のステータスビットフラグを返します。

// （termios.hの185～196行目を参照）。

// TIOCMBIS: MODEMのステートコントロールピン1本の状態（trueまたはfalse）を設定します。

// TIOCMBIC: MODEMのステートコントロールピン1本の状態をリセットすることができます。

// TIOCMSETです。MODEMのステータスピンの状態を設定します。ビットがセットされた場合、対応するステータスが

```
// ピン          case TIOCMGET:
がアサ
ートさ
```

れます。

```
236
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: ソフトウェアキャリア検出フラグを読み取ります（1 - オン、0 - オフ）。

```
//
tio
css
oft
car
:
ソ
フ
ト
ウ
ェ
ア
キ
ャ
リ
ア
検
出
フ
ラ
グ
を
設
定
し
ま
す
（
1 -
オ
ン
、
0 -
オ
フ
）
。
244
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).

共通ボーレートとボーレートファクターの対応関係を表10-9に示す。

表10-9 ボーレートとボーレート係数の表	Baud rate factor		Baud rate	Baud rate factor	
	MSB,LSB	Value		MSB,LSB	Value

Baud rate					
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.

次の章では，数学コプロセッサの基本的な機能と使用される基本的なデータ型について詳しく説明し，Linuxカーネルで数学コプロセッサをソフトウェアでエミュレーションするための実装コードを指定します。

11 Math Coprocessor (math)

カーネルディレクトリkernel/mathディレクトリには，リスト11-1に示すように，数学コプロセッサのシミュレーションプログラムが格納されており，合計9つのCプログラムが含まれています。本章の内容は，具体的なハードウェア構造と密接に関係しているため，読者はインテルのCPUやコプロセッサの命令コード構造について深い知識を持っている必要があります。幸いなことに、この内容はカーネルの実装とはほとんど関係がないので、この章を飛ばしてもカーネルの実装を完全に理解することはできません。しかし、本章の内容を理解していれば、システム・レベルのアプリケーション（アッセンブ ルや逆アセンブルなど）の実装や、コプロセッサの浮動小数点ハンドラの開発に非常に役立 つでしょう。

リスト 11-1 linux/kernel/math

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

Linux 0.1xはもちろん、Linux 0.9xのカーネル開発の初期には、数学コプロセッサチップ80387（また

はその互換チップ）が高価だったため、一般のPCでは常に贅沢品でした。そのため、以下のような場合を除いては

が大量に科学的な計算をする場合や、特に必要とされる場所では、80387チップは一般的なPCには搭載されない。現在のインテル・プロセッサーには数学コプロセッサ機能が組み込まれているが、現在のOSではコプロセッサのエミュレーターコードは不要になっている。しかし、8037エミュレーションプログラムは、完全にアナログ8037チップの処理構造と命令コード構造の解析に基づいているため、この章では、8037エミュレーションプログラムの概要を説明します。ですから、この章の内容を学んだ後は、80387コプロセッサのプログラミング方法を完全に理解できるだけでなく、アセンブラやディスアセンブラのようなシステムレベルのプログラムを書くのにも役立つのです。

80386 PCに80387 mathコプロセッサチップが搭載されていない場合、CPUがコプロセッサ命令を実行すると、「device nonexistent」という例外割り込みが発生します。この例外処理の処理コードはsys_call.sの158行目から始まります。オペレーティングシステムが初期化時にCPU制御レジスタCR0のEMビットを設定済みであれば、math_emulate.cプログラムのmath_emulate()関数が呼び出され、コプロセッサの各命令をソフトウェアで「解釈」します。

11.1.1 Linux 0.12カーネルの数学コプロセッサエミュレータmath_emulate.cは、80387チップのコプロセッサ命令の実行方法を完全にエミュレートします。コプロセッサ命令を処理する前に、プログラムはまず、データ構造などを用いてメモリ上に「ソフトな」8037環境を構築します。この環境では、8037内部のスタックアキュムレータ群ST[]、コントロールワードレジスタCWD、ステータスワードレジスタSWD、フィーチャーワードTWD（TAGワード）レジスタをすべてエミュレートし、例外の原因となっている現在のコプロセッサ命令のオペコードを解析し、特定のオペコードに応じて対応する数学シミュレーション演算を行います。したがって、math_emulate.cプログラムの処理を説明する前に、80387の内部構造と基本的な動作原理を紹介する必要がある。

11.1.2 Floating point data type

1. ここでは、コプロセッサで使用される浮動小数点データの種類に注目します。まず、整数のいくつかの表現について簡単におさらいしてから、浮動小数点数のいくつかの標準的な表現と、80387の演算で使われる一時的な実数表現について説明します。
2. Integer data type

インテルの32ビットCPUでは、基本的な符号なしデータタイプは、バイト、ワード、ダブルワードの3種類で、それぞれ8ビット、16ビット、32ビットです。符号なしの数値の表現は簡単です。バイトの各ビットは2進数を表し、その位置によって重みが異なります。例えば、8ビットの符号なし2進数0b10001011は次のように表現できる。

U = 0b10001011 = 1 x 27 + 0 x 26 + 0 x 25 + 0 x 24 + 1 x 23 + 0 x 22 + 1 x 21 + 1 x 20 = 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).

さらに、コンピュータの中で負の値を持つ整数データの表現は、通常3種類あります。2の補数、バイアス数、符号の大きさの表現です。表11-1は、これら3つの形式で表現される値の一部です。

表11-1 整数のいくつかの表現方法 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 -- 2146473647.

数値のバイアス表現は、通常、浮動小数点フォーマットのインデックス・フィールド値を表現するために使用されます。指定された偏った値に数値を加えると、その数値の偏った数値表現になります。表11-1からわかるように、この表現の数値は、符号なしの数値とは桁違いの大きさを持っています。したがって、この表現は、数値の比較が容易であり、つまり、偏った表現の数値の大きな値は、必ず符号なしの数値の大きな値となり、他の2つの表現はそうではないということになります。

符号付きの数値表現では、符号の表現専用のビット（正の数は0、負の数は1）があり、その他のビットは符号なし整数で表現される値と同じである。浮動小数点数の有効数字（仮数）部分はこの表現方法を採用しており、符号ビットは浮動小数点数全体の符号を表す。

また、80387エミュレータでは、図11-1に示すように、一時的な整数型と呼ぶフォーマットが使用されています。これは10バイト長で、64ビットの整数データ型を表現できる。下位8バイトは最大63個の符号なし数値を表すことができ、上位2バイトは最上位ビットのみで正負の値を表します。32ビットの整数値の場合は下位4バイトを使用し、16ビットの整数値は下位2バイトで表します。

図11-1 エミュレータがサポートする一時的な整数フォーマット

3. 2. BCD data type

BCD（Binary Coded Decimal）コード値は、2進数でコード化された10進数の値です。圧縮BCDコードでは、各バイトが2桁の10進数を表すことができ、4ビットごとに0～9の桁を表します。例えば、

10進数59を圧縮BCDコードで表現すると、0x01011001となります。圧縮されていないBCDの場合

符号は、各バイトが下位4ビットのみで1桁の10進数を表現します。

80387コプロセッサは、図11-2に示すように18ビットの10進数を表すことができる10バイトの圧縮BCDコードの表現と操作をサポートしています。一時的な整数形式と同様に、最上位バイトでは符号ビット（最上位ビット）のみで値の正負を表し、残りのビットは使用しません。BCDコードデータが負の場合、最上位バイトの有効な最上位ビットが負の値を示すために使用されます。それ以外の場合は、最上位バイトのすべてのビットが0になります。

図11-2 80387がサポートするBCDコードデータタイプ

4. Floating point data type

整数部と小数部（仮数部）を持つ数を実数または浮動小数点数と呼ぶ。実際には、整数とは分数部分が0の実数であり、実数集合の部分集合である。コンピュータは固定長のビットで数値を表現するため、すべての実数を正確に表現することはできません。コンピュータが実数を表現する際に、最も正確な実数値を固定長のビットで表現するためには、分数部に割り当てるビット数が固定されていないため、つまり、小数点が「浮く」ことができるため、コンピュータが表現する実数データ型は、浮動小数点数とも呼ばれています。プログラムの移植を容易にするために、現在、コンピュータでは実数を表現するのにIEEE規格754（または854）で規定された浮動小数点数表現を使用しています。

この実数表現の一般的なフォーマットを図11-3に示す。Significand部、Exponent部、Signビットで構成されています。Significand部は、整数の1と分数の1で構成されています。80387コプロセッサは3つの実数型をサポートしており、各部の使用ビット数を図11-4に示します。これらのデータ型は、80ビットのテンポラリーリアル（または拡張リアルと呼ばれる）形式を除いて、すべてメモリ上にのみ存在する。80387コプロセッサのデータレジスタに読み込まれると、一時実数形式に変換され、その形式で演算されます。

図11-3 浮動小数点の一般的なフォーマット

図11-4 80387コプロセッサの実数フォーマット

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 x 10^-2, or 10.34 x 10^0, or 1.034 x 10^1 or 0.1034 x 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 x 10^1. For binary numbers it is 1.XXXXX * 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.

フォーマットの指数フィールドには、数値を正規の形で表現するために必要な2の累乗が格納されています。前述したように、数値の大きさを比較しやすくするために、80387は指数の値を格納するために偏った数値形式を使用しています。短実数、長実数、拡張実数（仮実数）の偏った基数は、それぞれ127、1023、16383であり、対応する16進数はそれぞれ0x7F、0x3FF、0x3FFFである。したがって、短い実数の指数値である0b10000000は、実際には2^1（0b01111111＋0b00000001）を表しています。

また、一時実数とは、80387の内部演算で表される数値の形式のことです。その最上位の数字1はビット63に明示的に配置されており、与えられるデータ型（例えば、整数、短実数、BCDコードなど）にかかわらず、80387はこの一時実数形式に変換します。80387の目的は、精度を最大限に高め、演算中のオーバーフロー例外を最小限に抑えることです。明示的に1を示すのは、80387が演算中にこのビットを必要とするためです（非常に小さな値を表すために使用されます）。80387に入力された短い実数または長い実数が一時的な実数フォーマットに変換されると、ビット63に1が明示的に置かれます。

処理中、プログラムはlinux/math_emu.hファイルに一時的な実数のデータ構造を定義します。

```
typedef struct {
    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.

```
typedef struct {
      short m0,m1,m2,m3;
      short exponent;
} temp_real_unaligned;
```

5. Special real numbers

上の表の一部の値が表現できない場合と同様に、実数で表される値にも特別な意味を持つものがあります。80ビット長フォーマットの一時的な実数の場合、80387は表現可能な範囲の値をすべて使用するわけではありません。表11-2は、80387が使用している一時的な実数で表現できるすべての可能な値であり、有効数字列の破線の左側の1ビットが一時的な実数のビット63を示している、つまり値1のビットが明示されている。短い実数や長い実数にはこのビットがありませんので、表には擬似的な非正規化カテゴリは存在しません。特殊な値として、ゼロ、無限大、非正規化、疑似正規化、そしてシグナリングのNaN（Not a Number）とquiet NaNを見てみましょう。

表11-2 仮設実数が表す値の種類と範囲 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	
0/1	11...11	1	...	Signalling NaNs - SNaNs
0/1	11...11	1	00...01	
0/1	11...11	1	00...00	Infinite
0/1	11...10	1	11...11	
0/1	...	1	...	Normals
0/1	00...01	1	00...00	
0/1	00...00	1	11...11	
0/1	00...00	1	...	Pseudo-Denormals
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

指数値の0x11...11も予約済みです。

デノルマル数は、非常に小さな値を表すために使用される特別なクラスの値です。プログレッシブアンダーフローやプログレッシブ精度の低下を示すことがあります。この値は通常、正規化された数値（有効数字の最上位ビットがビット1になるまで左シフト）として表されることが要求されますが、一方で、正規化されていない数値の最上位桁は1ではありません。このとき、偏った指数0x00...00は、短い実数、長い実数、一時的な実数の指数値である2^-126、2^-1022、2^-16382を、それぞれ特別に表現したものです。この表現が特別なのは、偏った指数値0x00...01が、3つの実数型の同じ指数値2^-126, 2^-1022, 2^-16382もそれぞれ表しているからです。

擬似非正規級数とは、符号部の最上位ビットが1で、非正規級数のビットが0の値です。 擬似非正規級数は稀なもので、正規級数で表すこともできますが、そうではありません。特殊な偏りのある指数0x00...00は、正規化数の指数0x00...01と同じ値であることは上で説明したとおりなので、疑似非正規化クラス番号は正規化クラス値として表現することができます。

もう一つの特殊なケースはNaN（Not a Number）です。NaNには、シグナルを発するもの（シグナリングNaN）と、シグナルを発しないもの（クワイエットNaN）の2つの形態があります。シグナリングNaN（SNaN）が操作に使用され、クワイエットNaN（QNaN）が使用されない場合、無効な操作例外が投げられます。SNaNは、使用前にすべての変数が初期化されているかどうかを判断するために使用できる。その方法は、プログラムが変数をSNaN値に初期化することで、操作中に初期化されていない値が使用された場合には、例外が発生するようになっています。もちろん、NaNクラスの値は他の情報を格納するためにも使用できる。

80387自身はSNaNクラスの値を生成しませんが、QNaNクラスの値を生成します。無効な操作の例外が発生すると、80387はQNaNクラスの値を生成し、操作の結果は不定型の値になります。不定型の値は、特別なQNaNクラスの値です。各データ型には、不定値を表す数値があります。整数の場合は、最大の負の数がその不定値を表すために使用されます。

また、80387がサポートしていない一時的な実数値、つまり上記の表に記載されていない値もあります。80387がこれらのサポートされていない値に遭遇した場合は、無効な操作の例外が発生します。

11.1.3 Math Coprocessor Function and Structure

80386は汎用のマイクロプロセッサですが、その命令は数学的な計算にはあまり適していません。そのため、80386を使って数学的な計算をしようとすると、非常に複雑なプログラムをコンパイルする必要があり、実行効率が相対的に悪くなってしまう。80387は、80386の補助処理チップとして、プログラマーのプログラミングの幅を大きく広げた。今までプログラマーがやりそうになかったことが、コプロセッサを使うことで、簡単に、早く、正確にできるようになります。

80387には特別なレジスタセットがあり、80386が扱うことのできる桁数よりも大きい、または小さい桁数を直接操作することができます。80386では、数値の表現に2進数の補数を用います。この方法は、小数の表現には適していません。80387では、2の補数方式ではなく、80ビット（10-by-10）方式で数値を表現します。IEEE規格754で規定されている80ビット（10バイト）のフォーマットを採用している。このフォーマットは、幅広い互換性があるだけでなく、大きな（あるいは小さな）値を2進数で表現できるのが特徴だ。例えば、1.21×10^4932という大きな値を表すことができ、3.3×10^4932という小さな数値を扱うこともできる。80387は、小数点の位置を固定していません。値が小さい場合は小数点以下の桁数が多くなり、値が大きい場合は小数点以下の桁数が少なくなります。そのため、小数点の位置は「浮動小数点」とすることができる。これが「浮動小数点」という言葉の由来でもある。

浮動小数点演算をサポートするために、80387には図11-5に示すような3つのレジスタが搭載されています。(1) 8個の浮動小数点オペランドを一時的に格納するために使用され、これらのアキュムレータがスタック操作を行うことができる8個の80ビット・データ・レジスタ（アキュムレータ）、(2) 3つの16ビット・ステータスおよびコントロール・レジスタ：ステータス・ワード・レジスタSWD、コントロール・ワード・レジスタCWD、およびフィーチャー（TAG）レジスタ、(3) 4つの32ビット・エラー・ポインタ・レジスタ（FIP、FCS、FOO、FOS）は、8037の内部例外を引き起こした命令およびメモリ・オペランドを決定するために使用される。

図11-5 80387のレジスタ

1. Stack floating point accumulator

浮動小数点命令の実行中、8つの80ビット物理レジスタ・セットがスタック・アキュムレータとして使用されます。各80ビットのレジスタは物理的に固定された順序で配置されていますが（例：左から0～7）、スタックの現在のトップはST（例：ST(0)）で示されます。STの下にある残りのアキュムレータは、ST(i)という名前で示されます（i ＝ 1 -- 7）。どの80ビット物理レジスタが現在のスタックトップSTであるかについては、特定の演算処理によって指定される。図11-6に示すステータスワードレジスタのTOPと名付けられた3ビットフィールドには、現在のトップSTに対応する80ビット物理レジスタの絶対位置が格納されている。プッシュまたはロード操作により、TOPフィールドの値が1つデクリメントされ、新しいSTに新しい値が格納されます。プッシュ操作の後、元のSTはST(1)となり、元のST(7)は現在のSTとなります。つまり，すべてのアキュムレータの名前が，元のST(i)からST((i+1)&0x7)に変わったのです。ポップまたはストア操作は、現在のTOPレジスタSTから値を読み取ってメモリに格納し、TOPフィールドの値を1だけ増加させる。したがって、ポップ操作の後、元のST（すなわちST(0)）はST(7)となり、元のST(1)は新しいSTとなる。つまり、すべてのアキュムレータの名前は、元のST(i)からST((i-1)&0x7)へと変更されるのである。

STは、すべての浮動小数点命令の暗黙のオペランドとして使用されるため、アキュムレータのように動作します。他にオペランドがある場合は、第2オペランドは残りのアキュムレータST(i)のいずれか、またはメモリオペランドとなります。スタック内の各アキュムレータは、実数の一時的な実数形式で格納された80ビットの空間を提供し、最上位ビットは符号ビット、ビット78--64は15ビットの指数フィールド、ビット63--0は64ビットの有効数フィールドとなります。

浮動小数点命令は、このアキュムレータ・スタック・モードを最大限に活用するように設計されています。浮動小数点ロード命令（FLDなど）は、メモリからオペランドを読み込み、スタックにプッシュします。

2. 浮動小数点ストア命令は，スタックの現在の最上位から値を取り出し，メモリに書き込む．スタック内の値が不要になった場合は、ポップ演算も同時に行うことができます。和算や乗算などの演算では、現在のSTレジスタの内容を一方のオペランドとし、他のレジスタやメモリーから他方のオペランドを取り、計算後の結果をSTに保存する。また、STとST(1)間の演算には、「オペレート＆ポップアップ」というタイプの演算がある。この形式の演算では、ポップアップを行い、その結果を新しいSTに配置する。

3. Status and Control Register

16ビットの3つのレジスタ（TAGワード、コントロールワード、ステータスワード）は、浮動小数点命令の動作を制御したり、そのステータス情報を提供する。それらの具体的なフォーマットを図11-6に示しますが、以下で一つずつ説明していきます。

RC	Rounding Mode
00	Round to nearest(even)
01	Round down(toward −∞)
10	Round up(toward +∞)
11	Round toward0(Truncat)

PC	Precision
00	24-bit Sigle Precision
01	Reserved
10	53-bit (Long Real)
11	64-bit (Temp Real)

Stat	Mask	Condition
IE	IM	Invalid Operation
DE	DM	Denormal OPed
ZE	ZM	Zero Divide
OE	OM	Overflow
UE	UM	Underflow
PE	PM	Precision

Field	Description
PE, UE, OE, ZE, DE, IE	These bits indicate exceptions that occurred while executing fp instruction. Must be reset by program.
SF	Stack error flag. This flag is set if the invalid operation is due to the overflow of the accumulator, otherwise it is automatically reset.
B, ES	FPU Busy, Error Summary Status. If any unmasked exception in bit 5-0 occurs, these 2 bits are set, otherwise they are all 0.
C3, C2, C1, C0	The condition code, similar to EFLAGS of CPU, are used for floating point comparison instructions.
TOP	Top of Stack Pointer, used to indicate which 80-bit register is currently at the top of the stack, ie which physical register corresponds to accumulator ST.

Tag	Accumulator
00	Valid
01	Zero
10	Special:invalid (NaN, non-numeric, Denormal)
11	Empty

図11-6 コントロール、ステータス、タグレジスタのフォーマット

A. Control Word

16ビットのControl Wordは、80387の動作を制御するためのさまざまな処理オプションをプログラムするのに使用できます。これは3つの部分に分けられます。(1) 11--10ビットのRC(Rounding Control)は，計算結果を丸めるための丸め制御フィールド (2) 9-8ビットの

PC(Precision Control)は，指定したメモリに保存する前に計算結果の精度を調整するための精度制御フィールド

B. の位置を示します。その他の演算では、一時的な実数形式の精度を使用するか、命令で指定された精度を使用します。（3）ビット5--0は、コプロセッサの例外処理を制御するための例外マスクビットです。この6ビットは、80387で発生する可能性のある6つの異常に対応しており、それぞれを個別にマスクすることができる。特定の例外が発生し、その対応するマスクビットが設定されていない場合、80387はCPUに例外を通知し、CPUに例外int16を発生させます。しかし、対応するマスクビットがセットされていれば、80387はCPUに通知することなく、自分自身で異常な問題を処理し、修正します。このレジスタは、いつでも読み書き可能です。各ビットの具体的な意味を図11-6に示す。

C. Status Word

D. 80387は動作中、プログラムが特定の状態を検出するために、Status Wordのビットを設定します。例外が発生すると、CPUに例外の原因を判断させます。なぜなら、コプロセッサの6つの例外はすべて、CPUに同じ例外int16を発生させるからです。

E. Tag Word

4. タグ・ワード・レジスタは、8つの物理的な浮動小数点データ・レジスタに対する8つの2ビット・タグ・フィールドを含んでいます。これらのタグ・フィールドは、対応する物理レジスタがそれぞれ有効な浮動小数点値、ゼロ、または特殊な浮動小数点値を含んでいるか、あるいは空であることを示します。特殊な値とは、無限大、非数値、非正規化、非サポートなどの値です。タグ・フィールドは、アキュムレータ・スタックのオーバーフローの検出に使用できます。プッシュ操作がTOPをデクリメントし、空ではないレジスタを指している場合、スタックのオーバーフローが発生します。popオペレーションが空のレジスタをリードまたはポップしようとした場合、スタックのアンダーフローが発生します。スタックのオーバーフローとアンダーフローの両方で、無効な操作の例外が発生します。

5. Error-Pointer Register

6. エラー・ポインター・レジスタは、図11-6に示すように、最後に実行された命令と使用されたデータへの80387個のポインタを含む4つの32ビット80387レジスタです。最初の2つのレジスタFIP（FPU Instruction Pointer）とFCS（FPU Code Selector）は、最後に実行された命令の2つのオペコードへのポインタです（プレフィックス・コードは無視します）。FCSはセグメントセレクタとオペコード、FIPはセグメント内オフセットです。最後の2つのレジスタ、FOO（FPU Operand Offset）とFOS（FPU Operand Selector）は、命令メモリのオペランドへの最後のポインタです。FOSにはセグメントセレクタ、FOOにはセグメント内オフセット値が入ります。最後に実行されたコプロセッサ命令にメモリ・オペランドが含まれていない場合、最後の2つのレジスタ値は意味がありません。FLDENV、FSTENV、FNSTENV、FRSTOR、FSAVE、FNSAVEの各命令は、これら4つのレジスタの内容をロードまたはストアするために使用されます。最初の3つの命令は、コントロールワード、ステータスワード、フィーチャーワード、および4つのエラーポインタレジスタの合計28バイトのコンテンツをロードまたはストアします。コントロールワード、ステータスワード、フィーチャーワードはすべて32ビットで動作し、上位16ビットは0です。最後の3つの命令は、コプロセッサの全108バイトのレジスタの内容をロードまたはストアするために使用されます。

7. 4. Floating-point instruction format

コプロセッサのシミュレーションは、特定の浮動小数点命令のオペコードとオペランドを解析し、各命令の構造に従って、80386の通常の命令を使用して、対応するエミュレーション操作を行います。演算コプロセッサ80387には70以上の命令があり、表11-3のように5つのカテゴリに分かれています。

各命令の演算コードは2バイトで、1バイト目の上位5ビットは2進法の11011です。この5ビットの値(0x1b、10進27)は、ちょうどESC(エスケープ)という文字のASCIIコード値であるため、数学コプロセッサの命令はすべて視覚的にESCエスケープ命令と呼ばれている。なお、浮動小数点命令をシミュレートする際には、下位11ビットの値が決まれば、同じESCビットを無視することができます。

表11-3浮動小数点演算命令タイプ	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):

a) OP (Operation opcode) is the instruction opcode. In some instructions, it is divided into two parts: OPA and OPB.

b) 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.

c) P(Pop) indicates whether to perform a popup process after the operation: 0 - no need; 1 - pop up the stack after the operation.

d) d (destination) indicates the accumulator that saves the result of the operation: 0 - ST(0); 1 - ST(i).

e) MOD (Mode) and R/M (Register/Memory) are the operation mode field and the operand position field.

f) SIB (Scale Index Base) and DISP (Displacement) are optional follow-up fields with MOD and R/M field instructions.

a) また、浮動小数点演算命令のアセンブリ言語のニーモニックは、すべてFで始まります。FADD」「FLD」などです。また、以下のような標準的な表現もあります。

b) FI All instructions for operating integer data start with FI, such as FIADD, FILD, etc.

c) FB All instructions for operating BCD type data start with FB, such as FBLD, FBST, etc.

d) FxxP All instructions that perform a pop operation are terminated with the letter P, such as FSTP, FADDP, etc.

e) FxxPP All instructions that perform two pop operations are terminated with the letter PP, such as FCOMPP, FUCOMPP, etc.;

f) 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

math_emulate.cプログラムに含まれるすべての関数は、3つのカテゴリーに分けることができます。第1のクラスは、「デバイスが存在しない」という例外ハンドラのインタフェース関数math_emulate()であり、このクラスにはこのような関数が1つしかありません。第2のタイプは、浮動小数点命令エミュレーション処理のメイン関数do_emu()であり、このクラスにはこのような関数が1つしかありません。さらに、すべての関数は、他のCプログラムの関数を含め、シミュレーション操作の補助関数です。

80387コプロセッサ・チップを搭載していないPCでは、カーネルの初期化時にCR0にエミュレーション・フラグEM=1が設定されていると、CPUが浮動小数点演算に遭遇したときに例外int7が発生します。

という命令があり、このプログラムの476行目のmath_emulate(long false)関数が例外処理中に呼び出されます。

math_emulate()関数では、現在の処理でシミュレーションのコプロセッサ動作を使用していないと判断した場合、シミュレーションの80387コントロールワード、ステータスワード、フィーチャーワード（タグワード）を初期化し、コントロールワードのコプロセッサ例外マスクビット6個をすべてセットし、ステータスワードとタグワードをリセットしてから、シミュレーション処理メイン関数do_emu()を呼び出します。呼び出し時に使用するパラメータは、以下のように割り込み処理中のmath_emulate()関数を呼び出すリターンアドレスポインタです。info構造体は、実際には、CPUが割り込みint7を発生させてから徐々にスタックに押し込まれたスタック内のいくつかのデータの構造体ですので、システムコール実行時のカーネルスタック内のデータの分布と基本的には同じです。include/linux/math_emu.hファイルの11行目と、kernel/sys_call.sの冒頭を参照してください。

```
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.

また、特定の浮動小数点命令をシミュレートする際に、浮動小数点命令が無効であることが判明した場合、プログラムは直ちに放棄実行関数math_abort()を呼び出します。この関数は、現在のプロセスに指定された信号を送信し、スタックポインタespが割込みプロセスのmath_emulate()関数のリターンアドレス(math_ret)を指すように修正し、直ちに割込みハンドラに戻ります。

11.2.2 Code annotation

プログラム 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
```

// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、および
// signal manipulation function prototypes.
// <linux/math_emu.h> コプロセッサのエミュレーションのためのヘッダーファイルです。コプロセッサのデータ構造と
// a floating point representation structure are defined.
// <linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。
// commonly used functions of the kernel.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
// for segment register operations.

33 #include <signal.h> (英語)

34
35 #define　アラインドテンポリアル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 pointr to the simulated ST(x).
43
44 /*
45 * *We don't want these inlined - it gets too messy in the machine-code.*
46 */
// 同名の浮動小数点命令に対するシミュレーション関数を以下に示します。
49 47 static void fpop(void); 48
static void fpush(void);
50 static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b);
51 static temp_real_unaligned * __st(int i);
51
// 浮動小数点命令のシミュレーションを行う。
// まず、シミュレートされたI387にマスクされていない例外フラグが設定されているかどうかを確認します。
//構造体のステータス・ワード・レジスタを確認し、そうであれば、ステータス・ワードにビジー・フラグBを設定する。その後
// 命令ポインタを保存し、2バイトの浮動小数点命令コードを取り出すコード
ポインタEIPで//を入力し、そのコードを解析して意味に応じた処理を行うことができます。については
// コードの種類によって、Linusはシミュレーション処理にいくつかの異なるスイッチブロックを使用しています。
// 引数は、info構造体へのポインタです。
54 52 static void do_emu(struct info * info) 53 {...
55 unsigned short code;
56 temp_real tmp;
57 char * address;
57
// この関数は，まず，ステータスワードレジスタに例外フラグが設定されているかどうかをチェックし，もし
// そうすると，ステータスワードのビジーフラグB（ビット15）がセットされ，そうでなければBフラグがリセットされます。その後、我々は
// 元の命令ポインタを保存します。次に、この関数を実行するコードが、次のような場合を考えてみましょう。
// はユーザーコードです。そうでない場合（つまり、呼び出し元のコードセグメントセレクタが0x0fになっていない場合）、次のようになります。
// カーネルには、浮動小数点命令を使用するコードがありますが、これは許可されていません。そのため
// カーネルは、CS、EIP、「Needs Mathematical Simulation in
浮動小数点演算命令の「// the Kernel」情報。

```
58          if (I387.cwd & I387.swd & 0x3f)
59                  I387.swd |= 0x8000;                     // sets the busy flag B.
60          else
61                  I387.swd &= 0x7fff;                     // clear the flag B.
62          ORIG_EIP = EIP;                                 // save the original EIP.
63  /* 0x0007 means user code space */
64          if (CS != 0x000F) {                             // if not in user code space, STOP.
65                  printk("math_emulate: %04x:%08x\n\r",CS,EIP);
66                  panic("Math emulation needed in kernel");
67          }
```

// そして、ポインタEIPに2バイトの浮動小数点命令コードを取得します。なお、インテルの

// CPUは "Little endien "スタイルでデータを格納しているため、この時に取得されるコードは全く逆になります。

命令の1バイト目と2バイト目からの//。の順序を入れ替える必要があります。

// のバイトを「コード」にします。そして、最初のコードバイト（バイナリ11011）のESCビットをマスクアウトします。

```
// 次に、浮動小数点命令ポインタEIPがTSSセグメントのfipフィールドに保存されます。
// i387構造体で、CSをfcsフィールドに保存し、わずかに処理されたフローティング
fcsフィールドの上位16ビットには//point instruction codeが入ります。これらの値は
// プログラムを保存しておくことで、擬似的に本物のコプロセッサのように処理することができます。
// プロセッサの例外です。最後に，EIPが後続の浮動小数点演算命令を指すようにする
// またはオペランド。
        code = get_fs_word((unsigned short *) EIP);  // get 2 bytes floating-point code.
        bswapw(code);                                // exchange bytes.
        code &= 0x7ff;                               // Mask the ESC part in the code.
        I387.fip = EIP;                              // save EIP, code selector and code.
        *(unsigned short *) &I387.fcs = CS;
        *(1+(unsigned short *) &I387.fcs) = code;
        EIP += 2;                                    // point to next instruction code.

// そして、そのコードを解析し、意味に応じた処理を行います。異なるコードタイプの場合
//の値は、Linusは処理のためにいくつかの異なるスイッチブロックを使用します。
// (1) まず、命令のオペコードが固定のコード値を持つ場合（レジスタに依存しない
// のように）、以下のように処理されます。
// マクロmath_abort()は、コプロセッサのエミュレーション操作を終了させるために使用され、定義されています。
// ファイルlinux/math_emu.hの52行目にあります。実際の実装コードは、linux/math_emu.hの488行目にあります。
// のプログラムです。linux/math_emu.hファイルの50行目の前の記述を参照してください。
        switch (code) {
                case 0x1d0: /* fnop */    // like nop.
                        return;
                case 0x1d1: case 0x1d2: case 0x1d3:  // invalid code, send signal and exit.
                case 0x1d4: case 0x1d5: case 0x1d6: case 0x1d7:
                        math_abort(info,1<<(SIGILL-1));
                case 0x1e0:                  // FCHS - Change the ST sign bit: ST = -ST.
                ST(0).exponent ^= 0x8000;
                        return;
                case 0x1e1:                  // FABS - get absolute value. ST = |ST|.
                        ST(0).exponent &= 0x7fff;
                        return;
                case 0x1e2: case 0x1e3:   // invalid code. send signal and exit.
                        math_abort(info,1<<(SIGILL-1));
                case 0x1e4:                  // FTST - Test TS and set Cn in status word.
                        ftst(PST(0));
                        return;
                case 0x1e5:                  // FXAM - Check TS and modify Cn in status word.
                        printk("fxam not implemented\n\r");
                        math_abort(info,1<<(SIGILL-1));
                case 0x1e6: case 0x1e7:   // invalid code. send signal and exit.
                        math_abort(info,1<<(SIGILL-1));
                case 0x1e8:                  // FLD1 - Load constant 1.0 to accumulator ST.
                        fpush();
                        ST(0) = CONST1;
                        return;
                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                     printk("%04x fxxx not implemented\n\r",code + 0xc800);
132                     math_abort(info,1<<(SIGILL-1));
133             case 0x2e9:             // FUCOMPP - no order comparison.
134                     fucom(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) 次に、2バイト目の最後の3ビットがREGであるという命令を処理します。つまり
// 11011,XXXXXXX,REG "という形式のコードで、すでに "11011 "というプレフィックスがクリアされ

ています。

```
// 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:                      // FST 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 notimplemented\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) 次に、2バイト目の7--6がMOD、ビット2--0が--という形でコードを処理します。

// R/M、つまり「11011, XXX, MOD, XXX, R/M」となります。MODは各サブルーチンで処理されます。

// なので、まずコードをAND 0xe7（つまり0b11100111）にして、MODをマスクアウトさせます。

```
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);
```

```
                        fpop();
                        return;
        }
//(4) 2種類目の浮動小数点命令の処理は以下の通り。まず、数字の
インストラクションコードのビット10〜9のMFに応じて、指定されたタイプの//が取られます。
//となり、OPAとOPBの合計値に応じて別々に処理されます。その
//      switch (code >> 9) {
//つまり、「11011、MF、000、XXX、R/M」という形式の命令コードが処理されるのです。

                case 0:                 // MF = 00, short real (32-bit real).
                        get_short_real(&tmp,info,code);
                        break;
                case 1:                 // MF = 01, short integer (32-bit integer).
                        get_long_int(&tmp,info,code);
                        break;
                case 2:                 // MF = 10, long real (64-bit real).
                        get_long_real(&tmp,info,code);
                        break;
                case 4:         // MF = 11, a long integer (64-bit). should be 'case 3'!
                        get_short_int(&tmp,info,code);
```

```
431         }
//(5)浮動小数点演算命令の2バイト目のOPBコードを処理します。
            switch ((code>>3) & 0x27) {
432
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         }
    // 「11011, XX, 1, XX, 000, R/M」の形式の命令コードを処理します。
467         if ((code & 0x138) == 0x100) {              // FLD, FILD
468                         fpush();
469                         real_to_real(&tmp,&ST(0));
470                         return;
471         }
    // 残りは無効な命令です。
472         printk("Unknown math-insns: %04x:%08x %04x\n\r",CS,EIP,code);
473         math_abort(info,1<<(SIGFPE-1));
474 }
475
    // 例外的に呼ばれた8087エミュレーションインターフェイス関数 インタラプト int7.
    // 現在のプロセスがコプロセッサを使用していない場合は、コプロセッサの使用フラグを設定する
    // used_mathを入力し、80387のコントロールワード、ステータスワード、タグワードを初期化します。
    最後に
    // int 7 がこの関数を呼び出したときの戻り値を，パラメータとして使用して，フローティング
    // ポイント命令エミュレーション関数 do_emu(). パラメータfalseには_orig_eipを指定します。
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
    // シミュレーションを終了します。
    // 無効な命令コードや未実装の命令が処理された場合、関数
    // まず、プログラムの元のEIPを復元し、指定された信号を現在の
    // 処理を行います。最後に，int7がこれを起動したときの戻り値をスタックポインタで指定します.
    // 関数を使用して、直接割り込みに戻ります。この関数は、マクロの
    // math_abort(). 詳しい説明は、linux/math_emu.hファイルの50行目をご覧ください。
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
    // アキュムレータ・スタック プッシュ操作。
    // ステータスワードのTOPフィールドを1デクリメント（つまり7を加算）し、モジュロ7とする。
505 static void fpush(void)
506 {...
507         unsigned long tmp;
508
509         tmp = I387.swd & 0xffffc7ff;
510         I387.swd += 0x00003800;
511         I387.swd &= 0x00003800;
512         I387.swd |= tmp;
513 }
514
    // 2つのアキュムレータ・レジスタの値を入れ替えます。
515 static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b) 516 {。
517         temp_real_unaligned c;
518
519         c = *a;
520         *a = *b;
521         *b = c;
522 }
523
    // I387構造体のST(i)メモリポインタを取得する。
    // ステータスワードのTOPフィールド値を取り、指定された物理データレジスタ番号を加える
    // とmodulo 7を組み合わせて、最後にST(i)に対応するポインタを返すようにしました。
524 static temp_real_unaligned *       st(int i)
525 {。
526         i += I387.swd >> 11;            // Get the TOP field in the status word.
527         i &= 7;
528         return (temp_real_unaligned *) (i*10 + (char *)(I387.st_space));
529 }
530
```

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

プログラム 11-2 linux/kernel/math/error.c

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

```
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6
    // <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、および
    //     signal manipulation function prototypes.
    // <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_structや
    //     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
    // int16で呼ばれるコプロセッサのエラー処理関数。
    // 次のコードは、コプロセッサから送られてきたエラーを処理するためのものです。これにより、
    80387が
    // すべての例外フラグとステータスワードのビジービットをクリアします。最後のタスクがコプロセッ
    サを使用していた場合。
    // そして、コプロセッサのエラー信号フラグを設定します。戻り値の後、この関数は次のようにジャン
    プします。
    // ret_from_sys_callのシステムコール割り込み戻り場所で、実行を継続します。
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.

図11-7 一般命令の符号化形式

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

命令の 各命令には、少なくとも1バイトのオペコードが必要です。必要に応じて、命令のオペコードフィールドは、オペランドの種類と数を明示的に示すために使用されるMODRMオペランドインジケーターに従うかどうかを示します。メモリオペランドの場合、アドレスディスプレイスメントフィールドは、オペランドのオフセットを与えるために使用されます。MODRMフィールドのMODサブフィールドは、その命令にアドレス・オフセット・フィールドが含まれているかどうかと、その長さを示します。即値定数フィールドは、命令のオペコードで要求されるオペランドを与えますが、これは命令の中で与えられる最も単純なオペランドです。即値オペランド、レジスタオペランド、メモリオペランドのエンコーディングの詳細については、インテルのマニュアルを参照してください。

すべての命令のエンコード形式を図11-8に模式的に示す。同図では、10種類の符号化方式の命令フォーマットを示している。ここで、記号OPCodeまたはOPCはオペレーションコード、REGはレジスタフィールド、R/Mフィールドはオペランドとしてのレジスタを示すために使用されるか、またはアドレッシングモードを指定するためにMODフィールドと組み合わせて使用される。MODR/Mエンコーディングの中には、アドレッシングモードを示すために、2番目のアドレッシングバイト（SIBバイトと呼びます）を必要とするものがあります。このバイトには3つのサブフィールドの内容があります。(1)Scale - スケール（S）フィールドは、スケールファクターを指定します。(2)index - インデックス（The Idx）フィールドは、インデックスレジスタを指定します。

MOD	Disp Offset Size
00	Normally none. But if R/M or Base=101, then DS:Disp32
01	8 bits
10	32 bits
11	None

R/M, Base	Description
000	DS:Disp[EAX]
001	DS:Disp[ECX]
010	DS:Disp[EDX]
011	DS:Disp[EBX]
100	SS:Disp[ESP]
101	SS:Disp[EBP]
110	DS:Disp[ESI]
111	DS:Disp[EDI]

Idx	*
000	[EAX*S]
001	[ECX*S]
010	[EDX*S]
011	[EBX*S]
100	None
101	[EBP*S]
110	[ESI*S]
111	[EDI*S]

11	REG	REG

図11-8 インストラクション・エンコーディングとアドレッシング・フォーマットの概要

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

11.4.2 MOD=0b11の場合は、オフセットフィールドのない1バイト命令を意味します。R/Mフィールド=0b100でMOD=0b11以外の場合は、2バイトアドレスモードのアドレッシングを意味します。このとき、第2オペランド命令バイトSIB（Scale, Index, Base）を処理する関数sib()が呼び出され、オフセット値を取得して戻ります。R/Mフィールドが0b101、MODが0の場合は、シングルバイトのアドレスモードのエンコードに続いて32バイトのオフセット値が得られることを示します。それ以外の場合は、MODに従って処理されます。

11.4.3 Code annotation

プログラム 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
  // <stddef.h> 標準定義のヘッダファイルです。NULL, offsetof(TYPE, MEMBER)が定義されています。
  // <linux/math_emu.h> コプロセッサのエミュレーションのためのヘッダーファイルです。コプロセッサのデータ構造と
  //     a floating point representation structure are defined.
  // <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
  //     for segment register operations.
11 #include <stddef.h>.
12
13 #include <linux/math_emu.h>
14 #include <asm/segment.h>
15
  // 情報構造体における各レジスタのオフセット位置。 offsetof()を使用することで
16 // 構造体の中の指定されたフィールドのオフセット位置。include/stddef.hファイルを参照してください。
17 static int __regoffset[] = {
18         offsetof(struct info,_ eax),
19         offsetof(struct info,_ ecx),
20         offsetof(struct info,_ edx),
21         offsetof(struct info,_ ebx),
22         offsetof(struct info,_ esp),
23         offsetof(struct info,_ ebp),
24         offsetof(struct info,_ esi),
25         offsetof(struct info, edi)_
25 };
26
  // info構造体の指定された位置にあるレジスタの内容を取得します。
27 #define REG(x) (*(long        regoffset[(x)]+(char *) info))
*))( 28
  // 第2オペランド指示バイトSIB(Scale, Index, Base)の値を取得します。
31        29 static char * sib(struct info * info, int mod) 30 {。
```

```
32          unsigned char ss,index,base;
33          long offset = 0;
33
```

// SIBバイトはまずユーザーコードセグメントから取得し、次にフィールドビット値を取得します。

```
34          base = get_fs_byte((char *) EIP);
35          EIP++;
36          ss = base >> 6;                     // scale size.
37          index = (base >> 3) & 7;
38          base &= 7;
    // インデックスコードが0b100の場合は、インデックスオフセット値がないことを意味します。それ以外の場合は、インデックスオフセット
39  // 値のオフセット = レジスタの内容 * スケールファクター。
40          if (index == 4)
41                  offset = 0;
42          else
43                  offset = REG(index);
44          offset <<= ss;
    // 前のMODRMバイトのMODが0でない場合、またはBaseが0b101に等しくない場合、それは
    // baseで指定されたレジスタにオフセット値があること。したがって、オフセットは
    // ベースとなる対応するレジスタの内容に加算されます。MOD=1の場合，オフセット値は
45  // は1バイト（8ビット）です。それ以外の場合、MOD=2、またはbase=0b101の場合、オフセット値は4バイトになります。
46          if (mod || base != 5)
47          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          }
    // 最後にオフセット値を保存して返します。
53          I387.foo = offset;
54          I387.fos = 0x17;
55          return (char *) offset;
56  }
57
    // 命令コードのアドレッシングモードバイトを基に、実効アドレスを算出する。
60  58 char * ea(struct info * info, unsigned short code) 59 {...
61          unsigned char mod,rm;
62          long * tmp = &EAX;
63          int offset = 0;
63
    // まず、命令コードのMODフィールドとR/Mフィールドの値を取る。MOD=0b11であれば
    //は、オフセットフィールドのない1バイト命令を意味します。R/Mフィールド=0b100でMODが
    // 0b11ではなく、2バイトアドレスモードのアドレッシングを意味するので、sib()を呼び出してオフセット値を見つける
    // と返します。
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);
    // R/Mフィールドが0b101でMODが0の場合、シングルバイトのアドレスモードのエンコードを示す
```

//に続いて32バイトのオフセット値を入力します。その後、ユーザーコードの4バイトのオフセット値を取り、保存します。
//返します。

```
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          }
```

// 残りのケースについては、MODに従って処理されます。まず、コンテンツの値
R/Mコードの対応するレジスタの//がポインタtmpとして取り出されます。MOD=0の場合。
// オフセット値はありません。MOD=1の場合は、コードの後に1バイトのオフセット値が入ります。
MOD=2の場合。

```
            tmp = & REG(rm);

75
76          switch (mod) {
77                  case 0: offset = 0; break;
78                  case 1:
                    xp(15)
79                          offset = (signed char) get_fs_byte((char *) EIP);
80                          EIP++;
81                          break;
82                  case 2:
```

```
                        offset = (signed) get_fs_long((unsigned long *) EIP);
                        EIP += 4;
                        break;
                case 3:
                        math_abort(info,1<<(SIGILL-1));
        }
        I387.foo = offset;
        I387.fos = 0x17;
        return offset + (char *) *tmp;
}

```

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.

図11-9 ショートリアルからテンポラリーリアル形式への変換図

11.5.2 Code annotation

プログラム 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> ←ここをクリックしてください。
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
   // 短い実数を一時的な実数に変換します。
   // 短い実数は32ビット、その有効数字（仮数）は23ビットの長さです。
   // 指数は8ビットで、符号ビットが1つあります。
21 void short_to_temp(const short_real * a, temp_real * b) 22 {。
   // まず、短い実数が0である場合を扱います。
   // 一時的な実数bは0に設定され、一時的な実数の符号ビットは次のようになります。
23 // 短い実数の符号ビット、つまり指数の最上位ビットに応じて設定されます。
24         if (!(*a & 0x7fffffff)) {
25                 b->a = b->b = 0;                    // set significand of the temp real = 0.
26                 if (*a)
27                         b->exponent = 0x8000;       // set sign bit.
28                 else
29                         b->exponent = 0;
30                 return;
30         }
   // 一般的な短実では、まず、一時的なものに対応する指数値を決定します。
   //実数です。ここでは、整数値に偏った表現の概念を使う必要があります
   // 11.1 節参照）を使用しています。短い実数指数のバイアス番号は127、バイアスの
   一時的な実数の指数の//は16383です。したがって，指数の値を抽出した後に
   // ショートリアルの場合、バイアス値を16383に変更する必要があります。これにより、指数
   // の値を一時的な実数形式で表示します。また、短い実数が負の値の場合は
   // で一時的な実数の符号ビット（ビット79）を設定します。次に、仮数を設定します。その方法は
   // で短い実数を8ビット左にシフトし、最上位の23桁を
   仮の実数のビット62に仮の実数の//が入ります。仮実数のビット63
```

```
// 仮数を1にする必要があるので、OR 0x80000000の演算が必要です。最後に、ロー
// テンポラリー・リアルの32ビット有効数字がクリアされます。
```

```
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
```

// long realをtemporary realに変換します。

// このメソッドは、short_to_temp()とまったく同じですが、long realは64ビット長です。

// その有効数字（仮数）は52ビット、指数は11ビット、そして

// 1の符号ビット。また、ロングリアル指数の偏り値は1023です。

```
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
```

// 一時的な実数を短い実数に変換します。

// この手順は、short_to_temp()の逆ですが、精度の高い処理を必要とします。

// 丸めの問題。

```
55 void temp_to_short(const temp_real * a, short_real * b) 56 {。
```

// 指数部が0の場合、短い実数は、-0または0に設定されます。

```
/       if (!(a->exponent & 0x7fff)) {
/
/符号ビットがない場合は、表11-2を参照して
```

ください。

```
57
58                  *b = (a->exponent)?0x80000000:0;
59                  return;
60          }
```

// まず、指数部分の処理、つまり指数の偏りの量を（16383）
の場合、//は短い実数の偏った量127に置き換えられ、符号ビットが設定されます。
// それは負の数です。

```
61          *b = ((((long) a->exponent)-16383+127) << 23) & 0x7f800000;
62          if (a->exponent < 0)                        // set sign if negative.
63                  *b |= 0x80000000;
```

// 続いて、一時的な実数からシニフカントの上位23ビットを取得し、以下の処理を行います。

```
/           *b |= (a->b >> 8) & 0x007fffff;             // get higher 23bits of the temp real.
```

//コントロールワードの丸め設定に応じて、丸め動作を行います。

```
64
65          switch (ROUNDING) {
66                  case ROUND_NEAREST:
67                          if ((a->b & 0xff) > 0x80)
68                                  ++*b;
69                          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
```

// 一時的な実数を長い実数に変換します。

// 長い実数は64ビットの長さで、その有効数字（仮数）は52ビットの長さ、指数は

//は11ビットで、符号ビットが1つあり、指数の偏り値は1023です。

```
81 void temp_to_long(const temp_real * a, long_real * b) 82 {。
83          if (!(a->exponent & 0x7fff))
{ 84                b->a = 0;
85                  b->b = (a->exponent)?0x80000000:0;
86                  return;
87          }
88          b->b = (((0x7fff & (long) a->exponent)-16383+1023) << 20) & 0x7ff00000;
89          if (a->exponent < 0)
90                  b->b |= 0x80000000;
91          b->b |= (a->b >> 11) & 0x000fffff;
92          b->a = a->b << 21;
93          b->a |= (a->a >> 11) & 0x001fffff;
94          switch (ROUNDING) {
95                  case ROUND_NEAREST:
96                          if ((a->a & 0x7ff) > 0x400)
97                                  _asm_("add1 $1,%0 ; adc1 $0,%1"
98                                          :"=r" (b->a), "=r" (b->b)
99                                          :"" (b->a), "1" (b->b));
100                         break;
101                 case ROUND_DOWN:
102                         if ((a->exponent & 0x8000) && (a->b & 0xff))
103                                 _asm_("add1 $1,%0 ; adc1 $0,%1"
104                                         :"=r" (b->a), "=r" (b->b)
105                                         :"" (b->a), "1" (b->b));
106                         break;
107                 case ROUND_UP:
108                         if (!(a->exponent & 0x8000) && (a->b & 0xff))
109                                 _asm_("add1 $1,%0 ; adc1 $0,%1"
110                                         :"=r" (b->a), "=r" (b->b)
111                                         :"" (b->a), "1" (b->b));
112                         break;
113         }
114 }
115
```

// 一時的な実数を一時的な整数形式に変換します。

// 一時的な整数も10バイトで表されます。下位8バイトは符号なし整数

// の値を表し、上位2バイトは指数の値と符号ビットを表します。もし、最上位の

上位2バイトの//バイトが1の場合は負の数を示し、0の場合は

```c
// a positive number.
void real_to_int(const temp_real * a, temp_int * b)
{
        int shift = 16383 + 63 - (a->exponent & 0x7fff);
        unsigned long underflow;

        b->a = b->b = underflow = 0;
        b->sign = (a->exponent < 0);
        if (shift < 0) {
                set_OE();
                return;
        }
        if (shift < 32) {
                b->b = a->b; b->a = a->a;
        } else if (shift < 64) {
                b->a = a->b; underflow = a->a;
                shift -= 32;
        } else if (shift < 96) {
                underflow = a->b;
                shift -= 64;
        } else
                return;
        __asm_("shrd1 %2,%1,%0"                    // 32-bit shift right.
                :"=r" (underflow), "=r" (b->a)
                :"c" ((char) shift), "" (underflow), "1" (b->a));
        __asm_("shrd1 %2,%1,%0"
                :"=r" (b->a), "=r" (b->b)
                :"c" ((char) shift), "" (b->a), "1" (b->b));
        __asm_("shr1 %1,%0"
                :"=r" (b->b)
                :"c" ((char) shift), "" (b->b));
        switch (ROUNDING) {
                case ROUND_NEAREST:
                        __asm_("add1 %4,%5 ; adc1 $0,%0 ; adc1 $0,%1"
                                :"=r" (b->a), "=r" (b->b)
                                :"" (b->a), "1" (b->b)
                                ,"r" (0x7fffffff + (b->a & 1))
                                ,"m" (*&underflow));
                        break;
                case ROUND_UP:
                        if (!b->sign && underflow)
                                __asm_("add1 $1,%0 ; adc1 $0,%1"
                                        :"=r" (b->a), "=r" (b->b)
                                        :"" (b->a), "1" (b->b));
                        break;
                case ROUND_DOWN:
                        if (b->sign && underflow)
                                __asm_("add1 $1,%0 ; adc1 $0,%1"
                                        :"=r" (b->a), "=r" (b->b)
                                        :"" (b->a), "1" (b->b));
                        break;
        }
}
```

```
168
    // 一時的な整数を一時的な実数形式に変換します。
169 void int_to_real(const temp_int * a, temp_real * b) 170 {。
    // 元の値は整数なので、一時的に実数に変換する際に、指数が
    // は、偏った量である16383に加えて63を加えます。これにより，記号部には
            b->a = a->a;
171
172         b->b = a->b;
173         if (b->a || b->b)
174                 b->exponent = 16383 + 63 + (a->sign? 0x8000:0);
175         else {
176                 b->exponent = 0;
177                 return;
178         }
    // The normal real number after the conversion format is normalized, that is, the high significant
    // bit of the significand is not zero.
179         while (b->b >= 0) {
180                 b->exponent--;
181                 __asm_("addl %0,%0 ; adcl %1,%1"
182                         :"=r" (b->a), "=r" (b->b)
```

```
                                    : "" (b->a), "1" (b->b));
            }
}
```

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.

図11-10 一時的なリアルとシミュレーションのフォーマット間の変換

11.6.2 Code annotation

プログラム 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>を使用します。
21
   // 数値の負の数（2の補数）表現を取得します。
   // 演算内容は、一時的な実数の仮数（significand）を反転させて加算するものです。
   // 1を指定します。パラメータ'a'は，一時的な実在の構造体へのポインタである．の組み合わせは
   // そのフィールドaとbは、一時的な実数のシニフィカントです。
22 #define NEGINT(a) ♪ ♪ ♪
23 asm_("notl %0 ; notl %1 ; addl $1,%0 ; adcl $0,%1"\
24          :"=r" (a->a), "=r" (a->b) \
25          :"" (a->a), "1" (a->b))
26
   // 仮数を符号化しました。
   // つまり、一時的な実数を指数と整数の表現に変換します。
   // シミュレーションの操作を容易にするために そのため、ここではシミュレーション・フォーマットと
   呼んでいます。
   // 64ビットのバイナリの仮数を2ビット右に移動させる操作です（そのため、指数は
   // 2を加算する）。指数フィールドの最上位ビットは符号ビットなので、指数値が
   // が0より小さい場合、その数字は負の値になります。そこで、仮数を補数で表します。
   //（マイナスを取る）して、指数をプラスに取ることができます。このとき、仮数部の
   // には、2ビットシフトしたシニフィカントだけでなく、値のサインビットも含まれています。
27 static void signify(temp_real * a) 28 {。

   // 30 行目: %0 は a->a; %1 は a->b です。アセンブリ命令 "shldl $2, %1, %0" は次のことを実行します。
   // 倍精度（64ビット）の右シフトで，結合された仮数 <b, a> を
   // を2ビットだけ右に移動します。この移動は %1（a->b）の値を変更しないので、次のことも必要です。
29 // を2ビット右にシフトします。
```

```
30        a->exponent += 2;                        // increases exponent by 2.
31        __asm ("shrd1 $2,%1,%0 ; shr1 $2,%1"     // mantissa is shifted to right by 2 bits.
31        :"=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
```

// 仮数を無符号化します。
// エミュレーション形式を一時的な実数形式に変換します。つまり、実数で表される
// を指数で表し、整数を一時的な実数形式に変換します。

```
38 static void unsignify(temp_real * a) 39 {。
```

// 値が0の数値の場合は、処理せずにそのまま返します。それ以外の場合は、まずリセット
// 一時的なリアの符号ビットを判断して、上位32ビットのフィールドa->bの
仮数部の//には符号ビットがあります。もしそうなら、指数フィールドに符号ビットを追加し、また、表
// (補) 仮数を符号なしの数値にする。最後に、仮数部を正規化して
// 指数値はそれに応じてデクリメントされます。つまり，左シフトの操作を行います
// 仮数の最上位ビットが0にならないように(最後のa->bの値が見えるのは
// 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
```

// 浮動小数点の加算命令をシミュレートします。

// 一時的な実数パラメータ： src1 + src2 -> result.

```
59  57 void fadd(const temp_real * src1, const temp_real * src2, temp_real * result) 58 {...
60          temp_real a,b;
61          int x1,x2,shift;
61
```

// まず、2つの数字の指数値x1,x2を取ります（符号ビットを取り除いたもの）。

// 次に、変数aを最大値とし、変数'shift'を

```
62  // 指数的な差（つまり、差の2の倍数）を表す。
63          x1 = src1->exponent & 0x7fff;
64          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          }
    //両者の差が2^64以上と大きすぎる場合は、無視することができます。
    //小さい数字（つまりbの値）を返します。つまり、値aを直接返すことができます。そうでなければ、もし差が
    //が2^32以上の場合は、下位32ビットの値を無視して、小
    //そこで、bの上位ロングフィールド値b.bを32ビット右にシフト、つまり
    //それをb.a.にして、bの指数を32倍にします。つまり、指数の差は
    //を32で引いたものになります。この調整の後、追加された2つの数字のマンタは次のようになります。
73  //実質的に同じ範囲である。
74          if (shift >= 64) {
75                  *result = a;
76                  return;
76          }
77          if (shift >= 32) {
78                  b.a = b.b;
79                  b.b = 0;
80                  shift -= 32;
81          }
    //その後、微調整を行い、両者の指数が同じ値になるように調整します。そのためには
    //調整方法は、小数値bの仮数部を「シフト」ビット分だけ右に移動させます。
    //したがって、この2つの指数は同じで、同じオーダーになります。そこで、次のように追加します。
    //2つのマンチカンを 追加する前に、シミュレーションのフォーマットに変換して
            __asm_("shrd1 %4,%1,%0 ; shr1 %4,%1"     // Double precision (64 bit) right shift.
```

```
82
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_("add1 %4,%0 ; adc1 %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));
```

```
        unsignify(&a);                          // change back to temp real format.
        *result = a;
}

```

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

プログラム 11-6 linux/kernel/math/compare.c

```
/*
 * 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>を追加します。
12
   // ステータスワードのC3、C2、C1、C0の各条件ビットをリセットします。
13 #define clear_Cx() (I387.swd &= ~0x4500)
14
   // 一時的な実数は正規化されており、つまり指数と符号で表現されています。
   // 例えば、102.345は、1.02345 X 10^2と表されます。0.0001234は、1.234 X 10^-4と表されます。
   // もちろん、これらの数値は関数内では2進数で表現されます。
17  15 static void normalize(temp_real * a) 16 {
18          int i = a->exponent & 0x7fff;           // get the exponent (ignore sign bit).
19          int sign = a->exponent & 0x8000;        // get sign.
19
   // 一時的な実数aの64ビットの有効数字（仮数）が0であれば、aは次のようになります。
   //を0にするので、aの指数をクリアして戻ります。
20          if (!(a->a || a->b)) {
21                  a->exponent = 0;
22                  return;
23          }
   // a の仮数が左端にゼロ値のビットを持つ場合、仮数を左にシフトする。
   // と、指数値を調整（デクリメント）します。のMSB（最上位ビット）まで、b
   //フィールドは1（bが負の値として現れるとき）。最後に符号ビットを追加します。
24          while (i && a->b >= 0)
{ 25                i--;
26                  _asm_("add1 %0,%0 ; adc1 %1,%1"
27                          :"=r" (a->a), "=r" (a->b)
28                          :"" (a->a), "1" (a->b));
29          }
30          a->exponent = i | sign;
31 }
32
   // 浮動小数点命令をシミュレートする FTST（Floating-point Test）。
   // つまり、トップスタックのアキュムレータST(0)が0と比較され、その中の条件ビットが
   // ステータスワードは比較結果に応じて設定されます。ST > 0.0の場合，C3，C2，C0は
   // それぞれ 000; ST < 0.0 の場合、条件ビットは 001; ST == 0.0 の場合、条件ビットは
   // は100、そうでなければ条件ビットは111となります。
33 void ftst(const temp_real * a) 34 {。
35          temp_real b;
36
   // まず、ステータスワードのコンディションフラグがクリアされ、比較値b（ST）が
   // を正規化します。bが0ではなく，符号ビットがセットされている（負の数である）場合は
```

```
    // 条件ビットC0がセットされ、そうでなければ条件ビットC3がセットされます。
37          clear_Cx();
38          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
   // 浮動小数点演算命令FCOM（Floating-point Compare）をシミュレートします。
   // 2つのパラメータ src1, src2 を比較し、比較結果に応じてコンディションビットを設定する
   // 結果。src1 > src2 の場合、C3, C2, C0 はそれぞれ 000 となります。src1 < src2 の場合、条件は
   //ビットは001、2つが等しい場合は条件ビットは100となります。
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
   // 浮動小数点演算命令FUCOM（無順序比較）をシミュレートします。
   // この関数は、オペランドの1つがNaNである場合の比較演算に使用されます。
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

プログラム 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> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、および
   //     signal manipulation function prototypes.
   // <linux/math_emu.h> コプロセッサのエミュレーションのためのヘッダーファイルです。コプロセッサのデータ構造と
   //     a floating point representation structure are defined.
   // <linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。
   //     commonly used functions of the kernel.
   // <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
   //     for segment register operations.
12 #include <signal.h> (英語)
13
14 #include <linux/math_emu.h>
15 #include <linux/kernel.h>
16 #include <asm/segment.h>
17
   // ユーザーメモリ内のショートリアル（単精度実数）を取得します。
   // 浮動小数点演算命令コードのアドレッシングモードバイトの内容による。
   // と、情報構造体の現在のレジスタの内容、有効なアドレスの
   // （ファイルmath/ea.c参照）の短い実数が配置されているところを取得して
   // 対応する実数値がユーザーデータエリアから読み込まれます。最後に、ユーザーの短い実
   // 数値を一時的な実数に変換して（math/convert.c）返します。
   // Parameters: tmp - 一時的な実体へのポインタ; info - 情報構造体へのポインタ。
18 //コード - インストラクションコード
19 void get_short_real(temp_real * tmp,
20         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
   // ユーザーメモリのlong real（倍精度の実数）を取得します。
29 // この関数はget_short_real()と同じように処理されます。
30 void get_long_real(temp_real * tmp,
31         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

```
// ユーザーのメモリにある一時的な実数を取る。
// まず、浮動小数点演算命令のアドレッシングモードバイトの内容に応じて
// コードと情報構造体の現在のレジスタの内容から、有効な情報を得ることができます。
// 一時的な実数が置かれているアドレス（math/ea.c）を読み込んでから
```

```
// ユーザーデータ領域から対応する一時的な実数値を取得します。
// パラメータ：tmp - 一時的な実数へのポインタ，info - 情報構造体へのポインタ。
//コード - インストラクションコード
void get_temp_real(temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;

        addr = ea(info,code);                           // get effective address.
        tmp->a = get_fs_long((unsigned long *) addr);
        tmp->b = get_fs_long(1 + (unsigned long *) addr);
        tmp->exponent = get_fs_word(4 + (unsigned short *) addr);
}

// ユーザーのメモリにある短い整数を取得します。
// この関数はまず、内容に応じて短い整数の実効アドレスを取得します。
浮動小数点演算命令コードのアドレッシングモードバイトの//の内容と
// 情報構造体の現在のレジスタを読み、対応する整数値を
// ユーザーデータ領域に、一時的な整数形式で保存します。最後に、一時的な
// 整数値を一時的な実数に変換します。
// 一時的な整数も10バイトで表されます。下位8バイトは符号なし整数
// の値を表し、上位2バイトは指数値と符号ビットを表します。もし、最上位の
上位2バイトのうち、 // 有効なバイトが1の場合は負の数を意味し、 // 有効なバイトが2の場合は負の数
を意味します。
//ビットが0の場合、正の数を意味します。
// パラメータ：tmp - 一時的な実数へのポインタ，info - 情報構造体へのポインタ。
//コード - インストラクションコード
void get_short_int(temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        temp_int ti;

        addr = ea(info,code);               // get effective in the instruction code.
        ti.a = (signed short) get_fs_word((unsigned short *) addr);
        ti.b = 0;
        if (ti.sign = (ti.a < 0))           // set sign bit if it is a negative.
                ti.a = - ti.a;              // and unsignify the mantissa part.
        int_to_real(&ti,tmp);               // change to temp real.
}

// ユーザーのメモリにある長整数を取得し、一時的な実数形式に変換します。
// この関数は、上記のget_short_int()と同じように処理されます。
void get_long_int(temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        temp_int ti;

```

```
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
   // ユーザーメモリ内の64ビット長整数（拡張長整数）を取得します。
   // まず、浮動小数点演算命令のアドレッシングモードバイトの内容に応じて
   // コードと情報構造体の現在のレジスタの内容、実効アドレス
   64ビット長整数の // を取得し、対応する整数値を読み出す。
   // をユーザーデータ領域から削除し、一時的な整数として保存します。最後に、一時的な
   // 整数値を一時的な実数に変換します。
80 // パラメータ：tmp -一時的なリアルポインタ、info -情報ポインタ、code -命令コード。
81 void get longlong int(temp real * tmp,
82          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 ("not1 %0 ; not1 %1\n\t"       // and complement, carry adjustment.
91                          "add1 $1,%0 ; adc1 $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
   // 64ビットの整数（Nなど）を10倍します。
   // このマクロは、以下のBCDコード値を一時的な実数に変換する際に使用します。
   // 数字の形式です。その方法は、N<<1 + N<<3 です。
 97 #define MUL10(low,high) ＼(^o^)
 98 asm_("add1 %0,%0 ; adc1 %1,%1\n\t" \
99 "mov1 %0,%%ecx ; mov1 %1,%%ebx\n\t" \
100 "add1 %0,%0 ; adc1 %1,%1\n\t" \
101 "addl %0,%0 ; adcl %1,%1¥¥" ●
102 "addl %%ecx,%0 ; adcl %%ebx,%1"
● 103 :"=a"(Low),"=d"(High)
104 :「」（低），「1」（高）：「cx」，
「bx」） 105
   // 64ビットの加算。64ビットの<high,low>に32ビットの符号なし数値valを加算します。
106 #define ADD64(val,low,high) ●。
107 asm ("addl %4,%0 ; adcl $0,%1":"=r" (low),"=r" (high) ⋈⋈◍＞◡¨*)
108 : "低" (Low) , "1" (High) , "r" ((unsigned long) (val))
の順で表示されます。） 109
   // ユーザーのメモリにあるBCDコードの値を取り、一時的に実数フォーマットに変換する。
   // この関数は、まず、BCDコードの有効なアドレスを内容に応じて取得します。
   浮動小数点演算命令コードのアドレッシング・モード・バイトに含まれる//の内容と
   // 情報構造体に登録されているレジスタから、対応する10バイトのBCDコードを
   // ユーザーデータ領域（1バイトが符号に使用される）で、一時的に整数に変換される
```

```
// の形で表示されます。最後に、一時的な整数値が一時的な実数に変換されます。
// パラメータ：tmp - 一時的な実数へのポインタ，info - 情報構造体へのポインタ。
//コード - インストラクションコード
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
    // Get the memory effective address of the BCD code value, and then start processing from the
    // last BCD code byte (most significant bit). First obtain the sign bit of the BCD code value
    // and set the sign bit of the temporary integer. The 9-byte BCD code value is then converted
    // to a temporary integer format, and finally the temporary integer value is converted to a
    // temporary real number.
117         addr = ea(info,code);                       // get effective address.
118         addr += 9;                                  // point to the last (10th) byte.
119         i.sign = 0x80 & get_fs_byte(addr--);        // get sign bit.
120         i.a = i.b = 0;
121         for (k = 0; k < 9; k++) {                   // change to temporary integer.
122                 c = get_fs_byte(addr--);
123                 MUL10(i.a, i.b);
124                 ADD64((c>>4), i.a, i.b);
125                 MUL10(i.a, i.b);
126                 ADD64((c&0xf), i.a, i.b);
127         }
128         int_to_real(&i,tmp);                        // change to temporary real.
129 }
130
    // Save the result in the short (single precision) real format to the user data area.
    // この関数は，まず，命令コードの実効アドレス addr を取得し，次に
    // 一時実数の結果を短実数形式に変換して，その場所に格納する
    // of addr.
    // パラメータ：tmp - 一時的なリアルフォーマットの結果値，info - 情報構造体のポインタ。
131 //コード - インストラクションコード
132 void put_short_real(const temp_real * tmp,
133         struct info * info, unsigned short code)
133 {
134         char * addr;
135         short_real sr;
136
137         addr = ea(info,code);                       // get effective address.
138         verify_area(addr,4);                        // verify the data area.
139         temp_to_short(tmp,&sr);                     // convert to short real.
140         put_fs_long(sr,(unsigned long *) addr);     // store at location addr.
141 }
142
    // 結果をロング（倍精度）リアルフォーマットでユーザーデータ領域に保存します。
    void put_long_real(const temp_real * tmp,
```

```
143
144         struct info * info, unsigned short code)
145 {
146         char * addr;
147         long_real lr;
148
149         addr = ea(info,code);
150         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
    // 結果を一時的な実数形式でユーザーデータ領域に保存します。
    // この関数は，まず結果を保存するためのアドレス addr を取得し，それを確認した後に
    // そのアドレスに十分な（10バイトの）ユーザーメモリがある場合は、一時的なリアル
    // 番号を addr.
    // パラメータ：tmp - 一時的なリアルフォーマットの結果値，info - 情報構造体のポインタ。
156 //コード - インストラクションコード
157 void put_temp_real(const temp_real * tmp,
158         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
    // 結果を短い整数形式でユーザーデータ領域に保存します。
    // この関数は，まず結果を保存するためのアドレス addr を取得し，それを
    // 一時的な実数形式の結果を一時的な整数形式に変換します。もしそれが負の
    // 数、整数の符号ビットを設定します。最後に、整数をユーザーメモリに保存します。
    // パラメータ：tmp - 一時的なリアルフォーマットの結果値，info - 情報構造体のポインタ。
168 //コード - インストラクションコード
169 void put_short_int(const temp_real * tmp,
170         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
    // This function is handled in the same way as put_short_int() above.
    void put_long_int(const temp_real * tmp,
```

```c

        struct info * info, unsigned short code)
{
        char * addr;
        temp_int ti;

        addr = ea(info,code);                       // get effective address.
        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
    // 64ビット整数形式の結果をユーザーデータ領域に保存する。
    // この関数は，まず結果を保存するためのアドレス addr を取得し，それを
    // 一時的な実数形式の結果を一時的な整数形式に変換します。もしそれが負の
    // 数、整数の符号ビットを設定します。最後に、整数をユーザーメモリに保存します。
    // パラメータ：tmp - 一時的なリアルフォーマットの結果値，info - 情報構造体のポインタ。
196 //コード - インストラクションコード
197 void put_longlong_int(const temp_real * tmp,
198         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 %1\n\t"     // invert and add one.
207                         "addl $1,%0 ; adcl $0,%1"
208                         :"=r"(ti.a),"=r"(ti.b)
209                         :""(ti.a),"1"(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
    // 符号なしの数値<high,low>を10で割って、その余りをremに入れる。
214 #define DIV10(low,high,rem) ♪ ♪ ♪ ♪
215 __asm__("divl %6 ; xchgl %1,%2 ; divl %6" \
216         :"=d"(rem),"=a"(low),"=b"(high) \
217         :""(0),"1"(high),"2"(low),"c"(10))
218
    // 結果をBCDコード形式でユーザーデータ領域に保存します。
    // この関数は、まず結果を保存するためのアドレスaddrを取得し、ユーザ
    // 10バイトのBCDコードを格納するスペース。一時的な実数のフォーマットの結果は
    // を、BCDコード形式のデータに変換して、ユーザーメモリに保存します。負の場合は
    // メモリの最上位バイトの最上位ビットが設定されます。
    // パラメータ：tmp - 一時的なリアルフォーマットの結果値，info - 情報構造体のポインタ。
    //コード - インストラクションコード
221 219 void put_BCD(const temp_real * tmp,struct info * info, unsigned short code) 220 {
222         int k,rem;
223         char * addr;
224         temp_int i;
225         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

プログラム 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>を追加します。
12
   static void shift(int * c)
```

```
13
14 {
15         __asm__("mov1 (%0),%%eax ; add1 %%eax,(%0)\n\t"
16                 "mov1 4(%0),%%eax ; adc1 %%eax,4(%0)\n\t"
17                 "mov1 8(%0),%%eax ; adc1 %%eax,8(%0)\n\t"
18                 "mov1 12(%0),%%eax ; adc1 %%eax,12(%0)"
19                 ::"r"((long) c):"ax");
20 }
21
```

// 2つの一時的なリアルを掛け合わせ、その結果をcポインタ（16バイト）に配置する。

22 static void mul64(const temp_real * a, const temp_real * b, int * c) 23 {。

```
24          __asm__("mov1 (%0),%%eax\n\t"
25                  "mul1 (%1)\n\t"
26                  "mov1 %%eax,(%2)\n\t"
27                  "mov1 %%edx,4(%2)\n\t"
28                  "mov1 4(%0),%%eax\n\t"
29                  "mul1 4(%1)\n\t"
30                  "mov1 %%eax,8(%2)\n\t"
31                  "mov1 %%edx,12(%2)\n\t"
32                  "mov1 (%0),%%eax\n\t"
33                  "mul1 4(%1)\n\t"
34                  "add1 %%eax,4(%2)\n\t"
35                  "adc1 %%edx,8(%2)\n\t"
36                  "adc1 $0,12(%2)\n\t"
37                  "mov1 4(%0),%%eax\n\t"
38                  "mul1 (%1)\n\t"
39                  "add1 %%eax,4(%2)\n\t"
40                  "adc1 %%edx,8(%2)\n\t"
41                  "adc1 $0,12(%2)"
42                  ::"b"((long) a),"c"((long) b),"D"((long) c)
43                  :"ax","dx");
44 }
45
   // シミュレーション浮動小数点演算命令 FMUL（Floating-point Multiply）。
   // 仮設リアル src1 * scr2 -> 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
   // まず2つの数字の掛け算の符号を決める。符号が等しいのは
   // 符号ビット 両者のXOR。次に、掛けられた指数の値を計算します。指数値には
   // を乗算時に加算します。しかし、指数は偏った数値形式で格納されているので
   // 2つの数字の指数を足すと偏った量が2回加算されるので、それが
51 // 偏った数を引くために必要です（一時的な実数の偏った数は16383です）。
52          sign = (src1->exponent ^ src2->exponent) & 0x8000;
53          i = (src1->exponent & 0x7fff) + (src2->exponent & 0x7fff) - 16383 + 1;

   // 結果の指数が負になれば、2つの数値が掛けられたことになります。
   // でアンダーフローが発生します。そこで、符号付きのゼロの値を直接返します。
   // 結果の指数が0x7fffよりも大きい場合は、オーバーフローが発生したことを示すので
   // ステータスワードのオーバーフロー例外フラグを設定して返します。
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          }
   // 2つの数値の仮数を掛け合わせ、その結果が0でない場合、結果の仮数は
```

```
// を正規化します。つまり，結果として得られる仮数の値は左にシフトされ，最大の
// の有効ビットが1となり、それに応じて指数が調整されます。もし，16バイトの
```

```
// 2つの数字の仮数を掛け合わせて得られる結果は0で、指数も
// 最後に、掛け算の結果は一時的な実数「result」に保存されます。
        mul64(src1,src2,tmp);
        if (tmp[0] || tmp[1] || tmp[2] || tmp[3])
                while (i && tmp[3] >= 0) {
                        i--;
                        shift(tmp);
                }
        else
                i = 0;
        result->exponent = i | sign;
        result->a = tmp[2];
        result->b = tmp[3];
}

```

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

プログラム 11-9 linux/kernel/math/div.c

```
/*
 * linux/kernel/math/div.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * temporary real division routine.
 */

#include <linux/math_emu.h>を追加します。

// 4バイトの内容を1ビットだけ左にシフト（2倍）します。
static void shift_left(int * c) {。
        _asm volatile ("movl (%0),%%eax ; addl %%eax,(%0)\n\t"
                "movl 4(%0),%%eax ; adcl %%eax,4(%0)\n\t"
                "movl 8(%0),%%eax ; adcl %%eax,8(%0)\n\t"
                "movl 12(%0),%%eax ; adcl %%eax,12(%0)"
                ::"r"((long) c):"ax");
}

// ポインタcが指す4バイトの内容を1ビットだけ右にシフトする。
```

```
22 static void shift_right(int * c) 23 {。
24          _asm__("shrl $1,12(%0) ; rcrl $1,8(%0) ; rcrl $1,4(%0) ; rcrl $1,(%0)"
25                  ::"r"((long) c));
26 }
27
```

// 16バイトの減算関数。

// 最後に、借用があるかどうかによってokが設定されます。

// フラグ(CF=1)を設定します。借り入れがない場合（CF=0）はOK=1、そうでない場合はOK=0となります。

```
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 ; sbbl %%eax,4(%2)\n\t"
34                  "movl 8(%1),%%eax ; sbbl %%eax,8(%2)\n\t"
35                  "movl 12(%1),%%eax ; sbbl %%eax,12(%2)\n\t"
36                  "setae %%al":"=a"(ok):"c"((long) a),"d"((long) b));
37          return ok;
38 }
39
```

// 16バイトの分割機能。

// パラメータ a / b -> c. 減算を使ってマルチバイトの分割をシミュレートする方法です。

```
42  40 static void div64(int * a, int * b, int * c) 41 {。
43          int tmp[4];
44          int i;
45          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
```

// 浮動小数点命令をシミュレートする FDIV.

// 一時的な実数分割：src1 / src2 -> 結果。

```
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
```

```
        sign = (src1->exponent ^ src2->exponent) & 0x8000;
        if (!(src2->a || src2->b)) {
                set_ZE();
                return;
        }
        i = (src1->exponent & 0x7fff) - (src2->exponent & 0x7fff) + 16383;
        if (i<0) {
                set_UE();
                result->exponent = sign;
                result->a = result->b = 0;
                return;
        }
        a[0] = a[1] = 0;
        a[2] = src1->a;
        a[3] = src1->b;
        b[0] = b[1] = 0;
        b[2] = src2->a;
        b[3] = src2->b;
        while (b[3] >= 0) {
                i++;
                shift_left(b);
        }
        div64(a,b,tmp);
        if (tmp[0] || tmp[1] || tmp[2] || tmp[3])
{               while (i && tmp[3] >= 0) {
                        i--;
                        shift_left(tmp);
                }
                if (tmp[3] >= 0)
                        set_DE();
        } else
                i = 0;
        if (i>0x7fff) {
                set_OE();
                return;
        }
        if (tmp[0] || tmp[1])
                set_PE();
        result->exponent = i | sign;
        result->a = tmp[2];
        result->b = tmp[3];
}

```

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 mathod of the math coprocessor, and

math_emulate.cプログラムをメインラインとして、各エミュレーションコードプログラムの動作実装を説明します。

次の章では、カーネル0.12で使用されているMINIXファイルシステムの包括的な紹介を行います。MINIXファイルシステムの構造と様々なファイルタイプについて簡単に説明した後、ファイルシステムにアクセスするために使用されるキャッシュがどのように機能するかを詳しく説明します。また、基礎となる各ファイルや機能の役割、システムコールで提供されるファイルやディレクトリの管理機能など、各種ファイルのデータにアクセスするための主な機能についても簡単に説明します。その後、コードの詳細なコメントを始める前に、ブロックデバイス上のシンプルなファイルシステムの例が具体的に提示され、説明されました。

12 File System (fs)

本章では、Linuxカーネルにおけるファイルシステムの実装コードと、ブロックデバイス用のキャッシュマネージャについて説明します。リーナス氏は、Linux 0.12カーネルのファイルシステムを開発する際に、主に当時のMINIX OSを参考にし、MINIXファイルシステムの1.0バージョンを使用しました。したがって、この章の内容を読むときは、まずMINIXファイルシステムについて読んでください。また、キャッシュの動作原理や実装方法の紹介については、まずM.J.Bach氏の著書「Design of UNIX Operating System」を閲覧することができます。

リスト12-1に示すように、ファイルシステムの実装には18のソースファイルがあります。これらのファイルとその中の機能の相互参照は、5.10節に記載されています。

リスト 12-1 linux/fs

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

システムの動作もファイルシステムに動的に保存される必要があります。そのため、ファイルシステムではプログラムやデータを保存するために高速なデバイスを使用する必要があり、OSでは通常、ファイルシステムのデバイスとして大容量の情報を保存できるブロックデバイスを使用している。また、UNIX系OSでは通常、デバイスファイルを介してデバイスにアクセスするため、ファイルシステムの構成や実装が非常に複雑になります。

本章で述べた手順は大規模なものですが、5.10節のLinuxソースコードのディレクトリ構造の分析（図5-29参照）により、これらのファイルを4つのパートに分けて議論することができます。第1部は、高速バッファ(キャッシュ)管理プログラムに関するもので、主にハードディスクなどのブロックデバイスへの高速データアクセスの機能を実装しています。この部分の内容はbuffer.cプログラムに集約されています。第2部では、ファイルシステムの低レベルな汎用機能について説明しています。ファイル・インデックス・ノードの管理、ディスク・データ・ブロックの割り当てと解放、ファイル名とiノードの変換アルゴリズムなどが記述されている。プログラムの第3部は、ファイル内のデータの読み書きに関するもので、キャラクタ・デバイス、パイプ、ブロック・リード・ライト・ファイル内のデータへのアクセスを含む。プログラムの第4部は、主にファイルのシステム・コール・インターフェースの実装で、主にファイルのオープン、クローズ、作成、および関連するファイル・ディレクトリ操作のシステム・コールに関連するものである。

12.1.1 まず、MINIXファイルシステムの基本的な構造を紹介し、その後、4つのパートを個別に説明します。

12.1.2 MINIX file system

現在、MINIXのOSのバージョンは2.0、使用しているファイルシステムはバージョン2.0です。バージョン1.5以前のバージョンとは異なり、容量も拡張されています。しかし、本書で注釈を付けたLinuxカーネルはMINIXファイルシステムのバージョン1.0を使用しているため、ここではバージョン1.0のファイルシステムのみを簡単に紹介する。

MINIXのファイルシステムは、基本的には標準的なUNIXのファイルシステムと同じです。6つの部分で構成されています。(1) ブートブロック、(2) スーパーブロック、(3) iノードビットマップ、(4) ロジックブロックビットマップ、(5) iノード 、(6) データブロック。一般的なディスク・ブロック・デバイスの場合、そのパーツの分布は図12-1のようになる。

図12-1 MINIXファイルシステムの各部のレイアウトの模式図

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.

ブートブロックは、コンピュータの電源を入れたときにROM BIOSが自動的に読み取ることができる実行コードとデータのディスクブロックです。ただし、システム内のすべてのディスク装置がブート装置として使用されるわけではないので、ブートに使用されないディスクについては、このディスクブロックにコードが含まれていない場合もある。ただし、どのディスクブロックのデバイスも

は、MINIXファイルシステムのフォーマットの統一性を保つために、ブートブロックスペースを持っています。つまり、ファイルシステムは単にブロックデバイス上にブートブロックを格納するためのスペースを残しているのです。カーネルイメージファイルをファイルシステムに置いた場合、実際のブートローダをファイルシステムが存在するデバイスの最初のブロック（つまりブートブロックスペース）に格納し、ファイルシステム内のカーネルイメージファイルを取得して読み込ませることができます。

大容量のハードディスクブロック装置では、通常、図12-2に示すように、その上にいくつかのパーティションを分割し、各パーティションに異なる完全なファイルシステムを格納することができます。図では、4つのパーティションがあり、FAT32ファイルシステム、NTFSファイルシステム、MINIXファイルシステム、EXT2ファイルシステムが格納されています。ハードディスクの第1セクターはマスターブートセクターで、ハードディスクのブートコードとパーティションテーブルの情報が格納されています。パーティションテーブルの情報は、ハードディスク上の各パーティションの種類、開始位置パラメーターと終了位置パラメーター、ハードディスク内で占有されているセクターの総数を示しています。kernel/blk_drv/hd.cファイルの後にあるハードディスクのパーティションテーブル構造をご覧ください。

図12-2 ハードディスク装置のパーティションとファイルシステム

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.

Linux 0.12システムでは、ロードされたファイルシステムのスーパーブロックは、スーパーブロックテーブル（配列）super_block[]に格納されています。このテーブルには8つのエントリがあり、Linux 0.12システムでは同時に8つのファイルシステムをロードすることができます。スーパーブロックテーブルは、super.cプログラムのmount_root()関数で初期化されます。スーパーブロックテーブルは、super.cプログラムのmount_root()関数で初期化されます。read_super()関数では、新しく読

み込まれたファイルシステムのスーパーブロックがテーブルに設定され、put_super()関数でスーパーブロックが解放されます。

Field name	Data type	Description
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	Log_2(DiskBlocks/LogicBlocks)
s_max_size	long	Maximum file size
s_magic	short	FS magic number (0x137f)
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)

図12-3 MINIXファイルシステムのスーパーブロック構造

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.

また、スーパーブロックの構造から、論理ブロックのビットマップは最大8個のバッファブロック(s_zmap[8])を使用し、各ブロックサイズは1024バイトで、各ビットは1つのディスクブロックの占有状態を表していることがわかります。したがって、1つのバッファブロックは8192個のディスクブロックを表すことができ、8つのバッファブロックは合計65,536個のディスクブロックを表すことができるので、MINIXファイルシステム1.0がサポートするブロックデバイスの最大容量（長さ）は64MBとなります。

i-nodeは、ディスクデバイス上の各ファイルやディレクトリ名のインデックス情報を格納するために使用される。i-nodeのビットマップは、i-nodeが使用されているかどうかを示すために使用され、各ビットはi-nodeを表します。1Kブロックの場合、1つのディスクブロックは8192個のi-nodeの使用を表すことができる。論理ブロックのビットマップの場合と同様に、アイドル状態のi-nodeを探す関数は、全てのi-nodeが使用されている場合には0という値を返すため、i-nodeビットマップの1バイト目の最下位ビット（ビット0）と対応するi-node0はアイドル状態となり、ファイルシステム作成時には、i-node0の対応するビットマップのビット位置はあらかじめ1に設定されている。したがって、第1のi-nodeビットマップブロックでは、8191個のi-nodeの状態しか表現できない。

ディスクのi-node部分は、ファイルシステムにおけるファイル名やディレクトリ名のインデックスノードを保持しており、各ファイル名やディレクトリ名にはi-nodeが存在する。各i-node構造には、ファイルホストのID（uid）、ファイルのグループID（gid）、ファイルの長さ、アクセス修正時間、ディスク上のファイルデータブロックの位置など、対応するファイルやディレクトリに関する情報が格納される。図12-4に示すように、i-nodeの構造全体には合計32バイトが使用される。

Field name	Data type	Description
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] 2nd indirect block no. Note: for device file, zone[0] is device no. of the specified device file.
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.

図12-4 MINIXファイルシステムバージョン1.0のi-node構造

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.

図12-5 i-nodeモードフィールドの内容

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.

また、/dev/ディレクトリ内のデバイスファイルについては、ディスクデータ領域のデータディスクブロックを占有しない、つまりファイル長は0となります。デバイスファイル名のi-nodeは、それが定義するデバイスの属性とデバイス番号を保存するためだけに使用されます。デバイス番号は、デバイスファイルのiノードのzone[0]に格納される。

図12-6 i-nodeの論理ブロック配列の機能

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.

PCの場合、1セクタ（512バイト）の長さをブロック装置のデータブロック長として使うのが一般的です。MINIXファイルシステムでは、連続した2セクタのデータ（1024バイト）をディスクブロックと呼ばれる1つのデータブロックとして扱い、その長さはキャッシュのバッファブロック長と同じになります。ディスクの最初のディスクブロックから数えていく、つまりブートブロックは0番目のディスクブロックである。

上記の論理ブロックまたはブロックは、ディスクブロックの2の累乗倍になります。論理ブロックの

長さは、1、2、4、8個のディスクブロックの長さと同じになります。本書で取り上げるLinuxカーネルでは、論理ブロックの長さはディスクブロックの長さと等しいので、コードコメントではこの2つの用語は同じ意味になります。ただし、データロジックブロック（またはデータディスクブロック）という用語は、ディスク装置のデータ部分の最初のデータディスクブロックから番号を付けたディスクブロックを指します。

12.1.3 File Types, Attributes, and Directory Items

12.1.3.1 File types and attributes

UNIX系OSのファイルは、大まかに6つのカテゴリーに分けられる。(1)通常のファイル、(2)ディレクトリ名、(3)シンボリックリンクファイル、(4)名前付きパイプファイル、(5)文字デバイスファイル、(6)ブロックデバイスファイルである。シェルのコマンドラインプロンプトで「ls -l」コマンドを実行すると，図12-7に示すように，リストアップされたファイルのステータス情報からファイルの種類を知ることができる。

図12-7 コマンド「ls -l」で表示されるファイル情報

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.

レギュラーファイル（'-'）は、ファイルシステムが解釈しないファイルで、任意の長さのバイトストリームを含むことができます。

例えば、ソースファイル、バイナリ実行ファイル、ドキュメント、スクリプトファイルなどです。

ディレクトリ（'d'）は、UNIX系OSではファイルとしても表示されるが、ファイルシステム管理では、どのファイルがディレクトリに含まれ、それらがどのように整理されて階層的なファイルシステムを形成しているのかが分かるように、その内容を解釈する。

シンボリックリンク（'s'）は、異なるファイル名を持つ別のファイルを参照するために使用されます。シンボリックリンクは、他のファイルシステムのファイルに接続することができます。シンボリックリンクを削除しても、接続先のファイルには影響しません。また、「ハードリンク」と呼ばれる接続方法もあります。これは、ここで説明するシンボリックリンクのリンク先のファイルと同じステータスを持ち、一般的なファイルとして扱われますが、ファイルシステム（またはデバイス）をまたいでリンクすることはできず、ファイルのリンクカウント値が増加します。後述のリンクカウントの説明を参照してください。

名前付きパイプ（'p'）ファイルは、システムが名前付きパイプを作成したときに作成されるファイルで、関係のないプロセス間の通信に使用することができます。

キャラクタデバイス（'c'）ファイルは、ttyターミナル、メモリーデバイス、ネットワークデバイスなどのキャラクタデバイスにファイルを操作してアクセスするためのファイルです。

ブロックデバイス（'b'）ファイルは、ハードディスクやフロッピーディスクなどのデバイスにアクセスするために使用される。UNIX系OSでは、ブロックデバイスファイルとキャラクタデバイスファイルは、一般的にシステムの/devディレクトリに格納されている。

Linuxカーネルでは、ファイルタイプの情報は、対応するi-nodeのi_modeフィールドに格納され、上位4ビットで表現される。ファイルシステムを操作する際、システムは以下のようないくつかのマクロを使用しています。

は，S_ISBLK，S_ISDIR などのファイルタイプを決定します。これらのマクロは、include/sys/stat.hファイルで定義されています。

図では、ファイルタイプ文字の後に、1セット3文字で構成されたファイル許可属性が続き、ファイルホスト、同一グループのユーザー、他のユーザーのファイルに対するアクセス権を示すために使用されている。rwx」はそれぞれ、ファイルの読み取り、書き込み、実行の許可を示す。ディレクトリ名のファイルの場合、実行可能とは、そのディレクトリに入ることができることを意味する。ファイルのパーミッションを操作する場合、一般的には8進数で表します。例えば，「755」（0b111，101，101）は，ファイルホストがファイルの読み取り/書き込み/実行が可能であり，同じグループのユーザーなどがファイルの読み取り/実行が可能であることを示す。Linuxでは

0.12のソースコードでは、ファイルのパーミッション情報はi-nodeのi_modeフィールドにも格納されており、フィールドの下位9ビットは3つのパーミッションのセットを表すのに使用され、変数modeで表されることが多い。ファイルパーミッションに関するマクロは，include/fcntl.hというファイルで定義されている。

2.1.3.2 図中の「リンクカウント」フィールドは、ファイルがハードリンクによって参照された回数を示す。カウントがゼロになると、そのファイルは削除される。'Username'はファイルホストの名前、'Group name'はユーザーが所属するグループの名前を示す。

12.1.3.3 File System Directory Item Structure

Linux 0.12システムでは、MINIXファイルシステムのバージョン1.0を採用しています。そのディレクトリ構造とディレクトリエントリ構造は、従来のUNIXファイルシステムと同じで、include/linux/fs.hファイルで定義されています。ファイルシステムのディレクトリでは，ディレクトリ内のすべてのファイル名に対するディレクトリエントリが，ディレクトリファイルのデータブロックに格納されている。例えば，ディレクトリ名 root/ の下にあるすべてのファイル名のディレクトリエントリは，ディレクトリ名ファイル root/ のデータブロックに格納される。ファイルシステムのルート・ディレクトリにあるすべてのファイル名情報は、指定されたi-node（すなわち、番号1のi-node）のデータ・ブロックに格納される。ファイル名ディレクトリのエントリ構造は以下の通りである。

```
// include/linux/fs.hファイルで定義されています。
#define NAME_LEN 14                         // maximum name length.
#define ROOT_INO 1                          // root i-node.

// ファイルディレクトリのエントリ
構造 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.

ファイルが開かれると、ファイルシステムは、図12-8に示すように、与えられたファイル名に応じてそのi-node番号を見つけ、対応するi-node情報によってファイルが置かれているディスクブロックの位置を見つける。例えば，ファイル名 /usr/bin/vi の i-node 番号を見つけるために，ファイルシステムはまず固定の i-node 番号（1）を持つルート・ディレクトリから開始する，つまり，i-node 番号 1 のデータ・ブロックから名前を見つけるのである。usrのディレクトリエントリは、このようにファイル/usrのiノード番号を取得します。このiノード番号により、ファイルシステムはスムーズにディレクトリ「/usr」を得ることができ、その中にファイル名「bin」のディレクトリエントリを見つけることができる。このようにして、/usr/binのi-node番号もわかるので、ディレクトリ/usr/binのディレクトリの位置がわかり、その中のviファイルのディレクトリエントリを見つけることができます。最後に、ファイルパス名「/usr/bin/vi」のi-node番号が分かるので、i-node構造の情報を取得するために

ディスクを使用します。

図12-8 ファイル名からファイルのディスクブロックの位置を探す

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).

図12-9 ファイル名からそのデータブロックを取得する

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,

は、ハードリンクがファイルシステムを越えられません。

ハードリンクとは異なり、シンボリック・リンク・タイプのファイル名ディレクトリ・エントリは、対応するi-nodeを直接指していない。シンボリックリンク型のディレクトリエントリは、ファイルのパス名文字列を対応するファイルのデータブロックに格納します。シンボリックリンクのディレクトリエントリにアクセスすると、カーネルはファイルの内容を読み込んだ後、パス名文字列に基づいて指定されたファイルにアクセスする。したがって，シンボリックリンクはファイルシステムに限定されない場合がある。あるファイルシステムに、他のファイルシステムのファイル名を指すシンボリックリンクを作成することができる。

また、各ディレクトリには2つの特別なファイル・ディレクトリ・エントリがあり、その名前はそれぞれ「.」と「...」に固定されています。'.'ディレクトリ・エントリは、カレント・ディレクトリのi-node番号を与え、'.'ディレクトリ・エントリは、親ディレクトリのi-node番号を与える。したがって、相対パス名が与えられたとき、ファイルシステムはこれら2つの特別なディレクトリ・エントリをルックアップ操作に使用することができる。例えば，「.../kernel/Makefile」を探すには，まずカレント・ディレクトリの「...」ディレクトリ・エントリから親ディレクトリのi-node番号を取得し，その後，上述のプロセスに従ってルックアップ操作を行えばよい。

各ディレクトリ・ファイルのディレクトリ・エントリについて、そのi-nodeのリンク・カウント・フィールドは、そのディレクトリに接続されているディレクトリ・エントリの数も示しています。したがって，各ディレクトリ・ファイルのリンク・カウント値は少なくとも2であり，そのうちの1つはディレクトリ・ファイルを含むディレクトリ・エントリへのリンクであり，もう1つはディレクトリ内の「...」ディレクトリ・エントリへのリンクである。例えば，カレントディレクトリに'mydir'という名前のサブディレクトリを作成した場合，カレントディレクトリとサブディレクトリの間のリンクは図12-10のようになります。

図12-10 ディレクトリファイルのエントリとサブディレクトリのリンク

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.

12.1.3.4 Directory Structure Example

Linux 0.12システムを例に、ルートディレクトリ構造を見てみましょう。を実行した後、Linux 0.12システムのbochsシミュレーションでは、まずそのファイルシステムのルートディレクトリのエントリをリストアップします。その中には、暗黙の了解である'.'や'.'のディレクトリエントリも含まれます。次に、hexdumpコマンドを使用して、'.'または'.'ファイルのデータブロックの内容を表示すると、ルートディレクトリに含まれる各ディレクトリ項目の内容を見ることができます。

```
[/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

 [/]#   0001 002e 0000 0000 0000 0000 0000 0000       // .
 hexdu
 mp .
0000000
0000010 0001 2e2e 0000 0000 0000 0000 0000 0000       // ..
0000020 0002 6962 006e 0000 0000 0000 0000 0000       // bin
0000030 0003 6564 0076 0000 0000 0000 0000 0000       // dev
0000040 0004 7465 0063 0000 0000 0000 0000 0000       // etc
0000050 0005 7375 0072 0000 0000 0000 0000 0000       // usr
0000060 0115 6e6d 0074 0000 0000 0000 0000 0000       // mnt
0000070 0036 6d74 0070 0000 0000 0000 0000 0000       // tmp
0000080 0000 6962 2e6e 656e 0077 0000 0000 0000       // idle, not used.
0000090 0052 6d69 6761 0065 0000 0000 0000 0000       // image
00000a0 007b 6176 0072 0000 0000 0000 0000 0000       // var
00000b0
[/]#
```

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.

次に、etc/サブディレクトリのエントリを見てみましょう。etc/ディレクトリでもhexdumpコマンドを使うと、以下のようにetc/サブディレクトリに含まれるディレクトリエントリを表示することができます。

```
[/]# ls etc -la
 トータル     2 root     root          224 Mar 21  2004 .
 32
drwxr-xr-x
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

```
0000000 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.

12.1.4 High speed buffer (Buffer cache)

ファイルシステムがブロックデバイス内のデータにアクセスするためには、高速バッファが必須である。ブロックデバイス上のファイルシステムのデータにアクセスするためには、カーネルがブロックデバイスにアクセスして、その都度、読み出しや書き込みの操作を行えばよい。しかし、各I/O操作の時間は、メモリやCPUの処理速度に比べて非常に遅い。そこでカーネルは、システムのパフォーマンスを向上させるために、メモリ上に高速データバッファ（バッファキャッシュ）をオープンにし、ディスクのデータブロックと同じ大きさのバッファブロックに分割して使用・管理することで、ブロックデバイスへのアクセス回数を減らすようにしている。Linuxカーネルでは，図5-5に示すように，バッファキャッシュは，カーネルコード領域と主記憶領域の間に配置されている。バッファキャッシュには，各ブロックデバイスの中で最近使用されたデータブロックが格納されている。ブロックデバイスからデータを読み出す必要がある場合、バッファマネージャはまずバッファキャッシュを検索します。対応するデータがすでにバッファにある場合は、ブロックデバイスから再度データを読み出す必要はありません。データがバッファキャッシュにない場合は、ブロックデバイスのリードコマンドが発行され、データがキャッシュに読み込まれます。ブロックデバイスにデータを書き込む必要があるときは、システムはキャッシュ内の空きバッファブロックを要求して、データを一時的に保存する。実際にデバイスにデータを書き込むタイミングについては、デバイスのデータ同期機能によって実現されます。

12.1.5 ファイルシステム内の他のプログラムは、アクセスしたいデバイス番号とデータロジックブロック番号を指定して、ブロックリード・ライト関数を呼び出す。これらのインターフェース関数は、ブロックリード関数bread（）、ブロックアドバンスリードアヘッド関数breada（）、ページブロックリード関数bread_page（）である。ページブロックリード機能は、メモリの1ページに収容できるバッフ

ァブロック数（4ブロック）を一度に読み出す。

12.1.6 File System Lower Level Functions

- ファイルシステムの基本的な処理機能は、以下の5つのファイルに集約されています。
  - 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 iput() 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(), iput(), 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().

これらのファイルの機能間の階層関係を図12-11に示します。

図12-11 ファイルシステムの低レベル機能階層

12.1.7 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.

図12-12 ファイルデータアクセス機能

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.

file_dev.cプログラムのfile_read()およびfile_write()関数は、一般的な通常のファイルにアクセスするために使用されます。ファイルのi-nodeとファイル構造を与えることで、ファイルが配置されているデバイス番号と、ファイルの現在の読み書きポインタを知ることができます。

パイプの読み書き関数read_pipe()とwrite_pipe()はpipe.cファイルに実装されています。また、この

ファイルは

は、匿名のパイプを作成するシステムコールpipe()を実装しています。パイプは、主にプロセス間のデータ転送を先入れ先出しで行うために使われたり、プロセスの同期を取るために使われたりします。パイプには「名前付きパイプ」と「無名パイプ」の2種類があります。名前付きパイプはファイルシステムのオープンコールで作成され、無名パイプはシステムコールのpipe()で作成される。パイプを使用する際には、通常のファイルのread()、write()、close()関数が使用されます。無名パイプへのアクセスを共有できるのはpipe()コールの子孫のみであり、許可が与えられている限り、すべてのプロセスが名前付きパイプにアクセスできる。

パイプの読み書きについては、あるプロセスがパイプの一方の端にデータを書き込み、別のプロセスがパイプのもう一方の端からデータを読み込んでいることがわかります。カーネルがパイプラインのデータにアクセスする方法は、通常のフォーマルファイルのデータにアクセスする方法とまったく同じです。パイプ用のストレージを割り当てることと、通常のファイル用のスペースを割り当てることの違いは、パイプはi-nodeのダイレクトブロックのみを使用することです。カーネルは、i-nodeのダイレクトブロックをパイプラインのサーキュラーキューとして管理し、読み書きポインタを修正することで先入れ先出しの順序を確保している。

キャラクタデバイスファイルの場合、システムコールのread()とwrite()はchar_dev.cのrw_char()関数を呼び出します。

キャラクターデバイスには、コンソールターミナル（tty）、シリアルターミナル（ttyx）、メモリーキャラクターデバイスなどがあります。

また、カーネルは、ファイル構造、ファイルテーブルfile_table[]、メモリ内i-nodeテーブルinode_table[]を用いて、ファイルへの操作アクセスを管理します。これらのデータ構造やテーブルの定義は、ヘッダファイルinclude/linux/fs.hに記載されています。ファイル構造は以下のように定義されています。

```
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:

```
// open()とfcntl()で使用されるファイルアクセスモード。同時に使用できるのは3つのうち1つだけです。
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.
11                          // open()のファイル作成フラグ。上記のアクセスモードと一緒に「bit or」モードで使用することができます。
12 #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

メモリi-node構造体フィールドf_inodeは，このファイルの対応するi-nodeテーブル内のメモリi-node構造体アイテムを指している。ファイルテーブルは，カーネル内のファイル構造項目の配列である。Linux 0.12カーネルでは、ファイルテーブルは最大64個のアイテムを持つことができるので、システム全体で最大64個のファイルを同時に開くことができます。プロセスデータ構造（つまり、プロセス制御ブロックまたはプロセス記述子）では、オープンファイルのファイル構造ポインタ配列filp[NR_OPEN]フィールドが具体的に定義されています。ここでNR_OPEN=20なので、各プロセスは同時に20個までのファイルを開くことができます。ポインタ配列の項目のシーケンス番号はファイルディスクリプタ(ファイルハンドル)に対応し、項目のポインタはファイルテーブルでオープンされたファイル項目を指します。例えば，filp[0]は，プロセスが現在オープンしているファイル記述子0（ハンドル0）に対応するファイル構造体のポインタである。

カーネル内のi-nodeテーブルinode_table[NR_INODE]は、メモリi-node構造体の配列であり、NR_INODE=32であるため、現時点では、カーネルは同時に32個のメモリi-node情報しか格納できません。プロセスがオープンしたファイルとカーネルのファイルテーブルと対応するメモリi-nodeの関係は、図12-13で表すことができます。図中の一方のファイルはプロセスの標準入力として開かれ（ファイルハンドル0）、もう一方のファイルは標準出力として開かれます（ファイルハンドル1）。

図12-13 ファイルを開くためにプロセスが使用する構造体

12.1.8 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.

図12-14 ファイルシステム上位層の動作プログラム

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.

exec.cプログラムは、バイナリ実行ファイルやシェルスクリプトファイルの読み込みと実行を実装しています。主な関数はdo_execve()で、これはシステムコール割り込み(int 0x80)関数番号のCハンドラです。

NR_execve()で、exec()関数群の主要な実装関数です。

Fcntl.cは、ファイル制御システムコールであるfcntl()と、2つのファイルハンドル複製システムコールであるdup()とdup2()を実装しています。dup2()は新しいハンドルの値を指定する必要がありますが、dup()は現在の値が最も小さい未使用のハンドルを返します。ハンドルの複製操作は、主にファイルの標準的な入出力のリダイレクトやパイプライン操作に使用されます。

ioctl.cファイルは，入出力制御システムコールioctl()を実装しており，主にtty_ioctl()関数を呼び出して端末の入出力を制御する。

stat.cファイルは、ファイルステータス情報システムコールであるstat()およびfstat()を実装するために使用されます。stat()はファイル名で情報を取得し、fstat()はファイルハンドル(記述子)で情報を取得する。

12.1.9 Analysis of a File System in 360KB Floppy Disk

図12-1に示したファイルシステム構造の理解を深めるために、Linuxの

0.12システムでは、360KBのフロッピーイメージにMINIX 1.0のファイルシステムを作成し、hello.cというファイルのみを保存しています。

まず、Linux 0.12システムのbochs環境で以下のコマンドを実行し、ファイルシステムを作成します。

```
[/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 szie(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).

次に、mountコマンドを使ってデバイスをディレクトリ/mntにマウントし、ファイルhello.cをそこにコピーしてから、ファイルシステムをアンマウントします。これで、bochsの第2フロッピー・ドライブに対応するディスク・イメージ・ファイル(diskb.img)に格納されている1つのファイルだけを持つMINIXファイル・システムができあがりました。

では、このファイルシステムの具体的な内容を見てみましょう。ここでは便宜上、Linux 0.12系のhexdumpコマンドを使って内容を観察します。また、bochsシステムを終了して、バイナリを修正できるnotepad++などの編集プログラムを使って見ることもできます。デバイス /dev/fd1 で hexdump コマンドを実行すると、以下のように表示されます（少し整理されています）。

```
[/usr/r 44eb 4d90 6f74 6c6f 2073 0020 0102 0001  // 0x0000 - 0x03ff (1KB) is boot block.
oot]#
hexdu
mp
/dev/fd
1 |
more
0000000
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
*                                                //// The following is data block contents.
0002000 0001 002e 0000 0000 0000 0000 0000 0000  // 0x2000 - 0x23ff (1KB), root i-node data
0002010 0001 2e2e 0000 0000 0000 0000 0000 0000
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.

ディスクブロック1（0x0400～0x07ff、1KB）は、スーパーブロックの内容である。MINIXファイルシステムのスーパーブロックのデータ構造（図12-3参照）によると、表12-1に示すファイルシステムのスーパーブロックの情報は、合計18バイトであることがわかります。論理ブロックあたりのディスクブロック数を2としたときの対数が0であることから、MINIXファイルシステムでは、ディスクブロックサイズは論理ブロック（ブロック）サイズと等しい。

表12-1 360KBディスクにおけるMINIXファイルシステムのスーパーブロック 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.

ディスクブロック2のデータを見ると、1バイト目の値が0x07（0b0000111）、つまり、i-nodeビットマップの最初の3ビットが占有されていることがわかる。前述の説明からわかるように、第1ビット（ビット0）は予約されている。第2ビットと第3ビットは、それぞれファイルシステムのNo.1とNo.2のi-nodeが使用されていることを示しており、つまり、以下のi-node領域には既に2つのi-node構造が含まれていることになる。実際には、ノード1はファイルシステムのルートi-nodeとして使用され、ノード2はファイルシステム上の唯一のファイルhello.cに使用されるが、その内容は後述する。

ディスクブロック3（0x0c00～0x0fff、1KB）は、論理ブロックのビットマップの内容です。デ

ィスクの容量は360KBしかないので、ファイルシステムでは実際には360ビットを使用し、360/8＝45バイトとなります。特別な目的のためのブロックの数は ブートブロック1個＋スーパーブロック1個＋i-nodeビットマップブロック1個＋論理ブロックビットマップブロック1個＋i-nodeデータブロック4個＝8個である。論理ブロックビットマップは、ディスク上のデータ領域にディスクブロックが占有されていることを示すだけであることを考えると、使用された機能ブロックの数を削除した後、論理ブロックビットマップに必要な実際のビット数は360 - 8 = 352ビット（44バイトを占有）であり、さらに残りの未使用ビット0を加えると、ビットマップは合計353ビットを必要とする。最後の45バイト目（0xfe）のビットが1つだけ0になっているのはこのためです。

したがって、論理ブロックビットマップで論理ブロックのビットオフセット値nrがわかっている場合、それに対応する実際のディスクのブロック番号ブロックはnr ＋ 8 -1、すなわちblock ＝ nr ＋ s_firstdatazone - 1となります。

i-nodeビットマップと同様に、論理ブロックビットマップの第1バイトの最初の3ビットは既に占有されている。第1ビット（ビット0）は予約されており、第2ビットと第3ビットはディスクデータゾーンで2つのディスクブロック（論理ブロック）が使用されていることを示している。実際には、ビット1で表されるディスク・データ・ゾーンの最初のディスク・ブロックは、ルートのi-nodeがデータ情報（ディレクトリ・エントリ）を保存するために使用され、ビット2で表される2番目のディスク・ブロックは、ナンバー2のi-nodeが関連するデータ情報を保存するために使用される。なお、ここでいうデータ情報とは、i-nodeが管理するデータコンテンツのことであり、i-nodeの構造情報ではありませんのでご注意ください。i-node自体の構造情報は、i-node領域内のi-node構造専用のディスクブロック、つまりディスクブロック4～7に保存されます。

ディスクブロック 4--7 (0x1000 - 0x1fff, 4KB) 4つのディスクブロックを用いて、i-nodeの構造情報を格納する。ファイルシステムには120個のi-nodeがあり、各i-nodeは32バイトを占めているので（図12-4参照）、120×32＝3840バイト、つまり4つのディスクブロックが必要となる。以上のデータから，表12-2および表12-3に示すように，最初の32バイトにはルートi-nodeの内容が，その後の32バイトにはノード2の内容が格納されていることがわかる。

表12-2 No.1 ルートのi-node構造の内容 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.

表12-3 No.2 i-nodeの構造の内容 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]	Araay 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.

No.2 i-nodeのデータブロックも1ブロックのみで、ディスクデータゾーンの第2ブロックに位置し、ディスクブロック番号は9です。

ディスクブロック8（0x2000 - 0x23ff, 1KB）は、ルートi-nodeのデータであり、表12-4に示すように、3つのディレクトリ項目構造情報（48バイト）がある。

表12-4 ルート i-node のデータ内容 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

12.2.1 本章からは、fs/ディレクトリに格納されているプログラムを一つずつ解説、注釈していく。本章の第1節の説明によれば、本章のプログラムは、4つの部分に分けられる。(1)キャッシュ管理、(2)ファイルの基本操作、(3)ファイルのデータアクセス、(4)ファイルの高位アクセス制御、 の4つに分けられる。ここではまず、第1部の高速バッファ（またはバッファキャッシュ）管理の手順を説明します。このセクションには、buffer.cというプログラムが1つだけ含まれています。

12.2.2 Function

1. 以下では、まず物理メモリ上のキャッシュの具体的な位置を説明し、その初期化のプロセスを説明する。そして、バッファキャッシュの構造、使用されているリンクリスト構造とハッシュテーブル構造を示す。次に、重要なバッファブロック取得関数gltblk()とブロック読み出し関数bread()について詳しく説明します。最後に、システムの観点から、バッファキャッシュのアクセス処理と同期動作の方法を説明します。

2. Buffer cache physical location

buffer.cプログラムは、バッファキャッシュ（プール）を操作・管理するためのプログラムです。バッファキャッシュは、図12-15に示すように、カーネルコードブロックとメインメモリ領域の間に配置されています。バッファキャッシュは、ブロックデバイスとカーネル内の他のプログラムとの橋渡しの役割を果たします。ブロックデバイスのドライバに加えて、カーネルプログラムがブロックデバイス内のデータにアクセスする必要がある場合、キャッシュを経由して間接的に操作する必要があります。

図12-15 物理メモリ内のバッファキャッシュの位置

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.

3. Buffer cache initialization

キャッシュ全体は1024バイトのバッファブロックに分割されており、これはブロックデバイス上のディスク論理ブロックと全く同じサイズである。バッファキャッシュは、すべてのバッファブロックを含むハッシュテーブルとリンクリストによって管理されています。バッファの初期化処理では，初期化プログラムは，図12-16に示すように，バッファ全体の両端から開始し，バッファブロックのヘッダ構造と対応するバッファブロックをそれぞれ設定する。バッファの高位端は1024バイトのバッファブロックに分割され，低位端には各バッファブロックに対応するバッファヘッダ構造buffer_head（include/linux/fs.h, line 68）がそれぞれ設定される。このヘッダ構造は、対応するバッファブロックの属性を記述するために使用され、すべてのバッファヘッダをリンクリストに結合するために使用されます。この設定操作は、バッファブロックを分割するのに必要なバッファのメモリが不足するまで続けられます。

図12-16 バッファキャッシュの初期化

4. Buffer cache structure and linked list

すべてのバッファブロックのbuffer_headは、図12-17に示すように、二重リンクリスト構造にリンクされています。図中の free_list ポインタは、リンクリストの先頭ポインタであり、フリーブロックリストの最初の「最もアイドルな」バッファブロックを指しています。バッファブロックのバックポインタb_prev_freeは，バッファブロックリストの最後のバッファブロック，つまり，最近使われたばかりのバッファブロックを指しています．バッファブロックのヘッダデータ構造は

```
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.

フィールドb_countは、バッファマネージャが使用するカウントで、対応するバッファブロックが各プロセスで使用（参照）されている回数を示すため、このフィールドは、バッファブロックのプログラムの参照カウント管理に使用され、また、バッファブロックの他のフラグとは独立しています。参照カウントが0でない場合、バッファマネージャは対応するバッファブロックを解放することができません。フリーブロックとは、b_count=0のブロックのことで、b_count=0の場合は、対応するバッファブロックが使用されていない（フリー）ことを意味し、そうでない場合は使用されていることを意味します。プログラムアプリケーションのブロックについて、バッファマネージャがハッシュテーブルから既存の指定されたブロックを取得できる場合、ブロックのb_countは1だけインクリメントされます（b_count++）。バッファブロックが再適用される未使用のブロックである場合、そのヘッダ構造のb_countは1に等しく設定される。プログラムがブロックへの参照を解放すると、それに応じてそのブロックへの参照数がデクリメントされます（b_count--）。フラグb_lockは、他のプログラムが指定されたバッファブロックを使用し、ロックしていることを示すので、b_lockが設定されているバッファブロックでは、b_countは0より大きくなければなりません。

フィールドb_dirtは、バッファブロック内のコンテンツが、ブロックデバイス上の対応するデータブロックとは異なる内容に変更されたかどうかを示すダーティフラグです（遅延書き込み）。フィールドb_uptodateは、バッファブロック内のデータが有効であるかどうかを示すデータ更新（有効）フラグです。両フラグは、ブロックの初期化やリリース時に0に設定され、この時点でバッファブロックが無効であることを示します。バッファにデータが書き込まれたが、まだデバイスに書き込まれていない場合は、b_dirt = 1, b_uptodate = 0となります。データがブロックデバイスに書き込まれたとき、またはブロックデバイスからバッファブロックを読み込んだ直後、すなわちb_uptodate = 1のとき、データは有効になります。なお、新しいデバイスのバッファブロックが適用されたときに、b_dirtとb_uptodateの両方が1になる特殊なケースがあります。これは、バッファブロック内のデータがブロックデバイスとは異なるが、データはまだ有効（更新されている）であることを示しています。

図12-17 全バッファブロックの双方向サーキュラーリンクされたフリーリスト

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.

カーネルコードがバッファキャッシュ内のバッファブロックを使用する際には、データのデバイス番号（dev）と論理ブロック番号を指定し、バッファブロックリード関数のbread()、bread_page()、breada()を呼び出して操作します。これらの関数は、バッファ検索管理関数であるgetblk()を用いて、すべてのバッファブロックの中から一致するブロックや空きブロックを探します。この関数を以下に紹介します。システムがバッファブロックを解放する際には、brelse()関数を呼び出す必要があります。これらすべてのバッファブロックデータアクセスおよび管理関数の呼び出し階層は、図12-18で説明できます。

図12-18 バッファーマネジメント機能の階層的な関係

5. Hash queue of the buffer cache

要求されたデータブロックがバッファに読み込まれたかどうかをバッファキャッシュで迅速かつ効率的に調べるために、buffer.cプログラムでは、307個のbuffer_headポインタエントリを持つハッシュ配列テーブル構造を使用しています。ハッシュテーブルで使用されるハッシュ関数は、デバイス番号と論理ブロック番号を組み合わせたものです。このプログラムで使用されている具体的なハッシュ関

数は（デバイス番号 ^ 論理ブロック番号）Mod 307。図12-17において，ポインタb_prevおよびb_nextは，ハッシュテーブルにおいて，同一アイテム上の複数のバッファブロック間のハッシュ化に使用される双方向リンクであり，すなわち，ハッシュ関数によって算出された同じハッシュ値を持つバッファブロック

関数は、リンクリストの同じ配列項目にリンクされています。バッファブロックがハッシュキュー上でどのように動作するかについては、「Unix Operating System Design」の第3章の説明を参照してください。動的に変化するハッシュテーブル構造のある時点での状態については、図12-19を参照。

図12-19 ある瞬間のバッファブロックのハッシュキューの模式図

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.

6. Buffer block get function

前述の3つの関数は、いずれも実行時にgetblk()を呼び出し、適切な空きバッファブロックを取得します。この関数は、まずget_hash_table()関数を呼び出し、指定されたデバイス番号と論理ブロック番号のバッファブロックを求めてハッシュテーブルのキューを検索します。指定されたブロックが存在する場合は、直ちに対応するブロックヘッダ構造体へのポインタを返します。存在しない場合は、フリーリストのヘッダから始めて、フリーリストをスキャンして空きバッファを探します。検索の過程では，見つかったフリーバッファブロックの比較も行い，モディファイドフラグとロックフラグの組み合わせで与えられた重みに従って，どのフリーブロックが最も適しているかを判断します。見つかったフリーブロックがモディファイもロックもされていなければ、探し続ける必要はありません。この時、フリーブロックが見つからなければ、現在のプロセスはスリープ状態になり、引き続き実行する際に再度検索されます。フリーブロックがロックされている場合は、プロセスもスリープ状態にして、他（ブロックドライバ）のロック解除を待つ必要がある。スリープ待機中にバッファブロックが他のプロセスに占有されていた場合は、再度バッファブロックの検索を開始すればよい。それ

以外の場合は、バッファブロックが変更されているかどうかを判断し、変更されていれば、そのブロックをディスクに書き込み、ブロックのロックが解除されるのを待ちます。この時、バッファブロックが再び他のプロセスに占有されていると、再び完全に放棄されてしまうので、再びgetblk()の実行を開始しなければなりません。

上記のトスを経験した後、この時にもう一つの予期せぬ状況があるかもしれません。つまり、私たちがスリープしている間に、他のプロセスが私たちが必要とするバッファブロックをハッシュキューに追加しているかもしれないので、私たちは検索して

を最後に、ここではハッシュキューを使用しています。もし、本当に必要なバッファブロックがハッシュキューの中に見つかったとしたら、見つかったバッファブロックに対して上記のような判断をしなければならないので、再度、getblk()の実行を開始する必要があります。最終的には、プロセスで使用されておらず、ロックされておらず、クリーン（modifiedフラグが設定されていない）なフリーのバッファブロックを見つけました。そこで、そのブロックへの参照数を設定し、他のいくつかのフラグをリセットしてから、ブロックのバッファヘッダ構造をフリーリストのキューから削除しました。バッファブロックが属するデバイス番号と対応する論理番号が設定された後、ハッシュテーブルの対応するテーブルエントリのヘッダに挿入され、フリーリストキューの最後にリンクされます。空きブロックの検索はフリーリストのヘッダーから始まるので、このように、まずフリーリストのキューから直近に使用されたバッファブロックを取り出して使用し、その後フリーリストの末尾に再挿入するという動作は、LRUアルゴリズムを実現している。最後に、バッファブロックのヘッダへのポインタが返されます。getblk()の処理全体は図12-20をご覧ください。

- 以上の分析から，この関数は，新しい空きバッファブロックを取得するたびに，それを free_list ヘッダポインタが指すリンクリストの末尾に移動させることがわかります．つまり，バッファブロックがリンクリストの末尾に近ければ近いほど，そのバッファブロックが使用される時間が短くなるということです．したがって、対応するバッファブロックがハッシュテーブルで見つからない場合、新しいフリーのバッファブロックを検索する際には、free_listヘッダから検索を開始する。このように、カーネルがバッファブロックを取得するアルゴリズムは、以下の戦略を用いていることがわかります。
  - 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.

したがって、最も理想的な状況は、完全に空いているバッファブロック、つまりb_dirtとb_lockのフラグが0のバッファブロックを見つけることである。しかし、この2つの条件が満たされない場合は、b_dirtとb_lockのフラグに基づいて値を計算する必要がある。デバイス操作には時間がかかることが多いので、計算におけるb_dirtの比重を大きくする必要があるのだ。次に、結果の値が最も小さいバッファブロックを待ちます（バッファブロックがすでにロックされている場合）。最後に、フラグb_lockが0になると、待機していたバッファブロックの元の内容がブロックデバイスに書き込まれたことを示します。そのため、getblk()は空いているバッファブロックを取得します。

図12-20 Getblk()関数の実行フローチャート

7. Buffer block read function

以上の処理から、getblk()が返すバッファブロックは、新しいフリーブロックである場合もあれば、必要なデータが含まれているバッファブロックが、すでにバッファキャッシュに存在しているだけの場合もあることがわかります。そのため、データブロックの読み込み操作（bread()）では、バッファブロックの更新フラグを判定して、含まれているデータが有効かどうかを確認する必要があります。有効であれば、そのデータブロックを応用プログラムに直接返すことができる。そうでない場合は、デバイスの低レベルブロック読み書き機能(ll_rw_block())を呼び出し、同時に自分をスリープ状態にして、バッファブロックにデータが読み込まれるのを待ち、起きた後にデータが有効かどうかを判断する必要があります。有効であれば、このデータをアプリケーションプログラムに返すことができ、そうでなければ、デバイスへの読み出し動作は失敗し、データは取り出せない。したがって、バッフ

ァブロックは解放され、NULL値が返されます。図12-21は、bread()関数のブロック図である。breada()関数とbread_page()関数は、bread()関数と似ている。

図 12-21 Bread()機能の実行フロー図

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).

8. buffer access method and synchronization operation

以上のように、高速バッファは、ブロックデバイスのアクセス効率の向上とデータの共有化に重要な役割を果たしています。ドライバ以外にも、カーネルによるブロックデバイスへの他の上位プログラムの読み書き操作は、キャッシュマネージャを介して間接的に実装する必要があります。両者の主な連携は、キャッシュマネージャのbread()関数と、ブロックデバイスの低レベルインターフェース関数ll_rw_block()を介して行われます。上位プログラムがブロックデバイスのデータにアクセスしたい場合、bread()を通じてバッファマネージャに申請します。必要なデータがすでにバッファキャッシュにある場合は、バッファマネージャはプログラムに直接データを返す。必要なデータがまだバッファにない場合は、バッファマネージャはll_rw_block()を通じてブロックデバイスドライバに申請し、プログラムに対応するプロセスをスリープさせて待機させます。ブロックデバイスドライバが指定されたデータをキャッシュに入れた後、図12-22のように、バッファマネージャは上位プログラムにデータを返します。

図12-22 カーネル・ブロック・デバイス・アクセス動作

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.

sync_inodes()関数は、ディスクを直接扱うことはありません。進められるのは、バッファ内の情報との同期を取ることだけです。残りの操作は、バッファマネージャが実行を担当する必要があります。sync_inodes()がどのi-nodeがディスク上のものと異なっているかを知るためには、まずバッファの内容をディスク上の内容と一致させる必要があります。このようにしてsync_indes()は、現在のディスクのバッファ内の最新データと比較することで、どのディスクのinodeを修正・更新する必要があるかを知ることができます。最後に、バッファ・キャッシュとディスク・デバイスの2回目の同期操作が行われ、メモリ内のデータとブロック・デバイス内のデータが真に同期されるようになります。

12.2.3 Code annotation

プログラム 12-1 linux/fs/buffer.c

```
/*
 *  linux/fs/buffer.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  'buffer.c' implements the buffer-cache functions. Race-conditions have
 * been avoided by NEVER letting a interrupt change a buffer (except for the
 * data, of course), but instead letting the caller do it. NOTE! As interrupts
 * can wake up a caller, some cli-sti sequences are needed to check for
 * sleep-on-calls. These should be extremely quick, though (I hope).
*/

/*
 * NOTE! There is one discordant note here: checking floppies for
 * disk change. This is where it fits best, I think, as it should
 * invalidate changed floppy-disk-caches.
*/

// <stdarg.h> 標準パラメータのヘッダファイルです。で変数パラメータのリストを定義します。
//      form of macros. It mainly describes one type (va_list) and three macros (va_start,
//      va_arg and va_end) for the vsprintf, vprintf, and vfprintf functions.
// <linux/config.h> カーネル設定用のヘッダーファイルです。キーボードの言語とハードを定義する
//      disk type (HD_TYPE) options.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_structや
//      data of the initial task 0, and some embedded assembly function macro statements
//      about the descriptor parameter settings and acquisition.
```

```
// <linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。
//     commonly used functions of the kernel.
```

```
// <asm/system.h> システムのヘッダーファイルです。を定義する埋め込みアセンブリマクロです。
//     modifies descriptors/interrupt gates, etc. is defined.
// <asm/io.h> Io のヘッダーファイルです。の io ポートを操作する関数を定義します。
//     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
// 以下の変数'end'の値は、リンカーldによってコンパイル時に生成され、以下を示します。
// 図12-15に示すように、カーネルモジュールの終了位置です。この値を求めるには
// カーネルがコンパイルされたときに生成されたSystem.mapファイルのことです。ここでは、次のことを示すために使われています。
// バッファキャッシュは、カーネルコードの最後に始まります。
// 33行目のbuffer_wait変数は、眠っているタスクのキューヘッドポインタである
// フリーのバッファブロックを待っている間。b_wait ポインタとは異なる効果があります。
// をバッファブロックのヘッダ構造に追加します。タスクがバッファブロックを要求したときに、たまたま
// 利用可能な空きバッファブロックが不足している場合、タスクは buffer_wait スリープに追加されます。
// wait queueです。b_waitは、待機しているタスク専用の待機キューのヘッダであり
29 // 指定されたバッファブロック（つまり、b_wait に対応するバッファブロック）。
30 extern int end;
31 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.

// システムバッファキャッシュに含まれるバッファブロックの数を定義します。ここでは
// NR_BUFFERSは、linux/fs.hファイルの48行目で定義されている定数シンボルで、以下のように定義されています。
// fs.hの172行目でグローバル変数として宣言されているnr_buffersという変数として
//ファイルを作成しました。Linus氏は、暗にnr_buffersを示すようにコードを書いています。
// カーネルバッファが初期化された後も変更されない「定数」です。設定されるのは
// 初期化関数buffer_init() (371行目)の中の
34 int NR_BUFFERS = 0;                              // The number of buffer blocks.
35
//// 指定したバッファブロックのロックが解除されるのを待ちます。
// 指定されたバッファブロックbhがロックされていれば、プロセスを中断させずにスリープさせる
// をバッファキューの b_wait に入れます。バッファブロックのロックが解除されると、その待機中のすべてのプロセスが
// のキューが起こされます。割り込みが無効になった後にスリープ状態になりますが（cli）。
// このようにしても、他のプロセスコンテクストでの割込みに対する応答には影響しません。なぜなら、それぞれの
```

```
// プロセスは、レジスタEFLAGSの値を自身のTSSセグメントに保存し、現在の
// CPUのEFLAGSは、プロセスが切り替わると変化します。sleep_on()を使用しているプロセスが
// sleepはwake_up()で明示的に目覚めさせる必要があります。
36 static inline void wait_on_buffer(struct buffer_head * bh) 37 {。
38      // ロックされていた場合、プロセスはスリープ状態になり、ロックが解除されるのを待ちます。
39          cli();                          // disable int.
40          while (bh->b_lock)
41                  sleep_on(&bh->b_wait);
42          sti();                          // enable int.
42 }
43
```

//// デバイスのデータ同期
// sync_inodes()はファイルinode.cの59行目で定義されています。
// この関数は、まずi-node同期関数を呼び出し、変更されたすべてのi-nodeを書き込みます。
メモリのi-nodeテーブルの//をバッファキャッシュに移す。その後、キャッシュバッファ全体がスキャンされます。
//となり、変更されたバッファブロックに対してライトディスクリクエストが発生して
// バッファリングされたデータがディスクに書き込まれることで、キャッシュ内のデータが同期して
// デバイスになります。

```
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
```

//// キャッシュされたデータを、指定されたデバイス上のデータと同期させる。
// この関数はまず、バッファキャッシュ内のすべてのバッファブロックを検索し、ディスクに書き込む
// 指定されたデバイスのバッファのデータが変更されているかどうか（同期型
//操作）を行います。そして、インメモリのi-nodeテーブルデータをバッファキャッシュに書き込みます。その後、実行する
// 指定されたデバイスに対して、上記と同じ書き込み操作を行う dev again.

```
61  59 int sync_dev(int dev) 60
    {。
62         int i;
63         struct buffer_head * bh;
63
```

// で指定されたデバイスに対して、データの同期操作を行います。
// パラメータを使用して、デバイス上のデータとバッファキャッシュ内のデータを同期させます。
// その方法は、キャッシュ内のすべてのバッファブロックをスキャンして、バッファブロックが
指定されたデバイス dev の // をロックし、ロックされている場合は、スリープしてロック解除を待つ。
// そして、そのバッファブロックがまだ指定されたデバイスのバッファブロックであるかどうかが判断されます。
// 変更されている（b_dirtフラグが設定されている）場合は、書き込み操作を行う
// を表示します。の間にバッファブロックが解放されたり、他の目的に使用されたりした可能性があるからです。
// のバッファブロックであるかどうかを再度確認する必要があります。

```
64     // 指定されたデバイスを選択してください。
65         bh = start_buffer;                      // points to the beginning of cache.
```

```
66          for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
67                  if (bh->b_dev != dev)         // ignore blocks that doesn't belong to dev.
68                          continue;
69                  wait_on_buffer(bh);           // wait for the buffer to be unlocked.
70                  if (bh->b_dev == dev && bh->b_dirt)
71                          ll_rw_block(WRITE,bh);
71          }
```

// そして、i-nodeのデータをバッファキャッシュに書き込み、inode_tableのinodeを同期させます。
// をバッファキャッシュの情報と照合します。その後、キャッシュ内のデータは再び同期されます
//を、更新後のデバイス内のデータで行うことができます。2パスの同期動作はこちら
カーネルの実行効率を向上させるために、//を使用しています。バッファ同期のファーストパスは

//は、メモリ内の多くの「ダーティブロック」をクリーンにすることができるので、同期動作の
// i-nodeを効率的に行うことができます。この2回目のバッファ同期では
// i-nodeのデータとの同期操作によってダーティになったバッファブロック
デバイスの中に //。

```
        sync_inodes();
        bh = start_buffer;
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                if (bh->b_dev != dev)
                        continue;
                wait_on_buffer(bh);
                if (bh->b_dev == dev && bh->b_dirt)
                        ll_rw_block(WRITE,bh);
        }
        return 0;
}

void inline invalidate_buffers(int dev) {
        int i;
        struct buffer_head * bh;

        bh = start_buffer;
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                if (bh->b_dev != dev)           // Ignore blocks that doesn't belong to dev.
                        continue;
                wait_on_buffer(bh);             // wait for the buffer to be unlocked.
                if (bh->b_dev == dev)
        bh->b_uptodate = bh->b_dirt = 0;  }
}

/*
 * This routine checks whether a floppy has been changed, and
 * invalidates all buffer-cache-entries in that case. This
 * is a relatively slow routine, so we have to try to minimize using
 * it. Thus it is called only upon a 'mount' or 'open'. This
 * is the best way of combining speed and utility, I think.
 * People changing diskettes in the middle of an operation deserve
 * to loose :-)
 *
 * NOTE! Although currently this is only for floppies, the idea is
 * that any additional removable block-device will use this routine,
 * and that mount/open needn't know that floppies/whatever are
 * special.
*/
```

//// ディスクが交換されたかどうかを確認し、交換された場合はバッファブロックを無効にします。

113 void check_disk_change(int dev)
114 {。

```
115         int i;
116
    // 最初に、それがフロッピーデバイスかどうかをチェックします。
    // .メディア そうでない場合は終了します。次に、フロッピーディスクが交換されたかどうかをテストし、
    そうでなければ終了します。このようにして
117 // 関数 floppy_change() は blk_drv/floppy.c の 139 行目にあります。
118         if (MAJOR(dev) != 2)
119                 return;
120         if (!floppy_change(dev & 0x03))
121                 return;
    // フロッピーディスクが交換されたので、i-nodeのビットマップで占められているバッファブロックと
    // 対応するデバイスの論理ブロックビットマップが解放され、デバイスのi-nodeが
122 // とデータブロック情報で占められていたキャッシュブロックが無効になります。
123         for (i=0 ; i<NR_SUPER ; i++)
124                 if (super_block[i].s_dev == dev)
125                         put_super(super_block[i].s_dev);
126         invalidate_inodes(dev);
127         invalidate_buffers(dev);
126 }
127
    // 以下は、ハッシュ関数の定義と、ハッシュテーブルエントリの計算マクロです。
    // ハッシュテーブルの主な役割は、要素を探すのにかかる時間を短縮することです。を確立することで
    // 要素の格納場所とキーワード（ハッシュ関数）を対応させる。
    // 関数の計算によって、指定された要素をすぐに直接見つけることができます。
    // ハッシュ関数を構築する際の指針となるのは、その確率が
    任意の配列アイテムへのハッシュの//は実質的に同じです。ハッシュを作成する方法はたくさんありま
    す。
    // 関数を使用しています。ここでは、Linux 0.12が最もよく使われるキーワードresidual remainderを主
    に使用しています。
    // メソッドを使用しています。探しているバッファブロックには2つの条件があるので、デバイス番号
    // 'dev'とバッファブロック番号'block'を含むように設計されたハッシュ関数は、これらを含む必要があ
    ります。
    // 2つのキーの値。ここでの2つのキーワードのXOR演算は、単なる計算方法の1つです。
    // のキーバリューを使用しています。キーの値にモジュロ演算(%)を行うことで、その値が
    関数で計算された // は，すべて関数の配列アイテムの範囲内にあります.
128 #define _hashfn(dev,block) (((unsigned)(dev^block))%NR_HASH)
129 #define hash(dev,block) hash_table[_hashfn(dev,block)] 。
130
    //// ハッシュキューとフリーリストからバッファブロックを削除します。
    // ハッシュキューはダブルリンクリスト構造、バッファブロックのフリーリストは
    // 双方向のサーキュラーリンクリスト構造です。
133 131 static inline void remove_from_queues(struct buffer_head * bh) 132 {。
134 /* remove from hash-queue */
135         if (bh->b_next)
136                 bh->b_next->b_prev = bh->b_prev;
137         if (bh->b_prev)
```

```
138                 bh->b_prev->b_next = bh->b_next;
    // バッファブロックがキューの最初のブロックである場合、対応するエントリの
139     // ハッシュテーブルは、キュー内の次のバッファブロックを指します。
140         if (hash(bh->b_dev,bh->b_blocknr) == bh)
141                 hash(bh->b_dev,bh->b_blocknr) = bh->b_next;
140 /* remove from free list */
141         if (!(bh->b_prev_free) || !(bh->b_next_free))
142                 panic("Free block list corrupted");
143         bh->b_prev_free->b_next_free = bh->b_next_free;
144         bh->b_next_free->b_prev_free = bh->b_prev_free;
```

145 // フリーリストのヘッダがこのバッファブロックを指している場合は、次のバッファブロックに誘導されます。
146 if (free_list == bh)
147 free_list = bh->b_next_free;
147 }
148
//// バッファブロックをフリーリストの最後に挿入し、ハッシュキューに入れます。
151 149 static inline void insert_into_queues(struct buffer_head * bh) 150 {。
152 /* put at end of free list */
153 bh->b_next_free = free_list;
154 bh->b_prev_free = free_list->b_prev_free;
155 free_list->b_prev_free->b_next_free = bh;
156 free_list->b_prev_free = bh;
157 /* put the buffer in new hash-queue if it has a device */
// なお、ハッシュテーブルのアイテムが初めて挿入される際には、hash()の値が
//は間違いなくNULLなので、161行目で取得したbh->b_nextも間違いなくNULLです。そこで、163行目で
// bh->b_next が NULL でない場合にのみ，b_prev に bh の値を割り当てるべきです．つまり，163番目の
// の行に、「if (bh->b_next)」という文を追加する必要があります。この誤りはカーネルの後に修正されました。
158 // 0.96バージョン
159 bh->b_prev = NULL;
160 bh->b_next = NULL;
161 if (!bh->b_dev)
162 return;
163 bh->b_next = hash(bh->b_dev,bh->b_blocknr);
164 hash(bh->b_dev,bh->b_blocknr) = bh;
165 bh->b_next->b_prev = bh; // "if (bh->b_next)" should be added before this.
164 }
165
//// ハッシュテーブルを使って、指定されたデバイスと指定されたブロック番号のバッファブロックを調べる
// バッファキャッシュ内の 見つかった場合はバッファブロックポインタを，そうでない場合は NULL を返します．
166 static struct buffer_head * find_buffer(int dev, int block) 167 {。
168 struct buffer_head * tmp;
169
170 // 指定されたデバイス番号とブロック番号のバッファブロックのハッシュテーブルを検索します。
171 for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b_next)
172 if (tmp->b_dev==dev && tmp->b_blocknr==block)
173 return tmp;
174 return NULL;
174 }
175
177 176 /*
178 * Why like this, I hear you say... The reason is race-conditions.
179 * As we don't lock buffers (unless we are readint them, that is),
180 * something might happen to it while we sleep (ie a read-error

```
181  * will force it bad). This shouldn't really happen currently, but
182  * the code is ready.
182 */
    //// ハッシュテーブルを使って、バッファブロックを探します。
    // ハッシュテーブルを使って、バッファキャッシュの中から指定されたバッファブロックを探します。
    見つかった場合は
    // ブロックがロックされ、ブロックヘッダポインタが返されます。
183 struct buffer_head * get_hash_table(int dev, int block) 184 {。
```

```
185         struct buffer_head * bh;
186
187         for (;;) {
    // 指定されたデバイスと指定されたブロックのバッファキャッシュを探します。
    // 見つからなかった場合は、NULLを返して終了します。
189                 if (!(bh=find_buffer(dev,block)))
190                         return NULL;
    // 目的のブロックが見つかれば、その参照カウントを1つ増やして、待機する
    ブロックがロックされている場合は）ロック解除のための //。スリープ状態を経ているので
    // ブロックの正しさを検証し、バッファヘッドポインタを返す必要があります。
    // スリープ中にバッファブロックのデバイス番号やブロック番号が変更された場合、その参照カウントは
    // が取り消され、再び検索されます。
192                 bh->b_count++;
193                 wait_on_buffer(bh);       // Wait for the buffer to be unlocked (if locked).
194                 if (bh->b_dev == dev && bh->b_blocknr == block)
195                         return bh;
196                 bh->b_count--;
195         }
196 }
197
198 /*
200  * Ok, this is getblk, and it isn't very clear, again to hinder
201  * race-conditions. Most of the code is seldom used, (ie repeating),
202  * so it should be much more efficient than it looks.
202 *
203 * The algoritm is changed: hopefully better, and an elusive bugremoved.
204 */
    // 以下のマクロでは、ブロックのモディファイドフラグとロックフラグを
205 // 同時に、修正フラグの重みがロックフラグよりも大きくなるように定義されています。
206 #define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)

    //// バッファキャッシュ内の指定されたブロックを取得します。
    // 指定された（デバイス番号＆ブロック番号）のバッファブロックがすでにキャッシュにあるかどうかをチェックします。
    // 指定されたブロックが既にバッファキャッシュにある場合は、対応するバッファブロックヘッダ
    // が返されて終了します。そうでない場合は、デバイスNo.とブロックNo.に対応する新しいエントリが作成されます。
    207       // がキャッシュに設定される必要があり、対応するバッファヘッダポインタが返されます。
208 struct buffer_head * getblk(int dev,int block)
207 {
208         struct buffer_head * tmp, * bh;
209
210 repeat:
    // ハッシュテーブルを検索し，指定されたブロックがすでにバッファキャッシュに存在する場合は，そのブロックを返します.
211     // 対応するブロックのヘッドポインタを表示して終了します。
212         if (bh = get_hash_table(dev,block))
```

```
213                 return bh;
    // そうでなければ，フリーリストをスキャンしてフリーブロックを探す。まず，'tmp'には
214     // フリーリストの最初のフリーブロックのヘッダーを、その後の操作を行うためにループします。
215         tmp = free_list;
216         do {
    // バッファブロックが使用されている場合（参照カウントが0になっていない場合）、スキャンを続ける
    // 次のアイテムを表示します。b_count=0のブロック、つまり、現在参照されていないブロックに対しては
    // をキャッシュに入れた場合、必ずしもクリーン（b_dirt=0）やアンロック（b_lock=0）ではない。したがって、我々は
    // といった判断や選択を続けていく必要があります。例えば、次のようなブロックの場合
```

// b_count=0の場合、タスクがブロックの内容を書き換えてからリリースすると、ブロックb_countのタスクがbreada()を実行して、いくつかのファイルを事前に読み込んだとき、 // はまだ0ですが、b_lockは0にはなりません。
// ブロックの場合は、ll_rw_block()が呼ばれるとすぐにb_countをデクリメントします。ただし、このときに
// ハードディスクのアクセス操作がまだ進行している可能性があるので、b_lock=1だが、b_count=0である。

```
218                 if (tmp->b_count)
219                         continue;
```

// バッファブロックヘッダポインタbhが空の場合、またはフラグの重み（修正。
現在のtmpが参照しているブロックの // lock) がbhのフラグの重さよりも小さい。
// ならば，bhはtmpのブロックヘッダを指している。もし，tmpが指すバッファブロックヘッダが
// バッファブロックが変更されず、ロックフラグも設定されていないことを示します（つまり、BADNESS()
// = 0）の場合は、指定されたデバイスで対応するキャッシュブロックが取得されたことを意味します。
// となり、ループは終了します。そうでなければ、このループを実行し続けて
// BADNESS()の値が最も小さいバッファを見つけます。

```
221                 if (!bh || BADNESS(tmp)<BADNESS(bh)) {
222                         bh = tmp;
223                         if (!BADNESS(tmp))
224                                 break;
221                 }
222 /* and repeat until we find something good */
223         } while ((tmp = tmp->b_next_free) != free_list);
```

// ループチェックで全てのバッファブロックが使用されていると判断された場合（全てのバッファブロックのヘッダー参照
// の数が >0）、sleep はバッファの空きブロックが利用可能になるのを待ちます。このプロセスは、明示的に
空いているバッファブロックが利用可能になると、 // ウォークアップされ、次のように先頭にジャンプします。

```
224     // 空きバッファブロックを再検索する関数です。
225         if (!bh) {
226                 sleep_on(&buffer_wait);
227                 goto repeat;                    // jump to line 210.
227         }
```

// ここで実行すると、適当な空きバッファブロックが見つかったことになります。その後、バッファのロックが解除されるのを待ちます
// have to repeat the above search process again.

```
228         wait_on_buffer(bh);
229         if (bh->b_count)                        // occupied again ??
230                 goto repeat;
```

// バッファが変更された場合、データをディスクに書き込み、バッファのロックが解除されるのを待つ
// を再び使用します。同様に、バッファが再び他のタスクによって使用された場合、上記の検索処理は
//を繰り返した。

```
231
232         while (bh->b_dirt) {
233                 sync_dev(bh->b_dev);
234                 wait_on_buffer(bh);
235                 if (bh->b_count)                // occupied again ??
236                         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 */
```

// スリープ中に指定されたバッファブロックがハッシュテーブルに追加されたかどうかをチェックします。

```
239
240         if (find_buffer(dev,block))
241                 goto repeat;
242 /* OK, FINALLY we know that this buffer is the only one of it's kind, */
243 /* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
```

// では、このバッファブロックを占有し、参照カウントを1にして、モディファイドフラグをリセットしましょう。

```
244         // そして、有効（更新）フラグ。
245         bh->b_count=1;
246         bh->b_dirt=0;
247         bh->b_uptodate=0;
            // バッファブロックのヘッダは，まずハッシュキューとフリーブロックリストから削除され，次に
            // バッファブロックは、指定されたデバイスとその上の指定されたブロックに使用されます。バッファは
            //ブロックは、フリーリストに再挿入され、ハッシュキューの新しい位置に置かれ、最後に
248         // バッファブロックのヘッダが返されます。
249         remove_from_queues(bh);
250         bh->b_dev=dev;
251         bh->b_blocknr=block;
252         insert_into_queues(bh);
253         return bh;
251 }
252
            //// 指定したバッファブロックを解放します。
            // バッファブロックのロックが解除されるのを待ちます。その後、参照カウントが1つデクリメントされて
            // 空きバッファブロックを待っているプロセスが明示的にアウェイクしている。
255 253 void brelse(struct buffer_head * buf) 254
{...
256         if (!buf)                       // Returns if the block head pointer is invalid.
257                 return;
258         wait_on_buffer(buf);
259         if (!(buf->b_count--))
260                 panic("Trying to free free buffer");
261         wake_up(&buffer_wait);
261 }
262
264 263 /*
265  * bread() reads a specified block and returns the buffer that contains
266  * it. It returns NULL if the block was unreadable.
266 */
            //// デバイスからデータブロックを読み込む。
            // この関数はまず、指定されたデバイス番号に基づいて、キャッシュ内のバッファブロックを要求します。
            // devとブロック番号のブロックです。バッファブロックに既に有効なデータが含まれている場合は、直接
            // バッファブロックのポインタを返します.
            // をバッファブロックに格納し、バッファブロックポインタを返します。
267 struct buffer_head * bread(int dev,int block) 268 {。
269         struct buffer_head * bh;
270
            // バッファのブロックをキャッシュに適用します．戻り値がNULLの場合は，カーネルの
            //にエラーが出て停止します。その後、利用可能なデータがあるかどうかを判断します。データがあるかどうかを返します。
```

```
271     バッファブロック内の//は有効（更新）なので、直接使用することができます。
272             if (!(bh=getblk(dev,block)))
273                     panic("bread: getblk returned NULL\n");
274             if (bh->b_uptodate)
275                     return bh;
        // それ以外の場合は、ブロックデバイスの読み取り/書き込み機能であるll_rw_block()を呼び出します。
        //を使用して、リードデバイスブロックリクエストを生成します。その後、指定されたデータブロック
        を待って
        // 読み込んで、バッファのロックが解除されるのを待ちます。スリープが解除された後、もしバッファ
        が
        // 更新された場合は、バッファヘッドポインタを返して終了します。それ以外の場合は、次のように示
        します。
        // デバイスの読み込み操作に失敗したので、バッファを解放し、NULLを返して終了する。
```

```
        ll_rw_block(READ,bh);
        wait_on_buffer(bh);
        if (bh->b_uptodate)
                return bh;
        brelse(bh);
        return NULL;
}

//// メモリブロックをコピーします。
// 1ブロック（1024バイト）のデータを、「from」アドレスから「to」位置にコピーします。
#define COPYBLK(from,to)
asm ("cld¥n¥t"
        "rep|n|t" \
        "movsl|n|t" \
        ::"c" (BLOCK_SIZE/4),"S" (from),"D" (to) \
        :"cx","di","si")

/*
 * bread_page reads four buffers into memory at the desired address. It's
 * a function of its own, as there is some speed to be got by reading them
 * all at the same time, not waiting for one to be read, and then another
 * etc.
*/
//// デバイス上の1ページ（4つのバッファブロック）を、指定されたメモリアドレスに読み込みます。
// パラメータの'address'はページデータを保存するアドレス、'dev'は指定した
// b[4]は4つのデバイスデータブロック番号を含む配列です。この関数は
void bread_page(unsigned long address,int dev,int b[4])
```

```
296
297 {
298         struct buffer_head * bh[4];
299         int i;
300
    // This function is executed 4 times. According to the 4 block numbers placed in the array b[],
    // a page is read from the device dev and placed at the specified memory location 'address'.
    // For the valid block number given by the parameter b[i], the function first gets the buffer
    // block of the specified device and block number from the buffer cache. If the data in the
    // buffer block is invalid (not updated) then a read device request is made to read the
    // corresponding data block from the device. For the invalid block number b[i], we don't have
    // to deal with it. Therefore, this function can freely read 1-4 data blocks according to the
    // block number in the specified b[].
301         for (i=0 ; i<4 ; i++)
302                 if (b[i]) {                     // If the block number is valid.
303                         if (bh[i] = getblk(dev,b[i]))
304                                 if (!bh[i]->b_uptodate)
305                                         ll_rw_block(READ,bh[i]);
306                 } else
307                         bh[i] = NULL;
    // The contents of the 4 buffer blocks are then sequentially copied to the specified address.
    // Before we can copy (use) the buffer block, we need to sleep and wait for the buffer block
    // to be unlocked (if it is locked). In addition, because we may have slept, we also need to
    // check if the data in the buffer block is valid before copying. We also need to release the
    // buffer block after copying.
308         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  */
    //// Reads some of the specified blocks from the specified device.
    // The number of parameters is variable and is a series of specified block numbers. When the
    // function succeeds, it returns the block header of the first block, otherwise it returns NULL.
322 struct buffer_head * breada(int dev,int first, ...)
323 {
324         va_list args;
325         struct buffer_head * bh, *tmp;
326
    // まず、変数パラメータリストの最初のパラメータ（ブロック番号）を取ります。次に
    // 指定されたデバイスとブロック番号のブロックをキャッシュから削除します。もし、バッファブロックデータが
327     // 無効（更新フラグが設定されていない）の場合は、リードデバイスデータブロック要求を発行します。
328         va_start(args,first);
329         if (!(bh=getblk(dev,first)))
330                 panic("bread: getblk returned NULL\n");
331         if (!bh->b_uptodate)
332                 ll_rw_block(READ,bh);
    // 次に、変数パラメータリストの他の事前に読み込まれたブロック番号を順番に取り、次のようにします。
    // と同じですが、使用しないでください。なお、336行目にバグがあります。ここでは、bhが
    // be tmp. このバグは0.96のカーネルコードまで修正されませんでした。さらに、この
    // は事前に読み込まれた後続のデータブロックで、キャッシュに読み込まれるだけで使用されません。
    // そのため、337行目のステートメントでは、その参照カウントをデクリメントしてリリースする必要があります。
    //      while ((first=va_arg(args,int))>=0) {
    ブロックのことです（getblk()関数で
```

はブロックの参照カウントが増えるため）。

```
332
333                 tmp=getblk(dev,first);
334                 if (tmp) {
335                         if (!tmp->b_uptodate)
336                                 ll_rw_block(READA,bh);     // here 'bh' should be 'tmp'.
337                         tmp->b_count--;                    // release the pre-read block.
338                 }
339         }
```

// この時点で、変数パラメータテーブルのすべてのパラメータが処理されます。そして、次のように待ちます。

// ロックが解除された最初のバッファブロック（ロックされていた場合）。待ち時間が終わると、もし

// バッファブロック内のデータがまだ有効であれば、バッファブロックのヘッダポインタが返されます。

//　　va_end(args);

そうでなければ、バッファは解放され、NULLを返します。

```
340
341         wait_on_buffer(bh);
```

```
342         if (bh->b_uptodate)
343                 return bh;
344         brelse(bh);
345         return (NULL);
```

```
346 }
347
    //// バッファ初期化機能。
    // パラメータ buffer_end は、バッファキャッシュメモリの終端です。16MBのシステムでは
    //メモリの場合、バッファキャッシュエンドは4MBに設定されます。8MBのメモリを持つシステムでは、
    バッファエンドの
    //には2MBが設定されています。この関数は，ブロックヘッダ構造体を設定（初期化）し，対応する
    // バッファキャッシュの開始位置であるstart_bufferとbuffer_endからのデータブロック
    バッファキャッシュのすべてのメモリが割り当てられるまで、それぞれのバッファの最後に//をつける。
350 348 void buffer_init(long buffer_end) 349
    {。
351         struct buffer_head * h = start_buffer;
352         void * b;
353         int i;
353
    // まず、ハイエンド位置に基づいて、実際のバッファのハイエンド位置'b'を決定します。
    パラメータで指定されたバッファの//。バッファの上限が1Mbに等しい場合は、次のようになります。
    // ビデオメモリーやBIOSが640KB～1MBの範囲で使用されている場合、実際に使用できるバッファメ
    モリーは
    // は、ハイエンドで640KBとします。それ以外の場合は，バッファキャッシュメモリの上限が
    // 1MBを超えています。
354         if (buffer_end == 1<<20)
355                 b = (void *) (640*1024);
356         else
357                 b = (void *) buffer_end;
    // 以下のコードは、バッファキャッシュを初期化し、フリーブロックのサーキュラーリストを作成する
    ために使用されます。
    // とし、システム内のブロック数を取得します。演算処理は、バッファを分割して
    // バッファの最上位から1KBサイズのブロックを構築し、同時に
    // バッファの下端にあるバッファブロックを記述して、buffer_headを形成する
    // を二重に連結したリストにします。
    // 'h'はブロックヘッダ構造体へのポインタ、'h+1'は次のブロックヘッダアドレス
    メモリアドレスを連続して指している//の最後を指しているとも言えます。
    // h ブロックヘッダ。ブロック・ヘッダを格納するのに十分なメモリを確保するために
    // 構造体の場合、'b'が指すメモリブロックのアドレスは、以下の値以上である必要があります。
358 // 'h'のブロックヘッダの終わりに等しい、つまり、'b >= h+1'が必要です。
359         while ( (b -= BLOCK_SIZE) >= ((void *) (h+1)) ) {
360                 h->b_dev = 0;               // device no.
361                 h->b_dirt = 0;              // dirt flag (block modified flag).
362                 h->b_count = 0;             // block reference count.
363                 h->b_lock = 0;              // block lock flag.
364                 h->b_uptodate = 0;          // Block update flag (or data valid flag).
365                 h->b_wait = NULL;           // block waiting queue.
365                 h->b_next = NULL;       // points to next header with same hash value.
366                 h->b_prev = NULL;       // points to previous header with same hash value.
```

```
367                 h->b_data = (char *) b; // block data pointer.
368                 h->b_prev_free = h-1;   // point to the previous item in the free-list.
369                 h->b_next_free = h+1;   // point to the next item in the free-list.
370                 h++;                    // h points to next new buffer header location.
371                 NR_BUFFERS++;                   // counting buffer blocks.
    // If b is decremented to 1MB, skip 384KB (video memory) and let b point to 0xA0000 (640KB).
372                 if (b == (void *) 0x100000)
373                         b = (void *) 0xA0000;
374         }
    //次に，h に最後の有効なバッファブロックヘッダを指定し，フリーリストヘッダには
    //最初のバッファブロックのヘッダ；フリーリストのヘッダのb_prev_freeは，前の
```

```
//hの次のポインタは、最初のヘッダを指すので
//フリーリストは双方向のリング構造を形成します。最後に，ハッシュテーブルを初期化し，すべての
テーブル内の//ポインタをNULLにします。
        h--;                            // h points to the last block header.

        free_list = start_buffer;       // free list header points to first block header.
        free_list->b_prev_free = h;
        h->b_next_free = free_list;
        for (i=0;i<NR_HASH;i++)
                hash_table[i]=NULL;
}
```

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.

super.cプログラムは、主にファイルシステムのスーパーブロックへのアクセスと管理のための関数を含んでいます。bitmap.cプログラムは、ファイルシステムの論理ブロックビットマップとi-nodeビットマップを処理するために使用されます。truncate.cプログラムは、ファイルデータの長さをゼロにカットする関数truncate()のみを含んでいます。inode.cプログラムは、主にファイルシステムのi-node情報のアクセスと管理に関係しています。namei.cプログラムは、主に与えられたファイルパス名からその対応するi-node情報を見つけてロードする機能を完了するために使用されます。

12.3.1 ファイルシステムの機能部分の順序からすると、上記のプログラムの順序で説明すべきですが、super.cプログラムには、ファイルシステムのロード/アンロードやシステムコールに関するいくつかの高レベルの関数も含まれているため、それらはいくつかの他のプログラムの関数を使用する必要がありますので、inode.cプログラムを紹介した後で説明します。

12.3.2 Function

bitmap.cプログラムの目的と機能はシンプルで明確です。主に、ファイルシステムの論理ブロックやi-node構造の使用状況に応じて、論理ブロックビットマップやi-nodeビットマップのビットを占有/解放するために使用されます。論理ブロックビットマップの操作関数はfree_block()とnew_block()で、i-nodeビットマップの操作関数はfree_inode()とnew_inode()です。

関数 free_block() は、指定されたデバイスのデータ領域にある論理ブロックを解放するために使用します。具体的な動作としては、指定された論理ブロックに対応する論理ブロックビットマップのビットをリセットします。まず、指定されたデバイスのスーパーブロックを取得し、スーパーブロックに記載されているデバイスデータの論理ブロックの範囲に応じて、論理ブロック番号の有効性を判断します。次に、指定された論理ブロックがバッファキャッシュの中にあるかどうかを確認します。Yesの場合、対応するバッファブロックを解放する。次に、データゾーンの先頭から（1から数えて）データの論理ブロック番号を計算し、論理ブロックのビットマップを操作して対応するビットをリセットする。最後に、対応する論理ブロックビットマップを含むバッファブロックにおいて、論理ブロック番号に応じてモディファイドビットフラグを設定する。

new_block()関数は、ブロックデバイスに論理ブロックを要求し、論理ブロック番号を返し、そのブロックに対応する論理ブロックビットマップビットを設定するために使用されます。まず、指定されたデバイスdevのスーパーブロックを取得します。次に、論理ブロックのビットマップ全体を検索して、最初に0になるビットを探します。

ディスク装置のスペースがなくなった場合、この関数は0を返します。それ以外の場合は、最初に見つかった0ビットの位置に1が設定され、対応するデータ論理ブロックが占有されていることを示します。同時に、このコードは、そのビットを含む論理ブロックのビットマップが配置されているバッファブロックのモディファイドフラグを設定します。次に、データ論理ブロックのディスクブロック番号を計算し、対応するバッファブロックをバッファキャッシュに適用し、バッファブロックをクリアする。次に、このバッファブロックの更新フラグと修正フラグを設定する。最後に、他のプログラムが使用できるようにバッファブロックを解放し、ブロック番号（論理ブロック番号）を返す。

12.3.3 関数free_inode()は、指定されたi-nodeを解放し、対応するi-nodeビットマップビットをリセットするために使用されます。New_inode()は、デバイスに新しいi-nodeを作成し、新しいi-nodeへのポインタを返すために使用されます。主な演算処理は、メモリのi-nodeテーブルのアイドルi-nodeエントリを取得し、i-nodeビットマップからアイドルi-nodeを見つけることである。これら2つの関数の処理は、上記2つの関数と同様であるため、ここでは説明を省略する。

12.3.4 Code annotation

プログラム 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 */
   // <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
   // <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_structや
   //     data of the initial task 0, and some embedded assembly function macro statements
   //     about the descriptor parameter settings and acquisition.
   // <linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。
   //     commonly used functions of the kernel.
8  #include <string.h>                // The memset() function is used here.
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h>
12
   //// 指定されたアドレス（addr）の1024バイトのメモリブロックをクリアします。
   // 入力： eax = 0; ecx = ブロックの長さをロングワードで表したもの (BLOCK_SIZE/4); edi = ブロックの長さを指定します。
13 // 開始アドレス addr.
14 #define clear_block(addr) \
15 __asm  ("cld\n\t" \                // clear direction.
16          "rep\n\t" \                // repeated execution of stored data (0).
17          "stosl" \
17          ::"a" (0), "c" (BLOCK_SIZE/4), "D" ((long) (addr)):"cx", "di")
18
   //// 指定されたアドレスから始まる nr 番目のビットオフセットのビットを設定します。
```

// 'nr'は32より大きくてもOK! このマクロは、元のビット値を返します。
// 入力： %0 -eax (戻り値); %1 -eax(0); %2 -nr, ビットオフセット; %3 -(addr), addrの内容。
// 20行目では、ローカルレジスタ変数'res'を定義しています。この変数は、指定された
// EAXレジスターを使用することで、効率的なアクセスと操作が可能になります。この変数の定義方法は、主に
// インラインのアセンブリプログラムで使用されます。マクロ全体はステートメント式で、その値は
// は、最後の「res」の値です。
// 21行目のBTSL命令は、ビットのテストとセットに使用されます。指定されたビット値を保存します。
// ベース・アドレス（%3）とビット・オフセット（%2）をキャリー・フラグCFにセットして

// SETB命令は、キャリーフラグに応じてオペランド（%al）を設定するために使用されます。
19 // CFです。CF=1の場合は%al =1、そうでない場合は%al =0となります。
20 #define set_bit(nr,addr) ({\
21 register int res asm ("ax"); \
22 __asm__ volatile ("bts1 %2,%3\n\tsetb %%al": \
22 "=a" (res):"" (0), "r" (nr), "m" (*(addr))); \
23 res;})
24
//// 指定されたアドレスの先頭からnrビット目のオフセットのビットをリセットします。
// 元のビット値の逆数を返します。
// 入力： %0 -eax(戻り値); %1 -eax(0); %2 -nr, ビットオフセット; %3 -(addr), addrの内容。
// 27行目のBTRL命令は、ビットのテストとリセットに使用されます。コマンドSETNBが使用される
// キャリーフラグCFに応じてオペランド(%al)を設定します。CF = 1の場合、%al = 0、そうでない場合は%al
// = 1.
25 #define clear_bit(nr,addr) ({¥ 26
register int res asm ("ax"); ^ω^)
27 asm volatile ("ctrl %2,%3¥¥tsetnb %%al": ¥ 28 "=a"
(res):"" (0), "r" (nr), "m" (*(addr)); ^ω^)
29 res;})
30
//// アドレス「addr」から最初のゼロビットを探し、「addr」からのビットオフセットを返します。
// 入力： %0 - ecx (戻り値); %1 - ecx(0); %2 - esi(addr).
// 「addr」で指定されたアドレスから始まるビットマップの中で、最初に0になるビットを探します。
// と、アドレス addr からのビットオフセットを返します。addr」はのデータ領域のアドレスです。
// バッファブロックで、スキャンの範囲は1024バイト（8192ビット）です。
// 36行目のBSFL命令は、EAXレジスタの最初の非ゼロビットをスキャンするために使用されます。
33 // そのオフセットを EDX に配置します。
31 #define find_first_zero(addr) ({ ** 32
int res; })
34 __asm ("cld\n" \ // clear direction flag.
35 "1:\tlods1\n\t" \ // obtain [esi] -> eax
36 "not1 %%eax\n\t" \ // inverse each bits in eax
37 "bsf1 %%eax,%%edx\n\t" \ // offset of first non-zero bit -> edx
38 "je 2f\n\t" \ // jump forward to label 2 if eax = 0.
39 "add1 %%edx,%%ecx\n\t" \ // offset is added to ecx (first 0 bit in bitmap).
40 "jmp 3f\n" \ // jump forward to label 3 (end).
41 "2:\tadd1 $32,%%ecx\n\t" \ // ecx adds 32 bit offset if bit 0 not found.
41 "cmp1 $8192,%%ecx\n\t" \ // scanned 8192 bits (1024 bytes)?
42 "j1 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
//// ロジックブロックをデバイスのデータゾーンにフリーにします。
// 指定された論理ブロックに対応する論理ブロックビットマップのビットをリセットします。
// 成功すれば1を、そうでなければ0を返します。

49 47 int free_block(int dev, int block) 48 {。

50 struct super_block * sb;

51 struct buffer_head * bh;

51

// まず、デバイスdev上のファイルシステムのスーパーブロック情報を取得して

// パラメーターブロックの有効性は、データブロックスタートの論理ブロックに従ってチェックされます。

// の数と、ファイルシステム内の論理ブロックの総数を表しています。指定されたデバイスが
// スーパーブロックが存在しない場合はエラーになります。論理ブロック番号が
// ディスク上のデータゾーンの最初の論理ブロックのブロック番号、またはそれ以上
52 // デバイス上の論理ブロックの合計数を超えた場合、システムも停止します。

```
53          if (!(sb = get_super(dev)))                 // in fs/super.c, line 56.
54                  panic("trying to free block on nonexistent device");
55          if (block < sb->s_firstdatazone || block >= sb->s_nzones)
56                  panic("trying to free block not in datazone");
57          bh = get_hash_table(dev,block);
```

// 次に、ハッシュテーブルからブロックデータを探します。バッファブロックが見つかった場合、その有効性
//がチェックされます。このとき、参照数が1以上であれば、それは
// 他の誰かがバッファブロックを使用しているので、brelse()が呼び出され、b_countがデクリメントされます。
// を1だけ増やして終了します。そうでなければ、修正フラグと更新フラグをクリアして、データを解放します。
// ブロックを作成します。このコードの主な目的は、現在のロジックブロックが
58 // がバッファキャッシュに存在する場合は、対応するバッファブロックが解放されます。

```
59          if (bh) {
60                  if (bh->b_count > 1) {
61                          brelse(bh);         // ret after b_count--, still used by others.
62                          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          }
```

// そして、ロジックブロックのビットマップにあるブロックのビットをリセット（0に設定）します。最初に計算する
// データゾーンから始まるブロックの論理ブロック番号（1から数えて）。が表示されます。
// ロジックブロックのビットマップを操作して、対応するビットをリセットします。もし、そのビットがすでに
// 0であれば、システムが故障していることを意味し、停止します。1ブロックは1024バイトなので、8192ビットになります。
// これにより、「block / 8192」は、論理ビットマップ内のどのブロックが指定されたブロックであるかを計算することができます。
// と 'block & 8191 ' は、現在のブロックのビットオフセット位置を得ることができます。
67 // ロジックブロックのビットマップです。

```
68          block -= sb->s_firstdatazone - 1 ;          // block = block - (s_firstdatazone -1);
69          if (clear_bit(block&8191,sb->s_zmap[block/8192]->b_data)) {
70                  printk("block (%04x:%d) ",dev,block+sb->s_firstdatazone-1);
71          printk("free_block: bit already cleared\n"); 71     }
```

72 // 最後に、ロジックブロックのビットマップが配置されているバッファブロックのモディファイドフラグを設定します。

```
73          sb->s_zmap[block/8192]->b_dirt = 1;
```

```
74          return 1;
74 }
75
    //// デバイスに論理ディスクブロックを要求します。
    // この関数は、まずデバイスのスーパーブロックを取得し、最初のゼロ値を探します。
    論理ブロックのビットマップでは、 // （フリーブロックを表す）が表示されます。このビットは、次のように設定されます。
    // 対応する論理ブロックの取得が期待されます。そして、対応するバッファ
    // のブロックがロジックブロック用のバッファに取得されます。最後に、バッファブロックがクリアされます。
    // その更新フラグと修正フラグがセットされ、論理ブロック番号が返されます。もし
    // 実行が成功すると，この関数は，論理ブロック番号（ディスクブロック
    // 数）を、そうでなければ0を返します。
76 int new_block(int dev)
77 {。
```

```
78          struct buffer_head * bh;
79          struct super_block * sb;
80          int i,j;
81
    // まず、デバイスのスーパーブロックを取得します。次に、ファイルシステムの8つの論理ブロックのビットマップをスキャンします。
    // 最初の0値ビットを探し、フリーロジックブロックを見つけ、ブロック番号を取得します。
    // 論理ブロックが配置されます。8ブロックの論理ブロックのビットマップの全てのビットがスキャンされると
    // (i >= 8 or j >= 8192)の場合、0値のビットが見つからないか、ビットマップのバッファ・ブロック・ポインタが
82  // が無効(bh = NULL)の場合は0で終了します(空きロジックブロックがない)。
83          if (!(sb = get_super(dev)))
84          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;
    // 次に，見つかった新しい論理ブロックjに対応する論理ブロックビットマップのビット
    // がセットされています。対応するビットが既に設定されている場合は、システムにエラーが発生していることを示す
    // と停止します。それ以外の場合は、以下を格納する対応するバッファブロックの修正フラグを設定します。
    // ビットマップを表示します。ロジックブロックのビットマップは、ロジックブロックの占有率を示すだけなので
    ディスク上のデータゾーンの//、つまりロジックブロックビットマップのビットオフセット値が示すのは
    // データゾーンの先頭からのブロック番号なので、論理ブロック番号の
    // jを論理ブロック番号に変換するために、データゾーンの最初のブロックをここに追加する必要があります。
    // 新しい論理ブロック番号がシステム上の論理ブロックの合計数よりも大きい場合は
91  // デバイスにブロックが存在しません。アプリケーションが失敗したので、0を返して終了します。
92          if (set_bit(j,bh->b_data))
93                  panic("new_block: bit already set");
94          bh->b_dirt = 1;
95          j += i*8192 + sb->s_firstdatazone-1;
96          if (j >= sb->s_nzones)
97                  return 0;
    // そして、指定された論理ブロック番号のバッファキャッシュにバッファブロックを取得する
    // をデバイス上に表示し、バッファブロックへッダポインタを返します。なぜなら，論理ブロック
    // 取得しただけでは、参照カウントが1（getblk()で設定）でなければならず、1でない場合は
    //停止します。最後に、新しい論理ブロックがクリアされ、その更新フラグと修正フラグが
98  // を設定します。その後、対応するバッファブロックを解放し、論理ブロック番号を返します。
99          if (!(bh=getblk(dev,j)))
```

```
100                 panic("new_block: cannot get block");
101         if (bh->b_count != 1)
102                 panic("new block: count is != 1");
103         clear_block(bh->b_data);
104         bh->b_uptodate = 1;
105         bh->b_dirt = 1;
106         brelse(bh);
107         return j;
106 }
107
```

//// 指定したi-nodeをリリースします。

// この関数は、最初に

// パラメーターです。i-nodeがまだ使用されている場合、それを解放することはできません。そのときは、i-nodeのビットマップ

//は、スーパーブロック情報、i-nodeのビットマップの対応するビットを使用して操作されます。

```
void free_inode(struct m_inode * inode)
108
109 {
110         struct super_block * sb;
111         struct buffer_head * bh;
112
// まず、リリースする必要のあるi-nodeの有効性や正当性を判断します。もし、そのi-nodeが
// i-nodeポインタ＝NULLの場合は終了します。i-nodeのデバイス番号フィールドが0であれば、ノード
//は使用しません。そして、対応するi-nodeが占有しているメモリ領域をクリアし、リターンします。
// Memset()はファイルinclude/string.hの395行目で定義されていますが、これはメモリ領域
113     inodeポインタで指定された//は0で埋められ、sizeof(*inode)バイトが埋められます。
114         if (!inode)
115                 return;
116         if (!inode->i_dev) {
117                 memset(inode,0,sizeof(*inode));
118                 return;
118         }
// このi-nodeに他のプログラム参照がある場合、それはリリースできないことを示しています。
// カーネルコードに問題があり、システムがダウンしている場合。もし、ファイルリンクの数が
// they should not be released, but should be put back.
```

```
119         if (inode->i_count>1) {
120                 printk("trying to free inode with count=%d\n",inode->i_count);
121                 panic("free_inode");
122         }
123         if (inode->i_nlinks)
124                 panic("trying to free inode with links");
```

// i-nodeの有効性をチェックした後、i-nodeのビットマップを使用して操作を開始します。
// そのスーパーブロックの情報です。まず、i-nodeがあるデバイスのスーパーブロックを取ります。
// を配置し、デバイスが存在するかどうかをテストします。その後、i-nodeの範囲があるかどうかを判断します。
//の数が正しいことを示しています。i-nodeの番号が0に等しいか、または合計数よりも大きい場合は
// デバイス上のi-nodeはエラーになります（No.0のi-nodeは予約されています）。もし、ノードビットマップ
i-nodeに対応する//が存在しない場合はエラーになります。のi-nodeビットマップが存在しないので
// バッファブロックは8192ビットなので、i_num>>13（つまり、i_num/8192）は、s_imap[]エントリを取得することができます。

```
125     // 現在のi-node番号、つまりディスクブロックです。
126         if (!(sb = get_super(inode->i_dev)))
127                 panic("trying to free inode on nonexistent device");
128         if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
129                 panic("trying to free inode 0 or nonexistant inode");
130         if (!(bh=sb->s_imap[inode->i_num>>13]))
131                 panic("nonexistent imap in superblock");
```

// ここで、i-nodeに対応するノードビットマップのビットをリセットします。もし、そのビットがすでに
// が0になると，エラー警告メッセージが表示されます。最後に，i-nodeが入っているバッファの
// ビットマップの位置が変更され、i-node構造体が占有するメモリ領域が

```
132     // がクリアされます。
133         if (clear_bit(inode->i_num&8191,bh->b_data))
134                 printk("free_inode: bit already cleared.\n\r");
135         bh->b_dirt = 1;
136         memset(inode,0,sizeof(*inode));_
135 }
136
```

//// デバイスの新しいi-nodeを作成し、初期化して、新しいi-nodeへのポインタを返します。
// メモリのi-nodeテーブルでフリーのi-nodeエントリを取得し、i-nodeからフリーのi-nodeを見つける

```
// bitmap.
struct m_inode * new_inode(int dev)
{
        struct m_inode * inode;
        struct super_block * sb;
        struct buffer_head * bh;
        int i,j;

// First, obtain a free i-node entry from the in-memory i-node table (inode_table) andread
// the super block structure of the specified device. Then scan the 8-block i-node bitmap in
// the super block, look for the first 0-bit (look for a free node), and obtain the node number
// where the i-node is placed. If the i-node is not found after the scan, or the buffer block
// where the bitmap is located is invalid (bh = NULL), then the i-node in the previously requested
// i-node table is returned, and the null pointer is returned and exited (indicating no Idle
// i-node available).
        if (!(inode=get_empty_inode()))         // fs/inode.c, line 197.
                return NULL;
        if (!(sb = get_super(dev)))             // fs/super.c, line 56.
                panic("new_inode with unknown device");
        j = 8192;
        for (i=0 ; i<8 ; i++)
                if (bh=sb->s_imap[i])
                        if ((j=find_first_zero(bh->b_data))<8192)
                                break;
        if (!bh || j >= 8192 || j+i*8192 > sb->s_ninodes) {
                iput(inode);
                return NULL;
        }
// 使用されていないi-node番号jを見つけたので、対応する
jに対応するi-nodeのビットマップの//ビット（既に設定されている場合は
// カーネルが故障している）。次に，i-nodeビットマップがあるバッファブロックのmodifiedフラグを
設定する。
//が配置されます。最後に、i-node構造が初期化されます（i_ctimeは、コンテンツの
i-nodeの//を変更する）。
        if (set_bit(j,bh->b_data))
                panic("new_inode: bit already set");
        bh->b_dirt = 1;
        inode->i_count=1;                       // reference count.
        inode->i_nlinks=1;                      // number of file directory entry links.
        inode->i_dev=dev;                       // device number the i-node is located.
        inode->i_uid=current->euid;             // user id of the i-node.
        inode->i_gid=current->egid;             // group id.
        inode->i_dirt=1;                        // set the modified flag.
        inode->i_num = j + i*8192;              // Corresponds to i-node nr in the device.
        inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
        return inode;
}
```

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.

図12-23 i-nodeの論理ブロック接続の模式図

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

プログラム 12-3 linux/fs/truncate.c

```
/*
 * linux/fs/truncate.c
 *
 * (C) 1991  Linus Torvalds
 */

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
```

```
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
//      and constants.
#include <linux/sched.h>.

#include <sys/stat.h> (日本語)

//// 全ての1間接ブロックを解放する。
// パラメータ dev はファイルシステムのデバイス番号、block は論理ブロック番号。
// 成功すれば1を、そうでなければ0を返します。
        struct buffer_head * bh;

        unsigned short * p;
```

```
15          int i;
16          int block_busy;                         // The logic block has not been released.
17
    // まず、パラメータの有効性を判断します。論理ブロック番号が0の場合に返します。
    // その後、プライマリブロックが読み込まれ、その上で使用されていたすべての論理ブロックが解放され、さらに
    // この一次間接ブロックのバッファブロックが解放されます。関数 free_block() は
    // デバイス上の指定された論理ブロック番号のディスクブロックを解放するために使用される (fs/bitmap.c
18  // 47行目）。
19          if (!block)
20                  return 1;
21          block_busy = 0;
22          if (bh=bread(dev,block)) {
23                  p = (unsigned short *) bh->b_data;  // points to the block data area.
24                  for (i=0;i<512;i++,p++)             // a block can contains 512 blcok nr.
25                          if (*p)
26                                  if (free_block(dev,*p)) {
26                                          *p = 0;
27                                          bh->b_dirt = 1;    // set the modified flag.
28                                  } else
29                                          block_busy = 1;    // block is not released.
30          brelse(bh);             // release the buffer block occupied by the indirect block.
31          }
    // 最後にデバイス上の間接ブロックを解放します。しかし、論理ブロックがあっても
            if (block_busy)
32
33                  return 0;
34          else
35                  return free_block(dev,block);       // 1 if successful, otherwise return 0.
36  }
37
    //// すべてのセカンダリ間接ブロックを解除します。
```

```
// パラメータdevはデバイス番号、'block'はセカンダリの論理ブロック番号
// 間接的なブロックです。
static int free_dind(int dev,int block) {。
        struct buffer head * bh;
        unsigned short * p;
        int i;
```

```
44          int block_busy;                             // The logic block has not been released.
44
    // まず、パラメータの有効性を判断します。論理ブロック番号が0の場合に返します。
    // その後、コードは2次間接ブロックの第1レベルのブロックを読み込み、すべての
    // その上の論理ブロックが使用されたことを示し、バッファブロックを解放します。
45  第一階層のブロックの//。
46          if (!block)
47                  return 1;
48          block_busy = 0;
49          if (bh=bread(dev,block)) {
50                  p = (unsigned short *) bh->b_data;  // points to the block data area.
51                  for (i=0;i<512;i++,p++)             // 512 second-level blocks
52                          if (*p)
53                                  if (free_ind(dev,*p)) {  // release primary indirect block.
53                                  *p = 0;
54                                          bh->b_dirt = 1;  // set modified flag.
55                                  } else
56                                          block_busy = 1;  // block not released.
57          brelse(bh);             // release buffer block occupied by secondary indirect block.
58          }
    // 最後に、デバイス上の2次間接ブロックが解放されます。しかし、もし、ロジック
59  // 解放されていないブロックは0（失敗）を返します。
60          if (block_busy)
61                  return 0;
62          else
63                  return free_block(dev,block);
63 }
64
    //// ファイルデータを切り捨てます。
    // ノードに対応するファイルの長さが0になり，占有されているデバイス空間が
    // をリリースしました。
67  65 void truncate(struct m_inode * inode) 66 {。
68          int i;
69          int block_busy;
69
    // まず、指定されたi-nodeの有効性を判断します。通常のファイルでない場合は返します。
    // ディレクトリ、またはリンクのエントリです。その後、i-nodeの7つのダイレクトブロックを解放し、
    7つの論理的なすべての
70  // ブロックの項目をゼロにする。ブロックがビジー状態でリリースされていない場合は、block_busy フ
    ラグが設定されます。
71          if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode) ||
72               S_ISLNK(inode->i_mode)))
73                  return;
74 repeat:
75          block_busy = 0;
75          for (i=0;i<7;i++)
76                  if (inode->i_zone[i]) {                 // release if block nr is not zero.
77                          if (free_block(inode->i_dev,inode->i_zone[i]))
78                                  inode->i_zone[i]=0;
```

```
79                          else
80                  block_busy = 1; // set if not released. 81 }
82          if (free_ind(inode->i_dev,inode->i_zone[7]))  // release primory indirect blocks.
83                  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;
```

// その後、i-node modifiedフラグが設定され、まだ論理ブロックがある場合は
// "busy "のためにリリースされない場合、現在のプロセス実行中のタイムスライスを0に設定して切り替える
//を別のプロセスに渡して実行させ、しばらく待ってからリリース操作を再実行します。
// 最後に、ファイルの修正時間とi-nodeの変更時間が現在の時間に設定されます。
// マクロ CURRENT_TIME は、ヘッダファイル linux/sched.h の 142 行目で定義されており、以下のように定義されています。

```
90      // 1970:0:0:0以降の秒数値を(startup_time + jiffies/HZ)とする。
91          inode->i_dirt = 1;
92          if (block_busy) {
93                  current->counter = 0;           // current process time slice is set to 0.
94                  schedule();
95                  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

iget()関数は、デバイスdevから指定されたノード番号nrのi-nodeを読み出し、そのノードの参照カウントフィールド値i_countを1つインクリメントするための関数である。その動作フローを図12-24に示す。この関数は、まずパラメータdevの有効性を判断し、i-nodeテーブルからアイドルのi-nodeを取り出し、次にi-nodeテーブルをスキャンして、指定されたノード番号nrのi-nodeを見つけ、そのi-nodeの参照番号をインクリメントする。現在スキャンしているデバイスが指定されたデバイスではない場合、またはノード番号が指定されたノードと等しくない場合、スキャンを続ける。そうでなければ、指定されたデバイス番号とノード番号を持つi-nodeが見つかり、そのノードがロック解除され

るのを待っている（もしも今ロックされているならば）。i-nodeがアンロックされるのを待っている間に、ノードテーブルが変更される可能性があります。そのため、現在、i-nodeのデバイス番号が指定されたデバイス番号と等しくない場合や、ノード番号が指定されたノード番号と等しくない場合は、i-nodeテーブル全体を再度スキャンする必要がある。続いて、i-nodeの参照カウント値を1インクリメントし、i-nodeが他のファイルシステムのマウントポイントであるかどうかを判断します。

図12-24 iget()関数の動作フローチャート

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.

i-nodeが他のファイルシステムのインストールポイントでない場合は、対応するi-nodeが見つかったことを示すので、一時的に適用されているアイドルi-nodeはこの時点で破棄でき、見つかったi-nodeポインタが返されます。

指定されたi-nodeがi-nodeテーブルに見つからない場合は、前のアプリケーションのアイドルi-node

を使用してi-nodeテーブルにノードを確立し、対応するデバイスからi-node情報を読み取ります。

2. i-nodeポインタが返されます。

3. iput() function

iput()関数が行う機能は、iget()とは全く逆です。主に、i-nodeの参照カウント値を1デクリメントし、それがパイプのi-nodeであれば、待機中のプロセスをウェイクアップするために使用されます。i-nodeがブロックデバイスファイルのi-nodeであれば、デバイスをリフレッシュし、i-nodeのリンクカウントが0であれば、i-nodeが占有している全てのディスク論理ブロックを解放し、i-nodeが解放された後に返却する。iノードの参照カウント値i_countが1で、リンク数が0ではなく、コンテンツが変更されていない場合は、この時にiノードの参照カウントを1つデクリメントしてから返却します。なぜなら、iノードがi_count=0であれば、そのノードが解放されたことを意味するからである。この関数の動作フローもiget()と同様です。

■ ある時点でプロセスがi-nodeを継続的に使用する必要がない場合、iput()関数を呼び出してi-nodeの参照カウント・フィールドi_countの値をデクリメントし、またカーネルに他の処理を行わせるべきです。したがって、以下の操作を行った後、カーネルコードは通常、iput()関数を呼び出す必要があります。

- 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.).

また、プロセスの生成時には、カレントワーキングディレクトリpwd、カレントルートディレクトリroot、実行ファイルディレクトリexecutableの各フィールドが3つのi-nodeを指すように初期化され、3つのi-nodeの参照カウントフィールドもそれに合わせて設定されます。したがって、プロセスがカレントワーキングディレクトリを変更するシステムコールを実行する際には、システムコールのコードの中でiput()関数を呼び出して、まず使用中のi-nodeを戻し、次にプロセスのpwdがパス名の新しいi-nodeを指すようにする必要があります。同様に、プロセスのrootやexecutableフィールドを変更するにも、iput()関数を実行する必要があります。

4. bmap() function

_bmap()関数は、ファイルのデータブロックを対応するディスクブロックにマッピングするために使用される。inode」はファイルのi-nodeポインタ、「block」はファイルのデータブロック番号、「create」は対応するファイルデータブロックが存在しない場合に、対応するディスクブロックをディスク上に確立する必要があるかどうかを示す作成フラグである。本関数の戻り値は、デバイス上のファイルデータブロックに対応する論理ブロック番号(ディスクブロック番号)である。create=0の場合は、bmap()関数となる。create=1の場合は、create_block()関数となります。

通常のファイルのデータは、ディスクのデータゾーンに配置され、ファイル名は対応するi-nodeを通じてこれらのデータディスクブロックに関連付けられます。これらのディスク・ブロックの番号は、i-nodeの論理ブロック配列に格納されている。_bmap()関数は、主にi-nodeの論理ブロック配列

i_zone[]を処理し、i_zone[]の論理ブロック番号に応じて論理ブロック・ビットマップの占有率を設定する。12.1節の図12-6を参照してください。前述したように，i_zone[0]～i_zone[6]はファイルの直接論理ブロック番号を，i_zone[7]は間接論理ブロック番号を，i_zone[8]は二次間接論理ブロック番号を格納するために使用される。ファイルサイズが小さい（7K以下）場合は、ファイルが使用するディスクブロック番号をi-nodeの7つの直接ブロック項目に直接格納することができ、ファイルが少し大きい（7K+512K以下）場合は、プライマリ間接ブロック項目i_zone[7]を使用する必要があり、ファイルが大きい場合は、セカンダリ間接ブロック項目i_zone[8]が必要となる。そのため、ファイルが比較的小さい場合は、Linuxのディスクブロックへのアドレス指定の速度が速くなります。

12.5.2 Code annotation

プログラム 12-4 linux/fs/inode.c

```
1 /*
2  *  linux/fs/inode.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
//      and constants.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
//      used functions of the kernel.
// <linux/mm.h> メモリ管理用のヘッダーファイルです。ページサイズの定義と、いくつかのページ
//      release function prototypes.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
//      descriptors/interrupt gates, etc. is defined.
7 #include <string.h>
8 #include <sys/stat.h>.
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h>
12 #include <linux/mm.h>
13 #include <asm/system.h>
14
// デバイスデータブロックの総数を示すポインターの配列。各ポインタの項目は
// 与えられたメジャーデバイス番号の総ブロック数に、hd_sizes[].の各項目は
// ブロックアレイの合計数は、データブロックの合計数に対応します。
// サブデバイス番号で決まるサブデバイス。
15 extern int *blk_size[]; 16
17 struct m_inode inode_table[NR_INODE]={{0,},};     // In-memory i-node table (NR_INODE=32).
18
19 static void read_inode(struct m_inode * inode);  // read i-node information, line 297.
20 static void write_inode(struct m_inode * inode); // write i-node info to cache, line 324.
21
//// 指定したi-nodeが利用可能になるのを待ちます。
// i-nodeがロックされている場合、現在のタスクは中断されない待機状態になります。
// そして、i-nodeがロックを解除して明示的にタスクをウェイクアップするまで、i-waitキューに追加されます。
```

24 22 static inline void wait_on_inode(struct m_inode * inode) 23 {...

```
25          cli();
26          while (inode->i_lock)
27                  sleep_on(&inode->i_wait);          // kernel/sched.c, line 199.
28          sti();
28 }
29
```

//// i-nodeをロックする（与えられたi-nodeをロックする）。

```
// i-nodeがロックされている場合、現在のタスクは中断されない待機状態になり
// i-waitキューに追加される i_wait. i-nodeがロックを解除して明示的にタスクを起こすまで。
// そして、ロックします。
32  30 static inline void lock_inode(struct m_inode * inode) 31 {。
33          cli();
34          while (inode->i_lock)
35                  sleep_on(&inode->i_wait);
36          inode->i_lock=1;                        // set locked flag.
37          sti();
37  }
38
//// 指定されたiノードのロックを解除します。
// i-nodeのロックフラグをリセットし、i-waitで待っているすべてのプロセスを明示的にウェイクアップする
// queue i_wait.
41  39 static inline void unlock_inode(struct m_inode * inode) 40 {。
42          inode->i_lock=0;
43          wake_up(&inode->i_wait);                // kernel/sched.c, line 204.
43  }
44
//// メモリのi-nodeテーブルにあるデバイスdevの全てのi-nodeを解放します。
// メモリ上のi-nodeテーブル配列をスキャンし、指定されたアイテムが使用された場合はリリースします。
// デバイスです。
47  45 void invalidate_inodes(int dev) 46 {。
48          int i;
49          struct m_inode * inode;
49
// まず、ポインタにメモリのi-nodeテーブル配列の最初のアイテムを指定させて
// その中のすべてのi-nodeをスキャンする。それぞれのi-nodeについて、i-nodeがロック解除されるのを待つ（もし
// が現在ロックされているかどうか）を確認し、指定されたデバイスに属するかどうかを判断します。もしイエスであれば
// が解除され、すなわち、i-nodeのデバイス番号フィールドi_devが0に設定され、修正された
//フラグもリセットされます。その間に、まだ使用されているかどうかもチェックされます、つまり
// その参照カウントが0でないかどうかを確認し、0の場合は警告メッセージを表示します。
// 50行目のポインタ代入文「0+inode_table」は、「inode_table」と同等です。
50  // "&inode_table[0]" ですが、この方がわかりやすいかもしれません。
51          inode = 0+inode_table;                  // points to the first item of i-node table.
52          for(i=0 ; i<NR_INODE ; i++,inode++) {
53                  wait_on_inode(inode);           // wait for i-node to be available(unlocked).
54                  if (inode->i_dev == dev) {
55                          if (inode->i_count)     // if references is not 0, error warning.
56                                  printk("inode in use on removed disk\n\r");
57                  inode->i_dev = inode->i_dirt = 0;               // release i-node.
57                  }
58          }
```

```
}

//// すべてのi-nodeを同期させます。
// メモリのi-nodeテーブルにあるすべてのi-nodeを、デバイス上のi-nodeと同期させる。
void sync_inodes(void)
{
        int i;
```

```
64          struct m_inode * inode;
65
    // まず、メモリのi-node型のポインタに、i-nodeテーブルの最初のアイテムを指すようにします。
    // とし、テーブル内のすべてのノードをスキャンします。それぞれのi-nodeに対して、i-nodeが待つこ
    とで
    // ロックが解除され（現在ロックされている場合）、i-nodeが修正されたかどうかをチェックして
    // はパイプノードではありません。この場合，i-nodeはバッファキャッシュに書き込まれます。バッフ
    ァは
66  // マネージャープログラムが適切なタイミングでディスクに書き込んでくれます。
67          inode = 0+inode_table;                  // points to the first item of the table.
68          for(i=0 ; i<NR_INODE ; i++,inode++) {
69                  wait_on_inode(inode);                   // wait for i-node to be available.
70                  if (inode->i_dirt && !inode->i_pipe)    // modified and not a pipe node.
71          write_inode(inode);                             // write to the buffer cache.
71          }
72 }
73
    //// ファイルデータブロックのディスクブロックへのマッピング（bmap - block map）。
    // パラメータ：inode - ファイルのi-nodeポインタ，block - ファイル内のデータブロック番号。
    // create - ブロック作成フラグ。この関数は，指定されたファイルデータブロックを
    // デバイス上の論理ブロックを指定し、その論理ブロック番号を返します。もし、ブロック作成フラグ
    // が設定されていると、対応する論理ブロックが存在しない場合に、新しいディスクブロックが要求さ
    れます。
    // に対応する論理ブロック番号（ディスクブロック番号）と、デバイス上のファイル
    // データブロックが返されます。この関数は4つの部分で処理されます。(1) パラメータの有効性
    // チェック、(2)ダイレクトブロック処理、(3)プライマリー間接ブロック処理、(4)セカンダリー
    // 間接的なブロック処理です。
76  74 static int _bmap(struct m_inode * inode,int block,int create) 75 {...
77          struct buffer_head * bh;
78          int i;
78
    // (1) まず、パラメータの有効性を確認します。ファイルのデータブロック番号が以下の場合
    // 0の場合、カーネルは停止します。ブロック番号が (直接ブロック nr + 間接ブロック nr) より大きい
    場合は、カーネルは停止します。
79  // ブロック nr + 2 番目の間接ブロック nr）の場合、ファイルシステムの範囲外となります。
80          if (block<0)
81          panic("_bmap: block<0"); 81    if
(block >= 7+512+512*512)
82                  panic("_bmap: block>big");

    // (2) 次に、ファイルのブロック番号の大きさと、作成フラグの有無に応じて
    //セットされている場合は、別々に処理を行います。ブロック番号が7より小さい場合は
    // ダイレクトブロックで表現されます。このとき、作成フラグが設定されていて、論理的な
    // i-nodeのblockフィールドが0の場合、デバイスにディスクブロック（論理ブロック）を要求している。
    // と、ディスク上の論理ブロック番号が論理ブロックフィールドに記入されます。次にセット
    // i-node change time and modified flag, and finally return the logical block number. 関数は
```

```
83      // new_block()は、bitmap.cプログラムの76行目で定義されています。
84          if (block<7) {
85                  if (create && !inode->i_zone[block])
86                          if (inode->i_zone[block]=new_block(inode->i_dev)) {
87                                  inode->i_ctime=CURRENT_TIME;  // ctime - change time
88                          inode->i_dirt=1;                      // set modified flag.
88                          }
89                  return inode->i_zone[block];
90          }
```

```
// (3) ブロック番号が>>7で、(7 + 512)より小さい場合は、間接的に使用していることを意味する
// ブロックを作成します。そのため、以下では間接ブロックが処理されます。もしそれが作成操作で
// i-nodeの間接ブロックフィールドi_zone[7]が0の場合、そのファイルが
// 間接ブロックを初めて使用することになります。そのため、ディスクブロックを申請するためには
間接的なブロック情報を格納するために、 // 実際のディスクブロック番号を
// 間接ブロックフィールド。そして、i-node modified flagとmodification timeを設定します。もし、ディスクの
// ブロックの作成時に失敗すると、i-nodeの間接ブロックフィールドi_zone[7].
// はこの時点では0であり、0が返されます。または、パラメータ作成フラグは0だが、i_zone[7]の
//も0で、i-nodeに間接ブロックがないことを示しているので、マッピングディスク
// ブロックは失敗し、0を返して終了します。
        block -= 7;
        if (block<512) {
                if (create && !inode->i_zone[7])
                        if (inode->i_zone[7]=new_block(inode->i_dev)) {
                                inode->i_dirt=1;
                                inode->i_ctime=CURRENT_TIME;
                        }
                if (!inode->i_zone[7])
                        return 0;
// ここで、デバイス上のi-nodeの間接ブロックを読み、論理ブロック番号iを取得する
その上の「block」項目に // を入力します（各ブロック番号は2バイトを占めます）。作成操作の場合
// で、取得した論理ブロック番号が0の場合は、ディスクブロック
// そして、間接ブロックの'block'エントリを、そのブロックの論理ブロック番号に設定します。
// 新しいディスクブロック。そして、間接ブロックの修正フラグを設定します。もし、それがcreate
//の操作では、'i'はマッピング（検索）する必要のある論理ブロック番号です。最後に
// 間接ブロックが占有していたバッファブロックが解放され、論理ブロック番号が
新しいアプリケーションまたはディスク上の対応するブロックの // が返されます。
                if (!(bh = bread(inode->i_dev,inode->i_zone[7])))
                        return 0;
                i = ((unsigned short *) (bh->b_data))[block];
                if (create && !i)
                        if (i=new_block(inode->i_dev)) {
                                ((unsigned short *) (bh->b_data))[block]=i;
                                bh->b_dirt=1;
                        }
                brelse(bh);
                return i;
        }

// (4) プログラムがこの場所まで走ると、そのデータブロックがセカンダリに属することを示します。
//間接ブロックです。その処理は、主な間接ブロックと同様です。以下は
// 二次間接ブロックの処理について。まず，このブロックは数値から差し引かれて
プライマリ間接ブロックに収容されているブロック（512）の//を作成または検索して
// 作成フラグが設定されているかどうかで判断します。作成操作で、セカンダリの
// iノードの間接ブロックフィールドが0の場合、プライマリを格納するために新しいディスクブロック
が必要となる
```

// セカンダリ間接ブロックのブロック情報で、実際のディスクブロック番号が記入されている
//を2次間接ブロックフィールドに設定します。その後、i-node modified flagを設定して
// 修正時間です。同様に、ディスクブロックの作成時に失敗すると
// その場合、i-nodeの2次間接ブロックフィールドi_zone[8]は0であり、0が返されます。あるいは、それが
// 作成操作ではないが、i_zone[8]が0になり、間接的なものがないことが判明する
// i-nodeのブロックがあるので、マップされたディスクブロックは失敗し、0を返して終了します。

```
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;
    // ここで、デバイス上のi-nodeの2次間接ブロックを読み、論理ブロックを取得します。
    セカンダリー・インダイレクトのファーストレベル・ブロックの（ブロック／512）エントリにある //
    番号「i」の表示
    //ブロックです。作成操作であり、(block / 512)項目の論理ブロック番号が
    2次間接ブロックの第1レベルのブロックの//が0であれば、適用する必要がある。
    2次間接ブロックの第2レベルのブロック「i」としてディスクブロックを//して
    二次間接ブロックの第一階層のブロックの中で、 //(ブロック / 512) の項目は、次のように等しくなり
    ます。
    // 第2階層ブロックのブロック番号'i'を指定します。そして、第1階層のブロック修正フラグを
    セカンダリー・インダイレクト・ブロックの // 第1レベルのブロックをリリースします。
119     // ブロックを作成します。もし作成されていなければ、'i'は探す必要のある論理ブロック番号です。
120         if (!(bh=bread(inode->i_dev,inode->i_zone[8])))
121                 return 0;
122         i = ((unsigned short *)bh->b_data)[block>>9];
123         if (create && !i)
124                 if (i=new_block(inode->i_dev)) {
125                         ((unsigned short *) (bh->b_data))[block>>9]=i;
126                         bh->b_dirt=1;
126                 }
127         brelse(bh);
    // 2次間接ブロックの第2レベルのブロック番号が0の場合は
    // アプリケーションのディスクブロックが故障するか、元の対応するブロック番号が0であれば、0は
    // を返します。それ以外の場合は，2次間接ブロックの第2レベルのブロックを読み込んで
128     // デバイスで、2階層目のブロックのブロック項目にある論理ブロック番号を取得しています。
129         if (!i)
130                 return 0;
131         if (!(bh=bread(inode->i_dev,i)))
132                 return 0;
133         i = ((unsigned short *)bh->b_data)[block&511];    // limits to 511 (0x1ff).
    // 作成フラグが設定されていて、第2階層のブロックの論理ブロック番号が
    // ブロックが0であれば、データ情報を最終的に格納するブロックとしてディスクブロックを適用します。
    // そして、第2レベルのブロックのブロックアイテムを、新しい論理ブロック番号「i」と同じにします。
    // そして、第2レベルのブロックの修正フラグを設定します。最後に、第2レベルのブロックの
    // セカンダリ間接ブロックが解放され、新たに適用された論理ブロック番号の
        if (create && !i)
```

```
133
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
```

//// デバイス上のファイルデータブロックの論理ブロック番号を取得します。

// パラメータ：inode -ファイルのi-nodeポインタ、block -ファイルのデータブロック番号。

// 操作が成功した場合は、論理ブロック番号を返し、そうでない場合は0を返します。

```
142 int bmap(struct m_inode * inode,int block) 143
{。
```

```
144         return _bmap(inode,block,0);        // create flag = 0.
145 }
146
    //// ファイルブロックのためにデバイス上のディスクブロックを検索または作成します。
    // デバイス上のファイルデータブロックの対応するディスクブロックを探します。存在していない場合は
    // ブロックを作成し、対応するディスクブロック番号を返します。
    // パラメータ：inode -ファイルのi-nodeポインタ、block -ファイルのデータブロック番号。
    // 操作が成功した場合は、論理ブロック番号を返し、そうでない場合は0を返します。
147 int create_block(struct m_inode * inode, int block) 148 {。
149         return _bmap(inode,block,1);
150 }
151
    //// i-nodeを戻す（デバイスに書き戻す）。
    // この関数は、主にi-nodeの参照カウントを1だけデクリメントするために使用され、もしそれが
    // パイプi-node、待機中のプロセスを起こす。ブロックデバイスファイルのi-nodeであれば、デバイスの
    // が更新され、i-nodeのリンクカウントが0であれば、すべてのディスク論理ブロックが占有されます。
    i-nodeによる//が解除され、i-nodeが解除されます。
152 void iput(struct m_inode * inode) 153
{。
    // まず、パラメータで与えられたi-nodeの有効性をチェックし、i-nodeが
    // ロックを解除する（ロックされている場合）。i-nodeの参照カウントが0であれば、i-nodeの
    // はすでにアイドル状態になっています。カーネルは次に、それに対して put back 操作を要求します。
    // カーネル内のコードに問題があることを示しています。その後、エラーメッセージが表示され、マシンが
154     //がシャットダウンされます。
155         if (!inode)
156                 return;
157         wait_on_inode(inode);
158         if (!inode->i_count)
159                 panic("iput: trying to free free inode");
    // パイプのi-nodeであれば、パイプを待っているプロセスを起こして、参照の数
    // が1デクリメントされ、それでも参照カウントが0にならない場合は、関数が返されます。
    // それ以外の場合は、パイプラインが占有していたメモリページが解放され、参照回数が増えます。
    // ノードのモディファイドフラグ，パイプフラグがリセットされてから，関数が返される。パイプの場合
    //     if (inode->i_pipe) {
    i-node, inode->i_size に
```

は、メモリページアドレスが格納されます。get_pipe_inode()の231,237行目を参照してください。

```
159
160                 wake_up(&inode->i_wait);
161                  wake_up(&inode->i_wait2);          //
162                 if (--inode->i_count)
163                         return;
164                  free_page(inode->i_size);
165                 inode->i_count=0;
166                 inode->i_dirt=0;
167                 inode->i_pipe=0;
168                 return;
169         }
```

// i-nodeのデバイス番号フィールドが0の場合、このノードの参照カウントがデクリメントされる
// を1つずつ返していきます。例えば、パイプライン運用のi-nodeは、デバイス番号が
i-nodeの場合は、//0です。さらに、ブロックデバイスファイルのi-nodeであれば、その中の
// for the i-node to unlock.

```
170         if (!inode->i_dev) {
171                 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         }
```

// i-nodeの参照カウントが1より大きい場合、カウントは1だけデクリメントされて
// 直接返される（i-nodeがまだ使用中で解放できないため）、そうでない場合は
// i-nodeの参照カウント値は1（157行目でチェックされているので
// それは0である）。i-nodeのリンク数が0であれば，対応する
//ファイルが削除されます。その後、i-nodeの全ての論理ブロックがリリースされ、i-nodeがリリースされます。
// 関数 free_inode() は、実際に i-node の操作を解放するために使用されます、つまり、リセットします。
// i-nodeのビットマップビット、およびi-node構造のコンテンツをクリアします。

```
178回繰り返す。 if (inode->i_count>1) {
179
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         }
```

// i-nodeが修正された場合、i-nodeを更新するために書き戻し、i-nodeを待つ
// をロック解除する（ロックされている場合）。i-nodeを書き込むときにスリープすることがあるので、他のプロセスが
この時点ではi-nodeは//になっているので、上述のチェック手順（ラベルリピート）が必要です。

```
//      if (inode->i_dirt) {
は、プロセスが起こされた後、再び繰り返されます。
188
189                 write_inode(inode);         /* we can sleep - so do again */
```

```
                wait_on_inode(inode);
                goto repeat;
        }
// ここでプログラムが実行されると、i-nodeの参照カウントi_countは1、リンクの数
//が0ではなく、内容が変更されていないことを示しています。そのため、i-nodeの参照がある限り
// countを1だけデクリメントすれば、戻すことができます。このとき、i_count=0となっているのは
// i-nodeがリリースされました。
        inode->i_count--;
        return;
}

//// i-nodeテーブルからアイドルのi-nodeアイテムを取得します。
// 参照カウントが0のi-nodeを探し、クリアしてディスクに書き込み、それを返します。
// ポインタです。これで、参照カウントが1になりました。
struct m_inode * get_empty_inode(void)
{
        struct m_inode * inode;
        static struct m_inode * last_inode = inode_table;   // point to the first item.
        int i;

// last_inode ポインタを初期化して，i-node テーブル全体をスキャンします.
// i-nodeテーブルのヘッダを指定します。last_inodeが既にi-nodeの最後のアイテムを指していた場合は
// のテーブルをループし続けるために、i-nodeテーブルの先頭に再ポイントさせます。
// テーブルになります。last_inodeが指すi-nodeの参照カウントが0であれば、それは
```

```
// アイドル状態のi-nodeアイテムが見つかった場合、inodeをi-nodeに指します。もし、修正された
// 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                 }
```

// フリーのi-nodeが見つからない場合（inode = NULL）、デバッグ用にi-nodeテーブルが表示されます。

```
                if (!inode) {
```

// となり、システムを停止します。

```
214
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                 }
```

// そうでなければ、i-nodeのロックが解除されるのを待つ（再びロックされた場合）。i-nodeがフラグを変更した場合

//がセットされると、i-nodeはリフレッシュ（同期）されます。リフレッシュ時にはスリープする可能性があるため

// i-nodeのロックが解除されるのを待つために、再度サイクルする必要があります。

```
221                 wait_on_inode(inode);
222                 while (inode->i_dirt) {
223                         write_inode(inode);
224                         wait_on_inode(inode);
```

```
224                 }
225         } while (inode->i_count);
```

// i-nodeが他の人に占有されている（i-nodeの参照カウントが0ではない）場合は
// アイドルi-nodeを再度検索する必要がある。そうでなければ，アイドル状態のi-nodeアイテムが
要件を満たす//が見つかりました。その後、i-nodeアイテムのコンテンツがクリアされます。

```
226
```

// そして、参照カウントを1にして、i-nodeポインタを返します。

```
227         memset(inode,0,sizeof(*inode));
228         inode->i_count = 1;
229         return inode;
229 }
230
```

//// パイプのi-nodeを取得します。
// 最初にi-nodeテーブルをスキャンし、アイドルのi-nodeエントリを探し、空きメモリのページを取得します。
// をパイプラインが使用するようにします。そして、取得したi-nodeの参照カウントを2にする（reader
//とライター）、パイプの頭と尻尾が初期化され、パイプ型表現の
// i-nodeを設定します。i-nodeへのポインタを返し，失敗した場合はNULLを返します。

```
231 struct m_inode * get_pipe_inode(void)
232 {
233         struct m_inode * inode;
234
```

// まず、メモリのi-nodeテーブルからフリーのi-nodeを取得し、見つからない場合はNULLを返します。
//1にします。そして、i-nodeのためにメモリのページを申請し、ノードのi_sizeフィールドが指すようにします。
// をページに追加します。空きメモリがない場合，i-nodeは解放され，NULLが返されます。

```
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         }
    // そして、i-nodeの参照カウントを2にして、パイプの先頭と末尾のポインタをリセットします。
    // i-node論理ブロック番号配列のi_zone[0]とi_zone[1]にはそれぞれ
    // パイプのヘッダとパイプのテールポインタをそれぞれ指定します。最後に、i-nodeフラグを
241 // i-nodeのタイプをパイプして、i-nodeを返します。
242         inode->i_count = 2;                 /* sum of readers/writers */
243         PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0;  // Reset the pipe head and tail.
244         inode->i_pipe = 1;                          // pipe i-node.
245         return inode;
245 }
246
    //// デバイスからi-nodeを取得します。
    // パラメータ：dev - デバイス番号、nr - i-node番号。
    // 指定したi-node番号のi-node構造体の内容をデバイスから読み込んで
    // メモリのi-nodeテーブルを検索して、i-nodeポインタを返します。i-nodeテーブルの最初の検索
    // バッファキャッシュに配置されています。指定されたノード番号のi-nodeが見つかった場合、i-node
    の
    // ポインタはいくつかのチェック処理の後に返されます。それ以外の場合は，デバイスから i-node を読
    み込みます.
    // そして、i-nodeテーブルに配置され、i-nodeポインタが返されます。
247 struct m_inode * iget(int dev,int nr) 248 {。
249         struct m_inode * inode, * empty;
250
    // まず、パラメータの有効性を判断します。デバイス番号が0であれば、それは
    // カーネルコードに問題がある場合は、エラーメッセージを表示して停止します。そうでなければ、ア
    イドル状態のi-nodeが取られます。
    バックアップのためにi-nodeテーブルからあらかじめ//を入れておきます。その後、i-nodeテーブル全体
    をスキャンし、以下を探します。
    指定されたノード番号nrを持つi-nodeを//し、参照番号を1つ増やします。
    // ただし、現在スキャンされているi-nodeのデバイス番号が、指定された
    // デバイス番号またはノード番号が、指定されたノード番号と等しくない場合、スキャン
251 // を継続しています。
252         if (!dev)
253                 panic("iget with dev==0");
254         empty = get_empty_inode();
255         inode = inode_table;                    // points to the first item of table.
256         while (inode < NR_INODE+inode_table) {
257                 if (inode->i_dev != dev || inode->i_num != nr) {
258                         inode++;
259                         continue;
259                 }
    // 指定されたdevとnrを持つi-nodeが見つかった場合、そのノードはロック解除を待ちます（if
    // ロックされている）になります。ノードのロックが解除されるのを待っている間に、i-nodeテーブル
```

が変更される可能性があります。そのため
// コードが継続して実行される際にも、上記と同じチェックが必要です。もし変更が
// が発生した場合、i-nodeテーブル全体が再度スキャンされます。

```
261                 wait_on_inode(inode);
262                 if (inode->i_dev != dev || inode->i_num != nr) {
262                         inode = inode_table;
263                         continue;
264                 }
```

// これは、対応するi-nodeが見つかったことを意味します。このとき、i-nodeの参照カウントは

// 1つ増加し、さらに別のファイルのマウントポイントであるかどうかをチェックします。
// のシステムを確認します。そうであれば、マウントされたファイルシステムのルートノードを探します。もし、i-nodeが本当に
// 他のファイルシステムのマウントポイントでは、スーパーブロックテーブルが検索されます。
// このi-nodeにインストールされています。スーパーブロックが見つからない場合は、エラーメッセージが表示されます。
// 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                         }
```

// ここまでのコード実行では、ファイルシステムのスーパーブロックにインストールされた
// inodeが見つかりました。続いて、i-nodeをディスクに書き込んで元に戻し、その際には
// デバイス番号を、i-nodeにインストールされているスーパーブロックから取得したものとし、i-node
// 必要な数はROOT_INOに再設定され、1となります。
// マウントされたファイルシステムのルートi-node情報を取得します。

```
279                         iput(inode);
280                         dev = super_block[i].s_dev;
281                         nr = ROOT_INO;
282                         inode = inode_table;
283                         continue;
283                 }
```

// 最後に対応するi-nodeを見つけました。したがって、アイドルのi-nodeを破棄することができます
// この関数の最初に一時的に適用され、見つかったi-nodeを返します。

```
285                 if (empty)
286                         iput(empty);
287                 return inode;
287         }
```

// 指定されたi-nodeがテーブルに見つからない場合、そのi-nodeは以下の方法でテーブルに作成されます。
// 前のアプリケーションのアイドルi-node 'empty'を使用して、i-nodeの情報は
// 対応するデバイスから読み込まれ、i-nodeのポインタが返されます。

```
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
```

//// 指定されたi-nodeの情報を読み取る。
// 指定されたi-node情報を含むディスクブロックをデバイスから読み込み、その後
// 指定されたi-node構造にコピーされます。論理ブロック番号を決定するために
i-nodeが配置されているディスクブロックの//対応するデバイスのスーパーブロックの

// を計算するためのINODES_PER_BLOCKの情報を得るために、最初に読み込まなければなりません。

//ブロック番号です。i-nodeが配置されている論理ブロック番号を計算した後に
//の論理ブロックがバッファブロックに読み込まれ、その時点でのi-nodeの内容は、次のようになります。
// バッファブロックの対応する位置が、指定した位置にコピーされます。

```
297 static void read_inode(struct m_inode * inode) 298
{...
300         struct super_block * sb;
301         struct buffer_head * bh;
302         int block;
302
303     // まず、i-nodeをロックして、ノードがあるデバイスのスーパーブロックを取得します。
304         lock_inode(inode);
305         if (!(sb=get_super(inode->i_dev)))
306                 panic("trying to read inode without dev");
    // i-nodeが配置されているデバイスの論理ブロック番号 = (ブートブロック + super
    // ブロック) +iノードのビットマップブロック+論理ブロックのビットマップブロック+ (iノード番号
    -1) /iノード
    // ブロックごとに、図12-1を参照してください。i-nodeの番号は0から順番になっていますが、最初の
    // ノード0は使用されておらず, 対応するノード0構造はディスクに保存されていません。そのため
    // i-nodeを格納する最初のディスクブロックは、i-node番号を持つi-node構造を保持します。
    // 0--31ではなく1--32です。を計算する際には、1だけデクリメントする必要があります。
    // i-node番号に対応するi-nodeのディスクブロック番号。ここでは、論理的な
    // デバイスからi-nodeがあるブロックを取り出し、i-nodeの内容をコピーして
307     // inode ポインタで示される場所です。
308         block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
309                 (inode->i_num-1)/INODES_PER_BLOCK;
310         if (!(bh=bread(inode->i_dev,block)))
311                 panic("unable to read i-node block");
312         *(struct d_inode *)inode =
313                 ((struct d_inode *)bh->b_data)
314                         [(inode->i_num-1)%INODES_PER_BLOCK];
    // 最後に、読み込まれたバッファブロックが解放され、i-nodeのロックが解除されます。ブロックデバイスの場合
            brelse(bh);
```

```
313
314         if (S_ISBLK(inode->i_mode)) {
315                 int i = inode->i_zone[0];   // i_zone[0] contains dev nr for device file.
316                 if (blk_size[MAJOR(i)])
317                         inode->i_size = 1024*blk_size[MAJOR(i)][MINOR(i)];
318                 else
319                         inode->i_size = 0x7fffffff;
320         }
321         unlock_inode(inode);
322 }
323
```

//// i-nodeの情報をバッファブロックに書き込む。

// この関数は，指定されたi-nodeをキャッシュのバッファブロックに書き込んで

// キャッシュが更新されたときにディスクへの の論理ブロック番号を決定するためには

// i-nodeがあるデバイスでは、まずそのデバイスのスーパーブロックを読む必要があります。

// i-nodeが配置されている論理ブロック番号を計算した後、論理ブロック

// がバッファブロックに読み込まれ、i-nodeの内容が対応する

// バッファブロックの位置。

```
324 static void write_inode(struct m_inode * inode) 325 {。
327         struct super_block * sb;
328         struct buffer_head * bh;
```

```
329         int block;
329
    // i-nodeを最初にロックします。i-nodeが変更されていない場合や、デバイス番号が
    // i-nodeが0に等しい場合、i-nodeはロック解除されて終了します。されていないi-nodeについては
    // 修正された場合、その内容はバッファやデバイスにあるものと同じです。その後、スーパーブロック
    を取得
    i-node の // 。
330         lock_inode(inode);
331         if (!inode->i_dirt || !inode->i_dev) {
332                 unlock_inode(inode);
333                 return;
334         }
335         if (!(sb=get_super(inode->i_dev)))
336                 panic("trying to write inode without device");
    // そして、i-nodeが配置されている論理ブロックをデバイスから読み込んで、コピーします。
337     // i-node情報を、i-nodeアイテムの位置に対応する論理ブロックに。
338         block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
339                 (inode->i_num-1)/INODES_PER_BLOCK;
340         if (!(bh=bread(inode->i_dev,block)))
341                 panic("unable to read i-node block");
342         ((struct d_inode *)bh->b_data)
343                 [(inode->i_num-1)%INODES_PER_BLOCK] =
344                         *(struct d_inode *)inode;
    // その後、バッファモディファイドフラグを設定し、i-nodeのコンテンツが既に一致しているので
    // バッファに格納されているので、i-node dirtyフラグは0になります。最後に、i-nodeを含むバッファは
            bh->b_dirt=1;
344
345         inode->i_dirt=0;
346         brelse(bh);
347         unlock_inode(inode);
348 }
349
```

12.5.3 Information

None :)

12.6 super.c

12.6.1 Function

super.cプログラムには、ファイルシステムのスーパーブロックを操作する関数が含まれている。これらの関数はファイルシステムの低レベル関数に属し、主にget_super()、put_super()、read_super()といったファイル名やディレクトリ、パスを扱う上位関数で使用されます。また、ファイルシステムのロード/アンロードのシステムコールsys_umount()とsys_mount()、そしてルートファイルシステムのロード関数mount_root()があります。他のいくつかの補助関数はbuffer.cのものと似ています。

スーパーブロックは、主にファイルシステム全体の情報を格納する。その構造については、12.1項「全体的な機能の説明」の図12-3を参照してください。

get_super()関数は、指定されたデバイスの条件で、メモリのスーパーブロック配列の中から該当するスーパーブロックを検索し、スーパーブロックのポインタを返す関数である。そのため、指定されたデバイスで

関数が呼び出された場合、対応するファイルシステムがマウントされているか、少なくともスーパーブロックがスーパーブロック配列のエントリを占めていなければならず、そうでなければNULLを返します。

put_super()は、指定されたデバイスのスーパーブロックを解放するための関数である。スーパーブロックに対応するファイルシステムのi-nodeビットマップと論理ブロックビットマップが占有するバッファブロックを解放し、スーパーブロックテーブル（配列）super_block[]の対応するオペレーションブロックアイテムを解放する。本関数は、ユーザーがumount()を呼び出してファイルシステムをアンロードしたり、ディスクを交換したりする際に呼び出される。

read_super()関数は、指定されたデバイスのファイルシステムのスーパーブロックをバッファに読み込み、スーパーブロックテーブルに登録するための関数である。同時に、ファイルシステムのi-nodeビットマップと論理ブロックビットマップもメモリアレイのスーパーブロック構造に読み込まれる。最後にスーパーブロック構造体へのポインタを返す。

sys_umount()システムコールは、指定されたデバイスファイル名のファイルシステムをアンマウントするために使用され、sys_mount()はディレクトリ名にファイルシステムをロードするために使用されます。プログラムの最後の関数であるmount_root()は、システムをインストールする際に使用するルートファイルシステムで、システムの初期化時に呼び出されます。具体的な動作プロセスを図12-25に示す。

図12-25 mount_root()関数の目的

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 table file_table[], and displays the number of free disk blocks and idle i-nodes in the root file system.

12.6.2 mount_root()関数は、システムの初期化ファイルmain.cの中で、プロセス0が最初の子プロセス（プロセス1）を生成した後に呼び出され、システムからは一度だけ呼び出されます。具体的に呼び出される場所は、main.cプログラムの初期化関数init()のsetup()関数内です。setup()は実際にはシステムコールで、その実装コードは/kernel/blk_drv/hd.cの74行目から始まります。

12.6.3 Code annotation

プログラム 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  */
   // <linux/config.h> カーネル設定用のヘッダーファイルです。キーボード言語やハードディスクを定義する
   //      type (HD_TYPE) options.
   // <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
   //      of the initial task 0, and some embedded assembly function macro statements about the
   //      descriptor parameter settings and acquisition.
   // <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
   //      used functions of the kernel.
   // <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
   //      descriptors/interrupt gates, etc. is defined.
   // <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
   // <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
   //      and constants.
10 #include <linux/config.h>
11 #include <linux/sched.h>
12 #include <linux/kernel.h>
13 #include <asm/system.h>
14
15 #include <errno.h>
16 #include <sys/stat.h>
17
18 // デバイス内のデータとバッファキャッシュの同期をとる (fs/buffer.c, line 59)
19 int sync_dev(int dev);
20 void wait_for_keypress(void);        // kernel/chr_drv/tty_io.c, line 140.
20
21 /* set_bit uses setb, as gas doesn't recognize setc */
   //// 指定されたビットオフセットのビットをテストし、そのビット値を返します。
   // bitmap.cプログラムの19行目に定義されているインラインアセンブリマクロと似ていますが、以下のようになっています。
   // マクロは、ビット設定をテストして返すだけで、ビットには何の変更も加えていません（そのため、名前は
   // test_bit()の方が適切かもしれません）。
   // 入力： %0 - ax( res); %1 - eax(0); %2 - bitnr, ビットオフセット値; %3 - (addr), 開始アドレス。
   // 23行目では、ローカルレジスタ変数「res」を定義しています。この変数はeaxに保存されます。
   // レジスターへの効率的なアクセスと操作を実現します。マクロ定義全体は、ステートメント
   // 値が最後の「res」の値である式。24行目のBT命令は
   ビットのテストに使われる//。これは、アドレス addr (%3) で指定されたビットの値を
```

// ビットオフセット bitnr (%2) をキャリーフラグ CF に入れます。SETB命令を使用すると

22 // オペランド %al をキャリーフラグ CF に従います。CF = 1の場合、%al = 1、そうでない場合は%al = 0となります。

```
23 #define set_bit(bitnr,addr) ({  \
23 register int res asm ("ax"); \
24 asm ("bt %2,%3;setb %%al":"=a" ( res): "a" (0), "r" (bitnr), "m" (*(addr))); 25
res; })
26
27 struct super_block super_block[NR_SUPER];   // Super block table array (NR_SUPER = 8).
28 /* this is initialized in init/main.c */
29 int ROOT_DEV = 0;                           // Root file system device number.
30
```

//// スーパーブロックをロックします。

// スーパーブロックがロックされている場合、現在のタスクは中断されない待機状態になります
// 状態になり、スーパーブロックの待ち行列に追加される スーパーブロックのロックが解除されるまで s_wait
// タスクが明示的に起動されます。その後、ロックします。
// 以下の3つの関数（lock_super()、free_super()、wait_on_super()）には
// inode.cファイルの最初の3つの関数と同じ効果がありますが，オブジェクトが
33 // ここで操作されるのは、スーパーブロックに置き換えられます。31 static void lock_super(struct super_block * sb) 32 {。

```
34          cli();                          // disable interrupt
35          while (sb->s_lock)              // sleep if super block is already locked.
36                  sleep_on(&(sb->s_wait));  // kernel/sched.c, line 199.
37          sb->s_lock = 1;                 // lock it!
38          sti();                          // enable interrupt
38 }
39
```

//// 指定したスーパーブロックのロックを解除します。
// スーパーブロックのロックフラグをリセットし，以下を待っているすべてのプロセスを明示的にウェイクアップします.
// この待ち行列 s_wait (unlock_super()という名前を使えば、より適切かもしれません)。

42 40 static void free_super(struct super_block * sb) 41 {...

```
43          cli();
44          sb->s_lock = 0;                 // reset the lock.
45          wake_up(&(sb->s_wait));         // kernel/sched.c, line 188.
46          sti();
46 }
47
```

//// スーパーブロックが解除されるのを待つ眠り。
// スーパーブロックがロックされている場合、現在のタスクは中断されない待機状態になります
// 状態になり、スーパーブロックの待ち行列 s_wait に追加されます。スーパーブロックがロックを解除するまで
// と明示的にタスクを起こします。

50 48 static void wait_on_super(struct super_block * sb) 49 {...

```
51          cli();
52          while (sb->s_lock)              // sleep if super block is already locked.
53                  sleep_on(&(sb->s_wait));
54          sti();
54 }
55
```

//// 指定されたデバイスのスーパーブロックを取得します。
// 指定したデバイスdevのスーパーブロック構造をスーパーブロックテーブルで検索する
// （配列）。見つかった場合はスーパーブロックへのポインタを、そうでない場合はNULLポインタを返します。

56 struct super_block * get_super(int dev) 57 {。

```
58          struct super_block * s;         // superblock data structure pointer.
59
```

// まず、パラメータで与えられたデバイスの有効性をチェックします。デバイス番号が0の場合

```
// nullが返されます。次に、スーパーブロック配列の先頭を指定し、次のように開始します。
// スーパーブロック配列全体を検索して、指定されたデバイスのスーパーブロックを見つけます dev.
        if (!dev)
                return NULL;
        s = 0+super_block;
        while (s < NR_SUPER+super_block)
// 現在の検索語が、指定されたデバイスのスーパーブロックである場合、すなわちデバイス番号
```

スーパーブロックの // フィールド値が、関数パラメータで指定されたものと同じであれば
// スーパーブロックは，（他のプロセスによってロックされていた場合は）ロック解除されるのを待ちます．その間
スーパーブロックアイテムは、他のデバイスで使用される可能性があるため、 // 待機時間
// スリープが解除された後、指定されたスーパーブロックであるかどうかを再度確認します。
// デバイスです。そうであれば、スーパーブロックのポインタを返し、そうでなければ、スーパーブロックを検索します。
// の配列を再度作成するので、sは再びスーパーブロックの配列の先頭を指す必要があります。

```
66                  if (s->s_dev == dev) {
67                          wait_on_super(s);
68                          if (s->s_dev == dev)
69                                  return s;
70                          s = 0+super_block;
```

// 現在の検索語がない場合は、次の項目をチェックします。最後に、もし指定されたスーパーブロックが
// が見つからない場合は、NULLポインタが返されます。

```
72                  } else
70                          s++;
71          return NULL;
72 }
73
```

//// 指定したデバイスのスーパーブロックをリリース（戻す）します。
// デバイスが使用しているスーパーブロックアレイのアイテムを解放（s_dev=0を設定）して
// デバイスのinodeビットマップとロジックブロックビットマップによって占有されるバッファキャッシュブロック。もし
// スーパーブロックに対応するファイルシステムは、ルートファイルシステム、または他のファイルシステムである
// がそのiノードのいずれかにすでにインストールされている場合、スーパーブロックを解放することはできません。

```
76  74 void put_super(int dev) 75
    {。
77          struct super_block * sb;
78          int i;
78
```

// まず、パラメータの有効性と合法性を確認します。指定されたデバイスが、ルート
//ファイルシステムのデバイスを探すと、警告メッセージが表示されて戻ってきます。その後、ファイルを探して
// スーパーブロックテーブルの指定されたデバイスのシステムスーパーブロックです。もしスーパーブロックが
// ファイルシステムがインストールされているi-nodeが処理されていないことを示す、警告メッセージ
// が表示されて返されます。ファイルシステムのアンマウント操作では、s_imount には
79 この関数を呼び出す前に // null を指定しています。

```
80          if (dev == ROOT_DEV) {
81                  printk("root diskette changed: prepare for armageddon\n\r");
82                  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          }
```

// 指定されたデバイスのスーパーブロックを見つけたら、まずそれをロックして、次にその
// デバイス番号フィールド s_dev を 0 にする、つまりデバイス上のファイルシステムのスーパーブロックを解放する。
// その後、スーパーブロックが占有していた他のカーネルリソースを解放します、つまり
// ファイルシステムのi-nodeビットマップと論理ブロックのビットマップで占められたバッファブロックを
// バッファキャッシュです。以下の定数記号I_MAP_SLOTSとZ_MAP_SLOTSは、ともに
i-nodeのビットマップが占有するディスク論理ブロック数を示すために使用される//8。
// と論理ブロックのビットマップにそれぞれ対応しています。なお、これらのバッファブロックの内容が
// が変更された場合、バッファにデータを書き込むための同期操作が必要になります。
// ブロックをデバイスに送信します。この関数は最後にスーパーブロックのロックを解除して戻ります。

```
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
```

//// 指定されたデバイスのスーパーブロックを読み込む。

// 指定されたデバイスdevのファイルシステムのスーパーブロックが、すでにスーパーブロックテーブルにある場合。

// スーパーブロックエントリへのポインタが直接返されます。それ以外の場合は，スーパーブロックが読み込まれます

// デバイスデブからバッファブロックに入り、スーパーブロックテーブルにコピーされます。最後に

// スーパーブロックのポインタを返します。

```
99          struct super_block * s;
99
static struct super_block *read_super(int
```

t

d
e
v
)

1
0
0

{
:
:
.

```
        struct buffer_head * bh;
        int i,block;

// まずパラメータの有効性を確認し、次にデバイスがディスクを交換したかどうかを確認する
// (つまりフロッピーデバイスかどうか) を確認します。ディスクが交換された場合、ディスクの中のすべてのバッファブロックが
// バッファキャッシュが無効であり、無効化する必要がある、つまり、元々ロードされたファイルシステムの
//が解除されます。
        if (!dev)
                return NULL;
        check_disk_change(dev);
// デバイスのスーパーブロックがすでにスーパーブロックテーブルにある場合は、スーパーブロックへのポインタ
// が直接返されます。そうでない場合は，スーパーブロックの配列の中で，空いている項目を探します（つまり
// フィールドを持つアイテム s_dev=0）。配列が既に一杯の場合は null ポインタを返します。
        if (s = get_super(dev))
                return s;
        for (s = 0+super_block ;; s++) {
                if (s >= NR_SUPER+super_block)
                        return NULL;
                if (!s->s_dev)
                        break;
        }
// スーパーブロック配列に空きエントリが見つかったら、それを使ってファイルシステムを指定する
        s->s_dev = dev;
```

デバイス上の//dev.その後、スー

パーブロック構造のメモリフィールドが部分的に初期化されます。

```
116
117         s->s_isup = NULL;
118         s->s_imount = NULL;
119         s->s_time = 0;
120         s->s_rd_only = 0;
121         s->s_dirt = 0;
```

// スーパーブロックがロックされ、その情報がデバイスからバッファに読み込まれる
// bhで指し示されるブロック。スーパーブロックは2つ目の論理ブロック（ブロック1）にあります。
ブロックデバイスの //。スーパーブロックの読み取り操作が失敗した場合、スーパーブロック内のアイテムは
// 上記で選択された配列がリリースされ（つまり、s_dev=0）、アイテムのロックが解除されて、ヌルポインタ
// を返します。そうでなければ、読み込まれたスーパーブロックは、バッファブロックのデータ領域から
// スーパーブロック配列の対応する項目と、読み込んだ内容を格納するバッファブロックの

```
//      lock_super(s);
の情報を公開しています。
122
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);
```

// デバイスdevからファイルシステムのスーパーブロックを取得したので、チェックを開始します。
// このスーパーブロックの有効性を確認し、i-nodeのビットマップと論理ブロックのビットマップを読み取る
// デバイスになります。読み込みスーパーブロックのファイルシステムマジックナンバーフィールドが正しくない場合は
// は、そのファイルシステムが正しいものではないことを意味します。そのため、上記のように、項目が
// 上記で選択されたスーパーブロックの配列が解除され、アイテムのロックが解除されて、空の
// ポインタが返されます。このバージョンの Linux カーネルでは，MINIX ファイルシステムのバージョンのみが

```
//      if (s->s_magic != SUPER_MAGIC) {
1.0がサポートされており、マジックナンバーは0x137fです
```

。

```
131
132                 s->s_dev = 0;
133                 free_super(s);
134                 return NULL;
135         }
```

// 以下、デバイス上のi-nodeビットマップとロジックブロックビットマップデータの読み込みを開始します。
// まず、メモリスーパーブロック構造のビットマップ空間を初期化します。次に，i-nodeのビットマップ
//と論理ブロックのビットマップ情報は、デバイスから読み込まれて
// スーパーブロックの対応するフィールドです。i-nodeのビットマップは，論理ブロックの
// デバイスのブロック2から始まり、合計でs_imap_blocksブロックを占有します。ロジックの

```
136     // ブロックビットマップは、後続のブロックであるs_zmap_blocksのブロックを占有します。
137         for (i=0;i<I_MAP_SLOTS;i++)                     // initialize.
138                 s->s_imap[i] = NULL;
139         for (i=0;i<Z_MAP_SLOTS;i++)
140                 s->s_zmap[i] = NULL;
141         block=2;
142         for (i=0 ; i < s->s_imap_blocks ; i++)          // read i-node bitmap in the device.
143                 if (s->s_imap[i]=bread(dev,block))
144                         block++;
145                 else
146                         break;
147         for (i=0 ; i < s->s_zmap_blocks ; i++)      // Read logic block bitmap in the device.
148                 if (s->s_zmap[i]=bread(dev,block))
149                         block++;
150                 else
151                         break;
```

// 読み込んだビットマップブロックの数が、論理ブロックの数と一致しない場合は
// ビットマップが占めるべきであることから、ファイルシステムのビットマップに問題があることを示しており
// スーパーブロックの初期化に失敗します。そのため、以前に適用されていたすべてのリソース
// と占有されているものを解放することができます。つまり、i-nodeビットマップで占有されているキャッシュブロックと
// 論理ブロックのビットマップが解除されると、上記で選択したスーパーブロックの配列項目が解除されます。

```
152     // スーパーブロックアイテムのロックが解除され、NULLポインタが返されます。.
153         if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {
154                 for(i=0;i<I_MAP_SLOTS;i++)          // release buffer blocks used by bitmap.
155                         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         }
```

// それ以外はすべてOKです。さらに、アイドル状態のi-nodeを要求する関数では、もし
// デバイス上のすべてのi-nodeが使用された場合、ルックアップ関数は0の値を返します。
// したがって、0番目のi-nodeは利用できないので、ビットマップの1番目のブロックの最下位ビット
ファイルシステムが0番目のiノードを割り当てるのを防ぐために、//は1に設定されています。同じ理由で
// ロジックブロックのビットマップの最下位ビットも1に設定されます。 最後の関数では、ロックを解除して

```
//      s->s_imap[0]->b_data[0] |= 1;
```

スーパーブロックへのポインタを返します。

```
160
161         s->s_zmap[0]->b_data[0] |= 1;
162         free_super(s);
163         return s;
164 }
165
```

//// ファイルシステムをアンマウントする system-call.
// パラメータの「dev_name」は、ファイルシステムが置かれているデバイスのファイル名です。
// この関数は，まず，与えられたブロックデバイスファイル名に基づいてデバイス番号を取得し，次に
// ファイルシステムスーパーブロックの対応するフィールドをリセットし，バッファブロックを解放する
// スーパーブロックとビットマップによって占有されます。最後に，バッファキャッシュを同期させて
// デバイス上のデータを表示します。アンマウント操作が成功した場合は0を、そうでない場合は
// エラーコードです。

```
68  166 int sys_umount(char * dev_name)
    167 {
    169         struct m_inode * inode;
    170         struct super_block * sb;
    171         int dev;
```

171 // まずデバイスファイル名からそのi-nodeを見つけ、デバイス番号を取得します。その際には
// デバイスファイルで定義されたデバイス番号は、そのi-nodeのi_zone[0]に格納されます。445行目参照
fs/namei.cプログラムのシステムコールsys_mknod()の//を使用しています。また、ファイルシステムが
// ブロックデバイスに格納される必要があり、ブロックデバイスのファイルでない場合は、i-node dev_i
172 返したばかりの // を戻して、エラーコードを返します。

```
173         if (!(inode=namei(dev_name)))
174                 return -ENOENT;
175         dev = inode->i_zone[0];
176         if (!S_ISBLK(inode->i_mode)) {
177                 iput(inode);                            // fs/inode.c, line 150.
178                 return -ENOTBLK;
178         }
```

// OK、デバイス番号を取得するために取得したi-nodeがミッションを完了したので、元に戻します。
// ここではi-nodeです。そして、ファイルシステムをアンマウントするための条件が満たされているかどうかをチェックします。
// デバイスがルートファイルシステムの場合は、アンマウントできず、ビジーエラーが返されます。
// デバイス上のファイルシステムのスーパーブロックが、スーパーブロックテーブルに見つからない場合。
// またはデバイス上のファイルシステムがインストールされていない場合は、エラーコードが返されます。もし、i-node
がマウントされている // にフラグ i_mount が設定されていない場合、警告メッセージが表示されます。すると
// i-nodeテーブルを調べて、デバイス上のファイルを使用しているプロセスがあるかどうかを確認し、もし
179 // そうすると、ビジーのエラーコードを返します。

```
180         iput(inode);
181         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;
```

// デバイス上のファイルシステムのアンマウント条件が満たされたので、起動できるようになった

// 実際のアンマウント操作を実装します。まず、マウントされたi-nodeのマウント・フラグをリセットします。

// マウントされている場合は、i-nodeをリリースし、スーパーブロックのマウントされたi-nodeフィールドを次のように設定します。

// を NULL にして，デバイスファイルシステムのルート i-node を戻す。のポインタを設定します.

```
//      sb->s_imount->i_mount=0;
```

//スーパーブロックにマウントされたファイルシステムのルートiノードをNULLにする。

```
189
190         iput(sb->s_imount);
191         sb->s_imount = NULL;
192         iput(sb->s_isup);
193         sb->s_isup = NULL;
```

// 最後に、デバイス上のスーパーブロックと、ビットマップが占有しているバッファブロックを解放します。

```
// then return 0 (unmounted successfully).
        put_super(dev);

        sync_dev(dev);
        return 0;
}

//// Mount (new) file system system-call.
// The parameter dev_name is the device file name, dir_name is the directory name to be mounted,
```

```
// and rw_flag is the read/write flag of the mounted file system. The location to be mounted
// must be a directory name, and the corresponding i-node is not occupied by other programs.
// Returns 0 if successful, otherwise returns an error number.
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
        struct m_inode * dev_i, * dir_i;
        struct super_block * sb;
        int dev;

// First, find the corresponding i-node according to the device file name to get the device
// number. For block special device files, the device number is in i_zone[0] of its i-node.
// In addition, since the file system must be in the block device, if not, the i-node dev_i
// just obtained is put back and the error code is returned.
        if (!(dev_i=namei(dev_name)))
                return -ENOENT;
        dev = dev_i->i_zone[0];
        if (!S_ISBLK(dev_i->i_mode)) {
                iput(dev_i);
                return -EPERM;
        }
```

// OK、デバイス番号を取得するために取得したデバイスファイルi-node dev_iが完了しました。
// の使命を果たしているので、ここに戻しておきましょう。次に、ファイルの保存先のディレクトリ名が
// のシステムがマウントされていることを確認します。その後、対応するiノードdir_iを取得します。
与えられたディレクトリ・ファイル名に // し、i-nodeの参照カウントが1でない場合（のみ
// ここで参照されています）、またはノード番号がルートファイルシステムの1である場合は、次のように戻します。

```
//iノードでは，エラーコードを返す。また，そのノードがディレクトリファイルノードでない場合は
// 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         }
    // マウントポイントがチェックされたので、新しいファイルシステムのスーパーブロックの読み込みを
    開始します。
    // ファイルシステムのスーパーブロックは、最初にスーパーブロックテーブルから検索して、読み取りは
    // デバイスがスーパーブロックテーブルにない場合。新しいファイルのスーパーブロックを取得した後
    // ファイルシステムをマウントしていないことを確認するために、まずそれをチェックします。新しい
    ファイルシステムがマウントされていないことを確認するために
    // 他の場所で、マウントされるi-nodeが（他のファイルシステムによって）占有されていないこと、そ
    うでなければ
223 // エラーが発生して戻ってきました。
224         if (!(sb=read_super(dev))) {
225                 iput(dir_i);
226                 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         }
    // 最後に、新しいファイルシステムのスーパーブロックの「マウントされたi-node」フィールドの
    「s_imount」を次のように設定します。
    // マウントされているディレクトリのi-nodeを指し、マウントフラグとnodeを設定する
            sb->s_imount=dir_i;
```

```
235
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
```

// まず、ディスクのi-node構造のサイズが要件（32バイト）を満たしているかどうかをチェックして
// コードを修正する際の不整合を防ぎます。続いて、ファイルテーブルの初期化（64項目）
// とスーパーブロックテーブル（8項目）があります。ここで，すべてのファイル構造における参照カウントは
// 0（アイドルを示す）に設定され、スーパーブロックテーブルの各構造体のデバイスフィールド
// は0に初期化されます（アイドル状態であることも示します）。ルートファイルシステムが配置されているデバイスが
//がフロッピーディスクの場合、"Insert root floppy and press ENTER "と表示され、キーを押して待ちます。

```
        if (32 != sizeof (struct d_inode))
                panic("bad i-node size");
        for(i=0;i<NR_FILE;i++)                          // initialize file table (64 items).
                file_table[i].f_count=0;
        if (MAJOR(ROOT_DEV) == 2) {
                printk("Insert root floppy and press ENTER");
                wait_for_keypress();
        }
        for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
                p->s_dev = 0;                           // initialize super block table (8).
                p->s_lock = 0;
                p->s_wait = NULL;
        }
```

// 上記の「余分な」初期化作業を行った後、ルートファイルシステムのマウントを開始しました。
// その後、ファイルシステムのスーパーブロックがルートデバイスから読み込まれ、ルートのi-nodeへのポインタが
メモリのi-nodeテーブルのファイルシステムの//(ノード1)を取得します。上のスーパーブロックが
// デバイスが故障するか、ルートノードが故障すると、メッセージが表示され、デバイスがダウンします。

```
        if (!(p=read_super(ROOT_DEV)))
                panic("Unable to mount root");
        if (!(mi=iget(ROOT_DEV,ROOT_INO)))              // ROOT_INO is defined as 1 in fs.h.
                panic("Unable to read root i-node");
```

// ここで、スーパーブロックとルートのi-nodeをセットアップし、その参照番号を増やします。
// ルートのi-nodeを3回分。なぜなら、i-nodeは次の数行のコードでも参照されるからです。
// さらに、iget()関数でのi-nodeの参照カウントはすでに1に設定されています。
// そして、マウントされたファイルシステムのルートのi-nodeのフィールド（s_isup）とi-nodeのフィールド（s_imount）
スーパーブロックの//がこのi-nodeに設定され、現在の作業ディレクトリと
// カレントプロセスのルートi-nodeも設定されます。この時点で、現在のプロセスは
// 番号1のプロセス（initプロセス）。

```
        mi->i_count += 3 ;      /* NOTE! it is logically used 4 times, not 1 */
        p->s_isup = p->s_imount = mi;
        current->pwd = mi;
        current->root = mi;
```

// 次に、ルートファイルシステムのリソースをカウントし、空きブロック数を数えて
// デバイス上のアイドルi-nodeの数です。まず、'i'には、デバイス上の
スーパーブロックで示されたデバイスの // 論理ブロックの数を数えて、空いている
論理ブロックビットマップの対応するビットの占有率に応じて、 // ブロック

// ここでは、マクロ関数set_bit()は、ビットのテストにのみ使用されています。"i&8191 "を使用して取得しています。

// 現在のビットマップブロック内のi-node番号の対応するビットオフセット値。"i>>13"

// free=0;

iをディスクブロックのビット数である8192で割ったものです。

```
268
269         i=p->s_nzones;
270         while (-- i >= 0)
271                 if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
272                         free++;
```

// デバイス上の空き論理ブロック数を表示した後、次に数を数えます。

デバイス上の空きi-nodeの//。まず、'i'をデバイス上のi-nodeの合計数とします。

// スーパーブロックで示されたデバイスを＋1（0のノードもカウントするために1を加える）して

```
    // 対応するビットの占有率に応じて，アイドル状態のi-nodeの数を計算する
    //をi-nodeのビットマップに表示します。最後に、空いているi-nodeの数と、i-nodeの合計数
    デバイスで利用可能な // が表示されます。
        printk("%d/%d free blocks\n\r",free,p->s_nzones);
        free=0;
        i=p->s_ninodes+1;
        while (-- i >= 0)
                if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))
                        free++;
        printk("%d/%d free inodes\n\r",free,p->s_ninodes);
}
```

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.

Linux 0.12では、MINIXファイルシステムのバージョン1.0を採用しています。そのディレクトリエントリ構造は、従来のUNIXファイルと同じで、include/linux/fs.hファイルで定義されています。ファイルシステムのディレクトリでは，すべてのファイル名に対応するディレクトリエントリが，ディレクトリファイル名のデータブロックに格納される。例えば，ディレクトリ名home/の下にあるすべてのファイル名のディレクトリ・エントリは，home/ディレクトリ名ファイルのデータ・ブロックに格納され，ファイルシステムのルート・ディレクトリの下にあるすべてのファイルは，指定されたi-node（すなわちノード1）のデータ・ブロックに格納される。各ディレクトリエントリには，以下のように，長さ14バイトのファイル名文字列と，そのファイル名に対応する2バイトのi-node番号のみが含まれる。

```
// include/linux/fs.hファイルで定義されています。
#define NAME_LEN 14                     // file name maximum length.
#define ROOT_INO 1                      // root i-node number.

// ファイルディレクトリのエントリ構造
struct dir_entry {
        unsigned short inode;           // i-node number.
        char name[NAME_LEN];            // filename string
};
```

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.

ファイルが開かれると、ファイルシステムは与えられたファイル名に基づいてi-node番号を見つけ、

ファイルが置かれているディスクブロックの位置を見つける。例えば、「/usr/bin/vi」というファイルのi-node番号を見つけるために、ファイルシステムはまず固定のi-node番号（1）を持つルートディレクトリからスタートする。すなわち，i-node番号1のデータブロックから，ファイル名「usr」のディレクトリエントリを見つけ，それによって，「/usr/bin/vi」のi-node番号を取得する。

ファイル'/usr'を取得する。i-node numberファイルシステムによれば、ディレクトリ'/usr'を正常に取得することができ、その中にファイル名'bin'のディレクトリ・エントリを見つけることができる。これもi-node numberで'/usr/bin'がわかるので、ディレクトリ'/usr/bin'の位置がわかり、その中にある'vi'ファイルのディレクトリ・エントリを見つけることができる。最後に、ファイルパス名'/usr/bin/vi'のi-node番号を取得しているので、そのi-node番号のi-node構造情報をディスクから取得することができる。

また、各ディレクトリには2つの特別なファイル・ディレクトリ・エントリがあり、その名前はそれぞれ「.」と「...」に固定されています。'.'ディレクトリ・エントリは、カレント・ディレクトリのi-node番号を与え、'.'ディレクトリ・エントリは、カレント・ディレクトリの直接の親ディレクトリのi-node番号を与える。したがって、相対パス名が与えられた場合、ファイルシステムはこれら2つの特別なディレクトリ・エントリをルックアップ操作に使用することができる。例えば，「.../kernel/Makefile」を探すには，まず，カレント・ディレクトリの「...」ディレクトリ・エントリから親ディレクトリのi-node番号を取得し，その後，上述のプロセスに従ってルックアップ操作を行えばよい。

プログラムのいくつかの主要な機能は、その前に詳細なコメントがあり、各機能やシステムコールの使用方法が明確であるため、ここでは説明しません。

12.7.2 Code annotation

プログラム 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> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ

```
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
```

// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。

```
//      used functions of the kernel.
```

// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義

```
//      for segment register operations.
```

// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。

```
// <fcntl.h> ファイル制御のヘッダファイルです。演算制御定数記号の定義は
//      used for the file and its descriptors.
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <const.h> 定数シンボルファイルでは、現在、i-nodeのi_modeフィールドのフラグを定義しています。
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む 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 // ファイル名で対応するi-nodeを探す内部関数です。
22 static struct m_inode * _namei(const char * filename, struct m_inode * base,
23         int follow_links);
23
   // 以下のマクロの右の式は、配列にアクセスする特別な方法です。これは
   // 配列名で表される配列項目（a[b]など）の値と
   // 配列の添え字は，配列ポインタ（アドレス）の値にオフセットを加えたものと同じです.
   // address*（a＋b）と同時に、a[b]という言葉も表現できることがわかります。
   //をb[a]の形で表示します。つまり、文字配列の項目形式「LoveYou」[2](または2["LoveYou"])の場合
   は
   //は、*("LoveYou"＋2)と同等です。また、文字列「LoveYou」がある位置は
   // メモリに格納されているのはそのアドレスなので、配列アイテム「LoveYou」[2]の値は
   // 文字列中のインデックス値が2で、ASCIIコード値が0x76の文字 "v"、または
   // 8進法で0166。C言語では、文字は次のようにしてASCIIの値で表すこともできます。
   // 文字のASCII値にバックスラッシュを加える。例えば、文字 "v "の場合
   // は「¥x76」または「¥166」と表現します。そのため、表示できない文字（例.
   //0x00--0x1fの範囲のASCIIコード値を持つ制御文字）は次のように表現できます。
   // そのASCIIコード値
   //
   // 以下は、ファイルアクセスモードのマクロです。x」はファイルアクセス（オープン）モードのフラグ
   が定義されています。
   ヘッダーファイルinclude/fcntl.hの7行目にある // このマクロは、対応する値をインデックス化します。
   ファイルアクセストークンxの値に基づいて、二重引用符で囲まれた文字列の中に//を入れます。
   // マークには8進法の制御文字が4つあります。「これは、読み取り、書き込みを示すものです。
   // と実行権限：それぞれr、w、rw、wxrwxrwxで、インデックスに対応しています。
   // xの値：0〜3。例えば、x が 2 の場合、マクロは 006 という 8 進法の値を返し、次のことを示します。
   // 読み込み可能性と書き込み可能性(rw)を示しています。また、O_ACCMODE = 00003の場合は、マ
   スクの
   // インデックス値 x.
24 #define ACC_MODE(x) ("¨¯´¯¨"[(x)&O_ACCMODE])
25
26 /*
28  * comment out this line if you want names > NAME_LEN chars to be
29  * truncated.  Else they will be disallowed.
29 */
30 /* #define NO_TRUNCATE */
31
32 #define MAY_EXEC 1
33 #define MAY_WRITE 2
34 #define MAY_READ 4
35
36 /*
37  *      permission()
38  *
39  * is used to check for read/write/execute permissions on a file.
40  * I don't know if we should look at just the euid or both euid and
```

```
41  * uid, but that should be easily changed.
42 */
    //// ファイルのアクセス権を検出します。
    // パラメータ：inode - ファイルのi-nodeポインタ，mask - アクセス属性マスク.
    // 戻り値：アクセスが許可されている場合は1、そうでない場合は0を返します。
43 static int permission(struct m_inode * inode,int mask) 44 {...
45          int mode = inode->i_mode;                   // File access mode.
```

46

47 /* special case: not even root can read/write a deleted file */

// i-nodeが対応するデバイスを持っているが、リンクカウントが0に等しい場合、以下を示します。

// ファイルが削除された場合には、リターンします。そうでない場合は、ファイルが削除されたときの

// プロセスは、i-nodeのユーザーID、ファイル所有者のアクセス許可と同じです。

// が取られます。それ以外の場合は、プロセスの有効なグループID（egid）がグループ

48 // i-nodeのidで、グループユーザーのアクセス権が取られます。

49 if (inode->i_dev && !inode->i_nlinks)

50 return 0;

51 else if (current->euid==inode->i_uid)

51 mode >>= 6;

52 else if (in_group_p(inode->i_gid))

53 mode >>= 3;

// 最後に、アクセス権がマスクコードと同じである場合、またはスーパー

54 // ユーザーの場合は1を、そうでない場合は0を返します。

55 if (((mode & mask & 0007) == mask) || suser())

56 return 1;

57 return 0;

57 }

58

60 59 /*

61 * ok, we cannot use strncmp, as the name is not in our data space.

62 * Thus we'll have to use match. No big problem. Match also makes

63 * some sanity tests.

63 *

64 * NOTE! unlike strncmp, match returns 1 for success, 0 for failure.

65 */

//// 文字列の照合・比較機能。

// パラメータ： len - 比較する文字列の長さ， name - ファイル名へのポインタ。

// de - ディレクトリエントリの構造です。

// 戻ります。66 static int match(int len,const char * name,struct dir_entry *

de) 67 {.

68 register int same_ asm ("ax"); // register variable.

69

// まず、関数のパラメータの有効性を判定します。ディレクトリエントリが

// ポインタが空であるか、ディレクトリエントリの i-node が NULL であるか、比較する長さが

// ファイル名の長さ NAME_LEN を超える。比較対象の長さが 0 に等しい場合でも

// ディレクトリエントリのファイル名が「...」であれば、同じと判断し、1（マッチ）を返します。

// 比較される長さ 'len' が NAME_LEN よりも小さく、ファイル名の長さが

// が len を超える場合は、0（一致しない）も返されます。

// 75行目でファイル名の長さが'len'を超えているかどうかを判断する方法としては

// name[len]がNULLかどうか。長さが len を超える場合は， name[len] は通常の文字です.

/ if (!de || !de->inode || len > NAME_LEN)

/

N

U

L

L

て

はないもの。長さ len の文字列名の場合，name[len]という文字は NULL でなければなりません．

```
70
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;
```

// 次に、埋め込みアセンブリステートメントを使用して、すばやく比較します。これは、文字列
// ユーザーデータスペース(fsセグメント)での比較です。

```
    // %0 - eax (比較結果、同じ); %1 - eax (eaxの初期値0); %2 - esi (名前)。
    // %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 */
    //// 指定されたファイル名のディレクトリ・エントリを、指定されたディレクトリから探し出します。
    // パラメータ。*dir - ディレクトリのi-nodeポインタ; name - ファイル名; namelen - ファイル名の長さ.
    // 指定されたディレクトリのデータ（ファイル）を検索して、ディレクトリエントリ
    // 与えられたファイル名に応じて特別な処理を行います。
    // ファイル名が「...」の場合の設定です。の151行目の前のコメントをご覧ください。
    関数の引数におけるポインタの役割については、 // linux/sched.c を参照してください。
    // 戻り値：成功した場合はキャッシュブロックポインター、返された場合はディレクトリエントリポインター
97  // at *res_dir. 失敗すると null ポインタが返されます。
98  static struct buffer_head * find_entry(struct m_inode ** dir,
99          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
    // 同様に、この機能でも判断して、妥当性を検証する必要があります
    // のパラメータを設定します。上記の30行目でシンボル定数NO_TRUNCATEを定義した場合、ファイル
    // 名前の長さが最大長のNAME_LENを超えると、処理されません。もし、NO_TRUNCATE
106 // が定義されていない場合、ファイル名の長さが最大長のNAME_LENを超えると切り捨てられます。
107 #ifdef NO_TRUNCATE
108         if (namelen > NAME_LEN)
109                 return NULL;
110 #else
111         if (namelen > NAME_LEN)
```

```
112                 namelen = NAME_LEN;
112 #endif
```

// まず、このディレクトリのディレクトリエントリの数を計算します。のi_sizeフィールドは
// ディレクトリのi-nodeには、このディレクトリに含まれるデータのサイズが含まれていますので

```
// Then null the pointer to the directory entry to be returned.
        entries = (*dir)->i_size / (sizeof (struct dir_entry));
```

```
113
114         *res_dir = NULL;
```

// 次に、ディレクトリエントリのファイル名が「...」である場合に特化します。もし、ルート
// 現在のプロセスで指定されたi-nodeは、関数パラメータで指定されたディレクトリと同じです。
// これは、このプロセスにとって、このディレクトリが擬似的なルートディレクトリであることを意味し、つまり
// プロセスは、ディレクトリ内のアイテムにのみアクセスでき、親ディレクトリにフォールバックすることはできません。
// つまり、このプロセスでは、このディレクトリがファイルシステムのルートディレクトリになっているということです。そのため
// ファイル名を「...」に変更する必要があります。
// そうでなければ、ディレクトリのi-node番号がROOT_INO（番号1）に等しい場合、それは確かに
// ファイルシステムのルートi-nodeなので、ファイルシステムのスーパーブロックが取られます。もし、そのような
// スーパーブロックでファイルシステムがマウントされているi-node（s_imount）が存在する場合、現在の
// 操作がs_imountノードにマウントされたファイルシステム内で行われた場合、元のi-nodeは
// まず、マウントされるi-nodeが処理されます。そこで、'*dir'に
マウントされているi-nodeに // し、このi-nodeへの参照数を増加させます。
// ひとりずつ。つまり、この状況を受けて、私たちは静かに「盗み」を実行したのです。

```
115 // コラム」プロジェクト :)
116 /* check for '..', as we might have to do some "magic" for it */
117         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 iput 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         }
```

// では早速、指定したファイル名のディレクトリエントリがどこにあるかを調べてみましょう。

// したがって、ディレクトリのデータを読み取る、つまり、データブロックを取り出す（論理
//ブロック）を、ディレクトリi-nodeに対応するブロックデバイスのデータゾーンに配置します。ブロックは
i-node構造体のi_zone[]配列には、これらの論理ブロックの // 番号が格納されています。
// 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;
```

// この時点で、与えられたファイル名に一致するディレクトリ・エントリを検索し、read
// データブロック。まず'de'にバッファブロックのデータブロック部分を指し示すようにして、ループで
// のディレクトリエントリ数を超えないように検索します。ここでiは
// ディレクトリエントリのインデックス。

```
135         i = 0;
136         de = (struct dir_entry *) bh->b_data;
```

```
137         while (i < entries) {
```

// 現在のディレクトリエントリデータブロックが検索され、一致するエントリがない場合は
// 見つかった場合、現在のディレクトリエントリデータブロックは解放され、次の論理的
ディレクトリの // ブロックです。ブロックが空の場合は、ディレクトリのすべてのエントリが
// が検索されていない場合、そのブロックはスキップされ、ディレクトリの次の論理ブロック
//が読み込まれます。ブロックが空でなければ、deはデータブロックを指定し、検索を続けます。
//を入れることができます。141行目のi/DIR_ENTRIES_PER_BLOCKでは、データブロック番号を
// ディレクトリ項目が現在検索されているディレクトリファイルと、bmap()関数（inode.c,
// 142行目）は、デバイス上の対応する論理ブロック番号を計算することができます。

```
139                 if ((char *)de >= BLOCK_SIZE+bh->b_data) {
140                         brelse(bh);
141                         bh = NULL;
142                         if (!(block = bmap(*dir,i/DIR_ENTRIES_PER_BLOCK)) ||
143                             !(bh = bread((*dir)->i_dev,block))) {
144                                 i += DIR_ENTRIES_PER_BLOCK;
145                                 continue;
145                         }
146                         de = (struct dir_entry *) bh->b_data;
147                 }
```

// 一致するディレクトリエントリが見つかった場合は、ディレクトリエントリ構造ポインタ「de」と
// ディレクトリエントリのi-nodeポインタ'*dir'とディレクトリエントリのデータブロックポインタ'bh'は
// を返し、この関数を終了します。そうでなければ，次のディレクトリエントリの比較を続けます

```
//                  if (match(namelen,name,de)) {
```

ディレクトリエントリデータブロックの

```
148
149                         *res_dir = de;
150                         return bh;
151                 }
152                 de++;
153                 i++;
154         }
```

// 指定されたディレクトリ内のすべてのディレクトリエントリが検索され、かつ
// and finally NULL is returned (failed).

```
155         brelse(bh);
156         return NULL;
157 }
158
159 /*
160  *      add_entry()
161 *
162  * adds a file entry to the specified directory, using the same
163  * semantics as find_entry(). It returns NULL if it failed.
164 *
165  * NOTE!! The inode part of 'de' is left at 0 - which means you
166  * may not sleep between calling this and putting something into
167  * the entry, as someone else might have used it while you slept.
168 */
```

//// 指定したディレクトリにファイルエントリを追加します。

// パラメータ： dir - ディレクトリのi-ノード， name - ファイル名， namelen - ファイル名の長さ.

```
169 // 戻り値: バッファブロックポインタ; res_dir - 返されたディレクトリエントリへのポインタ。
170 static struct buffer_head * add_entry(struct m_inode * dir,
171         const char * name, int namelen, struct dir_entry ** res_dir)
171 {
```

```
172         int block,i;
173         struct buffer_head * bh;
174         struct dir_entry * de;
175
    // 同様に、この機能でも判断して、妥当性を検証する必要があります
    // のパラメータを設定します。上記の30行目でシンボル定数NO_TRUNCATEを定義した場合、ファイル
    // 名前の長さが最大長のNAME_LENを超えると、処理されません。もし、NO_TRUNCATE
    // が定義されていない場合、ファイル名の長さが最大長のNAME_LENを超えると切り捨てられます。
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
    // それでは早速、指定されたファイル名のディレクトリエントリを、指定された
    //のディレクトリです。そのため、ディレクトリのデータを読む、つまりデータを取り出す必要があるのですが
    ディレクトリに対応するブロックデバイスのデータゾーン内の // ブロック（論理ブロック）です。
    // i-node（アイノード）です。これらの論理ブロックのブロック番号は、i_zone[]配列に格納されています。
    // i-nodeの構造。まず、その中に保存されている最初のダイレクトブロック番号を取得し、それを読み取って
    // 指定されたディレクトリアイテムのデータブロックをデバイスから取り出します。また、ファイル名の長さが
184     パラメータで指定された // が 0 の場合は、NULL を返します。
185         if (!namelen)
186                 return NULL;
187         if (!(block = dir->i_zone[0]))
188                 return NULL;
189         if (!(bh = bread(dir->i_dev,block)))
190                 return NULL;
    // この時点で、このディレクトリのi-nodeのデータブロックをループして
    // 最後の未使用の空のディレクトリ・エントリです。まず、ディレクトリ・エントリ・ポインタに、データ
    バッファブロックの // ブロック部分、つまり最初のディレクトリエントリです。ここでiは、インデックス
    ディレクトリ内のディレ
        i = 0;
```

クトリエントリの//。

```
190
191         de = (struct dir_entry *) bh->b_data;
192         while (1) {
```

// 現在のディレクトリエントリデータブロックが検索されたが、必要な空のディレクトリがある場合
// のエントリーが見つからなかった場合、現在のディレクトリ・エントリー・データ・ブロックがリリースされ、その後
// ディレクトリの次の論理ブロックが読み込まれます。ブロックを作成するのは、対応する論理
// ブロックが存在しません。読み込みや作成の操作に失敗した場合は null を返します。もし、バッファブロック
今回読み込んだディスクの論理ブロックデータが返す // ポインタが空の場合は
// 論理ブロックは存在しないので、新しく作られた空のブロックかもしれません。そこで我々は
// ディレクトリエントリ数に対するディレクトリエントリインデックス値
DIR_ENTRIES_PER_BLOCK
論理的なブロックが保持できることを示す // ブロックをスキップして検索を続行します。そうでなければ、新たに
// 読み込みブロックにはディレクトリエントリデータがあるので、ディレクトリエントリ構造ポインタは
// ブロックのデータ部分をバッファリングして、そこから検索を続けます。
// 196行目のi/DIR_ENTRIES_PER_BLOCKは、ブロック番号を計算するために
// 現在検索されているディレクトリ・エントリiが存在するディレクトリ・ファイルで、関数
// create_block()(inode.c, line 147)を使用して、デバイス上のロジックブロックを読み込んだり、作成したりすることができます。

```
194                 if ((char *)de >= BLOCK_SIZE+bh->b_data) {
195                         brelse(bh);
196                         bh = NULL;
197                         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                 }
```

// 現在、ディレクトリエントリ番号iにディレクトリエントリサイズを乗じたサイズである場合
// ディレクトリのiノードが示すディレクトリデータサイズi_sizeを超える場合は
// ファイルが削除されたことにより、空のエントリが残っていないことを示しています。そのため、新しい
//ディレクトリデータファイルの最後に追加する必要があるディレクトリエントリです。したがって、私たちは
// このディレクトリのディレクトリ・エントリを設定する必要があるので、ディレクトリのi-node番号を設定する
// エントリを空にして、ディレクトリファイルのサイズを更新します（さらに、ディレクトリの長さを
// エントリ）を設定し、変更するディレクトリのi-nodeを設定して、変更を更新する
// このディレクトリの時刻を現在の時刻に変換します。

```
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                 }
```

// 現在検索されているディレクトリ・エントリ'de'のi-node番号が空の場合、それは
// 使用されていないフリーのディレクトリ・エントリ、または追加された新しいディレクトリ・エントリが見つかります。
// したがって、ディレクトリの修正時刻を現在の時刻に更新して、ファイル
ユーザーデータスペースからディレクトリエントリのファイル名フィールドに // 名前を設定し
// 対応するディレクトリエントリのキャッシュバッファブロック修正フラグ。ポインタで返します。

```
                if (!de->inode) {
```

```
211
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
222     上記の // find_entry() 関数を修正して、この関数にしました :)。
223         brelse(bh);
224         return NULL;
224 }
225
    //// シンボリックリンクのi-nodeを探す。
    // パラメータ： dir - ディレクトリ i-node; inode - ディレクトリエントリ i-node。
    // 戻り値: シンボリックリンク先のファイルへの i-node ポインタ。エラーの場合は NULL。
 2          unsigned short fs;              // Used to temporarily save FS segment register.
 2
 6

 s
 t
 a
 t
 i
 c

 s
 t
 r
 u
 c
 t

 m
 =
 i
 n
```

o
d
e
*
f
o
r
l
o
w
=
r
i
n
k
(
s
t
r
u
c
t
m
=
i
n
o
d
e
*
d
i
r
,
s
t
r
u
c
t
m
=
i
n
o
d
e
*
i
n
o
d
e
)
2
2
7

```
{
228
229         struct buffer_head * bh;
230
```

// まず、関数のパラメータの有効性を判断します。ディレクトリのi-nodeが与えられていない場合。
// プロセス・タスク構造のルートiノードセットを使用し、リンク数を増やす
//を1つずつ表示します。ディレクトリエントリのi-nodeが与えられていない場合、ディレクトリのi-nodeは戻されて
// NULLが返されます。指定されたディレクトリエントリがシンボリックリンクでない場合は，直接
// ディレクトリ・エントリに対応するi-nodeです。

```
231
232         if (!dir) {
233                 dir = current->root;
234                 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         }
```

// 続いて、FSセグメントレジスタの値を取得します。FSは通常、0x17を指すセレクタを保持しています。
// タスク（ユーザー）データセグメントを指します。FSがユーザーデータセグメントを指していない場合や、最初の
// 与えられたディレクトリ・エントリi-nodeのダイレクト・ブロック・ナンバーが0に等しいか、エラーが発生した場合
// 最初のダイレクトブロックを読み込む際に，dirとinodeのノードを戻して，NULLを返す.
// そうでなければ、FSがユーザーデータセグメントを指すようになり、読み取りに成功したことになります。
// このシンボリックリンクのディレクトリエントリのファイルコンテンツが、すでに
// 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
// リンク先のファイル これで，シンボリックのi-node情報は必要なくなりました.
// のリンクディレクトリのエントリがあったので、それを戻しました。ここで問題が発生しました。つまり、処理されたユーザーデータの
カーネル関数による // は，デフォルトではユーザーデータ空間に格納されるべきであり，そのためにはFS

ユーザー空間からカーネル空間にデータを転送するための//セグメントレジスタ。しかし、そのデータは
// ここで処理する必要があるのは、カーネル空間である。したがって、正しく処理するためには
// カーネル空間にあるユーザーデータのために、FSセグメントレジスタを一時的にポイントする必要がある
// FS = 0x10とし，関数の後に元のFSを復元する。
// namei()が呼び出されます。最後に、バッファブロックが解放され、指し示されたファイルのi-nodeが

```
        iput(inode);
```

```
250
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 */
    //// 与えられたパス名の一番上のディレクトリのi-nodeを検索します。
    // パラメータ：pathname - パス名、inode - 開始ディレクトリを指定するi-node。
    // 戻り値：ディレクトリのi-nodeポインタ、失敗した場合はNULL。
264         char c;

static struct m_inode *get_dir(const char *pathname,
```

```
struct m_inode * i_node)
{
        const char * thisname;
        struct buffer_head * bh;
        int namelen,inr;
        struct dir_entry * de;
        struct m_inode * dir;

// まず、パラメータの有効性を判断します。もし、与えられたi-nodeポインタのinodeが
// ディレクトリが空の場合は、現在のプロセスの作業ディレクトリi-nodeが使用されます。
// ユーザーがパス名の最初の文字を'/'と指定した場合、パス名は
// 絶対的なパス名で、ルート（または疑似ルート）のi-nodeセットから開始する必要があります。
// 現在のプロセスのタスク構造にある そこで、指定されたディレクトリi-nodeを戻す必要があります。
// またはパラメータで設定され、プロセスで使用されるルートi-nodeを取得した後、インクリメントします。
i-nodeの // 参照カウントと、パス名の最初の文字'/'を削除します。これにより
// 現在のプロセスは、自分が設定したルートのi-nodeを開始点としてのみ使用できることを
//      if (!inode) {
を検索します。

                inode = current->pwd;                   // i-node of the working directory.
                inode->i_count++;
        }
        if ((c=get_fs_byte(pathname))=='/') {
                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         }
```

// そして、パス名に含まれる様々なディレクトリ名やファイル名をループします。ループの中で
// 処理の際には、まずディレクトリ名のi-nodeの有効性を判断します。
// 処理されているディレクトリ名の部分を変数'thisname'に指定します。
// 処理されています。i-nodeが、現在処理されているディレクトリ名の部分が
// がディレクトリタイプではない場合や、ディレクトリに入るためのアクセス権限がない場合は
// i-nodeが戻され、NULLが返されます。もちろん，ループに入るときには，i-nodeの
// カレントディレクトリの'inode'は、プロセスルートのi-nodeまたはカレントディレクトリのi-nodeです。

```
//      while (1) {
```

作業ディレクトリ、またはパラメータで指定された検索開始ディレクトリのi-nodeです。

```
283
284                 thisname = pathname;
285                 if (!S_ISDIR(inode->i_mode) || !permission(inode,MAY_EXEC)) {
286                         iput(inode);
287                         return NULL;
288                 }
```

// 各ループでは、パス名に含まれる1つのディレクトリ名を処理します。そのため、各ループの中では

// パス名の文字列からディレクトリ名を その方法は，検出された文字を検索して
// 現在のパス名ポインタ'pathname'から末尾の文字になるまで
//（NULL）または'/'文字です。この時点で、変数「namelen」は、正確には、「NULL」の長さになります。
// 現在処理されているディレクトリ名の部分を、変数'thisname'が指しているので

// ディレクトリ名部分の先頭です。このとき、その文字が終了文字であれば
// NULLの場合は、パス名の最後まで検索したことを示し、最後に指定した
// ディレクトリ名やファイル名に到達した場合は、i-nodeポインタを返します。
// 注意：パス名の最後の名前がディレクトリ名でもあり、'/'文字が
// が付加されていない場合は、最後のディレクトリ名のi-nodeを返すことはありません!
// 例えば パス名「/usr/src/linux」の場合、この関数はi-nodeのみを返します。
src/'ディレクトリ名の//。

```
290                 for(namelen=0;(c=get_fs_byte(pathname++))&&(c!='/');namelen++)
291                         /* nothing */;
292                 if (!c)
293                         return inode;
```

// カレントディレクトリ名部分（またはファイル名）を取得した後、ルックアップディレクトリを呼び出す
// エントリ関数 find_entry() で、指定された名前のディレクトリエントリを
// 現在処理中のディレクトリ。(見つからなかった場合は、i-nodeを戻してNULLを返す。)その後。
// 見つかったディレクトリにi-node番号「inr」とデバイス番号「idev」が取り出される。
// エントリーがあると、そのディレクトリ・エントリーを含むキャッシュ・ブロックが解放され、i-nodeが置かれます。
//を返します。そして、ノード番号'inr'のi-node 'inode'を取り、次のループを続けます。
ディレクトリエントリをカレントディレクトリとするパス名の中の // ディレクトリ名部分。もし
// 現在処理されているディレクトリエントリがシンボリックリンク名であれば、follow_link() を使用する
// が指し示すディレクトリエントリ名のi-nodeを取得します。

```
295                 if (!(bh = find_entry(&inode,thisname,namelen,&de))) {
296                         iput(inode);
297                         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 *
313  * dir_namei() returns the inode of the directory of the
314  * specified name, and the name within that directory.
314 */
```

//// 指定されたディレクトリ名のi-nodeと、トップレベルのディレクトリ名を取得します。
// パラメータ: pathname - ディレクトリのパス名; namelen - パス名の長さ; name - ファイル名
// 一番上のディレクトリ名が返されます。base - 検索を開始するディレクトリのi-nodeです。

```
// 戻り値：指定されたディレクトリの最上位ディレクトリのi-nodeと名前と長さ
// 名前です。エラー時にはNULLを返します。
static struct m_inode * dir_namei(const char * pathname,
        int * namelen, const char ** name, struct m_inode * base)
{
        char c;
        const char * basename;
        struct m_inode * dir;
```

```
321
    // まず、指定されたパス名のトップレベルディレクトリのi-nodeを取得し、次に検索
    // とパス名を検出し、最後の'/'文字の後の名前文字列を見つけ、それを計算します。
    // 長さを指定し、最上位ディレクトリのi-nodeポインタを返します。なお，最後の
    パス名の//文字がスラッシュ文字'/'の場合、返されるディレクトリ名は
    // 空であり、長さは0です。 しかし、返されたi-nodeポインタはまだi-nodeを指しています。
322     // ディレクトリ名の最後の'/'文字より前の部分です。
323         if (!(dir = get_dir(pathname,base)))        // base is the i-node of the starting dir.
324                 return NULL;
325         basename = pathname;
326         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
    //// 指定されたパス名のi-nodeを取得する（内部機能）。
    // Parameters: pathname - パス名; base - 検索開始ディレクトリ i-node;
    // follow_links - シンボリックリンクをフォローするかどうか、1 - はい、0 - いいえ。
333 // 対応するi-nodeを返します。
334 struct m_inode * _namei(const char * pathname, struct m_inode * base,
335         int follow_links)
335 {
336         const char * basename;
337         int inr,namelen;
338         struct m_inode * inode;
339         struct buffer_head * bh;
340         struct dir_entry * de;
341
    // まず、指定されたパス名の最上位ディレクトリのディレクトリ名を見つけて
    // そのi-node。存在しない場合は，NULLを返して終了します．もし，一番上の
    // 返された名前が0の場合は、パス名がディレクトリの最後の項目にちなんだ名前になっていることを意
    味します。
    // したがって、対応するディレクトリのi-nodeを見つけたので、それを返すことができます。
    // i-nodeを直接入力します。返された名前の長さが0でなければ、dir_namei()を呼び出します。
    // 新たに指定した開始位置を持つトップレベルのディレクトリ名を検索するために、再度関数を実行す
    る
    //      if (!(dir = dir_namei(pathname,&namelen,&basename)))
    ディレクトリベースで、
```

返された情報をもとに同様の判断をする。

```
311
312                 return NULL;
313         if (!namelen)                   /* special case: '/usr/' etc */
314                 return dir;
342         if (!(base = dir_namei(pathname,&namelen,&basename,base)))
343                 return NULL;
344         if (!namelen)                   /* special case: '/usr/' etc */
345                 return base;
```

// 続いて、指定されたファイル名のディレクトリエントリのi-nodeを、返されたトップレベルの
// ディレクトリです。もう一度注意してください! もし、最後にもディレクトリ名ですが、そこには
// の後に「/」が付いていない場合は、最後のディレクトリのi-nodeを返しません。例えば、以下のようになります。
// '/usr/src/linux' は 'src/' ディレクトリ名の i-node のみを返します。関数
// dir_namei()は、'/'で終わらない最後の名前をファイル名として扱いますが、これは必要なことです。
// この状況を処理するために、i-nodeのfind directory entry関数find_entry()を使用します。
// を別々に表示します。この時点で、'de'にはディレクトリ・エントリ・ポインタが見つかり、'dir'には

```
//ディレクトリエントリを含むディレクトリのi-nodeポインタ。
        bh = find_entry(&base,basename,namelen,&de);

        if (!bh) {
                iput(base);
                return NULL;
        }
// 次に、ディレクトリ・エントリのi-node番号を取り、以下を含むキャッシュ・バッファ・ブロックを解放します。
// ディレクトリエントリを、ディレクトリのi-nodeに戻します。その後、i-nodeを取りノード番号に対応する // アクセスタイムを現在時刻に修正して
// 修正されたフラグです。最後に，i-nodeポインターinodeが返されます。さらに，現在の
// 処理されたディレクトリエントリがシンボリックリンク名であれば、follow_link() を使って
        inr = de->inode;
```

```
351
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 */
    //// ファイルを開くために使用される名目上の関数です。
    // パラメータ： filename - ファイルのパス名， flag - ファイルを開く際のフラグで，以下の値を取ることができます。
    // O_RDONLY (読み取り専用)、O_WRONLY (書き込み専用)、O_RDWR (読み取りと書き込み)、O_CREAT などの値があります。
    //（作成）、O_EXCL（ファイルが存在してはならない）、O_APPEND（ファイルの最後に追加）などの
    // Mode - 指定されたファイルの許可属性。
    // 新しいファイルを作成します。これらの属性は，S_IRWXU（ファイルの所有者が読み取り，書き込み，および実行の権限を持つ
    // パーミッション）、S_IRUSR（ユーザーがファイルの読み取りパーミッションを持っている）、S_IRWXG（グループメンバーが読み取り。
    //書き込み、および実行権限）などがあります。新規に作成されたファイルの場合、これらの属性は
    // 将来のファイルへのアクセスにのみ使用され、読み取り専用のファイルを作成するオープンコールは
    // また、読み書き可能なファイルハンドルを返します。インクルードファイルsys/stat.h, fcntl.hを参照してください。
    // 戻り値：正常に返された場合は0、そうでない場合はエラーコードが返されます; res_inode - 戻り値
    int open_namei(const char * pathname, int flag, int mode,
388
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
```

// まず，関数のパラメータが合理的に処理されます。もし，ファイルアクセスモードフラグが
// 読み込み専用(0)だが、ファイル切り捨てフラグO_TRUNCが設定され、書き込み専用フラグO_WRONLYが
// をファイルオープンフラグに追加しています。その理由は、切り捨てフラグ O_TRUNC が
// ファイルが書き込み可能な場合にのみ有効です。そして，現在のファイルのアクセス許可マスクを使用して
// 処理し、与えられたモードで対応するビットをマスクし、通常のファイルフラグI_REGULARを追加します。
// このフラグは、開いているファイルがない場合、新規ファイルのデフォルトプロパティとして使用されます。
// が存在し、作成する必要があります。以下の411行目のメモを参照してください。

```
397         // が存在し、作成する必要があります。以下の411行目のメモを参照してください。
398         if ((flag & O_TRUNC) && !(flag & O_ACCMODE))
399                 flag |= O_WRONLY;
400         mode &= 0777 & ~current->umask;
401         mode |= I_REGULAR;                // regular file flag. See file include/const.h.
```

// そして、指定されたパス名とトップディレクトリにしたがって、対応するi-nodeを見つけます。
// の名前とその長さを指定します。このとき、トップディレクトリ名の長さが0の場合（例えば、'/usr/'）。
// ならば、操作が読み書き、作成、ファイル切り捨て0でない場合は
// ディレクトリ名のファイル操作が開いています。そのため、直接ディレクトリのi-nodeを返したり
// code is returned.

```
        if (!(dir = dir_namei(pathname,&namelen,&basename,NULL)))
```

```

                return -ENOENT;
        if (!namelen) {                 /* special case: '/usr/' etc */
                if (!(flag & (O_ACCMODE|O_CREAT|O_TRUNC))) {
                        *res_inode=dir;
                        return 0;
                }
                iput(dir);
                return -EISDIR;
        }
```

// そして、上記で取得したトップレベルのディレクトリ名のi-nodeに従って、ディレクトリのパス名の最後のファイル名に対応する // エントリ構造が検索されます。

```
//ディレクトリエントリのキャッシュブロックが同時に取得されます。もし、キャッシュブロックのポインタ
//がNULLの場合は、ファイル名に対応するディレクトリエントリが見つからないことを意味するので
//ファイルを作成することしかできません。このとき、ファイルを作成しない操作の場合は
//または、ユーザーがディレクトリに書き込む権利を持っていない場合は、i-nodeを戻します。
//ディレクトリを作成し、対応するエラーコードを返して終了します。
        bh = find_entry(&dir,basename,namelen,&de);
        if (!bh) {
                if (!(flag & O_CREAT)) {
                        iput(dir);
                        return -ENOENT;
                }
                if (!permission(dir,MAY_WRITE)) {
                        iput(dir);
                        return -EACCES;
                }
//これで、作成操作であり、書き込み許可を持っていると判断しました。したがって
//パス名で指定されたファイル名を使用するために、デバイス上の新しいi-nodeを申請します。
// flag. Then add a new directory entry in the specified directory dir.
                inode = new_inode(dir->i_dev);
```

```
421
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);
```

// 新しいディレクトリエントリを含むべきキャッシュブロックポインタがNULLの場合は
// は、ディレクトリエントリの追加操作が失敗したことを意味します。その後、エラーロールバック処理が実行されます。

```
// 新しいi-nodeの参照接続カウントが1だけデクリメントされ、i-nodeが置かれます。
// を返し、エラーコードexitを返します。それ以外の場合は、ディレクトリの追加操作は
// のエントリーは成功です。そこで、新しいディレクトリ・エントリの初期値をいくつか設定します。
// 番号を新たに適用されたi-nodeの番号に変更し、キャッシュブロック修正フラグを設定する。その後
// 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         }
```

// 上記（411行目）でファイルに対応するディレクトリエントリの取得に成功した場合は
// の名前である場合（すなわち、bhがNULLでない場合）、指定されたファイルがすでに存在することを示します。その後は
// ディレクトリ・エントリのi-node番号とデバイス番号が取得され、キャッシュ・ブロックの
//とディレクトリのi-nodeが解放されます。排他的操作フラグO_EXCLが設定されている場合。

```
//      inr = de->inode;
が、ファイルがすでに存在している場合は、エラーコード(ファイルが存在する)を返して終了します。
443
444         dev = dir->i_dev;
445         brelse(bh);
446         if (flag & O_EXCL) {
447                 iput(dir);
448                 return -EEXIST;
449         }
    // そして、ディレクトリ・エントリのi-nodeの内容を読みます。i-nodeがディレクトリのi-nodeであれば
    // そして、アクセスモードが書き込み専用や読み書き、またはアクセス許可がない場合は、元に戻します。
450     // i-nodeのアクセス許可エラーコードを返します。
451         if (!(inode = follow_link(dir,iget(dev,inr))))
452                 return -EACCES;
453         if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
454             !permission(inode,ACC_MODE(flag))) {
```

```
455                 iput(inode);
456                 return -EPERM;
456         }
```

// そして、i-nodeのaccess timeフィールドを現在の時刻に更新します。もし、切り捨てフラグが
// が設定されている場合は，ファイルサイズが0に切り捨てられます。最終的には，iノードへのポインタを返します.

```
457     // 0（成功）を返します。
458         inode->i_atime = CURRENT_TIME;
459         if (flag & O_TRUNC)
460                 truncate(inode);
461         *res_inode = inode;
462         return 0;
462 }
463
```

//// デバイスの特殊ファイルまたは通常のファイルノードを作成します。
// ファイルシステムノード（コモンファイル，デバイススペシャルファイル，名前付きパイプ）を作成します。
// モードとdevで指定された名前のファイル名。
// Parameters: filename - ファイル名のパス名; mode - 使用する許可を指定して
// 作成されたノードのタイプ、dev - デバイス番号。
// 戻り値：成功した場合は0、そうでない場合はエラーコードを返します。

```
466 464 int sys_mknod(const char * filename, int mode, int dev) 465 {。
467         const char * basename;
468         int namelen;
469         struct m_inode * dir, * inode;
470         struct buffer_head * bh;
471         struct dir_entry * de;
471
```

// まず、操作権限とパラメータの有効性を確認し、i-nodeの
// パス名に含まれるトップレベルのディレクトリを指定します。スーパーユーザでない場合は、アクセス許可を返します。
// エラーコードです。パス名のトップレベルディレクトリに対応するi-nodeが
// 見つかった場合は、エラーコードが返されます。トップファイル名の長さが0の場合は、与えられた
// パス名の最後にファイル名が指定されていないか、プロセスに書き込み権限がない場合

```
ディ        if (!suser())
```

ディレクトリの中に//がある場合は、ディ

レクトリのi-nodeを戻して、エラーコードを返します。

```
472
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         }
```

// そして、パス名で指定されたファイルが既に存在している場合は、そのファイルを検索します。
// 同名のファイルノードを作成することはできません。そのため，ディレクトリエントリの
// パス名に対応する最後のファイル名が既に存在する場合、そのファイルを含むバッファブロックは
// ディレクトリエントリが解放され、ディレクトリのi-nodeが戻され、エラー

```
484
```

// ファイルがすでに存在するというコードが返されます。

```
485         bh = find_entry(&dir,basename,namelen,&de);
486         if (bh) {
487                 brelse(bh);
488                 iput(dir);
489                 return -EEXIST;
489         }
```

// ファイル名のディレクトリエントリが見つからない場合は、新しいi-nodeを申請して
// i-nodeの属性モードを設定します。ブロックデバイスファイルを作成している場合や、キャラクタ
// デバイスファイルでは、i-nodeのダイレクト論理ブロックポインタ0をデバイス番号と同じにする。
// つまり、デバイスファイルの場合、i-nodeのi_zone[0]には、デバイス番号が格納されています。
// デバイスファイルで定義されたデバイスです。の修正時間とアクセス時間を設定します。

```
//      inode = new_inode(dir->i_dev);
```

i-nodeを現在の時間に変更し、i-nodeの修正フラグを設定する。

```
490
491         if (!inode) {           // put back inode and return error code if failed.
492                 iput(dir);
493                 return -ENOSPC;
```

```
494         }
495         inode->i_mode = mode;
496         if (S_ISBLK(mode) || S_ISCHR(mode))
497                 inode->i_zone[0] = dev;
498         inode->i_mtime = inode->i_atime = CURRENT_TIME;
499         inode->i_dirt = 1;
    // そして、この新しいi-nodeのために新しいディレクトリ・エントリを追加します。失敗した場合（キャッシュブロックポインタが
    ディレクトリのエントリーを含む // がNULLの場合）、ディレクトリのi-nodeを戻し、リセットします。
500 // 要求されたi-nodeの参照接続数を返し、i-nodeを戻し、エラーコードを返します。
501         bh = add_entry(dir,basename,namelen,&de);
502         if (!bh) {
503                 iput(dir);
504                 inode->i_nlinks=0;
505                 iput(inode);
506                 return -ENOSPC;
506         }
    // これでディレクトリエントリの追加操作も成功したので、この内容を
    // ディレクトリ・アイテムです。ディレクトリ・エントリのi-nodeフィールドを新しいi-node番号と同じにする。
    // そして、キャッシュブロック修正フラグを設定し、ディレクトリと新しいi-nodeを戻し、リリースします。
    //      de->inode = inode->i_num;
    //そして最後に0（成功）を返します。
507
508         bh->b_dirt = 1;
509         iput(dir);
510         iput(inode);
511         brelse(bh);
```

```
512         return 0;
513 }
514
    ////  Make a directory (used in system-call).
    // Parameters: pathname - the path name; mode - permission attribute used by the directory.
    // Returns: 0 if successful, otherwise returns an error code.
515 int sys_mkdir(const char * pathname, int mode)
516 {
517         const char * basename;
518         int namelen;
519         struct m_inode * dir, * inode;
520         struct buffer_head * bh, *dir_block;
521         struct dir_entry * de;
522
    // First check the validity of the operation permissions and parameters and get the i-node of
    // パス名にトップレベルのディレクトリが含まれています。トップレベルのディレクトリに対応するi-nodeが
    // パス名に含まれるディレクトリが見つからない場合は、エラーコードが返されます。トップファイル名が
    // 長さが0の場合、与えられたパス名の最後にファイル名が指定されていないことを意味します。
    // プロセスがディレクトリに書き込み権限を持っていない場合は、i-nodeを戻します。
523     // ディレクトリを作成し、エラーコードを返します。
524         if (!(dir = dir_namei(pathname,&namelen,&basename, NULL)))
525                 return -ENOENT;
526         if (!namelen) {
527                 iput(dir);
528                 return -ENOENT;
528         }
529         if (!permission(dir,MAY_WRITE)) {
530                 iput(dir);
531                 return -EPERM;
532         }
    // 次に、パス名で指定されたディレクトリ名が既に存在するかどうかを検索し、既に存在する場合は
    // が存在する場合、同じ名前のディレクトリノードを作成することはできません。したがって、もしディレクトリ
    // パス名の最後のディレクトリ名のエントリが既に存在する場合、そのエントリを含むバッファブロックは
    // ディレクトリエントリが解放され、ディレクトリのi-nodeが戻され、エラー
    // ファイルがすでに存在するというコードが返されます。そうでない場合は，新しいi-nodeを申請して
    // i-nodeのアトリビュートモード：新しいi-nodeに対応するファイルサイズを32に設定する
    //バイト（2つのディレクトリエントリのサイズ）、ノードの修正フラグを設定し、ノードの
    // 修正時間とアクセス時間です。2つのディレクトリエントリは、「...」と「...」のディレクトリに使用されます。
    //      bh = find_entry(&dir,basename,namelen,&de);
    //それぞれ
533
534         if (bh) {
```

```
535                 brelse(bh);
536                 iput(dir);
537                 return -EEXIST;
538         }
539         inode = new_inode(dir->i_dev);
540         if (!inode) {               // put back inode and return error code if failed.
541                 iput(dir);
542                 return -ENOSPC;
543         }
544         inode->i_size = 32;
545         inode->i_dirt = 1;
546         inode->i_mtime = inode->i_atime = CURRENT_TIME;
```

// 次に、新しいi-nodeのディレクトリ・エントリ・データを保存するためのディスク・ブロックを申請します。
// そして、i-nodeの最初のダイレクトブロックポインタをブロック番号と同じにします。もし、そのような
// アプリケーションが失敗したら、ディレクトリのi-nodeを戻し、新しく要求されたi-nodeをリセットする
// リンクカウント；新しいi-nodeを戻し、スペースエラーコードを返さない。そうでなければ，新しい
547 // i-node modified flag.
548 if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
549 iput(dir);
550 inode->i_nlinks--;
551 iput(inode);
552 return -ENOSPC;
552 }
553 inode->i_dirt = 1;
// ここで、新たに要求されたディスクブロックをデバイスから読み込みます（目的は、デバイスに
// 対応するブロックをバッファキャッシュに入れる）。同様に、エラーが発生した場合は、i-nodeを戻す
ディレクトリの // 要求されたディスクブロックを解放し，新たに要求された i-node リンクをリセットする。
554 // count; 新しいi-nodeを戻し、no spaceエラーコードを返して終了します。
555 if (!(dir_block=bread(inode->i_dev,inode->i_zone[0]))) {
556 iput(dir);
557 inode->i_nlinks--;
558 iput(inode);
559 return -ERROR;
559 }
// 次に、作成したディレクトリに2つのデフォルトの新しいディレクトリ・エントリ（'.'と'.'）データを作成します。
// ファイルをバッファブロックに格納します。まず、'de'にディレクトリを保持するデータブロックを指定します。
// エントリーのi-node numberフィールドを、新しく適用された
// i-node番号、nameフィールドには"."が設定されます。すると'de'は次のディレクトリ・エントリを指す
// 構造体に、親ディレクトリのi-node番号と名前「...」を格納します。
// そして、キャッシュブロックモディファイドフラグをセットし、バッファブロックをリリースします。モードを再初期化します。
// 新しいi-nodeのフィールドを作成し、i-node modifiedフラグを設定します。
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;
    // ここで、指定したディレクトリに新しいディレクトリ・エントリを追加して、i-nodeとディレクトリ
    を格納します。
    // 新しく作成されたディレクトリの名前。失敗した場合（バッファブロックポインタの
    // ディレクトリ・エントリーがNULLの場合）、ディレクトリのi-nodeを戻します；要求されたi-node
571     // 参照リンクカウントがリセットされ、i-nodeに戻されます。エラーコードを返して終了する。
572         bh = add_entry(dir,basename,namelen,&de);
573         if (!bh) {
574                 iput(dir);
575                 inode->i_nlinks=0;
576                 iput(inode);
577                 return -ENOSPC;
577         }
    // 最後に、新しいディレクトリ・エントリのi-nodeフィールドは、新しいi-node番号に等しく、その
    // キャッシュブロック修正フラグが設定され、ディレクトリと新しいi-nodeが戻され、バッファ
```

```
        de->inode = inode->i_num;
```

```
        bh->b_dirt = 1;
        dir->i_nlinks++;
        dir->i_dirt = 1;
        iput(dir);
        iput(inode);
        brelse(bh);
        return 0;
}

/*
 * routine to check that the specified directory is empty (for rmdir)
 */
//// Check if the directory is empty.
// Parameters: inode - the i-node pointer of the specified directory.
// Returns: 1 - the directory is empty; 0 - not empty.
static int empty_dir(struct m_inode * inode)
{
        int nr,block;
        int len;
        struct buffer_head * bh;
        struct dir_entry * de;

// First calculate the number of existing directory entries in the specified directory and check
// if the information in the two specific directory entries is correct. There should be at least
// 2 directory entries in a directory: the entries whose names are "." and "..". If the number
// of directory entries is less than 2 or the first direct block of the i-node of the directory
// does not point to any disk block number, or the direct block cannot be read, then the warning
// message is displayed, and 0 is returned. (failure).
        len = inode->i_size / sizeof (struct dir_entry);   // number of directory entries.
        if (len<2 || !inode->i_zone[0] ||
            !(bh=bread(inode->i_dev,inode->i_zone[0]))) {
```

```
601                 printk("warning - bad directory on dev %04x\n",inode->i_dev);
602                  return 0;
603         }
    // At this time, the buffer block indicated by 'bh' contains directory entry data. We let the
    // ディレクトリエントリポインタ 'de' バッファブロック内の最初のディレクトリエントリを指します。
    については
    // 最初のディレクトリ・エントリ（"."）の場合、そのi-node numberフィールドはi-node numberと同じ
    でなければなりません。
    カレント・ディレクトリの // 2番目のディレクトリ・エントリ（"..."）の場合、そのi-node番号フィール
    ド
    // はアップレイヤーディレクトリのものと等しく、0にはなりません。 したがって、i-node
    // 最初のディレクトリ・エントリの番号がカレント・ディレクトリの番号と一致しない場合、または
    // 2つ目のディレクトリ・エントリのi-node番号が0であるか、または2つのディレクトリのnameフィー
    ルドが
    // エントリーが"."と"..."に等しくない場合は、エラー警告メッセージが表示され、0が返されます。
604         de = (struct dir_entry *) bh->b_data;
605         if (de[0].inode != inode->i_num || !de[1].inode ||
606             strcmp(".",de[0].name) || strcmp("..",de[1].name)) {
607                 printk("warning - bad directory on dev %04x\n",inode->i_dev);
608                 return 0;
609         }
    // 次に、'nr'をディレクトリのエントリ番号（0から数えたもの）にし、'de'を
    // の残りの(len - 2)のディレクトリ・エントリをすべてループします。
    // i-node numberフィールドが0（使用済み）でないかどうか、ディレクトリを確認します。
```

```
610         nr = 2;
611         de += 2;
612         while (nr<len) {
```

// ディスクブロック内のディレクトリエントリが完全にチェックされており、ディレクトリ使用されている // エントリが見つからない場合、ディスクブロックのバッファブロックが解放され、次の
// ディレクトリデータファイルのディレクトリエントリを含むディスクブロックが読み込まれます。読み込みの
// 方法は、対応するデータブロック番号（nr/DIR_ENTRIES_PER_BLOCK）を計算することです。
// 現在検出されているディレクトリに応じたディレクトリファイルのエントリ項目
// アイテム番号「nr」を、bmap()関数でブロック番号を取得してから
// ブロックデバイス関数 bread() がディスクブロックをバッファキャッシュに読み込んで返します。
// バッファブロックへのポインタです。ディスクブロックが使用されていない（あるいは使用されたことがない，そのような
// ファイルが削除されたなどの理由で）、次のブロックの読み取りを続行し、読み取れない場合は
// の場合、エラーは0を返します。それ以外の場合は、'de'にブロック内の最初のディレクトリエントリを指定します。

```
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                 }
```

// 現在のディレクトリ・エントリについて、そのi-node numberフィールドが0に等しくない場合、それは次のことを意味します。
// そのディレクトリ項目が以前から使用されていて、ディレクトリが空ではないことを示している場合は
// バッファブロックが解放され、0が返されます。それ以外の場合は，すべてのディレクトリエントリディレクトリ内の//がクエリされていない場合、ディレクトリエントリ番号「nr」がインクリメントされて

```
// 'de'は次のディレクトリエントリを指しており、検出は継続されます。
                if (de->inode)
{ 625                   brelse(bh);
626                     return 0;
627             }
628             de++;
629             nr++;
630         }
// ここでコードが実行された場合、使用されたディレクトリエントリが
// ディレクトリ（もちろん最初の2つを除く）に移動した後、バッファブロックが解放され、1
// を返します。
631         brelse(bh);
632         return 1;                       // directory is empty!
633 }
634
//// ディレクトリを削除します。
// パラメータ： name - ディレクトリ名（パス名）です。
// を返します。成功した場合は0を、そうでない場合はエラー番号を返します。
635 int sys_rmdir(const char * name)
636 {.
638         const char * basename;
639         int namelen;
640         struct m_inode * dir, * inode;
641         struct buffer_head * bh;
642         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 filename
    // 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         }
    // そして、指定されたディレクトリのi-nodeとディレクトリ名に応じて、関数
    // find_entry()は、ディレクトリ・エントリを見つけるために、そのエントリを含むバッファ・ブロック
    'bh'を使用します。
    // ディレクトリ・エントリ、ディレクトリのi-node 'dir'、ディレクトリ・エントリ 'de' が返されます。
    // そして、ディレクトリ・エントリ「de」のi-node番号にしたがって、対応するi-nodeの
    // は、iget()関数を使って取得します。特定の操作の過程で、もしディレクトリ
    // パス名の最後のディレクトリ名のエントリーが存在しない場合、そのエントリーを含むバッファブロ
    ックは
    // ディレクトリ・エントリが解放され、ディレクトリのi-nodeが戻されると、エラー・コード
    // のファイルがすでに存在している場合は、その旨が返されます。ディレクトリエントリのi-nodeを取
    得する操作が
        // が間違っていると、ディレクトリのi-nodeが戻され、ディレクトリを含むバッファブロックが
    // 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         }
    // この時点では、ディレクトリのi-node 'dir' 、削除されるディレクトリ・エントリ 'de' があります。
    //とそれに対応するi-nodeがあります。以下では、削除の実行可能性を確認するために
    // この3つのオブジェクトの情報を
    //
    // ディレクトリに削除制限フラグが設定されていて、かつ、そのディレクトリの
    // プロセスがルートではなく、プロセスの実効ユーザーID（euid）が
    // i-nodeのユーザーIDは、現在のプロセスが削除する権限を持っていないことを意味します。
    // ディレクトリなので、ディレクトリのi-nodeとエントリーのi-nodeを戻して、次に
     //     if ((dir->i_mode & S_ISVTX) && current->euid &&
```

バッファブロックにエラーコードを返します。

```
663
664             inode->i_uid != current->euid) {
665                 iput(dir);
666                 iput(inode);
667                 brelse(bh);
668                 return -EPERM;
669         }
```

// ディレクトリエントリのi-nodeのデバイス番号が、i-nodeのデバイス番号と等しくない場合。
// このエントリーを含むディレクトリ、または削除するディレクトリの参照リンク数が
// が1より大きい場合（シンボルリンクがあることなど）、そのディレクトリは削除できません。
// そのため、削除されるディレクトリ名を含むディレクトリのi-nodeと、エントリのi-nodeを
// が解放され、バッファブロックが解放され、エラーコードが返されます。

```
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         }
```

// ディレクトリエントリのi-nodeがディレクトリのi-nodeと等しい場合、そのことを示します。
// "."ディレクトリを削除しようとしますが、これは許されません。すると、ディレクトリ・エントリのi-node
// 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         }
```

// 削除されるディレクトリのi-nodeの属性が、それがないことを示している場合は
// ディレクトリには、この削除操作の前提となるものが全く存在しません。すると、i-nodeの
削除されるディレクトリ・エントリの // とそのディレクトリの i-node が戻されると、バッファ
// ブロックが解除され、エラーコードが返されます。

```
682         if (!S_ISDIR(inode->i_mode))
{ 683               iput(inode);
684                 iput(dir);
685                 brelse(bh);
686                 return -ENOTDIR;
687         }
```

// 削除するディレクトリが空でない場合は、削除できません。すると、ディレクトリのi-node
// 削除されるディレクトリ名と削除されるディレクトリのi-nodeを含む
// が戻され、バッファブロックが解放され、エラーコードが返されます。

```
688         if (!empty_dir(inode))
{ 689               iput(inode);
690                 iput(dir);
691                 brelse(bh);
692                 return -ENOTEMPTY;
693         }
```

// 空のディレクトリの場合，ディレクトリ・エントリ・リンクの数は 2（上部の
//ディレクトリと自身）を削除します。削除するi-nodeのリンク数が、以下の値に等しくない場合
//2の場合は、警告メッセージが表示されますが、削除操作は続行されます。その後、i-nodeの
削除するディレクトリのディレクトリエントリの//番号フィールドが0に設定されていることを示します。
// ディレクトリエントリが使用されなくなり、バッファブロックモディファイドフラグが設定されてい
ることを示す
// バッファブロックを解放します。そして、削除されたi-nodeのリンク数を設定します。

```
//      if (inode->i_nlinks != 2)
ディ
```

レクトリを0（アイドルを示す）にして、i-nodemodifiedフラグを設定します。

```
694
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;
    // 次に、削除されたディレクトリ名を含むディレクトリのi-nodeリンクカウントを減らす
    //を1ずつ加算し、変更時刻と修正時刻を現在の時刻に修正し、修正した
    // ノードのフラグです。最後に、削除されるディレクトリ名を含むディレクトリi-ノード
    //と削除するディレクトリのi-nodeが戻され、0が返されます（削除の
    // 操作が成功した場合）。
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
    //// ファイル名に対応するディレクトリエントリを削除（解除）します。
    // ファイルシステムから名前を削除します。それがファイルへの最後のリンクであり、どのプロセスも
    // ファイルを開くと、そのファイルも削除され、占有していたデバイスの領域が解放されます。
    // パラメータ： name - ファイル名（パス名）です。
    // 戻り値：成功した場合は0、そうでない場合はエラーコードを返します。
711 709 int sys_unlink(const char * name) 710
{.
712         const char * basename;
713         int namelen;
714         struct m_inode * dir, * inode;
715         struct buffer_head * bh;
716         struct dir_entry * de;
716
    // まず、操作権限とパラメータの有効性を確認し、i-nodeの
    // パス名にトップレベルのディレクトリが含まれています。トップレベルのディレクトリに対応するi-
    nodeが
    // パス名に含まれるディレクトリが見つからない場合は、エラーコードが返されます。トップファイル
    名が
    // 長さが0の場合、与えられたパス名の最後にファイル名が指定されていないことを意味します。
    // プロセスがディレクトリに書き込み権限を持っていない場合は、i-nodeを戻します。
    // ディレクトリを作成し、エラーコードを返します。
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         }
    // そして、指定されたディレクトリのi-nodeとディレクトリ名に応じて、関数
    // find_entry()は、ディレクトリ・エントリを見つけるために、そのエントリを含むバッファ・ブロック
```

'bh'を使用します。

// ディレクトリ・エントリ、ディレクトリのi-node 'dir'、ディレクトリ・エントリ 'de' が返されます。

// そして、ディレクトリ・エントリ「de」のi-node番号にしたがって、対応するi-nodeの

```
//      bh = find_entry(&dir,basename,namelen,&de);
は
ig
et
()
関
数
を
使
っ
て
取
得
し
ま
す
。
727
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         }
    // この時点では、ディレクトリのi-node 'dir' 、削除されるディレクトリ・エントリ 'de' があります。
    //とそれに対応するi-nodeがあります。以下では、削除の実行可能性を確認するために
    // この3つのオブジェクトの情報を
    //
    // ディレクトリに削除制限フラグが設定されていて、有効なユーザーID（euid）が
    // プロセスはルートではなく、プロセスのeuidはi-nodeのユーザーidと等しくありません。
    // そして、プロセスのeuidがディレクトリi-nodeのユーザーidと等しくない場合、以下を示します。
    // 現在のプロセスがディレクトリを削除する権限を持っていないことを示します。その後、戻して
    // 削除されるディレクトリ名を含むディレクトリのi-nodeと
    // 削除するディレクトリを指定した後、バッファブロックを解放し、エラーコードを返します。
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         }
    // 指定されたファイル名がディレクトリの場合は、削除できません。そのときは、i-nodeの
    // ディレクトリのi-nodeとファイル名のディレクトリ・エントリが戻されると、その中のバッファ・ブロックは
    // ディレクトリエントリが解放され、エラーコードが返されます。
745         if (S_ISDIR(inode->i_mode))
{ 746               iput(inode);
747                 iput(dir);
748                 brelse(bh);
749                 return -EPERM;
750         }
    // i-nodeのリンクカウントが既に0の場合、警告メッセージが表示され、1に修正されます。
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         }
    // ここで、ファイル名に対応するディレクトリエントリを削除します。すると、i-nodeの番号
    ファイル名ディレクトリエントリの // フィールドが 0 に設定されている場合、そのディレクトリエントリが
    //解放され、バッフ
        de->inode = 0;
```

ァブロック修正フラグが設定され、バッファブロックが解放されます。

```
756
757         bh->b_dirt = 1;
758         brelse(bh);
```

// そして、ファイル名に対応するi-nodeのリンク数をデクリメントします。
// 1であれば、修正フラグが設定され、更新時刻は現在時刻に設定されます。最後に
// i-nodeとディレクトリのi-nodeを戻して、0（成功）を返します。もしそれが最後のリンクの
// ファイル、つまり、i-nodeのリンク数から1を引いた値が0になり、どのプロセスも開いていない状態
// この時点では、iput()がファイルを戻すために呼び出されたときに、ファイルは削除されます。
// i-node と占有していたデバイス空間を解放する。fs/inode.cの183行目を参照。

```
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
    //// Create a symbolic link.
    // Create a symbolic link (also known as a soft 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.
767 int sys_symlink(const char * oldname, const char * newname)
768 {
769         struct dir_entry * de;
770         struct m_inode * dir, * inode;
771         struct buffer_head * bh, * name_block;
772         const char * basename;
773         int namelen, i;
774         char c;
775
    // First 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, an error code
    // is returned. If the file name is not included in the new path name, the i-node of the new
    // path name directory is put back and the error code is returned. In addition, if the user
    // does not have permission to write in the new directory, then the link cannot be established.
    // Then put back the i-node of the new path name directory and return the error code.
776         dir = dir_namei(newname,&namelen,&basename, NULL);
777         if (!dir)
778                 return -EACCES;
779         if (!namelen) {
780                 iput(dir);
781                 return -EPERM;
782         }
783         if (!permission(dir,MAY_WRITE)) {
784                 iput(dir);
785                 return -EACCES;
786         }
    // ここで、ディレクトリで指定されたデバイスに新しいi-nodeを申請し、i-nodeを設定します。
    //モードをプロセスが指定したシンボリックリンクタイプとモードマスクコードに設定して
    //      if (!(inode = new_inode(dir->i_dev))) {
    i-node modified flag.
787
788                 iput(dir);
789                 return -ENOSPC;
790         }
791         inode->i_mode = S_IFLNK | (0777 & ~current->umask);
792         inode->i_dirt = 1;
    // シンボリックリンクのパス名の文字列を保存するためには、ディスクブロックを
    // i-nodeで、最初のダイレクトブロック番号i_zone[0]を、得られた論理的な
```

// ブロック番号を指定して、i-node modified flagを設定します。アプリケーションが失敗した場合は、元に戻して
// the new i-node, and return no space error code.

```
793         if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
794                 iput(dir);
795                 inode->i_nlinks--;
796                 iput(inode);
```

```
797                 return -ENOSPC;
798         }
799         inode->i_dirt = 1;
    // 新たに要求されたディスクブロックは、デバイスから読み込まれます（目的は、ディスクブロックに
    // ブロックをバッファキャッシュに格納します）。エラーが発生した場合は、ディレクトリのi-nodeを
    戻します; reset
    // 新しく要求されたi-nodeのリンクカウント；新しいi-nodeを戻して、エラーコードを返す。
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         }
    // これで、シンボリックリンク名の文字列をこのディスクブロックに入れることができます。ディスク
    ブロックの長さは
    // は1024バイトなので、デフォルトのシンボルリンク名の最大長は1024バイトにしかなりません。私た
    ちは
    // ユーザ空間のシンボリックリンク名の文字列を、ディスクブロックのあるバッファブロックにコピー
    する。
    // が配置されており、バッファブロックモディファイドフラグを設定しています。ユーザーが入力した
    文字列が
    // ヌルを入れずに終了した場合、バッファブロックのデータ領域の最後のバイトにヌルを入れる必要が
    あります。
    // その後、バッファブロックを解放し、対応するファイルのデータのサイズを
    シンボリックリンク名の文字列の長さと同じになるように // i-nodeを設定します。
    //      i = 0;
    修
    正
    さ
    れ
    た
    フ
    ラ
    グ
    。
806
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;
    // 次に、パス名で指定されたシンボリックリンクファイルの名前を検索します。もしそれがすでに
    //が存在する場合、同じ名前のディレクトリエントリi-nodeを作成することはできません。もし、対応
    する
    // シンボリックリンクのファイル名が既に存在する場合、ディレクトリエントリを含むバッファブロッ
    クは
    // リリースされ、新たに要求されたi-nodeのリンクカウントがリセットされ、ディレクトリのi-nodeが
    // が戻され、ファイルがすでに存在するというエラーコードが返されます。
```

```
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         }
```

// ここで、指定したディレクトリに、i-nodeを格納するための新しいディレクトリ・エントリを追加します。

新たに作成されたシンボリックリンクのファイル名の // 番号とディレクトリ名を指定します。失敗した場合（

// ディレクトリエントリを含むバッファブロックポインタがNULLの場合），ディレクトリのi-node

```
// and the error code is returned.
        bh = add_entry(dir,basename,namelen,&de);
```

```

        if (!bh) {
                inode->i_nlinks--;
                iput(inode);
                iput(dir);
                return -ENOSPC;
        }
```

```
// 最後に、新しいディレクトリ・エントリのi-nodeフィールドは、新しいi-node番号に等しく、その
// バッファブロック修正フラグが設定されると、バッファブロックが解放され、ディレクトリと新
//      de->inode = inode->i_num;
i-nodeが戻され、最後に0（成功）が返されます。

        bh->b_dirt = 1;
        brelse(bh);
        iput(dir);
        iput(inode);
        return 0;
}

//// 既存のファイルにファイル名のディレクトリエントリを作成します。
// 既存のファイルに新しいリンク（ハードリンクとも呼ばれる）を作成します。
// パラメータ: oldname - 元のパス名, newname - 新しいパス名。
// 戻り値：成功した場合は0、そうでない場合はエラーコードを返します。
int sys_link(const char * oldname, const char * newname) {.
        struct dir_entry * de;
        struct m_inode * oldinode, * dir;
        struct buffer_head * bh;
        const char * basename;
        int namelen;

// まず、元のファイル名を検証します。ディレクトリ名ではなく、存在している必要があります。そこで、まず
// 元のファイルパス名に対応するi-node 'oldinode'を取る。0であれば、それは
// エラーが発生し、エラーコードを返します。元のパス名がディレクトリに対応する場合は
//名の場合は、i-nodeが戻され、エラーコードも返されます。
        oldinode=namei(oldname);
        if (!oldinode)
                return -ENOENT;
        if (S_ISDIR(oldinode->i_mode)) {
                iput(oldinode);
```

```
850                 return -EPERM;
851         }
```

// そして、新しいパス名のトップレベルのディレクトリのi-node 'dir'を見つけて、それを返します。
// 最後のファイル名とその長さ。ディレクトリのi-nodeが見つからない場合は、そのi-nodeの
// 元のパス名が戻され、エラーコードが返されます。もし、ファイル名が
// 新しいパス名に含まれる、元のパス名のi-nodeと新しいパス名の
// ディレクトリが戻され、エラーコードが返されます。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         }
```

// デバイス間のハードリンクは構築できません。そのため、トップディレクトリのデバイス番号が
新しいパス名の//が元のパス名のデバイス番号と異なる場合、i-node
新しいパス名のディレクトリの//と元のパス名のi-nodeが戻されて
// のエラーコードを返します。また、ユーザーが書き込み権限を持っていない場合は

```
// and the i-node of the original path name are put back, and the error code is returned.
        if (dir->i_dev != oldinode->i_dev) {
```

```
862
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         }
```

// ここで新しいパス名が既に存在するかどうかをチェックし、存在する場合は
// リンクを設定します。その後、既存のディレクトリエントリを含むバッファブロックが解放されて
// 新しいパス名のディレクトリのi-nodeと、元のパス名のi-nodeが戻される。
// とエラーコードが返されます。

```
872         bh = find_entry(&dir,basename,namelen,&de);
873         if (bh) {
874                 brelse(bh);
875                 iput(dir);
876                 iput(oldinode);
877                 return -EEXIST;
878         }
```

// すべての条件が満たされたので、新しいディレクトリにディレクトリ・エントリを追加します。もしそれが
// 失敗したら、ディレクトリのi-nodeと元のパス名のi-nodeを戻して
// エラーコードを返します。それ以外の場合は，ディレクトリエントリのi-node番号が初期設定される
// 元のパス名のi-node番号と同じで、バッファブロック修正フラグが
新たに追加されたディレクトリ・エントリを含む // がセットされ、バッファ・ブロックがリリースされて

```
        bh = add_entry(dir,basename,namelen,&de);
```

```
879
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

プログラム 12-7 linux/fs/file_table.c

```
/*
 *  linux/fs/file_table.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する（file, buffer_head,
//     m_inode, etc.).
#include <linux/fs.h>

struct file file_table[NR_FILE];  // File table array (64 items in total).

```

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.

図12-26 各種ファイルとシステムコールのインターフェース機能

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.

ブロックデバイスは、ディスクブロック（バッファブロックと同じサイズ）単位で毎回ディスクへの読み書きを行うため、関数block_write()では、まずファイルポインタ'pos'の位置をブロック番号とブロック内のオフセット値に対応させ、ブロックリード関数bread()またはブロックリードアヘッド関数breada()を使用して、ファイルポインタが位置するデータブロックをバッファキャッシュのバッファブロックに読み込みます。その後、書き込まれるデータの長さ「chars」に応じて、ユーザーデータバッファから現在のバッファブロックのオフセット位置にデータがコピーされる。まだ書き込むべきデータがある場合は、次のブロックがキャッシュのバッファブロックに読み込まれ、ユーザーデータがバッファブロックにコピーされる。2回目以降のデータ書き込み時には、オフセット量は0になります。 図12-27参照。

図12-27 ブロックデータ読み書き動作ポインタ位置図

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

を実行します。この関数を呼び出す前に、システムは、ユーザーバッファが使用する仮想リニアアドレスを、メインメモリ領域の対応するメモリページにマッピングします。

block_read()関数は、データがバッファからユーザーが指定した場所にコピーされることを除けば、block_write()と同じように動作します。

12.9.2 Code annotation

プログラム 12-8 linux/fs/block_dev.c

```
1 /*
2  *  linux/fs/block_dev.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
 // <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
 // <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
 //     of the initial task 0, and some embedded assembly function macro statements about the
 //     descriptor parameter settings and acquisition.
 // <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
 //     used functions of the kernel.
 // <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
 //     segment register operations.
 // <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
 //     descriptors/interrupt gates, etc. is defined.
7 #include <errno.h>
8
9 #include <linux/sched.h>
10 #include <linux/kernel.h>
11 #include <asm/segment.h>
12 #include <asm/system.h>
13
 // デバイスデータブロックの総数を示すポインターの配列。そのポインタのそれぞれは
 // 指定されたメジャーデバイス番号の総ブロックの配列であるhd_sizes[]に の各項目は
 // ブロックアレイの合計数は、データブロックの合計数に対応します。
 // サブデバイス番号で決まるサブデバイス。
14 extern int *blk_size[];                    // blk_drv/ll_rw_blk.c, line 49.
15
 //// ブロックライト機能 - 指定されたサイズのデータを、デバイス上の指定されたオフセットに書き込みます。
 // パラメータ： dev - デバイス番号、 pos - デバイスファイル内のオフセットポインタ。
 // buf - ユーザースペースのバッファ; count - 転送するバイト数。
 // 書き込んだバイト数を返します。データが書き込まれていない場合やエラーが発生した場合は、エラーコードを返します。
```

// カーネルにとっての書き込み操作とは、キャッシュにデータを書き込むことです。データが最終的にデバイスに書き込まれた//をキャッシュマネージャーが判断して処理します。また、今回は
// ブロックデバイスは、ブロック単位で読み書きを行いますが、書き込み位置が
// ブロックの端に位置する場合は、データがあるブロック全体を先に読み出す必要があります。
// とし、書き込み位置からデータを埋めていきます。その後、データの完全なブロックを書き込む
// をディスクに送信します（つまり、バッファキャッシュに処理します）。
16 int block_write(int dev, long * pos, char * buf, int count) 17 {。
// まず、ファイルの位置'pos'をブロック番号'block'に変換し、その中に
// のディスクブロックの読み書きが開始されるオフセット位置「offset」と

```
18      // ブロックに書き込まれる最初のバイトを取得します。
19              int block = *pos >> BLOCK_SIZE_BITS;     // the block number where pos is located.
20              int offset = *pos & (BLOCK_SIZE-1);      // offset value in the data block.
21              int chars;
22              int written = 0;
23              int size;
24              struct buffer_head * bh;
25              register char * p;                       // local register variable.
25
        // ブロックデバイスファイルの書き込み時には、書き込みに必要なデータブロック数の合計が
        // は、もちろん、指定されたデバイスで許容されるデータブロックの最大数を超えてはなりません。
        // そのため、まず指定されたデバイスの総ブロック数を取り出して比較します
        関数のパラメータで指定された書き込みデータの長さを // 制限します。長さが指定されない場合は
26      // システム内のデバイスでは、デフォルトの長さである0x7fffff（2GBブロック）が使用されます。
27              if (blk_size[MAJOR(dev)])
28                      size = blk_size[MAJOR(dev)][MINOR(dev)];
29              else
30                      size = 0x7fffffff;
        // そして、書き込むバイト数「count」に対して、以下の処理をループさせます。
        データが完全に書き込まれるまで、 // 。ループの実行中に、データのブロック番号が
        // 現在書き込まれているデータが、指定されたブロックの合計数以上である場合
        // デバイスに書き込まれたバイト数を返して終了します。書き込むことができるバイト数は
        // 現在処理されているデータブロックに書き込まれるバイト数が計算されます。もし、バイト数が
        // が1ブロックに満たない場合は、カウントバイトだけを書けばよいのです。
        // 1ブロック分のデータを書き込みたい場合は、直接バッファブロックを申請し、ユーザーデータを置く
        // を読み込んでいます。そうでない場合は、パーシャルに書き込まれるデータブロックを読み込む必要
        があります。
        // のデータを読み、次の2ブロックのデータを先読みします。その後、ブロック番号をインクリメントし、
        準備
        // 次の操作のために バッファブロック操作が失敗した場合、書き込まれたバイト数は
          /     while (count>0) {
          /
          が返され、バイトが書き込まれていない場合はエ
```

ラーコード（負の数）が返されます。

```
30
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;
```

// 次に，ポインタ p は，まず，データがあるバッファブロック内の位置に向けられます。
// が書き込まれることになります。最後のループで書き込まれたデータが1ブロックに満たない場合は
// ブロックの先頭から必要なバイトを埋めて（修正して）いくため、オフセットは
// はデフォルトでゼロに設定されます。その後、ファイル内のオフセットポインタ 'pos' が転送されます。
// を今回の書き込みバイト数で割って、書き込みバイト数を
// 統計値'written'に蓄積されます。その後、バイト数の'count'を引く
今回書き込むべきバイト数（chars）から、書き込む必要のある//を選択します。その後、私たちは
// ユーザーバッファからの'chars'バイトを、バッファブロック内の
// 書き込みを開始します。コピーが完了すると、バッファブロックモディファイドフラグがセットされて
// バッファブロックが解放されます（つまり、バッファブロックの参照カウントが1デクリメントされます）。

```
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
```

//// ブロックリード機能 - 指定されたデバイスからユーザーバッファにデータを読み込みます。

// パラメータ： dev - デバイス番号、 pos - デバイスファイル内のオフセット、 buf - ユーザ空間のバッファ。

// count - 転送するバイト数。

// 戻り値：読み込んだバイト数。読み込んだバイト数がない場合やエラーの場合は、エラーコードが返されます。

```
58  56 int block_read(int dev, unsigned long * pos, char * buf, int count) 57 {。
59          int block = *pos >> BLOCK_SIZE_BITS;
60          int offset = *pos & (BLOCK_SIZE-1);
61          int chars;
62          int size;
63          int read = 0;
64          struct buffer_head * bh;
65          register char * p;                    // local register variable.
65
```

// ブロックデバイスファイルを読み出す際に、読み出すのに必要なブロック数の合計が、以下を超えることはできません。

// 指定されたデバイスで許容される最大のブロック数です。そのため、合計数

デバイスのブロックの//を最初に取り出して比較し、与えられたリードデータの長さを制限します。

// を関数パラメータで指定します。システム内のデバイスに長さが指定されていない場合は、デフォルトの

```
66  // 0x7fffff（2GBブロック）の長さが使用されます。
67          if (blk_size[MAJOR(dev)])
68                  size = blk_size[MAJOR(dev)][MINOR(dev)];
69          else
70                  size = 0x7fffffff;
```

// そして、「count」で読み取るべきバイト数に対して、以下の処理をループさせます。

// データの読み込みが完了するまで ループの実行中に、データのブロック番号が

// 現在読み込まれているデータが、指定されたブロックの合計数以上である場合

// デバイスで、読み込んだバイト数を返して終了します。その後、バイト数を計算する

現在処理されているデータブロックを読み出すための //。読み込む必要のあるバイト数が

// が1ブロックに満たない場合は、「count」バイトだけを読み取ります。その後、ブロック読み込み関数を呼び出します。

// breada（）で必要なデータブロックを読み込み、次の2ブロックのデータを事前に読み込みます。もし

// 読み取り操作にエラーがあった場合は、読み取ったバイト数を返します。バイトが読み込まれていな

い場合は
// エラーコードが返されます。その後、ブロック番号を1つ増やし、次の
//の操作を行います。

```
        while (count>0) {
                if (block >= size)
                        return read?read:-EIO;
                chars = BLOCK_SIZE-offset;
                if (chars > count)
                        chars = count;
                if (!(bh = breada(dev,block,block+1,block+2,-1)))
                        return read?read:-EIO;
```

```
80                  block++;
// 次にインタはずー読むスロのフロ内置す後ーみ操デがッ満いは要イブク頭読むがたあじフト設て必あすのフルオッイ'pos'読むト'chars'分進読むト統ポpまデを込ィブクッブク位指最ル読みのタロにな合必バをッ先ら込要る、かオッにしくがまそ、イのセポタを込イ数け、込イを値ンタタみデクッバァッのを。のプ込作ー1ブクた場、なトロのかみ必あめらめセを0定お要り。後ァ内フトンみバ数だめみバ数計
after this copy is completed.
                p = offset + bh->b_data;
```

'read' にし
累積す。し読
まそ　　るのカト
てみ、取要るン
必あウ値
'count'
か読らみ、取
るカトベウ　きン数
'chars' 引
を　　そ
く。てッブ、フロの指始ら
しバックが開か
ファ　　ト
'p' ユ　　ッに
す点 'chars' バイ　ーユバ
をザ　ファピ　ーユバファイを
フコ　ーザ　ッフポタにさ
コし　ッの　ンタ方動る。フロ
ッの　前移せバッブ
ファ　　ックの
is released

```
79
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

プログラム 12-9 linux/fs/file_dev.c

```
1 /*
2  *  linux/fs/file_dev.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
 // <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
 // <fcntl.h> ファイル制御のヘッダファイルです。演算制御定数記号の定義は
 //     used for the file and its descriptors.
```

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//　　of the initial task 0, and some embedded assembly function macro statements about the
//　　descriptor parameter settings and acquisition.
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
//　　used functions of the kernel.
 // <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
//　　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
```

//// ファイル読み込み機能 - i-nodeとファイル構造に基づいて、ファイル内のデータを読み込みます。
// i-nodeからはデバイス番号を知ることができ、ファイル構造からは現在の
// ファイル内の読み書き可能なポインタの位置を指定します。Buf」は、ユーザーのバッファの位置を指定します。
//スペース、「count」は読み取る必要のあるバイト数です。戻り値は
// 実際に読み込まれたバイト数、またはエラーコード（0未満）を表します。

```
17 int file_read(struct m_inode * inode, struct file * filp, char * buf, int count) 18 {。
19
20        int left,chars,nr;
21        struct buffer_head * bh;
21
```

// まず、パラメータの有効性を判断します。読み込むバイト数が以下の場合は0を返します。
// 0以下であること。読み込むバイト数が0にならない場合は、次のようになります。
データが完全に読み込まれるか、問題が発生するまで、//ループ操作が行われます。
// 読み込みループ操作中に、bmap()を使って対応する論理（ディスク）ブロックを取得する
i-nodeとファイルテーブル構造に基づいて、デバイス上のデータブロックの//番号'nr'
// の情報を提供します。nrが0でなければ、ディスクブロックをデバイスから読み込む。の場合は、ループを終了します。
// 読み取り操作が失敗します。nrが0の場合は、指定されたデータブロックが存在しないことを意味します。
// そのため、バッファブロックポインタはNULLに設定されています。(filp->f_pos)/BLOCK_SIZE を用いて
// ファイルのカレントポインタが置かれているデータブロック番号。

```
22
23        if ((left=count)<=0)
24                return 0;
25        while (left) {
26                if (nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE)) {      // inode.c, line 140.
27                        if (!(bh=bread(inode->i_dev,nr)))
28                                break;
```

```
29                  } else
30                          bh = NULL;
   // 次に、データブロック内のファイル読み書きポインタのオフセット値nrを計算してから
   // データブロックで読みたいバイト数は（BLOCK_SIZE - nr）です。次に比較します。
   // 今読み取る必要のあるバイト数を'左'にして、小さい値は数字の
   この操作で読み込まれるバイト 'chars' の // 数。(BLOCK_SIZE - nr) > left」の場合は、ブロック
   // が読み込まれる最後のデータで、そうでなければ次のブロックのデータを読み込まなければなりませ
   ん。
   // 次に、ファイルの読み取りと書き込みのポインタを調整します。ポインタはバイト数を進める 'chars'
   // 今回読み込まれるバイト数です。残りのバイト数「left」も差し引く必要があります。
   // 「chars」バイトで。
32                  nr = filp->f_pos % BLOCK_SIZE;
```

```
33                  chars = MIN( BLOCK_SIZE-nr , left );
34                  filp->f_pos += chars;
35                  left -= chars;
```

// 上記のデータがデバイスから読み込まれた場合、バッファブロック内の以下の場所をpに指定します。
// データが読み込まれると、'chars'バイトをユーザーバッファ'buf'にコピーし、そうでなければ
// ユーザーバッファを'chars'バイトのゼロで埋める。

```
37                  if (bh) {
38                          char * p = nr + bh->b_data;
39                          while (chars-->0)
40                                  put_fs_byte(*(p++),buf++);
41                          brelse(bh);
42                  } else {
43                          while (chars-->0)
44                                  put_fs_byte(0,buf++);
42                  }
43          }
```

// 最後に、i-nodeのアクセス時間が現在の時間に修正され、いくつかの
// 読んだバイト数が返されます。読んだバイト数が0の場合は、エラーコードが返されます。
// CURRENT_TIME は、include/linux/sched.h の 142 行目で定義されたマクロで、以下の計算に使用されます。
// UNIXの時間。つまり、1970年1月1日の0:00から現在の時刻までです。単位は秒です。

```
44          inode->i_atime = CURRENT_TIME;
45          return (count-left)?(count-left):-ERROR;
46 }
47
```

//// ファイル書き込み機能 - i-nodeとファイル構造に基づいて、ユーザーデータをファイルに書き込みます。
// i-nodeからはデバイス番号がわかり、ファイル構造からは
// ファイル内の現在の読み書き可能なポインタの位置を指定します。'buf'は、バッファの位置を指定します。
ユーザー空間での//、'count'は書き込まれるバイト数です。戻り値は
// 実際に書き込まれたバイト数、またはエラーコード（0より小さい）です。

```
48 int file_write(struct m_inode * inode, struct file * filp, char * buf, int count) 49 {。
51          off_t pos;
52          int block,c;
53          struct buffer_head * bh;
54          char * p;
55          int i=0;
55
56 /*
58  * ok, append may not work when many processes are writing at the same time
59  * but so what. That way leads to madness anyway.
59 */
```

// まず、データをファイルのどこに書き込むかを決めます。にデータを追加したい場合は
// ファイルの場合、ファイル書き込みポインタをファイルの最後に移動させ、そうでない場合は
// ファイルの現在の読み取りおよび書き込みポインタ。

```
61          if (filp->f_flags & O_APPEND)
62                  pos = inode->i_size;
```

```
63          else
64                  pos = filp->f_pos;
```

// そして、'i'（最初はゼロ）を書き込んだバイト数が少なくなると
// 指定された書き込みバイト数'count'よりも多い場合、以下の処理を行います。
// を周期的に行います。ループ操作では、まず、論理的な（ディスク）ブロック番号「block」を取得します。
// デバイス上のファイルデータブロック(pos/BLOCK_SIZE)に対応しています。もし、ディスクブロックが
// が存在しない場合は、作成します。作成されたディスクのブロック番号が0の場合、作成は失敗します。

// となり、ループが終了します。それ以外の場合は、デバイス上のディスクブロックを以下のように読み込みます。

```
65      // ディスクのブロック番号を入力し、エラーが発生した場合はループを終了します。
66              while (i<count) {
67                      if (!(block = create_block(inode,pos/BLOCK_SIZE)))
68                              break;
69                      if (!(bh=bread(inode->i_dev,block)))
70                              break;
```

// この時、バッファブロックポインタ'bh'は、先ほど読み込んだファイルデータブロックを指しています。

// これで、データ内のファイルの現在の読み書きポインタのオフセット'c'を見つけることができます。

// ブロックの中で、データが開始されたバッファブロックの位置へのポインタ'p'を指します。

書き込まれる // バッファブロックモディファイドフラグを設定します。ブロック内のカレントポインタに対して

// 書き込み位置の先頭から、合計でc = (BLOCK_SIZE - c) バイトを書き込むことができる

// をブロックの最後まで書き込みます。cが書き込まれる残りのバイト数より大きい場合は

// (count - i) とすると、今度は単純に c = (count - i) バイトと書きます。

```
72                      c = pos % BLOCK_SIZE;
73                      p = c + bh->b_data;
74                      bh->b_dirt = 1;
75                      c = BLOCK_SIZE-c;
76                      if (c > count-i) c = count-i;
```

// データを書き込む前に、次のループで書き込むファイル内の位置をあらかじめ設定する

// の操作を行います。そこで、書き込まなければならないバイト数だけ「pos」ポインタを前進させます。

// 「pos」ポジションの値が、現時点でのファイルの長さを超える場合は

// i-nodeのファイル長フィールドが修正され、i-nodeの修正フラグが設定されます。その後

// 今回書き込まれるバイト数は、書き込まれたバイト数に加算される 'i' for loop judgment.

// 続いて、'c'バイトがユーザーバッファ'buf'から、'buf'が指す開始位置にコピーされる。

// 'p'をキャッシュバッファブロックにコピーします。バッファブロックはコピー後に解放されます。

```
78                      pos += c;
79                      if (pos > inode->i_size) {
80                              inode->i_size = pos;
81                              inode->i_dirt = 1;
78                      }
79                      i += c;
80                      while (c-->0)
81                              *(p++) = get_fs_byte(buf++);
82                      brelse(bh);
83              }
```

// データがすべてファイルに書き込まれたときや、途中で問題が発生したときにループが終了します。

// 書き込み操作を行います。この時点で、ファイル修正時刻を現在の時刻に変更します。

// とし、ファイルの読み書きポインタを調整します。つまり、操作によってデータが追加されない場合は

// をファイルの末尾に移動させると、ファイルの読み書きポインタが現在の読み書きの

// 位置'pos'、ファイルi-nodeの修正時間が現在の時間に変更されます。

// 最後に、書き込んだバイト数を返します。書き込まれたバイト数が0の場合は

```
        inode->i_mtime = CURRENT_TIME;
```

```
84
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.

パイプラインの作成と初期化の際、プログラムは特にパイプラインのi-nodeを要求し、パイプラインにページバッファ(4KB)を割り当てます。パイプi-nodeのi_sizeフィールドはパイプバッファを指すように設定され、i_zone[0]フィールドにはパイプデータのヘッダポインタが、i_zone[1]フィールドにはパイプデータのテールポインタが格納される。パイプラインの読み出し動作では、パイプラインの終端からデータを読み出し、パイプテールポインタを読み出したバイト数だけ進め、パイプラインへの書き込み動作では、図12-28に示すように、データをパイプヘッダに書き込み、ヘッドポインタを何箇所か進める（ヌルバイトを指す）ことになる。

図12-28 パイプバッファ動作図

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.

write_pipe()関数は、リードパイプライン関数のように動作します。

システムコールのsys_pipe()は、無名のパイプを作るために使われます。これはまず、システムのファイルテーブルの2つのエントリを取得し、 現在のプロセスのファイルディスクリプタテーブルから2つの未使用のディスクリプタエントリを探し、 対応するファイル構造ポインタを保存します。その後、システム内のアイドルi-nodeを申請し、パイプラインのバッファブロックを取得する。その後、対応するファイル構造を初期化し、一方のファイル構造を読み取り専用モードに、もう一方のフ

ァイル構造を書き込み専用モードに設定する。最後に、2つのファイルディスクリプターがユーザーに渡されます。

また、上記の関数で使用されるPIPE_HEAD()、PIPE_TAIL()などのパイプライン操作に関連するいくつかのマクロは、include/linux/fs.hファイルの57～64行目で定義されています。

12.11.2 Code annotation

プログラム 12-10 linux/fs/pipe.c

```
1 /*
2  *  linux/fs/pipe.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナルの定義
//      manipulation function prototypes.
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <termios.h> 端末入出力機能のヘッダファイル。主に端末の入出力機能を定義しています。
//      interface that controls the asynchronous communication port.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
// <linux/mm.h> メモリ管理用のヘッダーファイルです。ページサイズの定義と、いくつかのページ
//      release function prototypes.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
//      segment register operations.
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
//      used functions of the kernel.
7 #include <signal.h>
8 #include <errno.h>
9 #include <termios.h>
10
11 #include <linux/sched.h>
12 #include <linux/mm.h>            /* for get_free_page */
13 #include <asm/segment.h>
14 #include <linux/kernel.h>.
15
//// パイプリード機能。
// パラメータの'inode'はパイプのi-node、'buf'はユーザーデータのバッファポインタであり
// 'count' は読んだバイト数です。
16 int read_pipe(struct m_inode * inode, char * buf, int count) 17 {。
18         int chars, size, read = 0;
19
// 読み込む必要のあるバイト数が0より大きい場合は、以下のようにループします。
// の操作を行います。ループ操作中、現在のパイプにデータがない場合（size=0）。
// そのノードを待っているプロセスが起こされます。これは通常、書き込みパイプラインプロセスです。
// パイプライタがなくなった場合、つまり、i-nodeの参照カウント値が小さくなった場合
// 2より大きい場合は、読み取りバイト数が返されます。現在、ノンブロッキングのシグナルがある場合は
// 受信した場合は、直ちに読み込んだバイト数を返して終了し、データがない場合は
```

// まだ受信していない場合は、システムコール番号を再起動して終了するように戻ります。それ以外の場合は
// 情報の到着を待つために、パイプ上でプロセスのスリープを行います。マクロPIPE_SIZEは
// include/linux/fs.hで定義されています。「restart system-call」の意味については
// kernel/signal.cプログラムです。

```
        while (count>0) {

                while (!(size=PIPE_SIZE(*inode))) {      // get data length in the pipe.
                        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
                  }
// この時点では、パイプライン（バッファ）にデータがあるので、「chars」の数を取得します。
// テールポインタからバッファの最後までのパイプを指定します。よりも大きい場合は
読まなければならないバイト数の // 'count' と同じにします。chars'が次の値より大きい場合
// 現在のパイプラインのデータの'サイズ'と同じになります。すると、その数は
// 読むべきバイト数 'count'は、この時までに読めるバイト数から減算され
// 既に読んだバイト数に追加される 'read'.
30                chars = PAGE_SIZE-PIPE_TAIL(*inode);
31                if (chars > count)
32                        chars = count;
33                if (chars > size)
34                        chars = size;
35                count -= chars;
36                read += chars;
// そして、'size'にパイプのテールポインタを指定させ、現在のパイプのテールポインタを調整します
（forward
// 'chars' バイト）。テールポインタがパイプの終端を超えると折り返します。のデータは
// のパイプがユーザーバッファにコピーされます。パイプのi-nodeでは，パイプバッファのブロックポ
インタ
/                size = PIPE_TAIL(*inode);
/
は
i
_
si
z
e
フ
ィ
ー
ル
ド
に
あ
り
ま
す
。
36
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        }
// 読み込みパイプ操作が終了すると、パイプラインを待っているプロセスが起こされて
42      // 読んだバイト数が返されます。
43        wake_up(& PIPE_WRITE_WAIT(*inode));
44        return read;
44 }
```

```
45
    //// パイプの書き込み機能。
    // パラメータの'inode'はパイプのi-node、'buf'はデータバッファのポインタ、'count'は
    // はパイプに書き込まれるバイト数です。
46 int write_pipe(struct m_inode * inode, char * buf, int count) 47 {。
48          int chars, size, written = 0;
49
    // 書き込まれるバイト数 'count' がまだ 0 より大きい場合は、ループします。
    // 以下の操作を行います。ループ操作の間、現在のパイプが満杯の場合（空きスペース
    // size = 0）の場合、パイプラインを待っているプロセスが起こされ、通常はリードパイププロセスが
    // が起こされます。パイプリーダがない場合、つまり、i-nodeの参照カウント値が小さい場合は
    // 2よりも大きい場合、SIGPIPEシグナルが現在のプロセスに送信され、書き込まれたバイト数が
    // を返します。0バイトが書き込まれた場合は、-1が返されます。それ以外の場合は、現在のプロセスを
    スリープさせます
    // パイプのプロセスがデータを読み取るのを待って、パイプを解放します。
50  // 空間になります。PIPE_SIZE()、PIPE_HEAD()などのマクロは、include/linux/fs.hというファイルで
    定義されています。
51          while (count>0) {
52                  while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
53                          wake_up(& PIPE_READ_WAIT(*inode));
```

```
54                      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              }
```

// プログラムはここで実行され、パイプラインに書き込み可能なスペース'size'があることを示す
// バッファです。そこで、パイプヘッドのポインタからバイト数「chars」を取り、最後に
//バッファになります。パイプの書き込み操作は、パイプの先頭ポインタから始まります。chars'が大きい場合
// 書き込まなければならないバイト数'count'よりも、'count'と同じにする。もし'chars'
// が現在のパイプラインの空き容量'size'より大きい場合、'size'と同じになります。その後
// 書き込まれるバイト数を「count」から「this」に書き込まれたバイト数を引いた値にします。
// 時刻の'chars'に、今回書き込んだバイト数を'written'に追加します。

```
60              chars = PAGE_SIZE-PIPE_HEAD(*inode);
61              if (chars > count)
62                      chars = count;
63              if (chars > size)
64                      chars = size;
65              count -= chars;
66              written += chars;
```

// そして、'size'にパイプのヘッドポインタを指定し、現在のパイプデータのヘッドポインタを調整します。
//（前方のcharsバイト）。ヘッドポインタがパイプの終端を超えると折り返します。その後
// ユーザーバッファから'chars'バイトをパイプの先頭ポインタにコピーします。のために
// パイプのi-nodeの場合、パイプバッファブロックポインタはi_sizeフィールドになります。

```
68              size = PIPE_HEAD(*inode);
69              PIPE_HEAD(*inode) += chars;
70              PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
71              while (chars-->0)
72          ((char *)inode->i_size)[size++]=get_fs_byte(buf++); 71      }
```

// 書き込みパイプ操作が終了すると、パイプを待っているプロセスが起こされて
72 // 書き込まれたバイト数が返されます。

```
73          wake_up(&PIPE_READ_WAIT(*inode));
74          return written;
74 }
75
```

//// パイプのsystem-callを作成します。
// ファイルハンドル（記述子）のペアをfildes配列に作成します。このファイルハンドルのペアは
// パイプのi-nodeを指します。パラメータ：Files - ファイルハンドルの配列。Fildes[0]が使用されます。
// でパイプのデータを読み込み、fildes[1]でパイプにデータを書き込みます。成功した場合は0、エラーの場合は-1を返します。

78 76 int sys_pipe(unsigned long * fildes) 77 {.

```
79          struct m_inode * inode;
80          struct file * f[2];                     // array of file structures.
81          int fd[2];                              // array of file handles.
82          int i,j;
82
```

// まず、システムファイルテーブルから2つのフリーアイテム（参照カウントフィールドを持つアイテム）を取り出します。
0の//）、参照カウントを1に設定します。 空きエントリが1つしかない場合、そのアイテムはリリースされます。

```
/       j=0;
/
/（参照カウントをリセット）。2つのフリーアイテムが見つからない場合は-1を返します。
83
84          for(i=0;j<2 && i<NR_FILE;i++)
85                  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;
    // 取得した2つのファイルテーブル構造項目のそれぞれに、ファイルハンドル番号が割り当てられる。
    // と、プロセスファイル構造配列の2つのアイテムがそれぞれ2つのファイルを指しています。
    // 構造体であり，ファイルハンドルはその配列のインデックス番号である。同様に，もし
    // フリーのファイルハンドルが１つしかない場合、そのハンドルは解放されます。２つのフリーのハンドルが見つからない場合は
    // 上記で取得した2つのファイル構造項目が解放され、-1が返されます。
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         }
    // 次に、関数get_pipe_inode()を使用して、パイプが使用するi-nodeを要求し、割り当てます。
    // パイプのバッファとしてメモリの1ページを使用します。失敗した場合は，2つのファイルハンドルとファイル
103     // 構造体のアイテムはそれに応じて解放され、-1が返されます。
104         if (!(inode=get_pipe_inode())) {                // fs/inode.c, line 231.
105                 current->filp[fd[0]] =
106                 current->filp[fd[1]] = NULL; 106
                    f[0]->f_count = f[1]->f_count = 0;
107                 return -1;
108         }
    // パイプi-nodeのアプリケーションが成功すると、2つのファイル構造が初期化されるので
    // 両方が同じパイプのi-nodeを指していることを確認し、readとwriteのポインタを0に設定します。
    // 1つ目のファイル構造のファイルモードが読み取りに設定され、2つ目のファイル構造のファイルモードが
    // ファイル構造が書き込みに設定されます。最後に，ファイルハンドル配列がユーザ空間にコピーされる
    // 配列を作成します。成功した場合は0を返して終了します。
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のコントロール機能です。

// パラメータ: pino - パイプiノードポインタ; cmd - 制御コマンド; arg - 引数。

// この関数は、成功すれば0を返し、そうでなければエラーコードを返します。

118 int pipe_ioctl(struct m_inode *pino, int cmd, int arg) 119 {...

// パイプラインの現在の読み取り可能なデータ長を取得するコマンドの場合は、パイプデータを

// 長さの値をユーザー引数で指定された場所に格納し、0を返します。それ以外の場合は

// 無効なコマンドのエラーコードが返されます。

```
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()は、メインデバイス番号が4のシリアルターミナルデバイスのリード/ライト関数で、ttyドライバを呼び出してシリアルターミナルのリード/ライト操作を実装しています。

rw_tty()はコンソール端末の読み書き機能で、メジャーデバイス番号は5です。実装原理はrw_ttyx()と同じですが、プロセスがコンソール操作を行えるかどうかに限定されます。

rw_memory()は、メモリデバイスファイルの読み書き関数で、メジャーデバイス番号は1で、メモリイメージのバイト操作を実装しているが、Linux 0.12カーネルでは、0.96バージョンでマイナーデバイス番号0、1、2の読み書き操作を実装し始めるまでは、実装されていない。

rw_char()は、キャラクタ・デバイスのリード・ライト操作のためのインターフェース関数である。他のキャラクタ・デバイスは、この関数を介して、キャラクタ・デバイス読み書き機能ポインタ・テーブル上の対応するキャラクタ・デバイスの操作を行う。ファイルシステムの操作関数open()、read()などは、すべてのキャラクタデバイスのファイルに対して操作を行う。

12.12.2 Code annotation

プログラム 12-11 linux/fs/char_dev.c

```
1 /*
2  *  linux/fs/char_dev.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
```

// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。

// <sys/types.h> 型のヘッダファイル。基本的なシステムデータ型が定義されています。

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ

```
//      of the initial task 0, and some embedded assembly function macro statements about the
```

```
//      descriptor parameter settings and acquisition.
```

// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
//　　used functions of the kernel.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
//　　segment register operations.
// <asm/io.h> io のヘッダーファイルです。の形で、ioポートを操作する関数を定義します。
//　　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
```

// キャラクターデバイスのリード/ライト機能のポインタタイプを定義します。

```
19 typedef (*crw_ptr)(int rw,unsigned minor,char * buf,int count,off_t * pos); 20
```

//// シリアル端子のリード/ライト操作機能。
// パラメータ： rw - リード/ライトコマンド、 minor - ターミナルサブデバイス番号、 buf - バッファ。
// cout - 読み書きされたバイト数; pos - 読み書き操作の現在のポインタ、このポインタ
//は端末操作には使えません。
// 戻り値：実際に読み書きされたバイト数です。失敗した場合は、エラーコードを返します。

```
21 static int rw_ttyx(int rw,unsigned minor,char * buf,int count,off_t * pos) 22 {。
23
24          return ((rw==READ)?tty_read(minor,buf,count):
25                  tty_write(minor,buf,count));
25 }
26
```

//// 端末のリード/ライト操作機能。
// rw_ttyx()と同じですが、プロセスが制御端末を持っているかどうかの検出が追加されます。

```
27 static int rw_tty(int rw,unsigned minor,char * buf,int count, off_t * pos) 28 {。
```

// プロセスが対応する制御端子を持たない場合は、エラーコードが返されます。
// そうでない場合は、端末が読み書き可能な関数 rw_ttyx() を呼び出して、実際の
// 読み取りまたは書き込みバイト数。

```
29
30          if (current->tty<0)
31                  return -EPERM;
32          return rw_ttyx(rw,current->tty,buf,count,pos);
32 }
33
```

//// メモリデータの読み書き。実装されていません。

```
34 static int rw_ram(int rw,char * buf, int count, off_t *pos) 35 {。
36          return -EIO;
37 }
38
```

//// 物理的なメモリデータのリード／ライト機能。実装されていません。

39 static int rw_mem(int rw,char * buf, int count, off_t * pos) 40 {。

```
41          return -EIO;
42 }
43
    //// カーネルの仮想メモリデータの読み書き機能です。実装されていません。
44 static int rw_kmem(int rw,char * buf, int count, off_t * pos) 45 {。
46          return -EIO;
47 }
48
    // ポートのリード/ライト操作機能。
    // パラメータ: rw - 読み取りまたは書き込みコマンド; buf - バッファ; cout - 読み取り/書き込みのバイト数。
    // pos - ポートのアドレス。戻り値：実際に読み書きしたバイト数。
49 static int rw_port(int rw,char * buf, int count, off_t * pos) 50 {。
51          int i=*pos;
52
    // 読み書きに必要なバイト数で、ポートアドレスが
    // 64kでは、1バイトのリードまたはライトの操作がサイクリックに実行されます。コマンドがリードの場合
    // 1バイトのコンテンツがポートiから読み込まれ、ユーザーバッファに置かれます。もし、コマンドが
    // 書き込まれると、ユーザーデータバッファから1バイトが取り出され、ポートiに出力されます。
53          while (count-->0 && i<65536) {
54                  if (rw==READ)
55                          put_fs_byte(inb(i),buf++);
56                  else
57                          outb(get_fs_byte(buf++),i);
58          i++;    // increase port address. [??] 59  }
    // そして、読み書きされたバイト数をカウントし、対応する読み書きポインタを調整します。
60  // そして、読み書きされたバイト数を返します。
61          i -= *pos;
62          *pos += i;
63          return i;
63 }
64
    //// メモリの読み書き操作機能です。メモリのマスターナンバーは1で、処理のみ
    0～5のサブデバイスの//がここに実装されています。
65 static int rw_memory(int rw, unsigned minor, char * buf, int count, off_t * pos) 66 {。
67  // メモリサブデバイスの番号に応じて、異なるメモリリード/ライト関数が呼び出されます。
68          switch(minor) {
69                  case 0:                 // device file name is /dev/ram0 or /dev/ramdisk.
70                          return rw_ram(rw,buf,count,pos);
71                  case 1:                 // device file name is /dev/ram1 or /dev/mem or ram.
72                          return rw_mem(rw,buf,count,pos);
73                  case 2:                 // device file name is /dev/ram2 or /dev/kmem.
74                          return rw_kmem(rw,buf,count,pos);
75                  case 3:                 // device file name is /dev/null.
76                          return (rw==READ)?0:count;      /* rw_null */
77                  case 4:                 // device file name is /dev/port
78                          return rw_port(rw,buf,count,pos);
79                  default:
```

```
80                          return -EIO;
80          }
```

```
81 }
82
// システム内のデバイスの数を定義します。
83 #define NRDEVS ((sizeof (crw_table))/(sizeof (crw_ptr)) 84
85 // キャラクターデバイスのリード/ライト機能のポインタテーブル。
86 static crw_ptr crw_table[]={
87         NULL,           /* nodev */
88         rw_memory,      /* /dev/mem etc */
89         NULL,           /* /dev/fd*/
90         NULL,           /* /dev/hd*/
91         rw_ttyx,        /* /dev/ttyx */
92         rw_tty,         /* /dev/tty */
93         NULL,           /* /dev/lp */
94         NULL};          /* unnamed pipes */
94
//// キャラクターデバイスのリード/ライト機能。
// パラメータ： rw - 読み取りまたは書き込みコマンド、 dev - デバイス番号、 buf - バッファ、 count -
数
// pos - 読み取りまたは書き込みポインタ.
// 戻り値：実際に読み書きしたバイト数。
95 int rw_char(int rw,int dev, char * buf, int count, off_t * pos) 96 {.
97         crw_ptr call_addr;
98
// デバイス番号がシステムデバイスの数を超える場合は、エラーコードが返されます。もし
// デバイスに対応するリード/ライト機能がない場合は、エラーコードが返されます。
// それ以外の場合は、対応するデバイスのリード/ライト操作機能が呼び出され
//      if (MAJOR(dev)>=NRDEVS)
実際に読み書きされたバイト数が返されます。
99
100                 return -ENODEV;
101         if (!(call_addr=crw_table[MAJOR(dev)]))
102                 return -ENODEV;
103         return call_addr(rw,MINOR(dev),buf,count,pos);
104 }
105
```

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.

read()システムコールは、まず与えられたパラメータの有効性を判断し、次にファイルのi-node情報に基づいてファイルのタイプをチェックします。パイプであれば、プログラムpipe.cのread関数を、キャラクタ・デバイス・ファイルであれば、char_dev.cのrw_char()キャラクタ・リード関数を、ブロック・デバイス・ファイルであれば、block_dev.cのブロック・デバイス・リード・オペレーションを実行し、読み取ったバイト数を返します。

write()システムコールの実装は read()と似ていますが、file_dev.c の file_read() というファイル読み取り関数を呼び出します。

2.13.2 lseek()システムコールは、ファイルハンドルに対応するファイル構造の現在の読み取り/書き込みポインタを修正します。読み書きポインタの移動ができないファイルやパイプファイルの場合、エラーコードが与えられ、直ちに返されます。

12.13.3 Code annotation

プログラム 12-12 linux/fs/read_write.c

```
1 /*
2  * linux/fs/read_write.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
```

// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。

```
//      and constants.
```

// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。

// <sys/types.h> 型のヘッダファイル。基本的なシステムデータ型が定義されています。

// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。

```
//      used functions of the kernel.
```

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ

```
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
```

// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。

```
//      segment register operations.
7 #include <sys/stat.h>
8 #include <errno.h>
9 #include <sys/types.h>
10
11 #include <linux/kernel.h>
12 #include <linux/sched.h>
13 #include <asm/segment.h>
14
15 // fs/char_dev.c, line 95.
16 extern int rw_char(int rw,int dev, char * buf, int count, off_t * pos);
17 // パイプ操作関数の読み込み fs/pipe.c, line 16.
18 extern int read_pipe(struct m_inode * inode, char * buf, int count);
19 // パイプ操作関数の記述 fs/pipe.c, line 46.
20 extern int write_pipe(struct m_inode * inode, char * buf, int count);
```

// fs/block_dev.cの56行目です。18 extern int block_read(int dev, off_t * pos, char * buf, int count);

```
19 // ブロックデバイスの書き込み機能 fs/block_dev.c, line 16.
20 extern int block_write(int dev, off_t * pos, char * buf, int count);
21 // ファイル操作関数の読み込み fs/file_dev.c, line 17.
```

```
22 extern int file_read(struct m_inode * inode, struct file * filp,
23                 char * buf, int count);
24     // ファイル操作関数を書く fs/file_dev.c, line 48.
25 extern int file_write(struct m_inode * inode, struct file * filp,
26                 char * buf, int count);
24
//// ファイルの読み取り/書き込みポインタを再配置する system-call.
```

// パラメータ fd はファイルハンドル、offset は新しいファイルの読み取りまたは書き込みポインタのオフセットです。
// で、origin はオフセットの開始位置です。3つのオプションがあります。SEEK_SET（0,
// ファイルの先頭から）、SEEK_CUR（1、現在の読み書き位置から）、SEEK_END
// （2、ファイルの最後から）。

```
27  25 int sys_lseek(unsigned int fd,off_t offset, int origin) 26 {.
28          struct file * file;
29          int tmp;
29
```

// まず，関数が提供するパラメータの有効性を判断します．もし，ファイルハンドル
// の値が、プログラム NR_OPEN(20)で開くファイルの最大数よりも大きい場合、または
// ハンドルのファイル構造ポインタが空であるか，対応するファイルのi-nodeフィールドが
// 構造体が空であるか、指定されたデバイスファイルのポインタが位置決めできない場合は、エラーコード
//が返されて終了します。ファイルに対応するi-nodeがパイプノードの場合は、エラーコード

```
30  // がexitに返されます。なぜなら、パイプのヘッドとテールのポインタは自由に動けないからです
31          if (fd >= NR_OPEN || !(file=current->filp[fd]) || !(file->f_inode)
32             || !IS_SEEKABLE(MAJOR(file->f_inode->i_dev)))
33                  return -EBADF;
34          if (file->f_inode->i_pipe)
35                  return -ESPIPE;
```

// そして、設定された位置決めフラグに従って、ファイル読み書きポインタを再配置する。
// Origin = SEEK_SET、つまり、ファイルの読み書きポインタの設定が必要になる
// をファイルの先頭の原点として設定します。オフセット値がゼロより小さい場合は、エラー

```
36  // コードが返され、そうでなければファイルの読み書きポインタをオフセットと同じにします。
37          switch (origin) {
38                  case 0:
39                          if (offset<0) return -EINVAL;
40                          file->f_pos=offset;
41                          break;
```

// Origin = SEEK_CUR であることから、読み書き可能なポインタは、次のように再配置される必要があります。
// ファイルの現在の読み取り/書き込みポインタの原点。もし、ファイルのカレントポインタに加えて
// オフセット値が0より小さい場合は、エラーコードが返されます。そうでない場合は、オフセット値が
// 現在の読み取り/書き込みポインタに追加されます。

```
43                  case 1:
44                          if (file->f_pos+offset<0) return -EINVAL;
45                          file->f_pos += offset;
46                          break;
```

// Origin = SEEK_ENDであることから、読み書き可能なポインタの再配置が必要となる
// ファイルの末尾からのオフセットです。このとき、ファイルサイズにオフセット値を加えた値が
// 0の場合はエラーコードを返します。それ以外の場合は，読み取り/書き込みポインタをファイル長に再配置する
//にオフセット値を加えたものです。

```
48                  case 2:
49                          if ((tmp=file->f_inode->i_size+offset) < 0)
```

```
50                                  return -EINVAL;
51                          file->f_pos = tmp;
52                          break;
```

// オリジンの設定が無効な場合は、エラーコードが返されます。最後に、ファイルの読み取り/書き込みの

// 再配置後のポインタの値を返します。

```
54                  default:
55                          return -EINVAL;
51          }
52          return file->f_pos;
53 }
```

```
54
//// Read file system-call.
// パラメータの'fd'はファイルハンドル、'buf'はユーザーバッファ、'count'は数を表す。
// 読むべきバイト数。
57  55 int sys_read(unsigned int fd,char * buf,int count) 56 {.
58          struct file * file;
59          struct m_inode * inode;
59
// この関数はまず，パラメータの有効性を判断します。もし，ファイルハンドルの値が
// プログラムの最大オープンファイル番号NR_OPENよりも大きい、またはバイトカウント値を
// 読み込んだ値が0より小さい場合や、ハンドルのファイル構造ポインタがNULLの場合は、エラーコードが
60  // を返して終了します。読み込むバイト数が0の場合は、0が返されます。
61          if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
62                  return -EINVAL;
63          if (!count)
64                  return 0;
// 次に、データを保持するために使用されるバッファメモリの上限を確認します。次に、ファイルのi-nodeを
// の属性（モード）に応じて、対応する読み出し操作関数を呼び出します。
// i-node（アイノード）。パイプファイルであり、リードパイプモードであれば、リードパイプの操作は
// を実行します。成功した場合は、読み込んだバイト数が返されます。それ以外の場合は，エラーコード
//が返されます。文字ファイルであれば、文字デバイスの読み取り操作を行い
// 読んだ文字数を返します。ブロックデバイスのファイルであれば、ブロックデバイスの読み取り
65  // 演算を行い、読み込んだバイト数を返します。
66          verify_area(buf,count);
67          inode = file->f_inode;
68          if (inode->i_pipe)
69                  return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
70          if (S_ISCHR(inode->i_mode))
71                  return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
72          if (S_ISBLK(inode->i_mode))
73                  return block_read(inode->i_zone[0],&file->f_pos,buf,count);
// ディレクトリファイルや通常のファイルであれば、まず読み込みバイト「count」の有効性を確認します。
// 読み込んだバイト数に、ファイルの現在の読み書きポインタの値を加えたものがあれば、それを調整します。
// ファイルサイズよりも大きい場合は、ファイルサイズを差し引いた値を読み取りバイト数として設定します。
// 現在の読み取り/書き込みポインタの値。読み取り番号が0に等しい場合は0が返されます。その後
74  // ファイルの読み取り操作を行い、読み取ったバイト数を返して終了します。
75          if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
76                  if (count+file->f_pos > inode->i_size)
77                          count = inode->i_size - file->f_pos;
```

```
78                  if (count<=0)
79                          return 0;
80          return file_read(inode,file,buf,count); 78  }
    // ここで実行されると、ファイルの属性を判断できないということになります。その後、プリント
79  // ノードファイルモードにして、エラーコードを返します。
80          printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
81          return -EINVAL;
81 }
82
    //// Write file system-call.
    // パラメータの'fd'はファイルハンドル、'buf'はユーザーバッファ、'count'は数を表す。
    // 書き込まれるバイト数
```

```
85      83 int sys_write(unsigned int fd,char * buf,int count) 84 {。
86              struct file * file;
87              struct m_inode * inode;
87
        // 同様に，まず，関数のパラメータの有効性をチェックします。もし，ファイルハンドルの値が
        // がプログラムの最大オープンファイル番号NR_OPENより大きい場合、またはバイトカウント値を
        // 読み込んだファイルが0より小さい場合や，ハンドルのファイル構造ポインタがNULLの場合は，エラーコード
        //      if (fd>=NR_OPEN || count <0 || !(file=current->filp[fd]))
        //が返されて終了します。読み込むバイト数が0の場合は、0を返します。
88
89                      return -EINVAL;
90              if (!count)
91                      return 0;
        // 次に、ファイルのi-nodeを取得します。これは対応する書き込み操作を呼び出すために使用されます。
        // i-nodeのモードに応じて機能します。パイプファイルで、書き込みパイプにある場合は
        //モードでは、パイプの書き込み操作が実行されます。成功した場合、書き込まれたバイト数は
        // それ以外の場合はエラーコードを返し、キャラクターデバイスファイルの場合は
        // 書き込まれた文字数を返す;ブロックの場合
        // デバイスファイルでは、ブロックデバイスの書き込み操作が行われ、書き込まれたバイト数が
        // が返されます。通常のファイルであれば、ファイルの書き込み操作を行って
92      // 書き込まれたバイト数を表示して終了します。
93              inode=file->f_inode;
94              if (inode->i_pipe)
95                      return (file->f_mode&2)?write_pipe(inode,buf,count):-EIO;
```

```
        if (S_ISCHR(inode->i_mode))
                return rw_char(WRITE,inode->i_zone[0],buf,count,&file->f_pos);
        if (S_ISBLK(inode->i_mode))
                return block_write(inode->i_zone[0],&file->f_pos,buf,count);
        if (S_ISREG(inode->i_mode))
                return file_write(inode,file,buf,count);
    // ここで実行されると、ファイルの属性を判断できないということになります。その後、プリント
    // i-nodeのファイル属性を設定し、エラーコードを返して終了します。
        printk("(Write)inode->i_mode=%06o\n\r",inode->i_mode);
        return -EINVAL;
}

```

12.13.4 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.

A. 通常、アプリケーションは、Linuxのシステムコールを直接呼び出さず、関数ライブラリ（libc.aなど）のサブルーチンを呼び出しています。しかし、何らかの効率化を図りたい場合は、もちろん直接呼び出すことができます。基本的なライブラリとしては、通常、以下のような基本的な関数やサブルーチンを集めたものを用意する必要があります。

B. System call interface functions;

C. Memory allocation management functions;

D. Set of signal processing functions;

E. String processing functions;

F. Standard input/output functions;

G. Other function sets, such as bsd functions, encryption and decryption functions, arithmetic operations functions, terminal operation functions, and network socket function sets.

これらの関数セットでは、システムコールがオペレーティングシステムの基本的なインターフェース関数となっています。システムコールを含む多くの関数は、Linuxのシステムコール・インターフェースを直接使用する代わりに、システムコール・インターフェース関数セットの標準的な名前を持つシステム関数を呼び出します。これにより、関数ライブラリをOSから大きく独立させることができ、関数ライブラリの移植性を高め、ユーザープログラムのシステム独立性を高くすることができます。新しいライブラリのソースコードでは、システムコールの部分（システムインターフェース部分）を新しいOSのものに置き換えるだけで、基本的に関数ライブラリの移植が完了します。

ライブラリのサブルーチンは、アプリケーションとカーネルシステムの間の中間層と考えることができます。その主な役割は、アプリケーションがシステムコールを実行する際の「ラッパー機能」を提供することに加え、カーネルに含まれない計算機能などの機能的な機能を提供することです。このようにして、呼び出しインターフェースを簡素化し、インターフェースをよりシンプルで覚えやすいものにするとともに、一部のパラメータ検証やエラー処理をこれらのラッピング関数で行うことで、プログラムの信頼性と安定性を高めることができます。

Linuxシステムでは、すべての入出力操作はファイルの読み書きによって行われます。すべてのペリフェラルがシステム内のファイルとして提示されるため、プログラムとペリフェラルの間のすべてのアクセスは、統一されたファイルハンドルを使って処理することができます。通常の場合、ファイルの読み書きを行う前に、まずオープンファイル操作を使って、開始する動作をOSに通知する必要があります。ファイルに書き込み操作を行う場合は、まずこのファイルを作成するか、ファイル内の以前の内容を削除する必要があるかもしれない。また、オペレーティング・システムは、これらの操作を実行する権限があるかどうかを確認する必要があります。問題がなければ、オープン操作はファイル記述子をプログラムに返します。ファイル記述子は、アクセスするファイルを決定するために、ファイル名を置き換えます。これは、MS-DOSのファイルハンドルと同じように機能します。この時点で、オープンファイルに関するすべての情報はシステムによって維持されており、ユーザープログラムはファイル記述子を使ってファイルにアクセスするだけでよい。

ファイルの読み取りと書き込みには、それぞれreadとwriteのシステムコールが使われており、ユーザープログラムは通常、ライブラリのread関数とwrite関数にアクセスすることで、これら2つのシステムコールを実行する。この2つの関数の定義は以下の通りである。

```
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)
{
ロングレスって
        asm volatile
        ("int $0x80")
            : "=a" ( res)
        : "" ( NR_read), "b" ((long)(fd)), "c" ((long)(buf)), "d" ((long)(count))); if ( res>=0)
    return int res; errno=-
    res;
    1を返す。
}
```

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

呼び出したプログラムに-1が返されます。

Linuxカーネルでは、読み取り操作はファイルシステムのread_write.cファイルに実装されています。上記のシステムコール割込みが実行されると、read_write.cファイルの55行目から始まるsys_read()関数が呼び出されます。sys_read()関数のプロトタイプは以下のように定義されています。

```
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.

- そして，カーネルコードは，ファイル記述子に対応する内部ファイルテーブルからファイルのiノードを取得し，そのノードのフラグ情報に従ってファイルの分類・判定を行い，次の種類の読み出し操作関数を呼び出して，実際に読み出したバイト数を返す。
  - 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) を使用する。この関数は、バッファキャッシュマネージャ fs/buffer.c のリードブロック関数 bread()を呼び出し、最後にブロックデバイスドライバの ll_rw_block()関数を呼び出して実際のブロックデバイスのリード操作を行う。
- 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.

読み取り操作のシステムコールが戻ってくると、ライブラリのread()関数は、システムコールの戻り値によって操作が正しいかどうかを判断することができます。戻り値が0より小さい場合は、読み取りエラーが発生したことを意味するので、エラー番号を反転させてグローバル変数errnoに格納し、-1の値をアプリケーションに返します。ユーザーがread()関数を実行してから、カーネル内で実際に動作するまでの全体の流れは図12-29を参照してください。

図12-29 Read()関数呼び出しの実行手順

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.

open.cプログラムは，主にファイルアクセス用のオペレーティングシステムコールを含む。exec.cプログラムは，主にプログラムのロードと実行関数execve()を含む。stat.cプログラムは，ファイルのステータス情報を取得するために使用される。fcntl.cプログラムは，ファイルアクセス制御管理を実

装する。ioctl.cプログラムは，デバイスへのアクセスを制御するために使用される。

12.14.1 Function

open.cプログラムは、ファイルの作成、オープン、クローズ、ファイルのホストや属性の変更、ファイルのアクセス許可の変更、ファイルの操作時間の変更、システムのファイルシステムのルート変更など、ファイル操作に関連する多くのシステムコールを実装しています。

12.14.2 Code annotation

プログラム12-13 linux/fs/open.c

```
1 /*
2  *  linux/fs/open.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <fcntl.h> ファイル制御のヘッダファイルです。演算制御定数記号の定義は
//     used for the file and its descriptors.
// <sys/types.h> 型のヘッダファイル。基本的なシステムデータ型が定義されています。
// <utime.h> ユーザータイムのヘッダーファイルです。アクセス時間と修正時間の構造体とutime()の
//     prototype are defined.
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
//     and constants.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/tty.h> ttyヘッダーファイルは、tty_io、シリアルのパラメータと定数を定義しています。
//     communication.
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
//     used functions of the kernel.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
//     segment register operations.
7 #include <string.h>
8 #include <errno.h>
9 #include <fcntl.h>
10 #include <sys/types.h>
11 #include <utime.h>
12 #include <sys/stat.h>
13
14 #include <linux/sched.h>
15 #include <linux/tty.h>
16 #include <linux/kernel.h>
17
18 #include <asm/segment.h>
```

19

//// Get file system status information.
// パラメータ dev はインストールされたファイルシステムを含むデバイス番号、ubuf は ustat
// 返されたファイルシステム情報を格納するための構造体バッファポインタ。このシステムコールの
// は、マウントされたファイルシステムの統計的なステータス情報を返すために使用されます。で0を返します。
// 成功して、ファイルシステムの空きブロックとアイドルiノードの合計数が追加される

ubufが指すustate構造体に // を入力します。ustat構造体は、以下で定義されています。
// include/sys/types.hです。

```
int sys_ustat(int dev, struct ustat * ubuf) {
        return -ENOSYS;                 // Return error code, not been implemented yet.
}

//// ファイルのアクセスと修正の時間を設定します。
// パラメータのfilenameはファイル名、timesはアクセス時刻と修正時刻
// 設定する必要がある。timesポインタがNULLでない場合、utimbufの時刻情報は
// 構造体は、ファイルのアクセス時間と修正時間を設定するために使用されます；もし times ポインタ
// がNULLの場合は、システムの現在の時刻がアクセス時刻と修正時刻に使われます。
// 指定されたファイルのドメイン。
int sys_utime(char * filename, struct utimbuf * times) {
        struct m_inode * inode;
        long actime,modtime;

// ファイルの時間はそのi-nodeに保存されているので、まず対応するi-nodeを
// ファイル名です。与えられたアクセス時間と修正時間の構造体ポインタの時間が
// NULLの場合、ユーザーが設定した時刻の値が構造体から読み込まれます。それ以外の場合は，現在の時刻
ファイルのアクセス時間と修正時間の設定には、システムの//が使われます。
        if (!(inode=namei(filename)))
                return -ENOENT;
        if (times) {
                actime = get_fs_long((unsigned long *) &times->actime);
                modtime = get_fs_long((unsigned long *) &times->modtime);
        } else
                actime = modtime = CURRENT_TIME;
// その後、i-nodeのアクセス時間フィールドと修正時間フィールドを修正し、i-nodeに
// i-node modified flag, put back i-node, and return 0.
        inode->i_atime = actime;
        inode->i_mtime = modtime;
        inode->i_dirt = 1;
        iput(inode);
        return 0;
}

/*
 * XXX should we use the real or effective uid?  BSD uses the real uid,
 * so as to make this call useful to setuid programs.
*/
//// ファイルのアクセス権を確認してください。
// パラメータモードは、チェックされたアクセス属性です。3つの有効なビットで構成されています。
R_OK
// 値4）、W_OK（2）、X_OK（1）、F_OK（0）のいずれかで、チェック対象のファイルが
// 読み込み可能、書き込み可能、実行可能、またはファイルが存在する。アクセスが許可されていれば
0 を、そうでなければ
```

// エラーコードを返します。

50 48 int sys_access(const char * filename,int mode) 49 {.

51 struct m_inode * inode;

52 int res, i_mode;

52

// ファイルのアクセスパーミッションは、ファイルのi-nodeにも格納されているので、まずは

// ファイル名に対応するi-nodeを取得します。検出されたアクセス属性モードは
下位3ビットの//なので、すべての上位ビットをクリアするには「AND」の8進数07が必要です。もしi-nodeが
ファイル名に対応する // が存在しない場合は、許可されていないエラーコードが返されます。
// i-nodeが存在する場合は、ファイル属性コード、i-nodeが戻されます。さらに

```
//      mode &= 0007;
57行目の"iput(inode);"という文は、61行目以降に配置するのが望ましいです。
53
54          if (!(inode=namei(filename)))
55                  return -EACCES;                     // error code: No access rights.
56          i_mode = res = inode->i_mode & 0777;
57          iput(inode);
```

// 現在のプロセスユーザがファイルの所有者である場合、ファイルの所有者属性が取られます。
// そうでなければ、現在のプロセスユーザーがファイル所有者と同じグループに属している場合、ファイルの
// group属性が取られ、そうでない場合はresの下位3ビットが「その他」のアクセス権となる
// ファイルの

```
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;" [??]
    // この時、resの下位3ビットは、以下の方法で選択されたアクセス属性ビットです。
    // 現在のプロセスとファイルの関係。では、この3つのビットを確認してみましょう。
    // ファイルがパラメータで照会された属性ビットモードを持っていれば，アクセスが許可されて
62  // 0を返します。
63          if ((res & 0007 & mode) == mode)
64                  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           */
    // 現在のユーザーIDが0（スーパーユーザー）で、マスク実行ビットが0の場合や、ファイルが
    // 誰でも実行・検索できる場合は0を、そうでない場合はエラーコードを返します。
70          if ((!current->uid) &&
71              (!(mode & 1) || (i_mode & 0111)))
72                  return 0;
73          return -EACCES;                             // error code: No access rights.
74 }
75
    //// 現在の作業ディレクトリを変更します。
    // パラメータfilenameは、ディレクトリ名です。
    // 操作が成功した場合は0を、そうでない場合はエラーコードを返します。
76 int sys_chdir(const char * filename) 77 {。
78          struct m_inode * inode;
79
    // カレントワーキングディレクトリを変更するには、カレントワーキングディレクトリのフィールドに
    プロセスタスク構造の//は、与えられたディレクトリ名のi-nodeを指します。そこで我々は
    // まず、与えられた名前のi-nodeを取ります。i-nodeがディレクトリのi-nodeでない場合、i-nodeの
    /       if (!(inode = namei(filename)))
    /
    /が戻され、エラーコードが返されます。
80
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          }
    // そして、プロセスの元の作業ディレクトリのi-nodeを解放して、それを指します。
86      // 新たに設定された作業ディレクトリのi-nodeを表示し、0を返します。
87          iput(current->pwd);
88          current->pwd = inode;
89          return (0);
89 }
90
    //// ルートディレクトリの変更。
    // 指定されたディレクトリ名をカレントプロセスのルートディレクトリ'/'に設定します。
    // 操作が成功した場合は0を、そうでない場合はエラーコードを返します。
91 int sys_chroot(const char * filename) 92 {.
93          struct m_inode * inode;
94
    // このシステムコールは、現在のプロセス・タスク構造のルート・フィールドを、次のように変更するためにに使用されます。
    // 与えられたディレクトリ名のi-nodeを指します。与えられた名前のi-nodeが
    // が存在する場合は、エラーコードが返されます。i-nodeがディレクトリi-nodeでない場合、i-nodeは
95      // 戻して、エラーコードを返します。
96          if (!(inode=namei(filename)))
97                  return -ENOENT;
98          if (!S_ISDIR(inode->i_mode)) {
99                  iput(inode);
100                 return -ENOTDIR;
100         }
    // その後、現在のプロセスの元のルートのi-nodeを解放し、それをリセットして
101     // 指定されたディレクトリ名を表示し、0を返します。
102         iput(current->root);
103         current->root = inode;
104         return (0);
104 }
105
    //// ファイルの属性（モード）を変更する。
    // パラメータのmodeには、新しいファイルモードを指定します。
    // 操作が成功した場合は0を、そうでない場合はエラーコードを返します。
106 int sys_chmod(const char * filename,int mode) 107
{。
108         struct m_inode * inode;
109
    // この関数は、指定されたファイルに新しいアクセスモードを設定します。なぜなら、ファイルのモードは
    // がファイルのi-nodeにある場合、まずファイル名に対応するi-nodeを取得する必要があります。
    // 現在のプロセスの実効ユーザーIDがファイルのユーザーIDと異なる場合
    // i-nodeがスーパーユーザーではない場合、ファイルのi-nodeは戻され、エラーコードが返されます。
110     // （アクセス権はありません）。
111         if (!(inode=namei(filename)))
```

```
112                 return -ENOENT;
113         if ((current->euid != inode->i_uid) && !suser()) {
114                 iput(inode);
115                 return -EACCES;
115         }
```

// そうでなければ、与えられたモードはi-nodeのファイル・モードを修正するために使用され、i-nodeは修正されます。

// フラグがセットされ、i-nodeが戻され、0が返されます。

```
116         inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777);
117         inode->i_dirt = 1;
118         iput(inode);
119         return 0;
120 }
121
```

//// ファイルの所有者を変更する。

// パラメータのuidはユーザー識別子（ユーザーID）、gidはグループIDです。

// 操作が成功した場合は0を、そうでない場合はエラーコードを返します。

```
122 int sys_chown(const char * filename,int uid,int gid) 123 {。
124         struct m_inode * inode;
125
```

// この呼び出しは、ファイルのi-nodeにユーザーとグループのIDを設定するために使用されますので、最初に

// 与えられたファイル名のi-nodeを。現在のプロセスがスーパーユーザでない場合は、与えられたファイル名の

```
126
```

// i-nodeで、エラーコード（アクセス権なし）を返します。

```
127         if (!(inode=namei(filename)))
128                 return -ENOENT;
129         if (!suser()) {
130                 iput(inode);
131                 return -EACCES;
131         }
```

// それ以外の場合は、パラメータで指定された値を使用して、ユーザーIDとグループIDを設定します。

```
//      inode->i_uid=uid;
```

//ファイルのi-nodeを設定してフラグを修正し、i-nodeを戻して0を返しま

す
。

```
132
133         inode->i_gid=gid;
134         inode->i_dirt=1;
135         iput(inode);
136         return 0;
137 }
138
```

//// 端末キャラクターのデバイスのプロパティを確認し、設定することができます。
// この補助関数は、次のシステムコールsys_open()でのみ使用されます。使用されるのは
// 現在のプロセスの属性とシステムのttyテーブルの変更と設定を確認する
// オープンファイルがtty端末のキャラクターデバイスの場合。
// 操作が成功した場合は 0 を返します。1を返すと失敗したことになり、その場合は
// 対応するキャラクターデバイスを開くことができません。

```
141 139 static int check_char_dev(struct m_inode * inode, int dev, int flag) 140 {。
142         struct tty_struct *tty;
143         int min;                                    // subdevice number.
143
```

// この関数は、メジャーデバイス番号が4（/dev/ttyxxファイル）の場合にのみ処理します。
//または5（/dev/ttyファイル）です。なお、/dev/ttyのサブデバイス番号は0です。 実際には、あるプロセスに
//制御端末、/dev/ttyはプロセス制御端末装置の同義名。
// つまり、/dev/tty デバイスは、/dev/ttyxx のいずれかに対応する仮想デバイスです。
// プロセスが実際に使用するデバイス。プロセスが制御端末を持っている場合、そのプロセスのためにtty
そのタスク構造の//フィールドは、デバイスNo.4のサブデバイス番号になります。
// では、現在のプロセスの制御端末デバイスを取得して、一時的に保存しましょう
// を後で使えるようにmin変数に入れておきます。パラメータがデバイス5（/dev/tty）を指定している場合は、次のようにします。
// プロセス構造体のttyフィールドの値と同じになるように、つまり、サブデバイスを取得します。
// デバイスNo.4の番号。それ以外の場合は、パラメータにデバイスNo.4のサブデバイスが与えられている場合。
// サブデバイス番号を直接取得します。取得したサブデバイス番号が、デバイスNo.

```
// 4が0より小さい場合は、プロセスに制御端末がないか、デバイス番号が
// が正しくない場合は、-1が返されます。
        if (MAJOR(dev) == 4 || MAJOR(dev) == 5) {
                if (MAJOR(dev) == 5)
                        min = current->tty;
                else
                        min = MINOR(dev);
                if (min < 0)
                        return -1;
// プライマリ疑似端末デバイスは、1つのプロセスでのみ独占的に使用することができます。そのため
// サブデバイス番号がプライマリ疑似端末を示し、オープンファイルのi-nodeの参照がある場合
// countが1より大きい場合は、デバイスが他のプロセスによって使用されたことを示します。 so
// キャラクターデバイスはもう開くことができないので、-1が返されます。それ以外の場合は
// ポインタ tty は，tty テーブルの対応する構造項目を指します.
// ファイルオープン操作フラグにO_NOCTTYフラグが含まれておらず、かつ、プロセスが
//はグループリーダーであり、制御端末を持たず、tty構造のセッションフィールド
// は0（その端末がまだどのプロセスグループの制御端末でもないことを示す）。
// とすると、このプロセス用のmin端末装置を制御端末として設定することができます。
// そのため、プロセスのタスク構造に端末デバイス番号フィールドttyの値を設定する
// をminに設定し、tty構造のセッション番号とグループ番号を
// それぞれ、プロセスのセッション番号とグループ番号に等しいです。
                if ((IS_A_PTY_MASTER(min)) && (inode->i_count>1))

                        return -1;
                tty = TTY_TABLE(min);
                if (!(flag & O_NOCTTY) &&
                    current->leader &&
                    current->tty<0 &&
                    tty->session==0) {
                        current->tty = min;
                        tty->session= current->session;
                        tty->pgrp = current->pgrp;
                }
// オープンファイル操作フラグにO_NONBLOCKフラグが含まれている場合は、関連する
// キャラクター端末機器の設定。読み取り操作を満足させるためには、最小の数
読み出す文字数の//が0で、タイムアウトのタイミング値が0に設定されており、端末機器の
//が非正規のモードに設定されています。ノンブロッキングモードは非正規モードでのみ動作します。
// このモードでは、VMINとVTIMEの両方が0に設定されている場合、何文字であっても
                if (flag & O_NONBLOCK) {
```

```
162
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
```

// プロパティは、将来のファイルへのアクセスにのみ適用されます。を作成するオープンコールは，読み取り専用の
また、 // ファイルは、読み取りと書き込みが可能なファイルハンドルを返します。呼び出し操作が成功すると
// ファイルハンドル（ファイルディスクリプター）が返され、そうでない場合はエラーコードが返されます。参照
int sys_open(const char * filename,int flag,int mode)

```
{
        struct m_inode * inode;
        struct file * f;
        int i,fd;

// 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).
        mode &= 0777 & ~current->umask;
        for(fd=0 ; fd<NR_OPEN ; fd++)
                if (!current->filp[fd])                 // free item found.
                        break;
        if (fd>=NR_OPEN)
                return -EINVAL;
// そして、現在のプロセスのclose_on_execファイルハンドルのビットマップを設定し、対応する
// bit. close_on_exec は、プロセスのすべてのファイルハンドルに対するビットマップフラグです。各ビットが表すのは
// 開いているファイル記述子で，そのファイルハンドルを閉じる必要があります。
// システムコールの execve() が呼び出されます。プログラムがfork()を使って子プロセスを作成すると
```

// 関数を呼び出すと、通常は子プロセスのexecve()関数を呼び出して、別の新しい
// のプログラムを作成し、子プロセスで新しいプログラムの実行を開始します。のビットが消えていたら
ファイルハンドルの // close_on_exec が設定されている場合，対応するファイルハンドルは，以下のときに閉じられます。
// execve()が実行された場合、そうでなければファイルハンドルは常に開いたままになります。
// ファイルがオープンされると、子プロセスではデフォルトでファイルハンドルもオープンされるので
// 対応するビットをここでリセットする必要があります。次に、ファイルテーブルの空きエントリを探します。
// (参照カウント0のエントリ)を開いているファイルのために、fにそのファイルを指し示すようにします。
// テーブルの配列を指定して検索を開始します。また、行のポインター割り当て「0+file_table」は
//　　current->close_on_exec &= ~(1<<fd);
//184は"file_table"と"&file_table[0]"に相当しますが、こちらの方がわかりやすいかもしれません。
183

```
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;
```

// この時点で、プロセスのファイルハンドルfdのファイル構造ポインタに
// 検索されたファイル構造を、ファイル参照カウントを1つ増やしてから
// 関数 open_namei() で開く操作を行います。戻り値が0より小さい場合は
// エラーを示しているので、先ほど適用したファイル構造を解除し、エラーコード
// が返されます。ファイルオープン操作が成功した場合，inodeは次のようなi-nodeポインタになります。

```
189
190         if ((i=open_namei(filename,flag,mode,&inode))<0) {
191                 current->filp[fd]=NULL;
192                 f->f_count=0;
193                 return i;
194         }
```

// 開かれたファイル(current->filp[fd]=f)->f_count++;
// 開かれたファイルのi-nodeのmodeフィールドに基づいて、ファイルのタイプを知ることができます。
についてはは

```
// ファイルの種類によっては、特別な処理が必要になることがあります。文字デバイスが
// ファイルが開かれたら、check_char_dev() 関数を呼び出して、現在の
// 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.
/* ttys are somewhat special (ttyxx major==4, tty major==5) */
```

```
195
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                 }
    // ブロックデバイスファイルが開かれている場合、ディスクが交換されたかどうかを確認します。置き換えられている場合は、すべての
203 // キャッシュバッファ内のデバイスのバッファブロックを無効にする必要があります。
204 /* Likewise with block-devices: check for floppy_change */
205         if (S_ISBLK(inode->i_mode))
206                 check_disk_change(inode->i_zone[0]);
    // ここで、開いたファイルのファイル構造を初期化します。ファイル構造のプロパティを設定する
    // とフラグを設定し、ハンドルの参照カウントを1にして、i-nodeフィールドを
207 // 開いているファイル、ファイルの読み書きポインタを0にして、最後にファイルハンドルを返します。
208         f->f_mode = inode->i_mode;
209         f->f_flags = flag;
210         f->f_count = 1;
211         f->f_inode = inode;
210         f->f_pos = 0;
211         return (fd);
```

```
}

//// Create file system call.
// パラメータのpathnameはパス名、modeは上記のsys_open()と同じです。
// 成功した場合はファイルハンドルを、そうでない場合はエラーコードを返します。
int sys_creat(const char * pathname, int mode) {.
        return sys_open(pathname, O_CREAT | O_TRUNC, mode);
}

// ファイルシステムコールを閉じる。
// パラメータ fd はファイルハンドルです。
// 成功した場合は0を、そうでない場合はエラーコードを返します。
int sys_close(unsigned int fd)
{.
        struct file * filp;

// まず、パラメータの有効性をチェックします。もし，与えられたファイルハンドルの値が
// プログラムが同時に開くことができるファイルの数 NR_OPEN を超えると、エラーコード（無効な
//パラメータ）が返されます。そして、ファイルハンドルの対応するビットをリセットします。
// プロセスのclose_on_execビットマップ。さらに、ファイルのファイル構造ポインタが
// ハンドルがNULLの場合は、エラーコードが返されます。
        if (fd >= NR_OPEN)
                return -EINVAL;
        current->close_on_exec &= ~(1<<fd);
```

```
        if (!(filp = current->filp[fd]))
                return -EINVAL;
// ここで，ファイルハンドルのファイル構造ポインタをNULLに設定します。もし，ハンドルの参照が
// ファイルが閉じられる前に，ファイル構造のカウントがすでに0になっていた場合，カーネルエラーが
発生する
// となり、マシンが停止します。そうでなければ，ファイル構造体の参照カウントがデクリメントされ
る
// この時点で、まだ0でなければ、他のプロセスが
// ファイルなので、0（成功）を返します。現在，参照カウントが0であれば，そのファイルには
// 処理の参照を行い、ファイル構造がフリーになりました。その後、ファイルのi-nodeを解放して
        current->filp[fd] = NULL;
```

```

        if (filp->f_count == 0)
                panic("Close: file count is 0");
        if (--filp->f_count)
                return (0);
        iput(filp->f_inode);
        return (0);
}

```

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()と、exec()関数群のカーネル実装関数です。他の5つの関連するexec関数は、一般的にライブラリ関数で実装され、最終的にはこの関数が呼び出されます。

Linuxカーネルバージョン0.12では、a.out形式の実行ファイルのみをサポートしています。このフォーマットは、シンプルでわかりやすく、入門的な学習に適しています。a.out形式の実行ファイルのヘッダには、以下のようなexecデータ構造があります。実行ファイルの基本的なフォーマットと内容が記述されています。a.out形式の詳細な説明は，第14章のinclude/a.out.hヘッダファイルを参照してください。

```
// 実行ファイルのヘッダ構造です。6 struct exec
{
  unsigned long a_magic;        /* Use macros N_MAGIC, etc for access */
  unsigned a_text;              /* length of text, in bytes */
```

```
10  unsigned a_data;          /* length of data, in bytes */
11  unsigned a_bss;           /* length of uninitialized data area for file, in bytes */
12  unsigned a_syms;          /* length of symbol table data in file, in bytes */
13  unsigned a_entry;         /* start address */
14  unsigned a_trsize;        /* length of relocation info for text, in bytes */
15  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

- を子プロセスで呼び出し、別の新しいプログラムをロードして実行します。この時点で、子プロセスのコード、データセグメント（ヒープ、スタックの内容を含む）は、新しいプログラムで完全に置き換えられ、新しいプログラムが子プロセスで実行されます。execve()関数の主な用途は以下の通りです。
  - Perform initialization operations on command line parameters and environment parameter space pages
  - -- 空間開始ポインタの初期設定，空間ページポインタ配列の(NULL)への初期化，実行ファイル名に応じたファイルのi-nodeの取得，パラメータ数と環境変数数の計算，ファイルタイプと実行許可の確認。
  - 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.

execve()の実行中、カーネルはfork()によってコピーされたオリジナルプログラムのページディレクトリとページテーブルのエントリをクリアし、対応するページを解放します。カーネルは、新たにロードされたプログラムコードのプロセス構造の情報を再設定し、コマンドラインパラメータや環境パラメータブロックが占有するメモリページを要求してマッピングし、コードの実行ポイントを設定するのみである。この時点では、カーネルは実行ファイルが置かれているブロックデバイスから新しいプログラムのコードとデータを直ちにロードするわけではない。execve()が終了して戻ると、新しいプログラムの実行が始まるが、最初の実行では間違いなくページフォルト例外が発生する。なぜなら、コードやデータがまだブロックデバイスからメモリに読み込まれていないからです。このとき、ページ・フォールト例外処理手順は、例外の原因となったリニア・アドレスに応じて、主記憶領域に新プログラム用のメモリ・ページ（メモリ・フレーム）を要求し、指定されたページをブロック・デバイスから読み出すとともに、リニア・アドレスに対応するメモリ・ページ・ディレクトリ・エントリとページ・テーブル・エントリを設定する。このような実行ファイルの読み込み方法は、メモリ管理の章で述べたように、ロード・オン・デマンドと呼ばれる。

また、新プログラムは子プロセスで実行されるため、子プロセスは新プログラムのプロセスとなり、新プログラムのプロセスIDは子プロセスのプロセスIDとなります。同様に、子プロセスのプロパティは、新プログラムのプロセスのプロパティとなり、オープンファイルの処理は、各ファイルディスクリプタのクローズオン実行フラグに関係する。linux/fs/fcntl.cファイルの記述を参照すると、プロセス内の各オープンファイルディスクリプタは、クローズオン実行フラグを持っている。プロセス制

御構造体では、このフラグは unsigned long integer close_on_exec で表され、各ビットが各ファイル記述子のフラグを表す。close_on_execの対応するビットにファイルディスクリプターがセットされていれば、execve()が実行されたときにディスクリプターはクローズされ、そうでなければディスクリプターは常にオープンされます。ファイル制御関数fcntl()を使ってこのフラグを特別に設定しない限り、カーネルのデフォルトの動作では、execve()の実行後もディスクリプターはオープンのままです。

以下に、コマンドラインパラメーターと環境パラメーターの意味を説明します。ユーザーがコマンドプロンプトでコマンドを入力すると、実行指定されたプログラムは入力されたコマンドを受け入れます。

コマンドラインからのライン引数を使用します。例えば、ユーザーが次のようなファイル名リストコマンドを入力した場合。

```
ls -l /home/john/
```

The shell process creates a new process and executes the /bin/ls command there. The three parameters ls, -l, and
また、/bin/ls実行ファイルのロード時にコマンドラインで指定した/home/john/は、新しいプロセスに継承されます。C言語をサポートしている環境では、プログラムのメイン関数main()を呼び出す際に、2つの引数を取ります。

```
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.

図 12-30 コマンドライン引数ポインタ配列 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 NULL, 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.

execve()関数は、コマンドラインの引数や環境空間に対して多くの処理動作を行います。各プロセスやタスクのパラメータや環境空間は、合計でMAX_ARG_PAGESページを持つことができ、そのサイズは128kBバイトに達することもあります。この空間へのデータの格納方法は、スタック操作と同様で、パラメータや環境変数の文字列は、想定される128kBの空間の端から逆に格納されます。初期状態では、プログラムはスペース内にスペースの終わり（128kB-4バイト）を指すオフセット値pを定義しています。このオフセットは、図12-31に示すように、データが格納されるにつれて後退していきます。図からわかるように、pは現在のパラメータ環境空間にどれだけの空き領域が残っているかを明確に示しています。copy_string()関数は、コマンドライン引数や環境文字列をユーザーメモリ空間からカーネルのフリーページにコピーするために使用されます。copy_string()を解析する際には、この図を参考にすることができます。

図12-31 パラメーターと環境変数の文字列スペース

After copy_string() is executed, the kernel code will adjusts 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.

図12-32 pを初期スタックポインタに変換する方法

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.

図 12-33 新プログラムスタックのポインタテーブルの模式図

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.

図 12-34 実行ファイルの読み込み時のスタック内のespおよびeipの変化

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

プログラム 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
```

// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナルの定義

```
//      manipulation function prototypes.
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
//      and constants.
// <a.out.h> a.outヘッダーファイルは、a.out実行ファイルのフォーマットといくつかのマクロを定義しています。
// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する
 (file,buffer_head,
//      m_inode, etc.).
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
//      used functions of the kernel.
// <linux/mm.h> メモリ管理用のヘッダーファイルです。ページサイズの定義と、いくつかのページ
//      release function prototypes.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
//      segment register operations.
#include <signal.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>
#include <a.out.h>

#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <asm/segment.h>

extern int sys_exit(int exit_code);         // kernel/exit.c, line 365.
extern int sys_close(int fd);               // fs/open.c, line 219.

/*
 * MAX_ARG_PAGES defines the number of pages allocated for arguments
 * and envelope for the new program. 32 should suffice, this gives
 * a maximum env+arg of 128kB !
*/
#define MAX_ARG_PAGES 32

//// ライブラリファイルを使用します。
// パラメータ：library - ライブラリファイルの名前。
// プロセス用のライブラリファイルを選択し、現在のライブラリのi-nodeフィールドを置き換える
```

// ここで指定されたライブラリファイルのi-nodeを持つプロセスのファイルです。もし、指定されたライブラリ
パラメータの // が空の場合、プロセスの現在のライブラリファイルが解放されます。
// 戻り値：成功した場合は0、そうでない場合はエラーコードを返します。

44 42 int sys_uselib(const char * library) 43 {。

```
45          struct m_inode * inode;
46          unsigned long base;
46
```

// まず、現在のプロセスが正常なプロセスであるかどうかを判断します。これを行うには
// 現在のプロセス空間のサイズ、なぜなら通常のプロセスの空間サイズは
// task_size (64mb)。そのため、プロセスの論理アドレス空間のサイズが

// をTASK_SIZEに設定した場合はエラーコード（無効なパラメータ）が返され、そうでない場合はライブラリファイルのinode
// が取られます。ライブラリファイル名ポインタが空の場合は，inodeにNULLを設定する。

```
        if (get_limit(0x17) != TASK_SIZE)
                return -EINVAL;
        if (library) {
                if (!(inode=namei(library)))            /* get library inode */
                        return -ENOENT;
        } else
                inode = NULL;
/* we should check filetypes (headers etc), but we don't */
```

// そして、プロセス・オリジナル・ライブラリ・ファイルをi-nodeに戻し、プロセス・ライブラリi-nodeをプリセットする。
// フィールドをnullにします。その後、プロセスのライブラリコードの位置を取得して、リリースの元のライブラリコードの//ページテーブルとメモリページが占有されている。最後に、プロセスに
// library i-node fieldが新しいライブラリi-nodeを指し、0（成功）を返します。読み込みと同様に
// 実行ファイルの場合は、実際のライブラリコードがメモリにロードされてから実行されます。
// を使用しています。また、ライブラリファイルのコードは、プロセス空間の最後に配置されているサイズの

// linux/sched.h, line 26.

```
 * create_tables() parses the env- and arg-strings in new user
 * memory and creates the pointer tables from them, and puts their
 * addresses on the "stack", returning the new stack pointer value.
*/
```

//// 新しいタスクスタックにパラメータと環境変数のポインタテーブルを作成します。
// 前述の図12-31から図12-33を参照してください。
// パラメータ：p - データセグメント内のパラメータと環境情報のオフセットポインタ。
// argc - パラメータの数、 envc - 環境変数の数。
// この関数は、スタックポインタを返します。

```
static unsigned long * create_tables(char * p,int argc,int envc) {
        unsigned long *argv,*envp;
        unsigned long * sp;

```

// スタックポインタは4バイト境界でアドレスされるので、ここではspを倍数にする必要があります。
// この時点で、spはパラメータ環境表の最後になります。次に、まず
// spダウン（ローアドレス方向）で、環境変数のポインタが占める空間を空ける
// をスタック上に置き、環境ポインタ envp がそれを指すようにします。ここでもう一か所、使用され

ている
// で最後にNULL値を格納します。spを再び下に移動し，コマンドライン引数のためのスペースを確保する
// ポインタを使用し、そのポインタに argv ポインタを指定します。同様に，より予約された場所を使用する
// でNULL値を保持します。この時点で sp は，パラメータポインタブロックの先頭を指しています.
// そして、環境パラメータブロックポインタenvpとコマンドラインパラメータをプッシュします。
// コマンドラインパラメータの数と同様に、ブロックポインタをスタックに格納します。
74 // 次のポインタプラス1の操作で、spが4バイト増加します。
75 sp = (unsigned long *) (0xfffffffc & (unsigned long) p);
76 sp -= envc+1; // That is sp = sp - (envc+1);

```
        envp = sp;
        sp -= argc+1;
        argv = sp;
        put_fs_long((unsigned long)envp,--sp);       // put in user space.
        put_fs_long((unsigned long)argv,--sp);
        put_fs_long((unsigned long)argc,--sp);
    // そして、コマンドラインのポインターと環境変数のポインターを
    // 上記で予約した対応する場所に、最後にNULLポインタを配置します。
        while (argc-->0) {
                put_fs_long((unsigned long) p,argv++);
        while (get_fs_byte(p++)) /* nothing */ ; // p points to next param string.    }
        put_fs_long(0,argv);
        while (envc-->0) {
                put_fs_long((unsigned long) p,envp++);
        while (get_fs_byte(p++)) /* nothing */ ; // p points to next param string.    }
        put_fs_long(0,envp);
        return sp;                  // returns the current new stack pointer constructed.
}

/*
 * count() counts the number of arguments/envelopes
 */
    //// パラメータの数をカウントします。
    // Parameters: argv - パラメータのポインタの配列，最後のポインタはNULLです.
    // ポインタ配列内のポインタの数をカウントし、そのカウント値を返します。
static int count(char ** argv) {
        int i=0;
        char ** tmp;

        if (tmp = argv)
                while (get_fs_long((unsigned long *) (tmp++)))
                i++;

        return i;
}

/*
 * 'copy_string()' copies argument/envelope strings from user
 * memory to free pages in kernel mem. These are in a format ready
 * to be put directly into the top of new user memory.
 *
 * Modified by TYT, 11/24/91 to add the from_kmem argument, which specifies
 * whether the string and the string array are from user or kernel segments:
 *
 * from_kmem     argv *        argv **
 *    0          user space    user space
 *    1          kernel space  user space
 *    2          kernel space  kernel space
 *
 * 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 */
    //// パラメータ文字列をプロセスのパラメータ空間と環境空間にコピーします。
    // パラメータ： argc - 追加される引数の数、 argv - 引数ポインタの配列。
    // page - 環境空間ページポインタへの引数の配列; p - オフセットポインタ
    // 引数テーブルのスペースに、コピーされた文字列のヘッダを常に指し示す。
    // from_kmem - 文字列のソースフラグです。
    // do_execve()関数では、p は引数の最後のロングワードを指すように初期化されます。
    // テーブル(128kB)のスペースに、引数の文字列を逆にコピーして、スタック操作で
    //モードになります。そのため、コピー情報の増加に伴い、pポインタは徐々に減少していきます。
    // で、常にパラメータ文字列の先頭を指します。文字列ソースフラグ from_kmem は
    // execve()にスクリプトファイルを実行する機能を持たせるためにTYT(Tytso)が追加した新しいパラメ
    ータである。
    // execve()にスクリプトファイルを実行する機能がない場合、すべてのパラメータ文字列は
    // ユーザーデータスペースの
127 // 戻り値：引数/環境空間の現在のヘッドポインタ、またはエラーの場合は0。
128 static unsigned long copy_strings(int argc,char ** argv,unsigned long *page,
129                 unsigned long p, int from_kmem)
129 {
130         char *tmp, *pag;
131         int len, offset = 0;
132         unsigned long old_fs, new_fs;
133
    // まず、カレントセグメントレジスタDS（カーネルデータセグメントを指している）とFS（ユーザー
    // データセグメント）の値を取得し、それぞれ変数new_fs、old_fsに格納します。
134     文字列とそのポインタがカーネル空間にある場合、 // カーネル空間を指すように fs を設定する。
135         if (!p)
136                 return 0;           /* bullet-proofing */ // offset pointer verification
137         new_fs = get_ds();
138         old_fs = get_fs();
139         if (from_kmem==2)           // string and string pointers all in kernel space.
140                 set_fs(new_fs);
    // 次に、パラメータをループし、最後のパラメータから逆にコピーを開始して、コピー
    // を指定されたオフセットアドレスに移動します。ループ内では，まず，現在の文字列ポインタを取得
    して
    // をコピーする必要があります。文字列がユーザースペースにあり、文字列配列（文字列ポインタ）が
    // がカーネル空間にある場合、まずFSセグメントレジスタをカーネルデータセグメントに設定します。
    // カーネルデータの文字列ポインタtmpを取得した直後にFSを復元します。
    // そうでない場合は、文字列ポインタがユーザ空間からtmpに直接取り込まれます。
    //      while (argc-- > 0) {
    FS値を変更する。
```

```
140
141                 if (from_kmem == 1)         // FS points to kernel space if ponters in it.
142                         set_fs(new_fs);
143                 if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
144                         panic("argc is wrong");
145                 if (from_kmem == 1)             // if string pointers in kernel.
146                         set_fs(old_fs);         // FS points back to user space.
```

// その後、ユーザースペースから文字列が取り出され、パラメータ文字列の長さlenは

// 計算され，その後，tmp は文字列の終わりを指す。もし，文字列の長さが

// この時点でパラメータと環境の空間に残っている自由長は、空間の

//では不十分です。その後、FSセグメントレジスタの値が（変更されていれば）復元され、0が返されます。

// しかし、パラメータ/環境変数が128KBであるため、これは起こらないはずです。

```
148                 len=0;                          /* remember zero-padding */
149                 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             }
```

// 次に、文字列をcharずつ逆にコピーして、パラメータと環境の最後に置く
//のスペースです。文字列の文字をループさせる過程で、まずは
// パラメータと環境の対応する位置にメモリページがあるかどうか
// のスペースを確保します。存在しない場合は、まずメモリのページを申請します。オフセット」は、その時の
// ページ内の現在のポインタのオフセット値。オフセット変数 'offset' は初期化されているので
この関数の先頭で // を 0 にすると、次のように '(offset-1 < 0)' のチェックが確実に行われます。
// 真にすると、'offset'がページ範囲内の現在のpポインタに再設定されます。

```
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.
```

// 文字列空間のページポインタの配列項目（page[p/PAGE_SIZE]）がゼロの場合、それは
// pポインタが配置されている空間メモリページがまだ存在しない場合は
// 空いているメモリページを探し、そのページポインタを配列に記入します。また、それと同時に
// ページポインタ pag が新しいページを指すようにします。空きページがない場合は 0 を返します。

```
162                             if (!(pag = (char *) page[p/PAGE_SIZE]) &&
163                                 !(pag = (char *) page[p/PAGE_SIZE] =
164                                   (unsigned long *) get_free_page()))
165                                     return 0;
166                             if (from_kmem==2)          // If string in kernel space,
167                                     set_fs(new_fs);    // FS points to kernel space.
167
168                     }
```

// そして、FSセグメントから文字列の1バイトをパラメータのオフセットにコピーして
// メモリページの環境変数 'pag'.

```
169                     *(pag + offset) = get_fs_byte(tmp);
170             }
171     }
```

// 最後に、文字列と文字列配列がカーネル空間にある場合は、元の値であるFS
// セグメント・レジスタが復元され、パラメータでコピーされたパラメータのヘッダ・オフセットが
//　　if (from_kmem==2)
と環境変数が返されます。

```
172
173                 set_fs(old_fs);
174         return p;
175 }
176
```

//// タスクのローカルディスクリプターテーブルを修正します。
// ローカルディスクリプターのセグメントベースアドレスと長さの制限を変更する
// テーブルLDTで、データセグメントの最後にパラメータと環境スペースのページを配置します。
// Parameters: text_size - a_textフィールドで指定されたコードセグメントの長さの上限。
// page - パラメータと環境変数のページポインタの配列。
// データセグメントの制限値（64MB）を返します。

```
177 static unsigned long change_ldt(unsigned long text_size,unsigned long * page) 178 {。
180         unsigned long code_limit,data_limit,code_base,data_base;
```

```
181         int i;
181
    // まず、コードとデータセグメントの長さの制限を64MBに設定し、コードセグメントを
    // 現在のプロセスのLDTにあるコード・セグメント・ディスクリプターのベース・アドレスです。コードセグメント記述子の
    // のベースアドレスは、データセグメントのベースアドレスと同じです。これらの新しい値は、その後
    // のコードセグメントとデータセグメントの記述子のベース長とセグメント長を再設定します。
    // LDTです。ここで注意していただきたいのは、新しいプログラムのコードとデータセグメントのベースアドレスが
    読み込み中の // は元のプログラムと同じなので、繰り返す必要はありません。ということです。
    // セグメントのベースアドレスを設定する186,188行目の2つのステートメントは冗長です。
182     // と省略することができます。
183         code_limit = TASK_SIZE;
184         data_limit = TASK_SIZE;
185         code_base = get_base(current->ldt[1]);        // include/linux/sched.h, line 277
186         data_base = code_base;
187         set_base(current->ldt[1],code_base);
188         set_limit(current->ldt[1],code_limit);
189         set_base(current->ldt[2],data_base);
190         set_limit(current->ldt[2],data_limit);
190 /* make sure fs points to the NEW data segment */
    // ローカルテーブルデータセグメント記述子（0x17）のセレクタをFSレジスタに入れる。
    // つまり、デフォルトでは、FSはタスクデータセグメントを指しています。そして、データが格納されているページを
    // データの最後にパラメータと環境空間（MAX_ARG_PAGESページ、128kBまで）で
    // セグメント。その方法は、AAの最初から、1ページずつ逆に入れていく。その方法は
    //です。プロセス空間のライブラリ・コードの位置の先頭から、逆算して
    // セグメントを1ページずつ表示します。ライブラリファイルのコードは、プロセス空間の端を占めており
     // は4MBの倍数です。関数 put_dirty_page()は、物理ページを
    // process logic space. See the mm/memory.c file.
191         __asm__("push1 $0x17\n\tpop %%fs"::);
192         data_base += data_limit - LIBRARY_SIZE;
193         for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
194                 data_base -= PAGE_SIZE;
195                 if (page[i])                          // put the page if exists.
196                         put_dirty_page(page[i],data_base);
197         }
198         return data_limit;              // return the length limit of data segment(64MB).
199 }
200
201 /*
202  * 'do_execve()' executes a new program.
204 203 *
205  * NOTE! We leave 4MB free at the top of the data-area for a loadable
206  * library.
206 */
    //// 他のプログラムを読み込んで実行します。execve()システムコールで呼び出されます。
```

// これは、システムコール割り込み（int 0x80）で呼び出される関数で、関数番号はNR_execveです。
// 関数の引数は、次のようにしてスタックに徐々に押し込まれる値です。
// 関数が呼び出されるまで、system-callが実行される（kernel/system_call.s, line 217）。
// これらの値（system_call.sファイル内）は以下の通りです。
// (1)89〜91行目でスタックにプッシュされたedx, ecx, ebxのレジスタ値は
それぞれ **envp, **argv, *filename への // の設定 (2) 関数の戻り値(tmp)
// 99行目でsys_call_tableのsys_execve関数が呼ばれたときに、スタックにプッシュされます。
// (3) 216行目に、プッシュされたシステム割り込みを呼び出すプログラムコードポインタeip
この関数 do_execve() を呼び出す前に、 // スタックに入れておきます。

```
// Parameters: eip - システム割り込みを呼び出すためのプログラムコードポインタ。
// tmp - _sys_execveを呼び出したときのシステム割り込みのリターンアドレス、使われていません。
// filename - 実行可能ファイル名へのポインタです。
// argv - コマンドライン引数ポインタの配列へのポインタ。
// envp - 環境変数のポインタの配列へのポインタです。
207 // 戻り値：成功した場合は戻りませんが、そうでない場合はエラーコードを設定して-1を返します。
208 int do_execve(unsigned long * eip,long tmp,char * filename,
209         char ** argv, char ** envp)
209 {
// 以下のローカル変数の一部の意味は以下の通りです。
// 213行目 - 引数と環境文字列(A&E)の空間ページポインタの配列です。
// 217行目 - シェルスクリプトファイルを実行するかどうかを制御するフラグです。
210     // 218行目 - 引数＆環境空間の末尾を指すのに使われる整数。
211         struct m_inode * inode;
212         struct buffer_head * bh;
213         struct exec ex;
214         unsigned long page[MAX_ARG_PAGES];          // array of page pointers in A&E space.
215         int i,argc,envc;
216         int e_uid, e_gid;                           // effective user and group ID.
217         int retval;
218         int sh_bang = 0;                            // control if we execute script file.
219         unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4; // p points to the end of the A&E space.
219
// 正式に実行ファイルの環境を整える前に、いくつかの準備をしておきましょう。
//作業を行います。カーネルは128KB（32ページ）のスペースを用意して、コマンドラインの引数や
// 実行ファイルの環境文字列変数。前の行では，pを初期状態である
// 128KBスペースの最後のロングワードに配置されます。初期化パラメータと
// pは、128KB空間内の現在の位置を示すために使用されます。
// さらに、パラメータeip[1]は、オリジナルのユーザープログラムコードのセグメントレジスタCS値
このシステムコールを呼び出す // セグメントセレクタはもちろんコードセグメントでなければなりません。
// 現在のタスクのセレクタ(0x000f)。これが値でない場合、CSはセレクタにしかなれません。
カーネルコードセグメントの // 0x0008 を使用しています。しかし、これは絶対に許されないことで、カーネル
//コードは常駐しており、交換することはできません。そのため、以下のような値になっています。
// eip[1]で正常かどうかを確認します。そして、128KBの引数と環境を初期化します。
// 文字列空間、すべてのバイトをクリアし、実行ファイルのi-nodeを取得します。次に、以下に従って
// 関数のパラメータを計算し、コマンドラインパラメータargcの数と
環境文字列の // envc. また、実行ファイルは通常のファイルである必要があります。
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         }
```

// そして、現在のプロセスが、指定された実行ファイルを実行する権利を持っているかどうかをチェックします。

// つまり、実行ファイルのi-nodeにあるモードに応じて、そのプロセスが

// 実行する許可を得ています。まず、「set-user_id」フラグと
// "set-group-id" フラグがモードに設定されます。これらの2つのフラグは、主に、一般的な
// ユーザーが passwd などの特権ユーザー（スーパーユーザー root など）のプログラムを実行できるようにします。
// パスワードを変更するものです。set-user-id フラグが設定されている場合は、その後の
// 実行プロセスには実行ファイルのユーザーIDが設定され、そうでない場合は
// 現在のプロセスの euid。実行ファイルのset-group-idが設定されている場合、有効なグループ
// 実行ファイルのグループIDに実行プロセスのID(egid)が設定され、それ以外は
// 現在のプロセスのegidに設定されます。ここでは、決定した2つの値を一時的に
// 変数 e_uid と e_gid に格納されます。

```
        i = inode->i_mode;
        e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
        e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
```

// ここで、プロセスのeuidおよびegidと、実行ファイルのアクセスモードを比較します。
// 実行ファイルがプロセスを実行しているユーザーのものであれば、ファイルモードの値がシフトする
// を6ビット分右に移動させ、下位3ビットはファイルのアクセス許可フラグ
// オーナーになります。それ以外の場合は、実行ファイルが現在のユーザーと同じグループに属していれば
//処理では、属性が3ビット右にシフトされ、最下位の3ビットが
// 実行ファイルグループのアクセス許可フラグ。それ以外の場合は、最下位3ビットの
この時の//モードは、他のユーザーが実行ファイルにアクセスするための権限です。
// 次に、現在のプロセスが実行ファイルを実行する権限を持っているかどうかを判断します。
// 最も低い3ビットの値になります。選択されたユーザーがファイルを実行する権利を持っていない場合（ビット
// 0は実行許可）、他のユーザーが何の権利も持っていない場合や、現在の
// プロセスユーザがスーパーユーザでない場合は、現在のプロセスが
// 実行ファイルを実行する権限を持つ。そして、実行不可能なエラーコードを設定し、次のようにジャンプします。

```
        // exec_error2で終了処理を行う。
        if (current->euid == inode->i_uid)
        i >>= 6;
        else if (in_group_p(inode->i_gid))
                i >>= 3;
        if (!(i & 1) &&
            !((inode->i_mode & 0111) && suser())) {
                retval = -ENOEXEC;
                goto exec_error2;
        }
```

// ここでコードが実行できる場合、現在のプロセスに実行権限があることを意味する
// 指定された実行ファイルを表示します。ですから、ここからは実行ファイルからデータを抽出する必要があります。
//ファイルのヘッダを解析し、その情報に基づいて実行環境を設定したり、ラン
// 別のシェルプログラムでこのスクリプトファイルを実行します。まず、実行の最初のブロックである
// ファイルがバッファブロックに読み込まれ、バッファブロックのデータがex構造体にコピーされます。

// 実行ファイルの最初の2バイトが文字'#!'であれば、それはスクリプトファイルです。
// スクリプトファイルを実行したい場合は、そのためのインタープリタ（シェルなど）を実行する必要があります。
// プログラム）を作成します。通常、スクリプトファイルの最初の行は「#!/bin/bash」となっていますが、これは次のように指定します。
// スクリプトファイルを実行するのに必要なインタープリタです。実行方法は、インタープリタを
スクリプトファイルの1行目から // 名前と次のパラメータ（ある場合）を入力して
// これらのパラメータとスクリプトファイル（実行ファイル）名を、インタープリタの
// コマンドライン引数スペース。
// その前に、もちろん、オリジナルのコマンドラインパラメーターと環境を
// 関数で指定された文字列を128KBのスペースに格納し、コマンドラインのパラメータを
ここで確立された//は、彼らの前に置かれます（逆になっているので）。最後に、このように
// カーネルがスクリプトファイルのインタプリタを実行します。そして、このようなパラメータを設定した後
スクリプトのファイル名として // を指定すると、インタプリタのi-nodeが取り出され、229行目にジャンプします。
// インタプリタを実行するために 実行されたコード229行目にジャンプする必要があるので

// 次のスクリプト処理コードの実行を禁止するフラグsh_bangを設定する。
スクリプトファイルを確認して処理した後に、再び // を表示します。以下のコードでは、このフラグ
//は、実行ファイルのコマンドライン引数を設定したことを示すためにも使われます。

```
        if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) {
```

```
246
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
```

// From here on, we extract the interpreter name and its parameters from the script file, and
// put the interpreter name, its parameters, and script file name into the environment parameter
// block. First copy the string after the character '#!' on the first line of the script file
// into the buffer buf, which contains the script interpreter name (for example /bin/sh), and
// possibly several parameters of the interpreter. Then process the contents of buf: replace
// the first newline with NULL and remove the space tab.

```
260                 strncpy(buf, bh->b_data+2, 127);
261                 brelse(bh);                     // release the buffer block.
262                 iput(inode);                    // put back the i-node of the script file.
263                 buf[127] = '\0';
264                 if (cp = strchr(buf, '\n')) {
265                         *cp = '\0';             // delete the space, tabs.
266                         for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
267                 }
268                 if (!cp || *cp == '\0') {       // if the line contains nothing, error!
269                         retval = -ENOEXEC;      /* No interpreter name found */
270                         goto exec_error1;
271                 }
```

// この時点で、スクリプトの名前で始まるコンテンツ（文字列）の行を取得します。
// インタープリターです。以下にその行を分析します。まず、最初の文字列を取得しますが、これは
// インタープリタ名を指定し、その時点でi_nameがその名前を指します。の後に文字がある場合は

```
// インタープリタ名ではなく、引数の文字列にして、i_argがその文字列を指すようにします。
                interp = i_name = cp;

                i_arg = 0;
                for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
                        if (*cp == '/')
                                i_name = cp+1;
                }
                if (*cp) {
                        *cp++ = '\0';           // add a NULL to the end of interpreter name.
                        i_arg = cp;             // i_arg points to the arguments.
                }
                /*
                * OK, we've parsed out the interpreter name and
                 *(optional) argument.
```

```
285                     */
```

// ここで、解析されたインタープリタ名i_name、そのパラメータi_arg、スクリプトファイルを置く必要があります。
// の名前をパラメータとしてインタープリタの環境とパラメータブロックに入力します。しかし、その前に必要なのは
// 関数が提供する元の引数や環境文字列の一部を最初に置くようにしています。
//で、最初に解析された行の内容をここに入れます。例えば、コマンドラインが
//が "example.sh -arg1 -arg2"、つまり実行ファイルがスクリプトファイルであることを示しています。もし、最初の行が
// の内容が "#!/bin/bash -iarg1 -iarg2 "であれば、最初にパラメータを配置した後に
// ここでは、新しいコマンドラインは次のようになります。
//   "bash -iarg1 -iarg2 example.sh -arg1 -arg2"
// ここでは、sh_bangフラグを設定し、オリジナルのパラメータや環境文字列を
関数のパラメータで提供された // を空間に入れる。環境文字列の数と
// 引数はそれぞれ envc と argc-1 です。コピーされない元の引数の一つ
//は元の実行ファイル名で、ここではスクリプトファイルの名前になっていて、次のようになります。
// は、以下のように処理されます。
// ここに注目してください! ポインタpが徐々に小さいアドレスに向かっていくのは
// コピー情報が増えるので、2つのコピー文字列関数が実行された後は
// 環境文字列ブロックは、プログラムのコマンドライン引数文字列ブロックの上に位置します。
//、pはプログラムの引数文字列の最初を指します。さらに、最後のパラメータである

copy_strings()の//(0)は、パラメータの文字列がユーザ空間にあることを示しています。

```
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                  */
```

// 次に、スクリプトのファイル名、インタープリタのパラメータ、インタープリタのファイル名を逆コピーする
// を引数と環境空間に配置します。エラーが発生した場合は、エラーコードが設定され、ジャンプ

//をexec_error1に変更します。また、この関数パラメータで指定されたスクリプトファイル名は
// はユーザースペースにあり、copy_strings() に与えられたスクリプトファイル名へのポインタはカーネルにあります。
// スペースが必要なので、このコピー文字列関数の最後のパラメータ（文字列ソースフラグ）は
// be set to 2, as shown in lines 301 and 304 below.

```
                p = copy_strings(1, &filename, page, p, 1);
```

```

		argc++;
		if (i_arg) {		// copy multiple arguments of the interpreter.
			p = copy_strings(1, &i_arg, page, p, 2);
			argc++;
		}
		p = copy_strings(1, &i_name, page, p, 2);
		argc++;
		if (!p) {
			retval = -ENOMEM;
			goto exec_error1;
		}
		/*
		 * OK, now restart the process with the interpreter's inode.
		 */
```

// 最後にインタープリタのi-nodeポインタを取得し、229行目にジャンプして実行します。
// インタープリターです。インタープリタのi-nodeを取得するためには、namei()を使う必要があります。
// 関数ですが、関数で使用するパラメータ（ファイル名）はユーザーデータから取得します。
// の空間、つまり、セグメントレジスタFSが指す空間からです。そのため
// namei()関数では、FSが一時的にカーネルのデータ空間を指すようにしておく必要があります。
// この関数は、カーネル空間からインタープリターの名前を取得し、デフォルトの設定を復元することができます。
// 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         }
```

// この時点で、バッファブロック内の実行ファイルヘッダー構造データがコピーされている
//を元にしています。そこで、まずバッファブロックを解放し、実行ヘッダー情報のチェックを開始します
// ex.では このカーネルでは、ZMAGICの実行ファイル形式のみをサポートしており、実行の
// ファイルコードは論理アドレス0から実行されるので、コードやデータを含む実行ファイルは
// 再配置情報はサポートされていません。もちろん、実行ファイルのサイズが大きすぎる場合は
// または実行ファイルが不完全であれば、実行することはできません。したがって、プログラムは
以下のような場合には // が実行されます。実行ファイルが実行可能ファイルでない場合
// (ZMAGIC)、またはコードとデータの再配置部分が0になっていない、または（コード部＋データ
// + 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         }
```

// さらに、実行ファイルの先頭のコードが境界に位置していない場合は
ページ(1024バイト)の//を超えてしまうと、実行できません。デマンドページング方式では
// 実行ファイルの内容がページ単位で読み込まれること、コードやデータが

```
//      if (N_TXTOFF(ex) != BLOCK_SIZE) {
```

//実行可能なファイルイメージは、ページ境界で開始する必要があります。

```
330
331                 printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
332                 retval = -ENOEXEC;
333                 goto exec_error2;
334         }
```

// sh_bangフラグが設定されていない場合は、指定された数のコマンドライン引数をコピーして
// 環境文字列をパラメータおよび環境空間に配置します。sh_bang フラグがすでに
// がセットされていると、スクリプト・インタープリターが実行されることを意味し、環境変数 page
// がコピーされているので、再度コピーする必要はありません。同様に、sh_bang が設定されておらず、設定が必要な場合は
// コピーされた後、ポインタpはコピー情報として小さなアドレスに向かって徐々に移動する

// が増加します。したがって、2つのコピー文字列関数が実行された後、環境文字列の
//ブロックはプログラムの引数文字列ブロックの上に位置し、pは最初の引数を指しています。
// プログラムの文字列です。実際には、pは128KBのパラメータと環境でのオフセット値

```
// 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 */
    // 前のセクションでは、実行するためのコマンドライン引数と環境変数を設定しました。
    関数のパラメーターから得られる情報に従って、 // 実行ファイルを作成しますが
    // そのための実質的な作業を行っていない、つまり初期化を行っていない。
    プロセスのタスク構造の//、ページテーブルの作成、などなど。では、仕事をしてみましょう。以来
    // 実行ファイルは、現在のプロセスの「ボディ」を直接使用する、つまり、現在の
    // プロセスがファイルを実行するプロセスに変換されるので、まず、リリース
    // 指定されたオープンを閉じるなど、現在のプロセスが占有しているいくつかのシステムリソース
    // ファイルを作成し、占有していたページテーブルやメモリページなどを解放します。そして、実行内容に応じて
    // ファイルのヘッダー構造、ローカルディスクリプターテーブルのディスクリプターの内容 LDTが使用される
    // 現在のプロセスが変更されると、コード・セグメントとデータ・セグメントの長さの制限が変更されます。
    // ディスクリプターが再設定され、取得したe_uidとe_gidを使って関連フィールドが設定される
    //をプロセスタスクに入れます。最後に、これを実行するプログラムのリターンアドレスeip[]を
    // システムコールは、実行ファイルのコードの先頭にポイントされます。このようにすると
    // システムコールが終了して戻ると、新しい実行ファイルのコードを実行します。
    // 新しい実行ファイルのコードとデータはまだメモリにロードされていませんが、注意してください。
    ファイル、パラメータ、環境ブロックから // get_free_page() を使用した
    // copy_strings()でデータを格納する物理メモリページを取得し、 put_page()を使って
    // change_ldt()でプロセスロジック空間の最後にデータを格納します。また、イン
    // create_tables()では、引数や環境を格納する際にもページフォルト例外が発生します。
    メモリマネージャが物理的なメモリページもマッピングするように，ユーザスタック上の // ポインタテーブル
    // ユーザーのスタックスペースに
    //
    // ここではまず、プロセスの元の実行プログラムのi-nodeを戻して
    // process executable fieldは、新しい実行ファイルのi-nodeを指します。その後、すべてのシグナルをリセット
    // 元のプロセスの処理ハンドルですが、SIG_IGNのためにリセットする必要はありません。
    // ハンドルになります。その後、close_on_execビットマップフラグに従って、指定されたオープンファイルを閉じて
345 // フラグをリセットします。
346         if (current->executable)
347                 iput(current->executable);
348         current->executable = inode;
349         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;
```

// その後、ベースアドレスと現在のプロセスで指定された長さの制限に従って
// 物理的なメモリページと、ページテーブルに対応して指定されたページテーブル自体が
オリジナルプログラムのコードとデータセグメントへの // を公開します。この瞬間、新しい
// 実行ファイルが主記憶領域のどのページも占有していないので、ページフォールトのアボート
// は、プロセッサが新しい実行ファイルコードを実際に実行するときに発生します。メモリ管理は
// その後、プログラムはページフォールト処理を行い、メモリページの申請と関連ページの設定を行います。
// 新しい実行ファイルのためのテーブルエントリーを読み、関連する実行ファイルページを
// メモリを使用しています。また、「最後のタスクがコプロセッサを使用した」が現在のプロセスを指している場合は
// NULL に設定され， used math フラグがリセットされます.

```
        free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
        if (last_task_used_math == current)
                last_task_used_math = NULL;
        current->used_math = 0;
```

// そして、LDTのディスクリプターベースアドレスとセグメント長を以下のように変更します。
// 新しい実行ヘッダー構造体のコード長フィールドa_textを、128KBの
データセグメントの最後に、 // 引数と環境スペースのページがあります。を実行した後
// 次の文では、pをデータの先頭からのオフセットに変更しています。
//のセグメントではなく、引数と環境空間のデータの先頭を指しています。
// つまり，pはスタックポインタの値に変換されています。次に，内部関数
// スタックに環境変数とパラメータ変数のポインタテーブルを作成する create_tables()
// プログラムのmain()の空間をパラメータとして、スタックポインタを返します。

```
        p += change_ldt(ex.a_text,page);
        p -= LIBRARY_SIZE + MAX_ARG_PAGES*PAGE_SIZE;
        p = (unsigned long) create_tables((char *)p,argc,envc);
```

// そして、プロセスフィールドの値を新しい実行ファイルの情報になるように修正します。
// つまり、プロセスタスク構造コードテールフィールドend_codeは、コードセグメント
// 実行ファイルの長さa_text；データテールフィールドend_dataはコードセグメントに等しい
// 実行ファイルの長さにデータセグメントの長さ（a_data + a_text）を加えたもの、およびプロセスの
// ヒープのエンドフィールド brk = a_text + a_data + a_bss. の終了位置を示すのにbrkが使われます。
// プロセスの現在のデータセグメント（未初期化部分を含む）、およびカーネルの
// プロセスにメモリを割り当てる際に、割り当ての開始位置を指定します。
// そして、プロセスのスタックスタートフィールドをスタックポインタがあるページに設定して
// プロセスの有効なユーザーIDと有効なグループIDを再設定します。

```
        current->brk = ex.a_bss +
                (current->end_data = ex.a_data +
                (current->end_code = ex.a_text));
        current->start_stack = p & 0xfffff000;
        current->suid = current->euid = e_uid;
        current->sgid = current->egid = e_gid;
```

// 最終的には、システムコール割り込みを最初に呼び出したプログラムポインタが置き換えられる
// 新しい実行ファイルのエントリーポイントを指すように、スタック上のコードポインターと一緒に

// スタックポインタが新しい実行ファイルのスタックポインタに置き換えられます。それ以降は
// return命令は、スタック上のデータをポップアップし、CPUが
// 新しい実行ファイルなので、最初にシステムを呼び出したプログラムには戻りません。
// 割り込みます。

```
        eip[0] = ex.a_entry;            /* eip, magic happens :-) */
        eip[3] = p;                     /* stack pointer */
        return 0;
exec_error2:
        iput(inode);                      // put back the i-node.
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.

```
7   // ファイルのステータス構造。すべてのフィールドは、ファイルのi-node構造から利用できます。
    6 struct stat {
8           dev_t   st_dev;     // device number that contains the file.
9           ino_t   st_ino;     // file i-node number.
10          umode_t st_mode;    // file type and mode.
11          nlink_t st_nlink;   // number of links to the file.
12          uid_t   st_uid;     // user identification number of the file.
13          gid_t   st_gid;     // group number of the file.
14          dev_t   st_rdev;    // device number (if the file is a char or block device file).
15          off_t   st_size;    // file size (bytes) if the file is a regular file.
16          time_t  st_atime;   // last access time.
17          time_t  st_mtime;   // last modified time.
18          time_t  st_ctime;   // i-node last changed time.
18 };
```

12.16.2 Code annotation

プログラム 12-15 linux/fs/stat.c

```
1 /*
2  *  linux/fs/stat.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
```

// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。

// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。

// and constants.

// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する（file, buffer_head,

// m_inode, etc.).

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ

```
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
//      used functions of the kernel.
 // <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
//      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
//// i-nodeを使ってファイルのステータス情報を取得します。
// パラメータのinodeはファイルのi-node、statbufはstatファイルの状態構造
// 取得した状態の情報を格納するために使用される、ユーザーデータ空間内のポインタです。
17  15 static void cp_stat(struct m_inode * inode, struct stat * statbuf) 16 {。
18          struct stat tmp;
19          int i;
19
// まず、データを格納するのに十分なメモリ領域を確認（または確保）し、一時的に
// 対応するファイルのi-nodeの情報をtmpに送る。最後に，これらのステータス情報
20  // がユーザーバッファにコピーされます。
21          verify_area(statbuf,sizeof (struct stat));
22          tmp.st_dev = inode->i_dev;        // device number that contains the file.
23          tmp.st_ino = inode->i_num;        // file i-node number.
24          tmp.st_mode = inode->i_mode;      // file type and mode.
25          tmp.st_nlink = inode->i_nlinks;   // number of links to the file.
26          tmp.st_uid = inode->i_uid;        // user id.
27          tmp.st_gid = inode->i_gid;        // group id.
28          tmp.st_rdev = inode->i_zone[0];   // device no.(if the file is a char/block file).
29          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 の時間になっています。
32          for (i=0 ; i<sizeof (tmp) ; i++)
33                  put_fs_byte(((char *) &tmp)[i],i + (char *) statbuf);
34 }
35
//// ファイル・ステータス・システム・コール機能。
// 与えられたファイル名に応じて、関連するファイルのステータス情報を取得する。
// パラメータ statbuf は、状態情報を格納するためのユーザバッファポインタです。
// 成功した場合は0を、エラーが発生した場合はエラーコードを返します。
36 int sys_stat(char * filename, struct stat * statbuf) 37 {。
38          struct m_inode * inode;
39
```

```
//まず、ファイル名から対応するi-nodeを見つけ、ファイルの状態をコピーします。
//i-nodeの情報をユーザーバッファに入れ、最後にi-nodeを戻します。
        if (!(inode=namei(filename)))
                return -ENOENT;
```

```
42          cp_stat(inode,statbuf);
43          iput(inode);
44          return 0;
45 }
46
```

//// シンボリックリンクファイルの状態をシステムコールする機能です。
// 与えられたファイル名に応じて、関連するファイルのステータス情報を取得します。もし、シンボリックな
ファイルパス名に // リンクファイル名が含まれている場合、シンボルファイル自体のステータス情報は
//が取られ、リンクが踏まれていない。
// パラメータ statbuf は、ファイルのステータス情報を格納するためのユーザバッファポインタです。
// 成功した場合は0を、エラーが発生した場合はエラーコードを返します。

```
47 int sys_lstat(char * filename, struct stat * statbuf) 48 {。
49          struct m_inode * inode;
50
```

// まず、ファイル名に応じて対応するi-nodeを探します。それがシンボリックリンクの場合
//ファイルがある場合は、リンクをたどりません。その後、i-nodeのファイルステータス情報がコピーされます。

```
51
```

// をユーザーバッファに入れて、i-nodeを戻します。

```
52          if (!(inode = lnamei(filename)))     // get the i-node without following the link.
53                  return -ENOENT;
54          cp_stat(inode,statbuf);
55          iput(inode);
56          return 0;
56 }
57
```

//// ファイルハンドルを使って、ファイルの状態を取得します。
// 与えられたファイルハンドルに基づいて、ファイルのステータス情報を取得します。
// パラメータfdは指定されたファイルのハンドル（記述子）、statbufは
// ステータス情報を保持するユーザバッファポインタ。
// 成功した場合は0を、エラーが発生した場合はエラーコードを返します。

```
60
58 int sys_fstat(unsigned int fd, struct stat * statbuf) 59 {...
61          struct file * f;
62          struct m_inode * inode;
62
```

// まず、ファイルハンドルに対応するファイル構造を取得して、i-nodeの
//ファイルを表示します。その後、i-nodeのファイル・ステータス情報がユーザー・バッファにコピーされます。
// ファイルハンドルの値が、以下の方法で開くことのできるファイルの最大数よりも大きい場合は
// プログラムが NR_OPEN であるか，ハンドルのファイル構造ポインタが NULL であるか，i-node フィールドが

```
63
```

対応するファイル構造体の // が NULL の場合はエラーとなり，エラーコードが返されます。

```
64          if (fd >= NR_OPEN || !(f=current->filp[fd]) || !(inode=f->f_inode))
65                  return -EBADF;
66          cp_stat(inode,statbuf);
67          return 0;
67 }
```

68

```
//// Read symbolic link file system-call.
// この関数は、シンボリックリンクファイルの内容（つまり、パス名の文字列である
シンボリックリンクで指し示されたファイルの//）をユーザーバッファに配置します。もしバッファが
// が小さすぎる場合は、シンボリックリンクの内容が切り捨てられます。
// パラメータ： path はシンボリックリンクファイルのパス名、 buf はユーザーバッファ、 bufsiz は
// バッファの長さ。
// 戻り値：成功した場合はバッファ内の文字数、失敗した場合はエラーコード。
```

```
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
```

// まず、関数のパラメータの有効性を確認・検証し、調整します。ユーザーが
// バッファサイズ bufsi は 1 から 1023 の間でなければなりません。次に、シンボリックリンクファイルのi-nodeを取得します。

```
//      if (bufsiz <= 0)
```

//とし、ファイルのデータ内容の最初のブロックを読みます。そして、i-nodeを戻します。

```
76
77                 return -EBADF;
78         if (bufsiz > 1023)
79                 bufsiz = 1023;
80         verify_area(buf,bufsiz);
81         if (!(inode = lnamei(path)))
82                 return -ENOENT;
83         if (inode->i_zone[0])
```

```
                bh = bread(inode->i_dev, inode->i_zone[0]);
        else
                bh = NULL;
        iput(inode);
```

// ファイルデータの内容が正常に読み込まれた場合、最も多いbufsiz文字がコピーされる

// ファイルをユーザバッファにコピーする際，NULL文字はコピーされません。最後に，バッファを解放します.

```
        if (!bh)

                return 0;
        i = 0;
        while (i<bufsiz && (c = bh->b_data[i])) {
                i++;
                put_fs_byte(c,buf++);
        }
        brelse(bh);
        return i;
}
```

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.

関数 dup() および dup2() が返す新しいファイルハンドルは、ファイルテーブルのエントリを

元のハンドルがコピーまたは複製されます。例えば、プロセスが他のファイルを開いていないときに、dup()またはdup2()関数を使用し、新しいハンドルを3に指定した場合、関数実行後のファイルハンドルは図12-35のようになります。

図12-35 dup(1)，dup2(1,3)関数実行後のファイル構造

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().

・AT&TのSystem IIIで採用されたfcntl()関数は、主に開いているファイルのプロパティを変更するために使用される。この関数は、パラメータに制御コマンドcmdを持つ関数で、4つの機能を統合しています。

・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 file handle flag close_on_exec. To set the flag, the third parameter 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

プログラム 12-16 linux/fs/fcntl.c

```
1 /*
```

```
2  * linux/fs/fcntl.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
```

// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。

// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。

// <sys/types.h> 型のヘッダファイル。基本的なシステムデータ型が定義されています。

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ

```
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
```

// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。

```
//      used functions of the kernel.
```

// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。

```
//      segment register operations.
```

// <fcntl.h> ファイル制御のヘッダファイルです。演算制御定数記号の定義は

```
//      used for the file and its descriptors.
```

// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む 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
```

//// ファイルハンドル（ファイル記述子）を複製します。

// パラメータfdは複製するファイルハンドル、argは最小数を指定します。

// 新しいファイルハンドルの

// 新しいファイルハンドルまたはエラーコードを返します。

18 static int dupfd(unsigned int fd, unsigned int arg) 19 {。

// この関数はまず，関数パラメータの有効性をチェックします。もし，ファイルハンドルの値が

// がプログラムのオープンファイルの最大数NR_OPENよりも大きい場合、またはファイル構造がハンドルの // が存在しない場合は、エラーコードが返されます；指定された新しいハンドル値が

// argが開いているファイルの最大数よりも大きい場合は、エラーコードも返されます。備考

20 // ファイルハンドルが，実際にはファイル構造ポインタの配列項目のインデックス番号であることを示します。

```
21          if (fd >= NR_OPEN || !current->filp[fd])
22                  return -EBADF;
23          if (arg >= NR_OPEN)
```

```
24                  return -EINVAL;
// 次に、プロセスのファイル構造ポインタ配列の中から、インデックス番号を持つエントリを探します。
// argと同じかそれ以上で、まだ使用されていない。見つかった新しいハンドルが
//      while (arg < NR_OPEN)
//オープンファイルの最大数NR_OPEN(つまり、空きアイテムがない)の場合は、エラーコードが返されます。
24
25                  if (current->filp[arg])
26                          arg++;
27                  else
28                          break;
```

```
29          if (arg >= NR_OPEN)
30                  return -EMFILE;
    // そうでなければ，見つかったアイドルアイテム（ハンドル）について，ハンドルの対応するビットがリセットされる
    // をフラグ close_on_exec で指定しています。つまり、exec()クラスの関数を実行すると、作成されたハンドルは
    dup()を使用した // は閉じられません。そして，ファイル構造体のポインタは，ポインタ
    // を元のハンドルfdに戻し、ファイル参照数を1つ増やします。最後に
31  // 新しいファイルハンドルargが返されます。
32          current->close_on_exec &= ~(1<<arg);
33          (current->filp[arg] = current->filp[fd])->f_count++;
34          return arg;
34 }
35
    //// ファイルハンドルsystem-call2を複製します。
    // 与えられたファイルハンドルoldfdを複製すると、新しいファイルハンドルの値はnewfdと同じになります。もしnewfd
    // がすでに開いている場合は、先に閉じてください。
    // パラメータ：oldfd -- 元のファイルハンドル、newfd -- 新しいファイルハンドル。
    // 新しいファイルハンドルを返します。
38  36 int sys_dup2(unsigned int oldfd, unsigned int newfd) 37 {。
39          sys_close(newfd);               // closed the newfd first if already opened.
40          return dupfd(oldfd,newfd);      // duplicate and return the new handle.
40 }
41
    //// ファイルハンドルのsystem-callを複製する。
    // 与えられたファイルハンドルoldfdを複製し、新しいハンドルの値は現在の最小値になります。
    // 未使用のハンドル値です。
        // Parameters: fildes -- 複製されるファイルハンドル。
// 新しいファイルハンドルを返します。
42 int sys_dup(unsigned int fildes) 43 {
44          return dupfd(fildes,0);
45 }
46
    //// ファイル・コントロール・システム・コールです。
    // パラメータfdはファイルハンドル、cmdは制御コマンド（include/fcntl.h, lines
    // 23-30); argは、コマンドによって意味が異なります。重複ハンドルコマンドの場合
    // F_DUPFD, arg は新しいファイルハンドルに取ることのできる最小の値で，セットファイルの場合は
    // 操作とアクセスフラグのコマンド F_SETFL, arg が新しいファイル操作とアクセスモードになります。
    // ファイルロックコマンド F_GETLK, F_SETLK, F_SETLKW では，arg はフロックへのポインタ
    // 構造体になっています。ただし，このカーネルでは，ファイルロック機能は実装されていない。
    // 返します。エラーが発生した場合、すべての操作は-1を返します。成功した場合、F_DUPFDは、新しい
    // F_GETFD はファイルハンドルの close_on_exec フラグを返し，F_GETFL はファイルハンドルの
    // ファイル操作とアクセスのフラグ。
```

```
47 int sys_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg) 48 {。
49         struct file * filp;
50
   // まず、与えられたファイルハンドルの有効性をチェックし、それに従って処理を行います。
   // を別のコマンドに変更する cmd. ファイルハンドルの値が最大数よりも大きい場合は
   // プロセスでファイルを開く NR_OPEN、またはハンドルのファイル構造ポインタが NULL の場合。
51 // エラーコードが返されます。
52         if (fd >= NR_OPEN || !(filp = current->filp[fd]))
53                 return -EBADF;
```

```
54          switch (cmd) {
55                  case F_DUPFD:           // duplicate file handle.
56                          return dupfd(fd,arg);
57                  case F_GETFD:           // get the close_on_exec flag bit for the file.
58                          return (current->close_on_exec>>fd)&1;
59                  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

プログラム 12-17 linux/fs/ioctl.c

```
1 /*
2  *  linux/fs/ioctl.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
```

// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。

// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。

// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。

//　　and constants.

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ

//　　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, 118行目. 15
```

// 入出力制御(ioctl)機能のポインタタイプを定義する。

```
16 typedef int (*ioctl_ptr)(int dev,int cmd,int arg); 17
```

// システム内のデバイスタイプの数を取得します。

```
18 #define NRDEVS ((sizeof (ioctl_table))/(sizeof (ioctl_ptr))) 19
```

// Ioctl操作関数ポインタテーブル。

```
20          NULL,           /* nodev */
static ioctl_ptr ioctl_table[]={{}}。
21
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
```

//// 入力と出力の制御システム-コールです。

// この関数はまず，パラメータで与えられたファイルディスクリプタが有効かどうかを判断します。

// そして、i-nodeのファイルモードに応じてファイルタイプをチェックし、関連する

// ファイルの種類に応じたioの処理機能。

// パラメータ：fd - ファイル記述子（ハンドル）、cmd - コマンドコード、arg - パラメータ。

// 戻り値：成功した場合は0、そうでない場合はエラーコードを返します。

```
33  31 int sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg) 32 {。
34          struct file * filp;
35          int dev,mode;
35
```

// まず、与えられたファイルハンドルの有効性を判断します。ファイルハンドルが

```
開放可能なファイルの // の数が少ないか，対応するハンドルのファイル構造ポインタが NULL の場合は
// のエラーコードが返されます。ファイル構造がパイプiノードに対応している場合には
// プロセスが権利を持っているかどうかに従って、パイプIOを実行するかどうかを判断するために
// を使ってパイプを操作します。実行の許可があればPipe_ioctl()が呼ばれ、なければ
// 無効なファイルのエラーコードが返されます。
        if (fd >= NR_OPEN || !(filp = current->filp[fd]))
                return -EBADF;
        if (filp->f_inode->i_pipe)
                return (filp->f_mode&1)?pipe_ioctl(filp->f_inode,cmd,arg):-EBADF;
```

```
// その他のタイプのファイルの場合は、対応するファイルのモードを取り、タイプを決定する
ファイルの // を適宜変更する。ファイルがキャラクタデバイスファイルでもブロックデバイスファイル
でもない場合は
// ファイルの場合、エラーコードが返されます。キャラクタまたはブロックデバイスファイルの場合は、
デバイス番号
//は、ファイルのi-nodeから取得します。の数よりも大きい場合は、デバイス番号が
システム内の//デバイスには、エラー番号が返されます。
        mode=filp->f_inode->i_mode;
        if (!S_ISCHR(mode) && !S_ISBLK(mode))
                return -EINVAL;
        dev = filp->f_inode->i_zone[0];          // device no. for device type file.
        if (MAJOR(dev) >= NRDEVS)
                return -ENODEV;
// そして，IO制御テーブルioctl_tableにしたがって，ioctl関数ポインタの
// 対応するデバイスが見つかり、その関数が呼び出されます。もし、デバイスが
// 表の中で該当する関数を選択すると、エラーコードが返されます。
        if (!ioctl_table[MAJOR(dev)])
                return -ENOTTY;
        return ioctl_table[MAJOR(dev)](dev,cmd,arg);
}

```

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.

この問題を解決する方法はたくさんあります。1つの方法は、同時にアクセスする必要のあるファイルディスクリプターごとにプロセスを設定することです。しかし、この方法では、これらのプロセス間の通信を調整する必要があり、この方法は複雑である。また、すべてのファイル記述子をノンブロッキング形式にして、各ファイル記述子に読み出し可能なデータがあるか、書き込み可能なデータがあるかをプログラム上で周期的に検出する方法がある。しかし，このループ検出法はプロセッサ時間を多く消費するため，マルチタスクOSではこの方法は推奨されない。3つ目の方法は、非同期I/O技術を使う方法です。原理的には、ディスクリプターにアクセスできるようになったら、カーネルがシグナルを使ってプロセスに通知するというものです。このような「通知」信号は各プロセスに1つしかないので、複数のファイルディスクリプターを使用する場合は、やはり各ファイルディスクリプターをノンブロッキング状態にして、その信号を受け取ったときに、どのディスクリプターが準

備できているかをテストする必要がある。

もうひとつの良い方法は、selcet.cプログラムのselect()(sys_select())関数を使ってこの状況を処理することです。select()関数は、もともとBSD 4.2オペレーティングシステムに搭載されていたもので、BSD Socketネットワークプログラミングをサポートするオペレーティングシステムで使用できます。この関数は主に、複数のファイル記述子（またはソケットハンドル）を同時に効率的にアクセスする必要がある状況に対処するために使用されます。この機能の主な動作原理は、ユーザーが提供した複数のファイル記述子をカーネルが同時に監視することである。ファイルディスクリプターの状態が変化していなければ，呼び出したプロセスをスリープ状態にし，ディスクリプターの1つがアクセス可能な状態であれば，プロセスに戻り，プロセスにどの

記述子または記述子の準備ができています。

select()関数のプロトタイプは、include/unistd.hファイルの277行目で以下のように定義されています。

```
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).

ファイルディスクリプターセット型fd_setは，include/sys/types.hファイルで定義されており，符号なしのロングワードとして定義されている。その各ビットがファイル記述子を表し、図12-36の上半分に示すように、ロングワード内のビットのオフセット位置値がファイル記述子の値となる。

図12-36 ファイルディスクリプターセットは1ビットにつき1つのディスクリプターを表す

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.
```

記述子セット変数を宣言した後、ユーザプログラムはまずFD_ZERO()でクリアし、次にFD_SET()またはFD_CLR()を使用して、指定された記述子に対応するビットをセット/リセットする必要がある。FD_ISSETは、select()が戻ってきたときに、ディスクリプターセットの指定されたビットがまだセットされているかどうかをテストするために使用される。select()が戻ってきたとき、3つのディスクリプターセットの中でまだセットされているビットは、対応するファイルディスクリプ

ターの準備ができていることを示している（読み取り、書き込み、または例外）。なお，これらのマクロでは，第2引数に記述子セットのポインタを用いる必要がある。

select()関数の最後のパラメータであるtimeoutは、任意のディスクリプターの準備が整うまで、プロセスがselect()を待つ最大時間を指定するために使用されます。型の構造体へのポインタである。

timeval（include/sys/time.hファイルで定義されています）を以下のように使用します。

```
構造体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.

タイムアウトポインタがNULLではなく、構造体の両フィールドの値が0の場合は、待ち時間がないことを意味する。この時点で、select()関数は、指定されたすべてのディスクリプターの状態をテストし、すぐに戻ることができる。2つの時間フィールドの少なくとも一方の値が0でない場合、select()関数はしばらく待ってからリターンする。待機期間中にディスクリプターの準備が整った場合は、直接リターンし、この時、2つのタイムフィールド値は残りの待機時間値を示すように修正される。設定された時間内にディスクリプターの準備ができていない場合、select()は0を返す。また、待ち時間中にシグナルによって中断されることがあり、-1を返す。

一般に、select()が-1を返す場合はエラーを示し、select()が0を返す場合は、指定された条件で準備ができているディスクリプターがないことを示し、select()が正の値を返す場合は、ディスクリプターセットの中でアクセスの準備ができているファイルディスクリプターの数を示している。このとき，3つのディスクリプターセットのうち，まだセットされているビットに対応するディスクリプターがレディディスクリプターとなる。

Linux 0.12カーネルでは、システムコールには3つまでのパラメータしか用意されておらず、select()には5つのパラメータがあるため、ユーザプログラムがselect()関数を呼び出すと、ライブラリファイル(例えばlibc.a)内のselect()は、最初のパラメータのアドレスを、カーネル内のシステムコールsys_select()へのポインタとして渡します。システムコールは、まず "バッファ "の中のパラメータを壊し、 do_select()関数を呼び出してパラメータを処理します。そして、do_select()が戻ると、その結果がユーザーデータの「バッファ」に書き込まれます。以下は、Linux 0.1xシステムのlibcライブラリにおけるselect()関数のソースコード実装である。

```
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 {。
    // まず、戻り値の変数resを定義し、レジスタ変数fooebxを定義します。
    // を最初のパラメータへのポインタとして使用します。その後、システムコールのインラインアセンブリコードを使用して、セット
07  // eax = システムコールの関数番号を選択し、ebx は最初のパラメータの nd ポインタです。
08      long  res;
09      register long_ fooebx _ asm_ ("bx") = (long) &nd;
09      _asm volatile ("int $0x80"
10          : "=a" (__res)
11          : "0" (_NR_select),"r" (_fooebx));
    // 最後に、戻り値が0以上の場合は値を返し、そうでない場合は
    // グローバルなエラー番号変数 errno が設定され、-1 が返されます。
12      if (_res >= 0)
13          return (int)  res;
14      errno = -  res;
15      return -1;
```

16 }

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.

sys_select()関数のコードは、select()関数の前後のパラメータのコピーや変換を主に担当しており、select()操作の主な作業はdo_select()関数で行われています。do_select()は、まず、ファイルディスクリプターセットの各ディスクリプターの有効性をチェックし、該当するディスクリプターセットの関数check_XX()を呼び出して各ディスクリプターをチェックし、さらにディスクリプターセットで現在準備ができているディスクリプターの数をカウントする。いずれかの記述子の準備ができていれば、この関数は直ちに戻り、そうでなければ、プロセスはadd_wait()関数を呼び出して、現在のタスクを対応する待ち行列に挿入し、do_select()関数でスリープ状態に入る。タイムアウトが経過した後もプロセスの実行が継続している場合や、ディスクリプターが配置されている待機キュー上のプロセスが覚醒したために、プロセスはディスクリプターの準備ができているかどうかを再度確認する。do_select()関数は、free_wait()関数を使用して、すでに待ち行列に入っている待ちタスク（もしあれば）を目覚めさせてから、繰り返しチェック操作を実行します。

select.cプログラムでは、プログラムの37行目から45行目と下記の図12-37に示すように、ディスクリプターの待機処理中に待機テーブルwait_tableを使用しています。select_table型のwait_tableには、有効な項目数フィールドnrと配列entry[NR_OPEN * 3]があり、各配列項目はwait_entry構造体となっています。wait_tableの有効項目数フィールドnrには、ディスクリプターセット内のディスクリプターが関連する待機キューで待機しているwait_entryエントリの数が記録されています。wait_entry構造体には2つのフィールドがあり、wait_addressポインタフィールドは、現在処理中の記述子に対応するタスクの待ち行列ヘッダを指すのに使用され、old_taskフィールドは、待ち行列ヘッダポインタが元々指していた待ちタスクを指すのに使用される。

図12-37 ウェイトテーブル構造図

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

を、キューのヘッドポインタが元々指していたタスクに変更します。次に、待ち行列の先頭ポインタが現在のタスクを指すようにします。最後に、waitテーブルの有効なアイテムカウント値nrを1つインクリメントします。

例えば，リードバッファキューが空で，端末ttyの文字入力を待っているディスクリプタに対して，対応する端末のリードバッファキュー「セカンダリ」には，バッファキュー内の読み取り可能な文字を待つタスク待ち行列ヘッダポインタproc_listが用意されている（ファイルinclude/linux/tty.hの26行目のttyキュー構造を参照）。バッファ・セカンダリに読める文字がない場合、select.cプログラムは、add_wait()関数を使って現在のタスクを待ち行列に追加します。wait_entryフィールドのwait_address = proc_listを作成し、フィールドold_taskにproc_listが元々指していたタスクを指定します。もし、proc_listが元々どのタスクも指していなかった場合、old_task=NULLとなります。そして、proc_listに現在のタスクを指させる。この処理を図12-38に示します。図中、(a)は元々のタスクがadd_wait()関数を呼び出す前にキューの先頭ポインタを待っている状態、(b)はadd_wait()を実行した後にエントリを待っている形を示しています。なお、図ではwaitテーブルのwait_entryエントリが1つだけ表示されています。

図12-38 ウェイトテーブルに1つのウェイトアイテムを追加する

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.

図12-39 待機中の新しいタスクが待機キューに挿入される

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.
待機中のリソースが利用可能になると、例えば、ttyのリードバッファキューの二次バッファに文字が入力された場合、待機中のヘッドポインタが指すタスクが起こされます。のです。

タスクは、そのtmpポインタが指すタスクをウェイクアップします。select()を実行しているタスクが起こされると、すぐにfree_wait()関数を実行します（コードの204行目参照）。タスクがウェイクアップし、待ち行列の先頭ポインタがこのタスクを指していれば（*wait_address == current）、free_wait()関数は、old_taskが指している後続のタスクを直ちにウェイクアップさせます。free_wait()の機能は、sleep_on()がタスクを起こすコードと全く同じであることがわかります。select関数を実行しているタスクが覚醒し、他のプロセスがsleep_on()関数を呼び出して待ち行列にスリープした場合、待ち行列の先頭ポインタが現在のプロセスを指していない(*wait_address != current)ので、まずこれらのタスクを覚醒させる必要があります。操作方法は、待ち行列の先頭が指しているタスクをレディ状態にする（state

= 0）に設定し、自身を非中断可能な待機状態にします。つまり、タスク自身は、これらの後続のキューイングされたタスクが実行を開始するのを待ってから、自分自身を目覚めさせるのです。そして、スケジューラーを再実行します。

2.19.2 また、Linuxカーネルに実装されているselect()は、実行中にtimeoutが指す構造体のフィールド値を、残りの待ち時間を反映して修正（デクリメント）しますが、他の多くのOSのselect()の実装ではこのような処理は行われないため、select()の実行中にtimeout構造体の値にアクセスするLinuxプログラムでは問題が発生することに注意してください。同様に、ループ内でタイムアウトを初期化せず、select()関数を何度も使用するプログラムは、Linuxシステムに移植すると問題が発生します。そのため、select()が戻ってきたとき、timeoutが指す構造体は初期化されていない状態であると考えるべきです。

12.19.3 Code annotation

プログラム 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
```

// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する（file, buffer_head,
//　　m_inode, etc.).
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
//　　used functions of the kernel.
// <linux/tty.h> ttyヘッダーファイルは、tty_io、シリアルのパラメータと定数を定義しています。
//　　communication.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//　　of the initial task 0, and some embedded assembly function macro statements about the
//　　descriptor parameter settings and acquisition.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されて

います。
//　　segment register operations.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
//　　descriptors/interrupt gates, etc. is defined.
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
//　　and constants.
// <sys/types.h> 型のヘッダファイル。基本的なシステムデータ型が定義されています。
// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <const.h> 定数ファイルでは，現在，i-nodeのi_modeフィールドのフラグのみを定義しています。
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <sys/time.h> timeval構造体とitimerval構造体が定義されています。
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナルの定義
//　　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 */
```

// 各プロセスはそれぞれEFLAGSフラグレジスタを持っていることに注意してください。

```
36
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
```

//// ウェイトテーブルにウェイトキューを追加することができます。

// not ready ディスクリプタの wait queue ポインタを wait_table に追加する。このとき，パラメータ

// *wait_address は，ディスクリプターに関連付けられたウェイトキューのヘッダーポインタです。例えば、以下のようになります。

// ttyのリードバッファキューの待ち行列の先頭ポインタはproc_listです。パラメータpは

// do_select()で定義された wait テーブル構造体へのポインタです。

```
47 static void add_wait(struct task_struct ** wait_address, select_table * p) 48 {...
49         int i;
50
```

// まず、ディスクリプターに対応する待ち行列があるかどうかをチェックし、ない場合はリターンします。その後、イン

// ウェイトテーブルでは、パラメータで指定されたキューポインタを検索して、それがすでに
// が待ち受けに設定されている。設定されていれば、コードはすぐに戻ります。このチェックは主に
パイプのファイルディスクリプターのための//。例えば、パイプが読み込みを待っている場合、そのパイプには

```
51      //すぐに書けるように
52              if (!wait_address)
53                      return;
53              for (i = 0 ; i < p->nr ; i++)
54                      if (p->entry[i].wait_address == wait_address)
55                              return;
        // そして、ディスクリプターの待ち行列のヘッダポインタをwait_tableに格納して
        // wait table entry の old_task フィールドは、wait queue header で指定されたタスクを指します。
        // ポインタ（ない場合はNULL）を指定して，待ち行列のヘッダに現在のタスクを指定させます．最後に
56      // ウェイトテーブルのアイテムカウント値nrが1だけ増加します。
57              p->entry[p->nr].wait_address = wait_address;
58              p->entry[p->nr].old_task = * wait_address;
59              *wait_address = current;
59              p->nr++;
60      }
61
        //// ウェイトテーブルの各ウェイトキューを解放します。
        // パラメータ p は wait table 構造体へのポインタです。この関数（204,207行目）は
        // do_select()関数で現在のプロセスが起こされたときに呼び出されて
        // 各待ち行列にある他のタスクを待ち行列にアップします。とほとんど同じです。
        // kernel/sched.cにあるsleep_on()関数の後半部分、その説明を参照。
64      62 static void free_wait(select_table * p) 63 {...
65              int i;
66              struct task_struct ** tpp;
66
        // ウェイトテーブルのエントリーのウェイトキューのヘッダー（トータルnrエントリー）が示すのは
        // 後から追加された他の待機タスクがある場合（例えば、他のプロセスがsleep_on()
        // 関数を使用して待機キューをスリープさせた場合）は、キューのヘッダーが現在の
        //の処理を行うためには、まずこれらのタスクを起こす必要があります。操作方法は、タスクに
        キューヘッダが指す // をレディ状態（state = 0）にし，自身を
        // 割り込み禁止の待機状態、つまり、これらの後続のキューイングされたタスクを待って
        // 起こされ、このタスクを起こすために再度実行します。そこで、スケジューラーを再実行します。
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                      }
        // ここでの実行は、現在のエントリーのウェイトキューのヘッダーフィールドwait_addressを示す
        待機テーブルの // は、現在のタスクを指します。これがNULLの場合は，現在のタスクが
        // スケジュールコードに問題があり、警告メッセージが表示されています。その後、待ち時間を
        // キューのヘッダは、先にキューに入ったタスクへのポインタです（76行目）。もし、ヘッドポインタが
        // この時点でNULLではなくタスクを指していれば、キューにはまだタスクが存在していることになります。
```

// (*tppは空ではない)なので、タスクはレディ状態に設定され、ウェイクアップします。最後に、タスクに
// ウェイトリストのアイテムカウントフィールドnrを0にして、ウェイトテーブルをクリアします。

```
75                  if (!*tpp)
76                          printk("free_wait: NULL");
77                  if (*tpp = p->entry[i].old_task)
78                          (**tpp).state = 0;
78          }
79          p->nr = 0;
```

```
80 }
81
    //// i-node に応じた tty を取得します。
    // ファイルがi-nodeに従った文字端末デバイスファイルであるかどうかを確認する。もしそうであれば
    // tty構造体のポインタを返し、そうでなければNULLを返します。
82 static struct tty_struct * get_tty(struct m_inode * inode) 83 {。
84          int major, minor;
85
    // キャラクターデバイスファイルでない場合はNULLを、メジャーデバイス番号が
86  // 5（コントロール端子）または4ではない。
87          if (!S_ISCHR(inode->i_mode))
88                  return NULL;
89          if ((major = MAJOR(inode->i_zone[0])) != 5 && major != 4)
90                  return NULL;
    // メジャーデバイス番号が5の場合、プロセスのttyフィールドがその端末デバイスとなる
    //番号、そうでない場合はキャラクターデバイス番号のマイナーデバイス番号と同じになります。
    // 端末のデバイス番号が0より小さい場合は、プロセスが制御されていないことを示す
    // 端末を使用していないのでNULLを返し、そうでない場合は
91  // 対応する tty 構造体のポインタ。
92          if (major == 5)
93                  minor = current->tty;
94          else
95                  minor = MINOR(inode->i_zone[0]);
96          if (minor < 0)
97                  return NULL;
98          return TTY_TABLE(minor);
97 }
98
100 99 /*
101  * The check_XX functions check out a file. We know it's either
102  * a pipe, a character device or a fifo (fifo's not implemented)
102 */
    //// リードインレディを確認する。
    // ファイル読み取り操作の準備ができているかどうか、つまり、端末の読み取りバッファのキューが
    // secondary に読み込むべき文字があるかどうか、パイプファイルが空でないかどうか。
        // パラメータwaitはwaitテーブルポインタ，inodeはファイルのi-nodeポインタです.
// 103 static int check_in(select_table * wait, struct m_inode * inode) 104 {...
続きを読む
105 struct tty_struct * tty; 106
    // まず、inodeに従って、get_tty()を呼び出し、ファイルがtty端末（文字）であるかどうかをチェックす
    る
    // デバイスファイルです。のセカンダリリードバッファのキューに文字があるかどうかをチェックしま
    す。
    // 端子がある場合は1を返します。この時点で secondary が空であれば、現在の
    // タスクは二次待機キューに追加されます proc_list と 0 を返します。パイプファイルの場合は
    // 現在のパイプに文字が入っているかどうかをチェックし、入っていれば1を返し、入っていなければ
```

```
(パイプが空)。
// 現在のタスクをパイプのi-nodeの待ち行列に追加し、0を返す。なお、この時の
// PIPE_EMPTY()マクロは、パイプの現在のヘッドポインタとテールポインタの位置を使って
// パイプが空の場合 パイプのi-nodeのi_zone[0]とi_zone[1]フィールドには、現在の
// パイプの先頭と末尾のポインター。
        if (tty = get_tty(inode))
                if (!EMPTY(tty->secondary))
```

```
110                     return 1;
111             else
112                     add_wait(&tty->secondary->proc_list, wait);
113     else if (inode->i_pipe)
114             if (!PIPE_EMPTY(*inode))
115                     return 1;
116             else
117                     add_wait(&inode->i_wait, wait);
118     return 0;
118 }
119
```

//// チェックライトアウトの準備ができました。

// ファイルの書き込み操作の準備ができているかどうか、つまり、空きのある場所があるかどうかを確認する

// 端末の書き込みバッファキューwrite_qに入っているか、パイプファイルがいっぱいになっていないか。

// パラメータwaitはwaitテーブルポインタ，inodeはファイルのi-nodeポインタです.

// 120 static int check_out(select_table * wait, struct m_inode * inode) 121 {...

```
122     struct tty_struct * tty;
123
```

// まず、i-nodeに従って、get_tty()を呼び出して、ファイルがtty端末であるかどうかをチェックします。

// （文字）デバイスファイルです。Yesの場合、書き込みバッファのキューに空きがあるかどうかをチェック write_q

// を書きます。空きスペースがない場合は1を返し、現在のタスクを

// write_qの待ち行列proc_listにして0を返す。パイプファイルであれば

// パイプラインに文字を書き込むための空き領域がある。ある場合は1を返し、ない場合は

// そうでない場合（パイプラインが満杯の場合）、現在のタスクはパイプラインの待ち行列に追加される

```
124     // i-nodeで0を返します。
125     if (tty = get_tty(inode))
126             if (!FULL(tty->write_q))
127                     return 1;
128             else
129                     add_wait(&tty->write_q->proc_list, wait);
130     else if (inode->i_pipe)
131             if (!PIPE_FULL(*inode))
132                     return 1;
133             else
134                     add_wait(&inode->i_wait, wait);
135     return 0;
135 }
136
```

//// 異常な状態を確認する。

// ファイルが異常な状態であるかどうかを確認します。ターミナルデバイスファイルの場合、カーネルは常に

// パイプファイルの場合、2つのパイプディスクリプターのうち、どちらか一方が閉じられていれば1を返します。

// 今回は、そうでなければ、現在のタスクをi-nodeの待ち行列に追加し、0を返します。

// パラメータwaitはwaitテーブルポインタ，inodeはファイルのi-nodeポインタです.

// 例外条件が発生した場合は1を、そうでない場合は0を返します。

```
137 static int check_ex(select_table * wait, struct m_inode * inode) 138 {...
139         struct tty_struct * tty;
140
141         if (tty = get_tty(inode))
142                 if (!FULL(tty->write_q))
143                         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
```

//// select()の内部機能です。
// do_select()は、カーネルがselect()システムコールを実行するための実際のハンドラです。これは
// この関数は，まずディスクリプターセット内の各ディスクリプターの有効性をチェックし，次に
// 関数check_XX()で各ディスクリプターセットのディスクリプターをチェックし、その数をカウントします。
// ディスクリプタセットで現在準備ができているディスクリプタの いずれかのディスクリプターが準備できている場合は
// この関数はすぐに戻りますが、そうでない場合はプロセスがスリープ状態になり、プロセス
// は、タイムアウトが切れるか、待ち行列のプロセスが終了しても実行し続けます。

```
154 記述子があるところの//を覚醒させる。
155 int do_select(fd_set in, fd_set out, fd_set ex,
156         fd_set *inp, fd_set *outp, fd_set *exp)
156 {
157         int count;                              // the number of ready descriptors
158         select_table wait_table;
159         int i;
160         fd_set mask;
161
```

// まず，3つのディスクリプターセットがORされ，有効なディスクリプターのビットマスクを
//記述子セットをマスクで取得します。その後、現在のプロセスをループして
// 各ディスクリプターが有効で、ディスクリプターセットに含まれているかどうか。ループ内では，各
//ディスクリプターが判定されると、マスクが1ビット右にシフトします。したがって、それに基づいて
// マスクの最下位ビットで、対応するディスクリプターの有無を判断します。
// は、ユーザーが与えたディスクリプターセットにあります。有効な記述子は，パイプファイル記述子でなければならない。
// キャラクタデバイスのファイル記述子またはFIFO記述子のいずれかであり、残りのタイプは
// 無効なディスクリプターとしてEBADFエラーを返す。

```
162
163         mask = in | out | ex;
164         for (i = 0 ; i < NR_OPEN ; i++,mask >>= 1) {
165                 if (!(mask & 1))                // not in the descriptor set.
166                         continue;
167                 if (!current->filp[i])          // file not opened.
168                         return -EBADF;
169                 if (!current->filp[i]->f_inode) // file i-node is null.
170                         return -EBADF;
171                 if (current->filp[i]->f_inode->i_pipe) // valid: a pipe file.
172                         continue;
173                 if (S_ISCHR(current->filp[i]->f_inode->i_mode)) // valid: a char dev file.
```

```
173                         continue;
174                 if (S_ISFIFO(current->filp[i]->f_inode->i_mode)) // valid: a FIFO file.
175                         continue;
176                 return -EBADF;                          // all the rest are invalid.
177         }
```

// 3つのディスクリプターセットの各ディスクリプターが準備できているかどうかを確認するためにループすることから始めましょう

//（運用）となります。この時点で「マスク」は、現在ディスクリプターのマスクとして使用されている

// 処理されています。ループ内の3つの関数check_in()・check_out()・check_ex()

```
// は、ディスクリプターがリードイン、ライトアウトの準備ができているかどうかを判断するために使用されます。
//の状態になります。ディスクリプタが動作可能な状態であれば，該当するビットが設定されます。
//ディスクリプターがセットされ、レディディスクリプターのカウント数がインクリメントされる
178 // 1です。183行目の「mask += mask」という記述は、「mask << 1」と同じです。
179 repeat:
180         wait_table.nr = 0;
181         *inp = *outp = *exp = 0;
182         count = 0;
183         mask = 1;
184         for (i = 0 ; i < NR_OPEN ; i++, mask += mask) {
// この時点でチェックしたディスクリプターが、読み取り操作ディスクリプターセットの中にあり、かつ
// ディスクリプタが読み取り操作の準備ができている場合，ディスクリプタセットの対応するビットが
// が1に設定され、レディディスクリプターのカウントが1つ増加します。
186                 if (mask & in)
187                         if (check_in(&wait_table,current->filp[i]->f_inode)) {
186                                 *inp |= mask;           // set the bit in the set.
187                                 count++;                // ready number.
188                         }
// この時点でチェックしたディスクリプターが、書き込み操作用ディスクリプターセットに入っていて
// ディスクリプタが書き込み操作の準備ができている場合，ディスクリプタセットの対応するビットが
// が1に設定され、レディディスクリプターのカウントが1つ増加します。
190                 if (mask & out)
191                         if (check_out(&wait_table,current->filp[i]->f_inode)) {
192                                 *outp |= mask;
193                                 count++;
193                         }
// この時点でチェックしたディスクリプターが異常ディスクリプターセットの中にあり、かつ、ディスクリプター
// が異常な状態である場合には、EXディスクリプターセットの対応するビットを1にして
// 異常なディスクリプターの数を1つ増やします。
195                 if (mask & ex)
196                         if (check_ex(&wait_table,current->filp[i]->f_inode)) {
196                                 *exp |= mask;
197                                 count++;
198                         }
199         }
// プロセスのすべてのディスクリプターをチェックした後、レディディスクリプターがない場合は
// (count==0)で、プロセスがノンブロッキング・シグナルを受信しておらず、かつ、待機中の
// この時点でディスクリプターを設定するか、待ち時間が経過していない場合は、現在のプロセスを
//の状態を割り込み可能なスリープ状態にして、スケジューラーを実行して他のタスクを実行します。
// カーネルがこのタスクを再びスケジューリングする際には、free_wait()を呼び出してタスクを目覚めさせます。
// 関連する待ち行列のタスクの前後に、リピートラベルにジャンプします（行
200     // 178）を見て、気になる記述子があるかどうかを再判断します。
201         if (!(current->signal & ~current->blocked) &&
```

```
202             (wait_table.nr || current->timeout) && !count) {
203                 current->state = TASK_INTERRUPTIBLE;
204                 schedule();
205                 free_wait(&wait_table);       // the task is awakened and returned to here.
206                 goto repeat;
206         }
    // この時点でcountが0になっていない場合、あるいはシグナルを受信した場合、あるいは待機時間が
    // が立ち上がり、待つべきディスクリプターがない場合、free_wait()を呼び出して
207     // 待機キューに入ってから、レディディスクリプターの数を返す。
208         free_wait(&wait_table);
```

```
209         return count;
209 }
210
211 /*
 * Note that we cannot return -ERESTARTSYS, as we change our input
 * parameters. Sad, but there you are. We could do some tweaking in
 * the library function ...
 */        // The parameter *timeout is changed during processing.
```

//// select()システムコール機能。

// この関数内のコードは、主にパラメータのコピーと変換を行います。

// とselect()関数の動作後に行われます。select()の主な仕事は、do_select()

// の関数です。Sys_select()は、まずselect()関数のパラメータを分解してコピーします。

によって与えられたバッファポインタに従って，ユーザデータ空間からカーネル空間に // を転送します。

// パラメータを設定し、待機時間を設定した後、do_select()を呼び出します。を返した後。

// 結果はユーザースペースにコピーバックされます。

// パラメータバッファは，ユーザ領域にあるselect()関数の最初のパラメータを指します.

// 返り値が0より小さい場合は、実行中にエラーが発生し、返り値が

// が0の場合，指定された待機時間内に操作可能なディスクリプターがないことを意味します。

```
217         //時間。戻り値が0より大きい場合は、準備の整ったディスクリプターの数を示す。
218 int sys_select( unsigned long *buffer )
217 {
218 /* Perform the select(nd, in, out, ex, tv) system call. */
```

// 最初に、select()関数を分解するためのいくつかのローカル変数を定義します。

// ポインタの引数で渡される引数。

```
220         int i;
221         fd_set res_in, in = 0, *inp;            // read in fd set.
222         fd_set res_out, out = 0, *outp;         // write out fd set.
223         fd_set res_ex, ex = 0, *exp;            // abnormal fd set.
224         fd_set mask;                            // descriptor value range (nd) mask code.
225         struct timeval *tvp;                    // wait time structure pointer.
226         unsigned long timeout;
226
```

// その後、パラメータが分離され、ユーザーデータ領域からローカルポインタにコピーされます。

// 変数と、3つのディスクリプターセットin（読み込み）、out（書き込み）、ex（例外・異常）がある

// は，ディスクリプタセットポインタが有効であるか否かに応じて，それぞれ取得される。ここで

//マスクもディスクリプターセットの変数です。の最大ディスクリプター値+1に基づいて、3つの

// 記述子のセット（つまり、最初のパラメータndの値）は、すべてのマスクに設定されます。

// ユーザープログラムが関心を持つ記述子。例えば，nd = 4の場合，mask = 0b00001111となります。

// （合計32ビット）。

```
228         mask = ~((~0) << get_fs_long(buffer++));
229         inp = (fd_set *) get_fs_long(buffer++);
230         outp = (fd_set *) get_fs_long(buffer++);
231         exp = (fd_set *) get_fs_long(buffer++);
232         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);
```

//次に、待機（スリープ）時間の値「timeout」をtime構造体から取り出すことを試みます。まずは
//「タイムアウト」は最大（無限）の値に初期化され、「タイムアウト」の時間値を

// 時間構造がユーザー空間から取得され、現在のティック値のジフ?
// システムが変換されて追加され、最後に時間を刻む値「タイムアウト」が
// が待つ必要があることを取得します。この値を使って、現在のプロセスが必要とする遅延時間を設定します。
// を待ちます。また、241行目のtv_usecフィールドはマイクロ秒の値です。これを次の式で割ると
// 1000000は対応する秒数を与え、それにシステムのティックを掛けています。

tv_usecをtickの値に変換する、// per second HZ。

```
239
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.
```

// select()の主な作業は、関数do_select()で行われます。
// 関数は、処理結果をユーザーデータ領域にコピーし、それを
// ユーザーになります。競合状態を避けるためには、do_select()を呼び出す前に割り込みを無効にする必要があります。
// とし、関数が戻った後に有効にします。
// do_select()が戻った後、プロセスの待ち時間フィールドのタイムアウトがまだ大きい場合
現在のシステムのタイミングの目盛り値よりも // ジフティに、ディスクリプターの準備ができていることを示します。
// タイムアウトの前に。そこでここでは、タイムアウトまでの残り時間値を記録して
// であれば、この値をユーザーに返します。プロセスの待機遅延フィールドのタイムアウトが
// すでに現在のシステムのjiffiよりも小さいか等しい場合は、do_select() が
// タイムアウトで返されたので、残り時間の値がゼロになります。

```
247         cli();                              // disable int.
248         i = do_select(in, out, ex, &res_in, &res_out, &res_ex);
249         if (current->timeout > jiffies)
250                 timeout = current->timeout - jiffies;
251         else
```

```
252                 timeout = 0;
253         sti();                                  // enable int.
    // 次にプロセスのタイムアウトフィールドをクリアします。返されたレディディスクリプターの数が
    do_select()で // が0より小さい場合は、実行エラーとなるので、エラー番号を返す。
    // その後、処理されたディスクリプターセットの内容と、遅延時間構造の内容を書き戻す
    // をユーザーのバッファスペースに配置します。また、時間構造のコンテンツを書き込む際には
    // tickの時間単位で表される残りの遅延時間を秒とマイクロ秒に変換する
254     // 値を表示します。
255         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));
```

```
                put_fs_long(timeout/HZ, (unsigned long *) &tvp->tv_sec);  // seconds.
                timeout %= HZ;
                timeout *= (1000000/HZ);
                put_fs_long(timeout, (unsigned long *) &tvp->tv_usec);    // microseconds.
        }
// この時点で利用可能なレディディスクリプターがなく、ノンブロッキングシグナルを受信した場合。
//中断されたエラー番号が返される。それ以外の場合は，レディディスクリプターの数
// を返します。
        if (!i && (current->signal & ~current->blocked))
                return -EINTR;
        return i;
}
```

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.

次の章では、インテル80X86が提供するセグメンテーション機能とページング機能を利用してメモリを管理するLinuxの具体的な方法とコードの実装を中心に説明します。また、Copy-on-Write機構やDemond Loading機構の原理と実装についても説明します。