# 10 Character Device Driver

Linux 0.12カーネルでは、キャラクターデバイスには主に制御端末デバイスとシリアル端末デバイスがあります。本章のコードは、これらのデバイスの入出力を操作するためのものである。端末ドライバの基本的な動作原理については、モーリス・J・バッハ氏の著書「The Design of the UNIX Operating System」やその他の関連書籍を参考にすることができる。本章には，リスト10-1に示すように，合計7つのソースコードファイルがあり，そのうち2つはアセンブリ言語でプログラムされています。これらのファイルは，kernel/chr\_drv/ ディレクトリにあります。

リスト 10-1 linux/kernel/chr\_drv

|  |  |
| --- | --- |
| ファイル名 サイズ 最終更新日(GMT) Desc. Makefile 3618 バイト 1992-01-12 19:49:17 console.c 23327 バイト 1992-01-12 20:28:33 keyboard.S 13020 バイト 1992-01-12 15:30:51 pty.c 1186 バイト 1992-01-10 23:56:45 rs\_io.s 2733 バイト 1992-01-08 06:27:08 serial.c 1412 バイト 1992-01-08 06:17:01 tty\_io.c 12282 バイト 1992-01-11 16:18:46  tty\_ioctl.c | 6325 bytes 1992-01-11 04:02:37 |

## 10.1 Main Functions

### この章のプログラムは、3つのパートに分けられます。第1部はRS-232シリアルラインドライバで、rs\_io.sとserial.cが含まれています。第2部はコンソールドライバで、キーボード割り込みドライバkeyboard.Sとコンソール表示ドライバconsole.cが含まれています。第3部は端末ドライバと上位プログラムとのインターフェース部分で、端末入出力プログラムtty\_io.cと端末制御プログラムtty\_ioctl.cが含まれています。以下では、まず端末制御ドライバの実装の基本原理を説明し、次にこれらの基本機能を3つのパートに分けて説明します。その後、各ソースファイルの詳細を説明し、注釈を付ける。

### 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拡張したり、バックスペースを入力して、それまでに入力した文字の削除を制御したりしている。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ローモードである。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。

Pro

cess Read/Write

Terminal Upper Layer

Interface

L

ine Discipline

D

evice Driver

Character Device

図10-1 端子ドライバ制御フロー

### 10.1.2 Types of Terminal Devices Supported by Linux

端末は文字型の装置で、多くの種類がある。ttyはTeletypeの略語で、Teletype社が製造した最も古い端末装置で、テレタイプライターのような形をしています。Linux 0.1xシステムのデバイスファイルのディレクトリ/dev/には、通常、次のような端末デバイスファイルが入っています。

/ crw-rw-rw- 1 root tty 5, 0 Jul 30 1992 tty // 制御端末 crw--w--w- 1 root tty 4, 0 Jul 30 1992 tty0 // 仮想端末のエイリアスです。

crw--w--w- 1 root tty 4, 1 Jul 30 1992 console // コンソール crw--w--w- 1 root other 4, 1 Jul 30 1992 tty1 // 仮想端末1 crw--w--w- 1 root tty 4, 2 1992 年 7 月 30 日 tty2 crw--w--w- 1 root tty 4, 3 1992 年 7 月 30 日 tty3 crw--w--w- 1 root tty 4, 4 1992 年 7 月 30 日 tty4 crw--w--w- 1 root tty 4, 5 1992 年 7 月 30 日 tty5 crw--w--w- 1 root tty 4, 6 1992年7月30日 tty6 crw--w--w- 1 root tty 4, 7 1992年7月30日 tty7 crw--w--w- 1 root tty 4, 8 1992年7月30日 tty8 crw-rw-rw- 1 root tty 4, 64 1992年7月30日 ttys1 // シリアルポート端子1 crw-rw-rw- 1 root tty 4, 65 1992年7月30日 ttys2 crw--w--w- 1 root tty 4, 128 1992年7月30日 ptyp0 // プライマリ疑似端子です。

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 1992 年 7 月 30 日 ttyp0 // スレーブ pty crw--w--w- 1 root tty 4, 193 1992 年 7 月 30 日 ttyp1 crw--w--w- 1 root tty 4, 194 1992 年 7 月 30 日 ttyp2 crw--w--w- 1 root tty 4, 195 1992 年 7 月 30 日 ttyp3

1. これらの端末機器ファイルは、以下の種類に分けられます。
2. Serial port terminal (/dev/ttySn)
   1. serial port terminal is a terminal device that is connected using a computer serial port. The computer treats each serial port as a character device. For some time these serial port devices are often referred to as terminal devices because its primary use at that time was to connect to a terminal. The device file names corresponding to these serial ports are /dev/ttyS0, /dev/ttyS1, etc., and the device numbers are (4, 64), (4, 65), etc., respectively corresponding to COM1 and COM2 under the DOS system. To send data to a port, you can redirect the standard output to these special file names on the command line. For example, typing "echo test > /dev/ttyS1" at the command prompt will send the word "test" to the device connected to the ttyS1 port.

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

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。

1. 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。

1. 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。
2. 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。

1. 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。
2. Other types of terminals

### 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。

### 10.1.3 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; // 端末の io プロパティと制御文字 int pgrp; // 所属するプロセスグループ。

int stopped; void (\*write)(struct tty\_struct \* tty); // ttyの書き込み関数へのポインタ struct tty\_queue read\_q; // ttyの読み込みキュー struct tty\_queue write\_q; // ttyの書き込みキュー。

struct tty\_queue secondary; // ttyの補助的なキュー（または正規のキュー）。

}; extern struct tty\_struct tty\_table[]; // tty構造体の配列。

Linuxカーネルは、配列tty\_table[]を使用して、システム内の各端末デバイスに関する情報を格納しています。

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; // 現在のデータの統計。

// 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; // バッファ内のデータのヘッダ。

unsigned long tail; // データのテールポインタ。

struct task\_struct \* proc\_list; // このバッファキューを待っているプロセス char buf[1024]; // キューのバッファです。

};

各tty文字キューのバッファ長は1Kバイトです。リードバッファキューread\_qは、キーボードやシリアルターミナルから入力されたオリジナルの文字列を一時的に格納するために使用され、ライトバッファキューwrite\_qは、コンソールディスプレイやシリアルターミナルに書き込まれたデータを格納するために使用されます。補助キューsecondaryは、read\_qから取り出されたデータを（ICANONフラグに基づいて）カノニカルモードプログラムで処理（フィルタリング）したデータを格納するために使用され、Cookedモードデータとも呼ばれます。これは、バックスペース文字などの元データに含まれる特殊文字を正規プログラムが変換した後の正規入力データであり、文字行単位でプログラムに読み込まれて使用される。セカンダリーキューの文字を読むには、上位端末のリード関数tty\_read()を使用する。

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構造体も含まれており、その内容は以下の通りです。

/ struct termios { unsigned long c\_iflag; /\* 入力モードフラグ \*/ unsigned long c\_oflag; /\* 出力モードフラグ \*/ unsigned long c\_flag; /\* 制御モードフラグ \*/ unsigned long c\_lflag; /\* ローカルモードフラグ \*/ unsigned char c\_line; /\* 行の規律 \*/ // または速度 unsigned char c\_cc[NCCS]; /\* 制御文字 \*/

};

ここで、c\_iflagは、設定されている入力モードフラグである。Linux 0.12カーネルでは、POSIX.1で定義されている11種類の入力フラグをすべて実装しており、その内容はtermios.hヘッダーファイルに記述されている。端末デバイスドライバは、これらのフラグを使って、端末から入力された文字をどのように変換（フィルタリング）するかを制御する。例えば，入力された改行文字（NL）をキャリッジリターン（CR）に変換する必要があるかどうか，入力された大文字を小文字に変換する必要があるかどうか（以前は大文字しか入力できない端末デバイスがあったため）などである。Linux 0.12カーネルでは、関連するハンドラはtty\_io.cファイルのcopy\_to\_cooked()です。また，include/termios.hファイルの86〜99行目も参照してください。

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行目から183行目を参照。 以上の4種類のフラグセットは、いずれも符号なしの長さで、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・リードを使用する際には、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構造の中のVTIMEとVMINという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、以下の情報が表示されます。

[/root]# stty

---------Characters----------

intr: '^c' quit: '^\' erase: '^h' kill: '^u' eof: '^d'。

time: 0 min: 1 swtc: '^@' start: '^q' stop: '^s'

SUSP: '^z' eol: '^@' eol2: '^@' lnext: '^v'

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

----------Control Flags---------

-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

-----------Local Flags---------

ISIG ICANON -XCASE ECHO -ECHOE -ECHOK -ECHONL -NOFLSH -TOSTOP ECHOCTL ECHOPRT ECHOKE -FLUSHO -PENDIN -IEXTEN rows 0 cols 0

その中で、マイナス記号は設定がないことを示しています。また、現在のLinuxシステムでは、これらの情報をすべて表示するには「stty -a」と入力する必要があり、表示形式も異なります。

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。

Console

Serial Term

inal 1

Serial Terminal 2

tty\_table

termios struct

O

ther fields

Write fun

c

P

ointer

tty read queue

(

)

read\_q

tty write queue

)

(

write\_q

tty auxiliary

queue

)

secondary

(

tty\_struct

tty\_queue

Other fields

Data

head

Data

tail

Buffer

(

buf

)

図10-2 ターミナルプログラムのデータ構造

### 10.1.4 Canonical mode and non-canonical mode

#### 以上のことからわかるように、端末プログラムが入力データまたは出力データを処理する方法に応じて、端末の作業モードを正規モードと非正規モードまたは生モードに分けることができる。正規のモードでは、端末装置を通過するデータは、関連する規則や規格に従って修正・変換された後、送信される。使用される変換機能は、ラインディシプリンまたはラインプロシージャモジュールと呼ばれることが多い。非正規モードの場合、ラインディシプリンプログラムは、端末とプロセス間のデータ転送のみを行い、データの変換処理は行わない。以下では、この2つのモードで使用される処理ルールと、カーネルコードへの実装について詳しく説明します。

#### 10.1.4.1 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元に返されます。

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。この文字と行末文字EOLおよびEOL2は、copy\_to\_cooked()関数ではキャリッジリターンとして扱われます。読み取り操作関数で遭遇した文字は直ちにリターンされます。EOF文字はキューに入れられずに無視されます。

#### 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。

#### 10.1.4.2 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。

### 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。

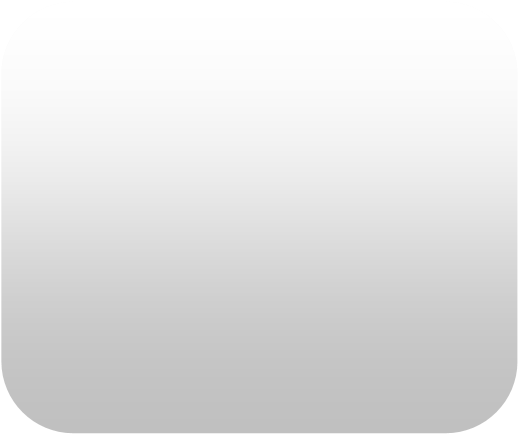
### 10.1.5 Console Terminal and Serial Terminal Devices

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と直接情報交換を行います。 keyboard.sとconsole.cの2つのプログラムは、実は、Linuxシステムのホストにあるハードウェア端末装置をモニターとキーボードを使ってエミュレートするプログラムとよく似ています。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。この2つのプログラムが実現する機能は、シリアル端末装置の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）と同等であり、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る。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。

S

ame

functions



Terminal

I

/

O

program tty\_io.c

console.c

keyboard.s

Monitor

Keyboard

Console

& Keybd Code

Comm Program

Monitor

Keyboard

rs\_io.s

Kernel

Hardware

S

erial Terminal Device

Console

C

ommunication

図10-3 コンソール端末とシリアル端末のデバイス図

#### 10.1.5.1 Console Driver

Linux 0.12カーネルでは，ターミナル・コンソール・ドライバには，keyboard.Sプログラムとconsole.cプログラムが含まれている。keyboard.Sは，ユーザが入力した文字を処理して読み取りバッファ・キューread\_qに入れ，copy\_to\_cooked()関数を呼び出してread\_qの文字を読み取り，それを補助バッファ・キューsecondaryに変換する。console.cプログラムは、コンソール端末で受信したコードの出力表示処理を実装している。

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echo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。

主な機能

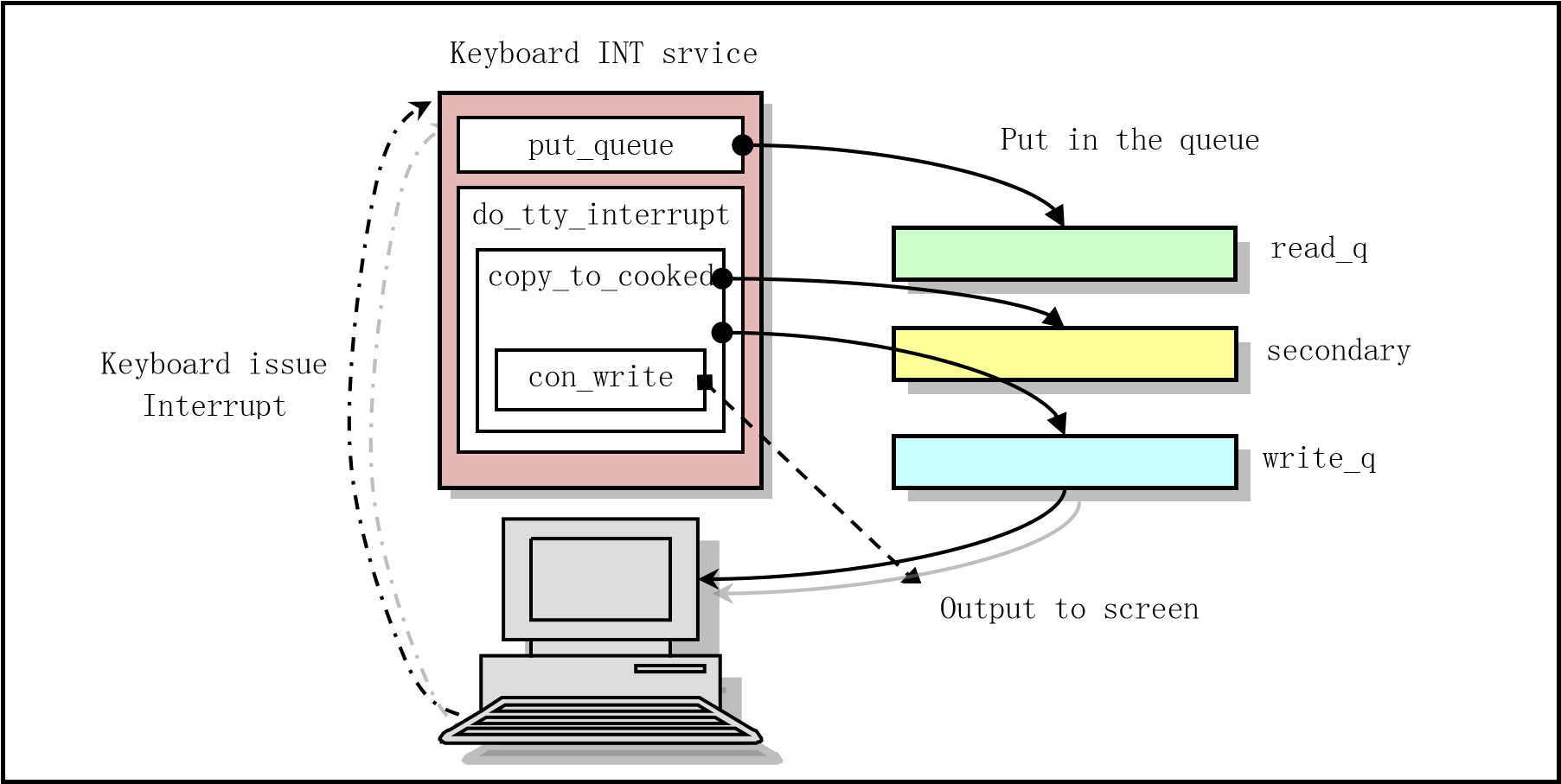


図10-4 コンソールキーボード割り込みサービス

プロセスがttyの書き込み操作を行うために、ターミナルドライバは文字を1つずつ処理します。書き込みバッファのキューwrite\_qが満杯になっていないときは、ユーザーバッファから文字を取り出して処理し、write\_qに入れます。ユーザーデータがすべてwrite\_qキューに入るか、この時点でwrite\_qが満杯になると、端末構造体tty\_structで指定された書き込み関数が呼び出され、write\_qバッファキュー内のデータがコンソール画面に出力される。コンソール端末の場合，書き込み関数はcon\_write()であり，これはconsole.cプログラムで実装されている．

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。

echo

secondary

read\_q

write\_q

Char

d

ev Interface

S

yscall read,write

Host screen

Host

Keyboard

read\_write.c

char\_dev.c

In File System

Char device driver

Console

Terminal D

evice

tty\_write()

tty\_read()

copy\_to\_cooked()

con\_write()

console.c

keyboard.S

put\_queue()

tty\_io.c

NOTE

:

The functions

tty\_write(), tty\_read(),

and copy\_to\_cooked() are

all in tty\_io.c

主な機能

図10-5 コンソールキャラクターキューと機能の関係

#### 10.1.5.2 Serial Terminal Driver

シリアル端子の操作を行うプログラムは、serial.cとrs\_io.sです。serial.cプログラムは、シリアルポートの初期化と、送信ホールドレジスタエンプティ割り込みのマスク解除によるシリアル割り込みの送信文字操作の有効化を行います。rs\_io.s プログラムは、シリアル割り込みのハンドラです。主に、シリアル割り込みが発生する4つの理由に応じて処理を行います。

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。

secondary

read\_q

write\_q

Char Dev Interface

Syscall

read,write

Screen

Keyboard

read\_write.c

char\_dev.c

In File system

C

har device driver

Serial Terminal Device

tty\_write()

tty\_read()

echo

copy\_to\_cooked()

write\_char

rs\_io.s

read\_char:

tty\_io.c

NOTE

The functions

:

tty\_write(), tty\_read(),

and copy\_to\_cooked() are

all in tty\_io.c

C

omm, display and keyboard firmware

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

図10-6からわかるように、シリアル端末とコンソールの主な違いは、シリアル端末ではコンソール操作の表示とキーボードのプログラムconsole.cとkeyboard.Sをrs\_io.sというプログラムに置き換えているだけで、あとの処理方法は全く同じである。

### 10.1.6 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節の記述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

1. /\*
2. \* Thanks to Alfred Leung for US keyboard patches
3. \* Wolfgang Thiel for German keyboard patches 10 \* Marc Corsini for the French keyboard

11 \*/

12

1. /\* KBD\_FINNISH for Finnish keyboards
2. \* KBD\_US for US-type
3. \* KBD\_GR for German keyboards
4. \* KBD\_FR for Frech keyboard
5. \*/

18 #define KBD\_FINNISH // 後で文字マッピングコード表を選択する際に使用するキーボードの種類を定義します。 18 #define KBD\_FINNISH // リーナスはフィンランド人である

19

20 .テキスト

21 .globl \_keyboard\_interrupt // 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を設定するために使用されます。 21 .globl \_keyboard\_interrupt // keyboard\_interrupt()と同じです。

22

1. /\*
2. \* these are for the keyboard read functions
3. \*/
4. // 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。
5. size = 1024 /\* must be a power of two ! And MUST be the same 27 as in tty\_io.c !!!! \*/

1. // キーボードバッファキューのデータ構造 tty\_queue // (include/linux/tty.h, line 22)の各フィールドのオフセットを以下に示します。
2. head = 4 // Offset of the head field in the tty\_queue structure.
3. tail = 8 // Offset of the tail field.
4. proc\_list = 12 // the wait process field offset (wait for this buffer).
5. 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）を使用します。各バイトフラグの // 各ビットの意味は以下の通りです。

// (1) 'mode' は，特殊キーの押下状態フラグである。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)、シフトキーの // 状態を示すのに使用されます。

// bit 7 - caps キーが押された状態、bit 3 - 右 ctrl キーが押された状態。

// bit 6 - caps キーの状態; bit 2 - 左の ctrl キーが押された状態。

// ビット5 - 右altキーが押された状態、ビット1 - 右シフトキーが押された状態 // ビット4 - 左altキーが押された状態、ビット0 - 左シフトキーが押された状態

// (2) 'leds'はキーボードインジケータを示すステータスフラグ、つまりnum-lock, caps-lock, scroll-lockを示すLEDチューブの状態を示すフラグです。

// ビット7-3はすべてゼロ、使用しない。ビット1のnum-lock（初期設定1）、//ビット2のcaps-lock、ビット0の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。

1文字または2文字のスキャンコードによって、 // 受信されます。通常、スキャンコード0xe0を受信した場合は、それに続く文字がまだ1つあることを // 意味し、スキャンコード0xe1を受信した場合は、それに続く文字が2つあることを // 意味します。プログラムリストの後の説明をご覧ください。

1. // bit1=1 0xe1を受信、bit0=1 0xe0を受信。
2. mode: .byte 0 /\* caps, alt, ctrl and shift mode \*/
3. leds: .byte 2 /\* num-lock, caps, scroll-lock mode (nom-lock on) \*/
4. e0: .byte 0

36

1. /\*
2. \* con\_int is the real interrupt routine that reads the
3. \* keyboard scan-code and converts it into the appropriate
4. \* ascii character(s).
5. \*/ // 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が要求に応答すると、 // 割り込みハンドラはキースキャンコードを読み込みます。割り込みハンドラは、キーボードコントローラポート（0x60）から // キースキャンコードを読み込み、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、すぐにキーボードコントローラに応答します

// 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

1. // do\_tty\_interrupt()が呼ばれ（実際にはcopy\_to\_cooked()が呼ばれる）、read\_qの中の文字を処理して // 二次補助キューに入れます。 42 \_keyboard\_interrupt:
2. pushl %eax
3. pushl %ebx
4. pushl %ecx
5. pushl %edx
6. push %ds
7. push %es
8. movl $0x10,%eax // Set ds and es to kernel data segment.
9. mov %ax,%ds
10. mov %ax,%es
11. movl \_blankinterval,%eax
12. movl %eax,\_blankcount // preset black screen time count (blankinterval).
13. xorl %eax,%eax /\* %eax is scan code \*/ 55 inb $0x60,%al // Read scan code to al.
14. 56 cmpb $0xe0,%al // スキャンコードが0xe0の場合、set\_e0にジャンプ 57 je set\_e0
15. cmpb $0xe1,%al // jump to set\_e1 if scan code is 0xe1
16. je set\_e1
17. call key\_table(,%eax,4) // key handler: key\_table + eax\*4 (see line 513).
18. 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

1. 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、 // すぐにキーボードの動作を可能にするために使用されます。
2. e0\_e1: inb $0x61,%al // Get port B state, PB7 is used to enable/disable keyboard.
3. jmp 1f // delay for awhile.
4. 1: jmp 1f
5. 1: orb $0x80,%al // set bit7 of al.
6. jmp 1f
7. 1: jmp 1f
8. 1: outb %al,$0x61 // set port B bit7 to disable keyboard.
9. jmp 1f
10. 1: jmp 1f
11. 1: andb $0x7F,%al // reset bit7 of al.
12. outb %al,$0x61 // reset port B bit7 to enable keyboard.
13. movb $0x20,%al // send EOI signal to 8259A chip.
14. outb %al,$0x20
15. pushl $0 // The console tty = 0, and pushed as a parameter.
16. call \_do\_tty\_interrupt // converted to canonical mode
17. addl $4,%esp // Discard the parameters of the stack.
18. pop %es
19. pop %ds
20. popl %edx
21. popl %ecx
22. popl %ebx
23. popl %eax
24. iret
25. set\_e0: movb $1,e0 // When 0xe0 is received, bit0 of e0 flag is set.
26. jmp e0\_e1
27. set\_e1: movb $2,e0 // When 0xe1 is received, bit1 of e0 flag is set.
28. jmp e0\_e1

89

1. /\*
2. \* This routine fills the buffer with max 8 bytes, taken from
3. \* %ebx:%eax. (%ebx is high). The bytes are written in the 93 \* 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

1. // を8ビット移動させ（つまり、ahの値をalに移動させ、bl→ah、bh→bl）、上記の処理をalに対して // 繰り返します。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を保存し、 // その後、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。 95 put\_queue:
2. pushl %ecx
3. pushl %edx // Get read buffer queue pointer.
4. movl \_table\_list,%edx # read-queue for console
5. movl head(%edx),%ecx // Take the queue head pointer -> ecx.
6. 1: movb %al,buf(%edx,%ecx) // put char in al into the head pointer position.
7. incl %ecx // moved forward by 1 byte.
8. andl $size-1,%ecx // Adjust the head pointer.
9. cmpl tail(%edx),%ecx # buffer full - discard everything
10. je 3f
11. shrdl $8,%ebx,%eax // Move ebx right 8 bits into eax, ebx is unchanged.
12. je 2f // Are there still characters? if no then jump.
13. shrl $8,%ebx // Move ebx value to the right by 8 bits
14. jmp 1b
15. 2: movl %ecx,head(%edx) // If all chars in queue, the head pointer is saved.
16. movl proc\_list(%edx),%ecx // Waiting process pointer for this queue?
17. testl %ecx,%ecx // Check if there are processes waiting for this queue.
18. je 3f // No, then jump (to line 114).
19. movl $0,(%ecx) // Yes, the process is awake.
20. 3: popl %edx
21. popl %ecx
22. ret 117

// ここから先は、ジャンプテーブルkey\_tableのポインタに対応する各キーのサブルーチンであり、 // 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は、 // 513行目から始まります。

// 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。

1. ctrlまたは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またはaltキーが押されたことを意味し，モードフラグのビットは， // ctrlまたはaltに対応してセットされる。
2. ctrl: movb $0x04,%al // 0x04 (bit2), the left ctrl key in mode.
3. jmp 1f
4. alt: movb $0x10,%al // 0x10 (bit4), the left alt key in mode.
5. 1: cmpb $0,e0 // e0 set? (key pressed is right ctrl/alt key)?
6. je 2f // no, jump (to line 124).
7. addb %al,%al // yes, change to right-click flag (bit3 or bit5).
8. 2: orb %al,mode // Set the corresponding bit in the mode flag.
9. 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

1. // モードフラグの対応するビットです。操作では，e0フラグが設定されているかどうかで， // キーボードの右側のctrlキーなのかaltキーなのかを判断する必要があります。126 unctrl: movb $0x04,%al // bit2はモードでは左のctrlキー用です。
2. jmp 1f
3. unalt: movb $0x10,%al // 0x10 is the bit 4 for the left alt key in mode.
4. 1: cmpb $0,e0 // e0 set? (Is the right ctrl/alt key released?)?
5. je 2f // no, jump (to line 132)
6. addb %al,%al // yes, then change to reset flag bit (bit3 or bit5).
7. 2: notb %al // Reset corresponding bit in mode flag.
8. andb %al,mode
9. ret 135
10. // このコードは，左右のシフトキーが押されたり離されたりしたときのスキャンコードを処理して， // モードの対応するビットをセットしたりリセットしたりします。
11. lshift:
12. orb $0x01,mode // left shift key is pressed, set mode bit0.
13. ret
14. unlshift:
15. andb $0xfe,mode // left shift key is released, reset mode bit 0.
16. ret
17. rshift:
18. orb $0x02,mode // right shift key is pressed, set mode bit1.
19. ret
20. unrshift:
21. andb $0xfd,mode // right shift key is released, reset mode bit1.
22. ret 148
23. // このコードはcapsスキャンコードを処理します。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。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）とledsフラグのcaps-lockビット（ビット2）を反転させ、capsキーが押されているフラグビット（ビット7）をセットします。149 caps: testb $0x80,mode // bit7をセット ? (capsが押された？)
24. jne 1f // yes, jump to line 169.
25. xorb $4,leds // flip caps-lock(bit2) in leds flag.
26. xorb $0x40,mode // flip caps state (bit6) in mode flag.
27. orb $0x80,mode // set bit7 (caps pressed) in mode flag.
28. // このコードは、ledsフラグに従って、LEDインジケータをオンまたはオフにします。
29. set\_leds:
30. call kb\_wait // Wait for controller input buffer to be empty.
31. movb $0xed,%al /\* set leds command \*/
32. outb %al,$0x60 // Send kbd command 0xed to port 0x60.
33. call kb\_wait
34. movb leds,%al // get leds flag as parameter.
35. outb %al,$0x60
36. ret
37. uncaps: andb $0x7f,mode // caps released, then reset bit7 in mode flag.
38. ret
39. scroll:
40. testb $0x03,mode // ctrl key pressed too?
41. je 1f // no, jump (to line 169).
42. call \_show\_mem // yes, display memory info (mm/memory.c, line 457) 168 jmp 2f
43. 1: call \_show\_state // no, show process state info (kernel/sched.c, 45)
44. 2: xorb $1,leds // flip bit0 of leds if scroll key pressed.
45. 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 // ledsフラグに基づいてLEDインジケータをオンまたはオフにします。

174

1. /\*
2. \* curosr-key/numeric keypad cursor keys are handled here.
3. \* checking for numeric keypad etc.
4. \*/

// 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後、「Ctrl-Alt-Del」キーのコンビネーションが押されたかどうかを判断します。コンビネーションキーが押された場合は、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。

1. // テンキーのスキャンコード範囲は[0x47 - 0x53]です。表からわかるように、テンキーのキーのスキャンコードの範囲は[0x47 - 0x53]であり、 // 方向キーとして使用する場合には、プリアンブルコードe0となる。
2. cursor:
3. subb $0x47,%al // If scan code not in range [0x47 - 0x53], it ret (198).
4. jb 1f // That is, scan code is not generated from numeric keypad.
5. cmpb $12,%al // (0x53 - 0x47 = 12)
6. ja 1f
7. jne cur2 /\* check for ctrl-alt-del \*/
8. // これが12になると、'del'キーが押されたことを示しているので、 // 続けて'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。
9. testb $0x0c,mode // ctrl key pressed? jump if not.
10. je cur2
11. testb $0x30,mode // alt key pressed?
12. 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

1. // 移動処理を行います。ただし、num-lockランプが点灯していて（キーパッドがテンキーとして使用されていることを示す）、シフトキーも押されている場合は、このときのキーパッドのキーを // カーソル移動の操作として扱います。
2. cur2: cmpb $0x01,e0 /\* e0 forces cursor movement \*/ // e0 flag set ? 190 je cur // jump to cur if e0 is set.
3. testb $0x02,leds /\* not num-lock forces cursor \*/ // test leds flag.
4. je cur // if num's LED is on, do cursor movement operation.
5. testb $0x03,mode /\* shift forces cursor \*/
6. 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

1. キーに合わせて // 取り出し、バッファキューに入れます。一度にキャラクタキューに入れる文字数は、 // <=8が必要なので、実行するためには、 // put\_queueにジャンプする前に、ebxをクリアする必要があります。95行目のコメントをご覧ください。
2. xorl %ebx,%ebx
3. movb num\_table(%eax),%al // obtain the digital char with eax as index.
4. jmp put\_queue // parameter: 1-8 chars in ebx:eax.
5. 1: ret

199

// 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 // 文字がまず得られる。文字が<='9' (5, 6, 2, 3)の場合は、Page Up, Page Dn, Insert, // Deleteキーのいずれかであることを意味し、'~'という文字を機能文字列に追加する必要があります。

// 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

1. // キャラクターキュー。
2. cur: movb cur\_table(%eax),%al // get cursor char to al.
3. cmpb $'9,%al // if <='9' (5,6,2 or 3) it's page up/dn or Ins/Del key
4. ja ok\_cur // it needs to add char '~'.
5. movb $'~,%ah
6. ok\_cur: shll $16,%eax // move content of ax to high word of eax.
7. movw $0x5b1b,%ax // put 'esc [' in ax
8. xorl %ebx,%ebx
9. jmp put\_queue

208

1. #if defined(KBD\_FR)
2. num\_table:
3. .ascii "789 456 1230." // ASCII code associated to keys on numeric keypad.
4. #else 213 num\_table:
5. 214 .ascii "789 456 1230," 215 #endif
6. cur\_table:
7. .ascii "HA5 DGC YB623" // associated to the arrow, Ins, Del keys on keypad.

218

1. /\*
2. \* this routine handles function keys 221 \*/

// ファンクションキーのスキャンコードをエスケープ文字列に変換し、 // 読み取りキューに格納する。このコードで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

1. // ファンクションキーテーブルfunc\_tableを照会して、対応する // エスケープ文字列を取得し、文字列キューに格納する。ファンクションキーが押されている状態で、Altキーが押された場合、 // 制御端末の切り替え動作のみが行われる。
2. func:
3. subb $0x3B,%al // F1 scan code is 0x3B 224 jb end\_func // ret if not a function key 225 cmpb $9,%al // F1--F10 ?
4. jbe ok\_func // jump if yes
5. subb $18,%al // key F11, F12 ?
6. cmpb $10,%al // key F11 ?
7. jb end\_func // ret if order no less than F11's.
8. cmpb $11,%al // key F12 ?
9. ja end\_func // ret if order no great than F12's.
10. ok\_func:
11. testb $0x10,mode // left alt pressed ?
12. jne alt\_func // jump to change console if yes.
13. cmpl $4,%ecx /\* check that there is enough room \*/ 236 jl end\_func // ret if not enough space (4 bytes).
14. movl func\_table(,%eax,4),%eax // get escape character sequence of the key.
15. xorl %ebx,%ebx 239 jmp put\_queue
16. // 次のコードは，仮想制御端子を変更するための alt + Fn キーの組み合わせを処理します。このとき、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、 // 仮想制御端子の番号に対応しています。
17. alt\_func:
18. pushl %eax // push the index number as virtual console number.
19. call \_change\_console // chr\_dev/tty\_io.c, line 87.
20. popl %eax // discard the parameter. 244 end\_func:

245回目

246

1. /\*
2. \* function keys send F1:'esc [ [ A' F2:'esc [ [ B' etc.
3. \*/
4. func\_table:
5. .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
6. .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。

1. // 対応するキーのスキャンコードは，先に定義したキーボードタイプ（FINNISH，US，GERMEN，FRANCH）に応じてASCII文字にマッピングされます。255 #if defined(KBD\_FINNISH)
2. key\_map:
3. .byte 0,27 // for scan code 0x00, 0x01;
4. .ascii "1234567890+'" // for scan code 0x02,...0x0c,0x0d. Bellow is similar.
5. .byte 127,9
6. .ascii "qwertyuiop}" 261 .byte 0,13,0
7. 262 .ascii "asdfghjkl|{" 263 .byte 0,0
8. .ascii "'zxcvbnm,.-"
9. .byte 0,'\*,0,32 /\* 36-39 \*/
10. .fill 16,1,0 /\* 3A-49 \*/
11. .byte '-,0,0,0,'+ /\* 4A-4E \*/
12. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
13. .byte '<
14. .fill 10,1,0

271

1. shift\_map: // mapping table when shift is pressed at the same time.
2. .byte 0,27
3. .ascii "!\"#$%&/()=?`"
4. .byte 127,9
5. .ascii "QWERTYUIOP]^"
6. .byte 13,0
7. .ascii "ASDFGHJKL\\["
8. .byte 0,0
9. .ascii "\*ZXCVBNM;:\_"
10. .byte 0,'\*,0,32 /\* 36-39 \*/
11. .fill 16,1,0 /\* 3A-49 \*/
12. .byte '-,0,0,0,'+ /\* 4A-4E \*/
13. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
14. .byte '>
15. .fill 10,1,0

287

1. alt\_map: // mapping table when alt key is pressed at the same time.
2. .byte 0,0
3. .ascii "\0@\0$\0\0{[]}\\\0" 291 .byte 0,0

292 .バイト 0,0,0,0,0,0,0,0,0,0,0 293 .バイト '～,13,0

1. 294 .byte 0,0,0,0,0,0,0,0,0,0,0 295 .byte 0,0,0
2. .byte 0,0,0,0,0,0,0,0,0,0,0
3. .byte 0,0,0,0 /\* 36-39 \*/
4. .fill 16,1,0 /\* 3A-49 \*/
5. .byte 0,0,0,0,0 /\* 4A-4E \*/
6. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
7. .byte '|
8. .fill 10,1,0

303

304 #elif defined(KBD\_US) // 私たちのキーボードのマッピングテーブルです。

305

1. key\_map:
2. .byte 0,27
3. .ascii "1234567890-=" 309 .byte 127,9
4. .ascii "qwertyuiop[]"
5. .byte 13,0
6. .ascii "asdfghjkl;'"
7. .byte '`,0
8. .ascii "\\zxcvbnm,./"
9. .byte 0,'\*,0,32 /\* 36-39 \*/
10. .fill 16,1,0 /\* 3A-49 \*/
11. .byte '-,0,0,0,'+ /\* 4A-4E \*/
12. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
13. .byte '<
14. .fill 10,1,0

321

322

1. shift\_map:
2. .byte 0,27
3. .ascii "!@#$%^&\*()\_+"
4. .byte 127,9
5. .ascii "QWERTYUIOP{}"
6. .byte 13,0
7. .ascii "ASDFGHJKL:\""
8. .byte '~,0
9. .ascii "|ZXCVBNM<>?"
10. .byte 0,'\*,0,32 /\* 36-39 \*/
11. .fill 16,1,0 /\* 3A-49 \*/
12. .byte '-,0,0,0,'+ /\* 4A-4E \*/
13. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
14. .byte '>
15. .fill 10,1,0 338
16. alt\_map:
17. .byte 0,0
18. .ascii "\0@\0$\0\0{[]}\\\0" 342 .byte 0,0

343 .バイト 0,0,0,0,0,0,0,0,0,0 344 .バイト '~,13,0

1. 345 .byte 0,0,0,0,0,0,0,0,0,0,0 346 .byte 0,0,0
2. .byte 0,0,0,0,0,0,0,0,0,0,0
3. .byte 0,0,0,0 /\* 36-39 \*/
4. .fill 16,1,0 /\* 3A-49 \*/
5. .byte 0,0,0,0,0 /\* 4A-4E \*/
6. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
7. .byte '|
8. .fill 10,1,0

354

1. 355 #elif defined(KBD\_GR) // ドイツ語キーボードのマッピングテーブル。356
2. key\_map:
3. .byte 0,27
4. .ascii "1234567890\\'"
5. .byte 127,9
6. .ascii "qwertzuiop@+" 362 .byte 13,0
7. .ascii "asdfghjkl[]^"
8. .byte 0,'#
9. .ascii "yxcvbnm,.-"
10. .byte 0,'\*,0,32 /\* 36-39 \*/
11. .fill 16,1,0 /\* 3A-49 \*/
12. .byte '-,0,0,0,'+ /\* 4A-4E \*/
13. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
14. .byte '<
15. .fill 10,1,0

372

373

1. shift\_map:
2. .byte 0,27
3. .ascii "!\"#$%&/()=?`"
4. .byte 127,9
5. .ascii "QWERTZUIOP\\\*"
6. .byte 13,0
7. .ascii "ASDFGHJKL{}~"
8. .byte 0,''
9. .ascii "YXCVBNM;:\_"

.バイト 0,'\*,0,32 /\* 36-39 \*/ のようになります。

.fill 16,1,0 /\* 3A-49 \*/ のようになります。

.バイト '-,0,0,0,'+ /\* 4A-4E \*/.

.バイト 0,0,0,0,0,0 /\* 4F-55 \*/ (注)

.バイト '>

.フィル 10,1,0

1. alt\_map:
2. .byte 0,0
3. .ascii "\0@\0$\0\0{[]}\\\0"
4. .byte 0,0
5. .byte '@,0,0,0,0,0,0,0,0,0,0
6. .byte '~,13,0
7. .byte 0,0,0,0,0,0,0,0,0,0,0 397 .byte 0,0
8. .byte 0,0,0,0,0,0,0,0,0,0,0
9. .byte 0,0,0,0 /\* 36-39 \*/
10. .fill 16,1,0 /\* 3A-49 \*/
11. .byte 0,0,0,0,0 /\* 4A-4E \*/
12. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
13. .byte '|
14. .fill 10,1,0

405

406

1. 407 #elif defined(KBD\_FR) // フランスのキーボードのマッピングテーブル。408
2. key\_map:
3. .byte 0,27
4. .ascii "&{\"'(-}\_/@)="
5. .byte 127,9
6. .ascii "azertyuiop^$"
7. .byte 13,0
8. .ascii "qsdfghjklm|"
9. .byte '`,0,42 /\* coin sup gauche, don't know, [\*|mu] \*/
10. .ascii "wxcvbn,;:!"
11. .byte 0,'\*,0,32 /\* 36-39 \*/
12. .fill 16,1,0 /\* 3A-49 \*/
13. .byte '-,0,0,0,'+ /\* 4A-4E \*/
14. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
15. .byte '<
16. .fill 10,1,0

424

1. shift\_map:
2. .byte 0,27
3. .ascii "1234567890]+" 428 .byte 127,9
4. .ascii "AZERTYUIOP<>"
5. .byte 13,0
6. .ascii "QSDFGHJKLM%"
7. .byte '~,0,'#
8. .ascii "WXCVBN?./\\"
9. .byte 0,'\*,0,32 /\* 36-39 \*/
10. .fill 16,1,0 /\* 3A-49 \*/

.バイト '-,0,0,0,'+ /\* 4A-4E \*/.

.バイト 0,0,0,0,0,0 /\* 4F-55 \*/ (注)

.バイト '>

.フィル 10,1,0

1. alt\_map:
2. .byte 0,0
3. .ascii "\0~#{[|`\\^@]}"
4. .byte 0,0
5. .byte '@,0,0,0,0,0,0,0,0,0,0
6. .byte '~,13,0
7. .byte 0,0,0,0,0,0,0,0,0,0,0 448 .byte 0,0
8. .byte 0,0,0,0,0,0,0,0,0,0,0
9. .byte 0,0,0,0 /\* 36-39 \*/
10. .fill 16,1,0 /\* 3A-49 \*/
11. .byte 0,0,0,0,0 /\* 4A-4E \*/
12. .byte 0,0,0,0,0,0,0 /\* 4F-55 \*/
13. .byte '|
14. .fill 10,1,0

456

1. #else
2. #error "KBD-type not defined"
3. #endif
4. /\*
5. \* do\_self handles "normal" keys, ie keys that don't change meaning
6. \* and which have just one character returns.
7. \*/

// 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

// ctrlボタンやaltボタンで同時に押された現在の文字と，その文字のASCIIコード値とを // 一定の変換を行います。最後に，変換された結果の文字が // 読み込みバッファのキューに格納される。

1. // 以下のコードは、まずモードフラグに基づいて、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。
2. do\_self:
3. lea alt\_map,%ebx // alt\_map table address -> ebx (use alt\_map table).
4. testb $0x20,mode /\* alt-gr \*/ // right alt key pressed ?
5. jne 1f // jump to label 1 if yes.
6. lea shift\_map,%ebx // otherwise, use shift\_map table.
7. testb $0x03,mode // any shift key pressed ?
8. jne 1f // jump to label 1 if yes.
9. lea key\_map,%ebx // otherwise, use key\_map table.

1. // これで、使用するマッピングテーブルを選択しました。次に、スキャンコードの値に応じて、マッピングテーブルの対応する文字を // 取得します。マッピングテーブルに対応する文字が // ない場合は、戻ります（turns to none、495行目）。
2. 1: movb (%ebx,%eax),%al // get the corresponding char from mapping table.
3. orb %al,%al // is it a non-zero ASCII code char ?
4. 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。

1. // 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。
2. testb $0x4c,mode /\* ctrl or caps \*/ // ctrl pressed or caps lock ?
3. je 2f // jump to label 2 if not.
4. cmpb $'a,%al // check to see if the char is in range of 'a'-- '}'
5. jb 2f // jum to label 2 if not.
6. cmpb $'},%al
7. ja 2f
8. 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 + ['@'--'\_'] の範囲の文字は

1. // 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る。例えば， // ctrl + 'M' を押すと，対応するキャリッジリターン制御文字が生成される。
2. 2: testb $0x0c,mode /\* ctrl \*/ // ctrl key pressed ?
3. je 3f // jump forward to label 3 if not.
4. cmpb $64,%al // check to see if char is in range [0x40--0x5F].
5. jb 3f // jump to label 3 if not.
6. cmpb $64+32,%al
7. jae 3f
8. subb $64,%al // convert to control char [0x00--0x1f].

1. // 左altキーが同時に押された場合、その文字のビット7が設定される。つまり、拡張文字セットの中で、値が0x7fより大きい文字は、この時点で // 生成することができる。
2. 3: testb $0x10,mode /\* left alt \*/ // left alt pressed ?
3. je 4f // jump to label 4 if not 491 orb $0x80,%al // set bit 7 of the char.

1. // 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。
2. 4: andl $0xff,%eax // only one character
3. xorl %ebx,%ebx
4. call put\_queue
5. none: ret

496

1. /\*
2. \* minus has a routine of it's own, as a 'E0h' before
3. \* the scan code for minus means that the numeric keypad
4. \* slash was pushed.
5. \*/
6. // 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'-'キーに対応していることに注意してください。 // 264行目と365行目をご覧ください。
7. minus: cmpb $1,e0 // e0 flag is set to 0x01 ? (e0 received ?) 503 jne do\_self // jump to normal processing of char if no.
8. movl $'/,%eax // otherwise replace '-' with '/'
9. xorl %ebx,%ebx
10. jmp put\_queue

507

1. /\*
2. \* This table decides which routine to call when a scan-code has been 510 \* gotten. Most routines just call do\_self, or none, depending if
3. \* they are make or break.
4. \*/ // Note that 'make' means a key is pressed, and 'break' means released.
5. key\_table:
6. .long none,do\_self,do\_self,do\_self /\* 00-03 s0 esc 1 2 \*/
7. .long do\_self,do\_self,do\_self,do\_self /\* 04-07 3 4 5 6 \*/
8. .long do\_self,do\_self,do\_self,do\_self /\* 08-0B 7 8 9 0 \*/
9. .long do\_self,do\_self,do\_self,do\_self /\* 0C-0F + ' bs tab \*/
10. .long do\_self,do\_self,do\_self,do\_self /\* 10-13 q w e r \*/
11. .long do\_self,do\_self,do\_self,do\_self /\* 14-17 t y u i \*/
12. .long do\_self,do\_self,do\_self,do\_self /\* 18-1B o p } ^ \*/
13. .long do\_self,ctrl,do\_self,do\_self /\* 1C-1F enter ctrl a s \*/
14. .long do\_self,do\_self,do\_self,do\_self /\* 20-23 d f g h \*/
15. .long do\_self,do\_self,do\_self,do\_self /\* 24-27 j k l | \*/
16. .long do\_self,do\_self,lshift,do\_self /\* 28-2B { para lshift , \*/
17. .long do\_self,do\_self,do\_self,do\_self /\* 2C-2F z x c v \*/
18. .long do\_self,do\_self,do\_self,do\_self /\* 30-33 b n m , \*/
19. .long do\_self,minus,rshift,do\_self /\* 34-37 . - rshift \* \*/
20. .long alt,do\_self,caps,func /\* 38-3B alt sp caps f1 \*/
21. .long func,func,func,func /\* 3C-3F f2 f3 f4 f5 \*/
22. .long func,func,func,func /\* 40-43 f6 f7 f8 f9 \*/
23. .long func,num,scroll,cursor /\* 44-47 f10 num scr home \*/
24. .long cursor,cursor,do\_self,cursor /\* 48-4B up pgup - left \*/
25. .long cursor,cursor,do\_self,cursor /\* 4C-4F n5 right + end \*/
26. .long cursor,cursor,cursor,cursor /\* 50-53 dn pgdn ins del \*/
27. .long none,none,do\_self,func /\* 54-57 sysreq ? < f11 \*/
28. .long func,none,none,none /\* 58-5B f12 ? ? ? \*/
29. .long none,none,none,none /\* 5C-5F ? ? ? ? \*/
30. .long none,none,none,none /\* 60-63 ? ? ? ? \*/
31. .long none,none,none,none /\* 64-67 ? ? ? ? \*/
32. .long none,none,none,none /\* 68-6B ? ? ? ? \*/
33. .long none,none,none,none /\* 6C-6F ? ? ? ? \*/
34. .long none,none,none,none /\* 70-73 ? ? ? ? \*/
35. .long none,none,none,none /\* 74-77 ? ? ? ? \*/
36. .long none,none,none,none /\* 78-7B ? ? ? ? \*/
37. .long none,none,none,none /\* 7C-7F ? ? ? ? \*/
38. .long none,none,none,none /\* 80-83 ? br br br \*/
39. .long none,none,none,none /\* 84-87 br br br br \*/
40. .long none,none,none,none /\* 88-8B br br br br \*/
41. .long none,none,none,none /\* 8C-8F br br br br \*/
42. .long none,none,none,none /\* 90-93 br br br br \*/
43. .long none,none,none,none /\* 94-97 br br br br \*/
44. .long none,none,none,none /\* 98-9B br br br br \*/
45. .long none,unctrl,none,none /\* 9C-9F br unctrl br br \*/
46. .long none,none,none,none /\* A0-A3 br br br br \*/
47. .long none,none,none,none /\* A4-A7 br br br br \*/
48. .long none,none,unlshift,none /\* A8-AB br br unlshift br \*/
49. .long none,none,none,none /\* AC-AF br br br br \*/
50. .long none,none,none,none /\* B0-B3 br br br br \*/
51. .long none,none,unrshift,none /\* B4-B7 br br unrshift br \*/
52. .long unalt,none,uncaps,none /\* B8-BB unalt br uncaps br \*/
53. .long none,none,none,none /\* BC-BF br br br br \*/
54. .long none,none,none,none /\* C0-C3 br br br br \*/
55. .long none,none,none,none /\* C4-C7 br br br br \*/
56. .long none,none,none,none /\* C8-CB br br br br \*/
57. .long none,none,none,none /\* CC-CF br br br br \*/
58. .long none,none,none,none /\* D0-D3 br br br br \*/
59. .long none,none,none,none /\* D4-D7 br br br br \*/
60. .long none,none,none,none /\* D8-DB br ? ? ? \*/
61. .long none,none,none,none /\* DC-DF ? ? ? ? \*/
62. .long none,none,none,none /\* E0-E3 e0 e1 ? ? \*/
63. .long none,none,none,none /\* E4-E7 ? ? ? ? \*/
64. .long none,none,none,none /\* E8-EB ? ? ? ? \*/
65. .long none,none,none,none /\* EC-EF ? ? ? ? \*/
66. .long none,none,none,none /\* F0-F3 ? ? ? ? \*/
67. .long none,none,none,none /\* F4-F7 ? ? ? ? \*/
68. .long none,none,none,none /\* F8-FB ? ? ? ? \*/ 577 .long none,none,none,none /\* FC-FF ? ? ? ? \*/

578

1. /\*
2. \* kb\_wait waits for the keyboard controller buffer to empty.
3. \* there is no timeout - if the buffer doesn't empty, we hang. 582 \*/ 583 kb\_wait:
4. pushl %eax
5. 1: inb $0x64,%al // read status of kbd controller.
6. testb $0x02,%al // test if input buffer is empty (0)
7. jne 1b // jump to label 1 if not empty.
8. popl %eax
9. ret
10. /\*
11. \* This routine reboots the machine by asking the keyboard
12. \* controller to pulse the reset-line low.
13. \*/

// 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

1. // その値に基づいて次の実行を指示します。値が0x1234の場合、BIOSはメモリ検出プロセスをスキップして // ウォームブートプロセスを実行します。値が0の場合は、コールドブート処理を行います。
2. reboot:
3. call kb\_wait // wait controller buffer to empty.
4. movw $0x1234,0x472 /\* don't do memory check \*/
5. movb $0xfc,%al /\* pulse reset and A20 low \*/
6. outb %al,$0x64
7. die: jmp die

### 10.2.3 Information

#### 10.2.3.1 PC/AT keyboard interface programming

PCのマザーボードに搭載されているキーボードインターフェースは、通常の同期式シリアルポートを簡略化したものともいえる専用のインターフェースである。このインターフェース回路はキーボードコントローラーと呼ばれ、キーボードからシリアル通信プロトコルで送られてくるスキャンコードデータを受信する。マザーボード上で使用されているキーボードコントローラーは、インテル社製の8042チップまたはその互換品です。その概略図を図10-7に示します。現在のマザーボードには独立した8042チップは搭載されていませんが、マザーボード上の他の集積回路が互換性のために8042チップの機能をシミュレートします。そのため、キーボードコントローラのプログラミング方法は、現行のマザーボードでも適用可能です。さらに、チップの出力ポートP2は、他の目的にも使用されます。ビット0（P20端子）は，CPUのリセット動作を実現するために使用され，ビット1（P21端子）は，A20信号線をオンにするかどうかを制御するために使用されます。出力ポートのビット1が1の場合は，A20信号線がオン（ストローブ）になり，0の場合はA20信号線がディセーブルになります。

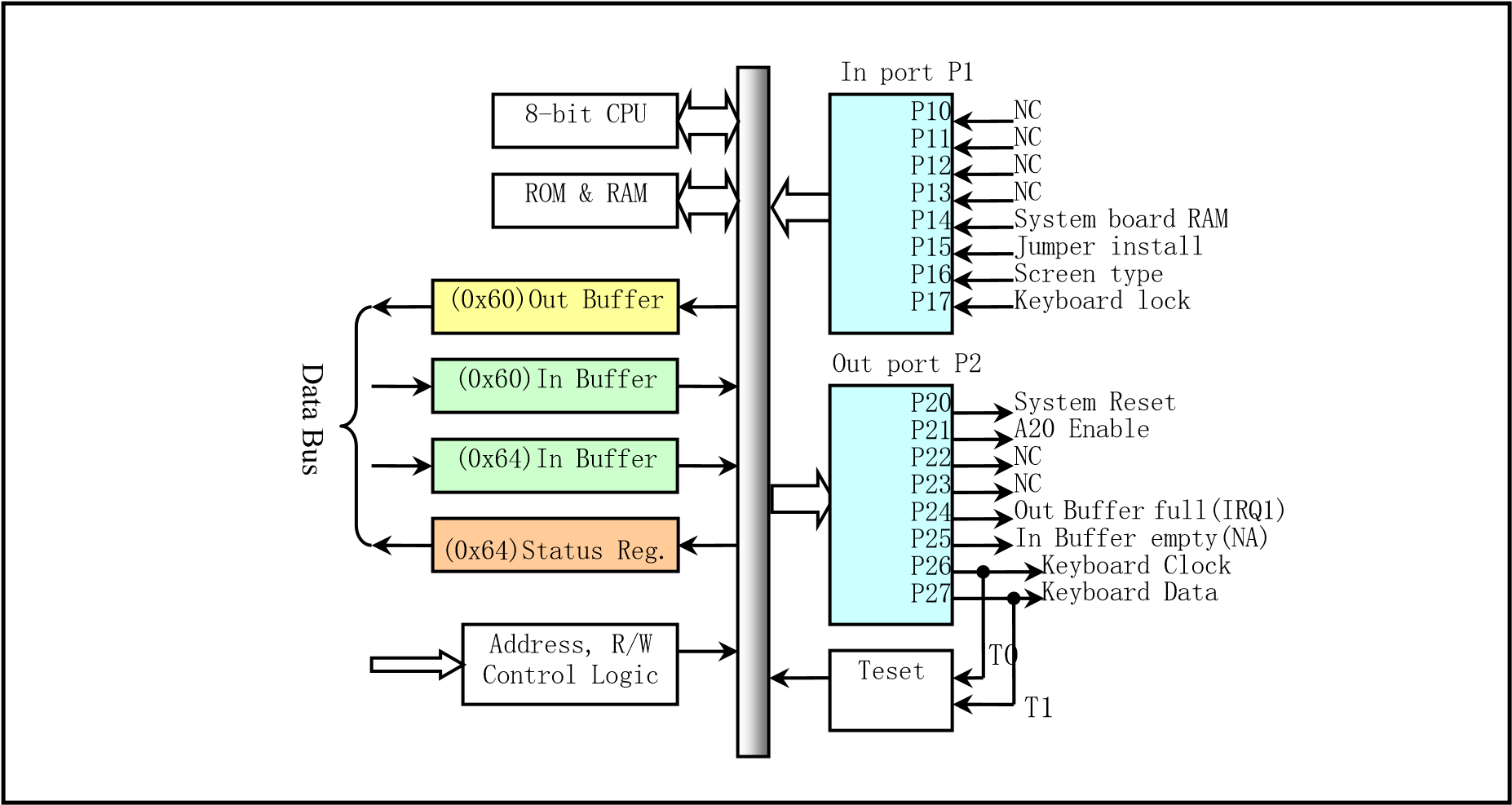


図10-7 キーボードコントローラ804Xの回路図

キーボードコントローラに割り当てられているIOポートの範囲は0x60～0x6fですが、実際にはIBM PC/ATでは0x60と0x64のポートアドレスのみを使用しています（XT互換機では0x61、0x62、0x63）。表10-1を参照してください。また、ポートの読み出し、書き込みの意味が違うので、主に4種類の操作があります。キーボードコントローラのプログラミングには、チップ内のステータスレジスタ、入力バッファ、出力バッファが関係します。

|  |  |  |  |
| --- | --- | --- | --- |
| 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 | データポートまたは  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 |  | このポートは、8255Aの出力ポートB(P2)のアドレスで、8255Aを使用したPC標準キーボード回路のハードウェアリセット処理です。このポートは、受信したスキャンコードに応答するために使用されます。方法としては、まずキーボードを無効にして、すぐにキーボードを再度有効にするというものです。操作しているデータは  Bit 7 =1 キーボードを無効にする、=0 キーボードを有効にする。  Bit 6 =0 Forces the keyboard clock to be low, so the keyboard cannot send any |
|  |  |  | のデータを提供しています。  Bits 5-0 These bits are independent of the keyboard and are used for Programmable Parallel Interface (PPI). |
| 0x64 | Read | ステータス  Register | ポートは8ビットのリードオンリーレジスタで、そのビットフィールドの意味は以下の通りです。  ビット7=1 キーボードからの送信データのパリティエラー（0、奇数パリティであることが望ましい）。  Bit 6=1 受信タイムアウト(キーボード転送ではIRQ1は発生しない)  ビット5=1 送信タイムアウト（キーボードが応答しない）。  Bit 4=0 Inhibit Switch. キーボードインターフェースが禁止されていることを示す。  Bit 3=1 入力バッファに書き込まれたデータは、コマンド（ポート0x64経由）です。  =0 入力バッファに書き込まれたデータは、パラメータ（ポート0x60経由）です。  ビット2 システム・フラグ・ステータス。0 = パワーオンまたはリセット、1 = セルフテスト合格。  ビット 1=1 入力バッファフル（0x60/64 ポートに 8042 用のデータあり）。  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

システムはポート0x60に1バイトを書き込みますが、これはキーボードコマンドを送信するためです。キーボードは、コマンドを受信してから20ms以内に応答する、つまりコマンドレスポンスを返す必要があります。また、一部のコマンドはパラメータ（これもポートに書き込まれます）を追いかける必要があります。コマンドリストを表10-2に示します。なお、すべてのコマンドは、特に指示がなければ、0xfaの応答コード（ACK）で返されます。

|  |  |  |
| --- | --- | --- |
| 表10-2 キーボードコマンド一覧  Command | Has  Params | Description |
| 0xed | Yes | セット/リセットモード表示。開くときは1、閉じるときは0に設定します。パラメータバイトです。  ビット7-3はすべて0として予約されています。  ビット2＝caps-lockキー。  ビット1＝num-lockキー。  Bit 0 = scroll-lock key. |
| 0xee | No | Diagnostic response. The keyboard should return 0xee. |
| 0xef |  | Reserved. |
| 0xf0 | Yes | スキャンコードセットの読み出し／設定を行います。パラメーターバイトは以下の通りです。  0x00 - 現在のスキャンコードセットを選択します。  0x01 - スキャンコードセット1を選択する（PC、PS/2 30などの場合）。  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: |
|  |  | ビット7は0として予約されています。  ビット6-5 遅延値。C=ビット6-5とすると、式:遅延値=(1 + C) \* 250ms; ビット4-0 スキャンコードを連続して送信するレート; B=ビット4-3; A=ビット2-0とすると、式:レート=1 / ((8 + A) \* 2 ^ B \* 0.00417)が成り立ちます。  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 | 1. キーボードのパワーオンリセット操作を行うことをBAT（Basic Assured Test）といいます。操作の流れは 2. The keyboard responds by sending the command 0xfa immediately after receiving the command; 3. The keyboard controller sets the keyboard clock and data lines high; 4. The keyboard begins to perform BAT operations; 5. If it is completed normally, the keyboard sends 0xaa; otherwise it sends 0xfd and stops scanning. |

#### 10.2.3.3 Keyboard Controller Commands

システムは入力バッファ（ポート0x64）に1バイトを書き込み、キーボードコントローラコマンドを送信します。これは1つのパラメータを取ることができますが、表10-3に示すように、パラメータはポート0x60に書き込むことで送信されます。

|  |  |  |
| --- | --- | --- |
| 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 | キーボードコントローラのコマンドバイトを書き込みます。パラメータバイトは (デフォルトは0x5d) ビット7は0として予約されています。  Bit 6 IBM PC互換モード（パリティ、システムスキャンコードへの変換、1バイトのPC BREAKコード）。  Bit 5 PCモード（スキャンコードのパリティチェックなし、システムスキャンコードへの変換なし）。  ビット4はキーボード操作を無効にします（キーボードクロックをLowにします）。  ビット3はオーバーライドを無効にし、キーボードロックの変換には機能しません。  ビット2 システムフラグ；1はコントローラが正常に動作していることを示す。  ビット1は0として予約されています。  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 | キーボードインターフェーステストを初期化します。返されたバイトの意味は  0x00 is error free; |
|  |  | 0x01 キーボードクロックラインがLow（常にLow、Lowスタック）。  0x02 キーボードのクロックラインが高い。  0x03 キーボードのデータラインが低い  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 | キーボードのLEDの状態を制御します。開くときは1、閉じるときは0に設定します。パラメータバイトです。  ビット7-3はすべて0として予約されています。  ビット2＝Caps-lockキー。  ビット1＝Num-lockキー。  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

PCはすべてノンエンコードのキーボードです。キーボードの各キーには左から右へ、上から下へとポジション番号がついており、PC XTとAT機のキーボードのポジションコードは大きく異なります。キーボードのマイクロプロセッサは、ボタンに対応するスキャンコードをシステムに送ります。キーが押されたときにキーボードが出力するスキャンコードは「メイク」スキャンコードと呼ばれ、キーが離されたときには「ブレイク」スキャンコードと呼ばれます。XTキーボードの各キーのスキャンコードを表10-4に示します。

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

キーボードの各ボタンには，それぞれ対応するスキャンコードがバイトの下位7ビット（ビット6～0）に含まれており，最上位ビット（ビット7）はボタンが押されたか離されたかを示します。ビット7=0はスキャン

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、メイクコードとスキャンコードも30（0x1e）で、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。 なお、一部の拡張キーについては、状況が若干異なる。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、表示文字は現在の属性を使って表示メモリに直接書き込まれます。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ったりする。

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

1. /\*
2. \* console.c
3. \*
4. \* This module implements the console io functions 11 \* 'void con\_init(void)'
5. \* 'void con\_write(struct tty\_queue \* queue)'
6. \* Hopefully this will be a rather complete VT102 implementation.
7. \*
8. \* Beeping thanks to John T Kohl.
9. \*
10. \* Virtual Consoles, Screen Blanking, Screen Dumping, Color, Graphics 18 \* Chars, and VT100 enhancements by Peter MacDonald. 19 \*/

20

1. /\*
2. \* 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 \* 割り込みがあります。トラップゲートを使用しているため、キーボード25 \* の割り込みを受けても割り込みが有効にならないことがわかっています。すべてが順調であることを願っています。

26 \*/

27

1. /\*
2. \* Code to check for different video-cards mostly by Galen Hunt,
3. \* <g-hunt@ee.utah.edu>
4. \*/

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

// 初期タスク0のデータと、組み込みアセンブリ関数のマクロ文

// ディスクリプタのパラメータ設定と取得について。

// <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。

// 通信です。

// <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

// ディスクタイプ（HD\_TYPE）のオプションです。

// <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。

// カーネルのよく使う機能

// <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。

// マクロの組み込みアセンブラの形式です。

// <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。

// ディスクリプタ/割込みゲートなどを変更することが定義されています。

// <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

// セグメント・レジスタ・オペレーションのための

// <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。

1. // <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。
2. #include <linux/sched.h>
3. #include <linux/tty.h>
4. #include <linux/config.h>
5. #include <linux/kernel.h>

37

1. #include <asm/io.h>
2. #include <asm/system.h>
3. #include <asm/segment.h>

41

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

44

// 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。

1. // 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。
2. #define DEF\_TERMIOS \
3. (struct termios) { \
4. ICRNL, \
5. OPOST | ONLCR, \
6. 0, \
7. IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, \
8. 0, \
9. INIT\_C\_CC \
10. }

54

55

1. /\*
2. \* These are set up by the setup-routine at boot-time:
3. \*/

// boot/setup.sのコメントや、セットアッププログラムが読み込んで保存したシステムパラメータテーブルを参照してください。

59

1. #define ORIG\_X (\*(unsigned char \*)0x90000) // original cursor colum no.
2. #define ORIG\_Y (\*(unsigned char \*)0x90001) // original cursor row no.
3. #define ORIG\_VIDEO\_PAGE (\*(unsigned short \*)0x90004) // current video page.
4. #define ORIG\_VIDEO\_MODE ((\*(unsigned short \*)0x90006) & 0xff) // display mode.
5. #define ORIG\_VIDEO\_COLS (((\*(unsigned short \*)0x90006) & 0xff00) >> 8) // screen colums.
6. #define ORIG\_VIDEO\_LINES ((\*(unsigned short \*)0x9000e) & 0xff) // screen rows. 66 #define ORIG\_VIDEO\_EGA\_AX (\*(unsigned short \*)0x90008) // current function.
7. #define ORIG\_VIDEO\_EGA\_BX (\*(unsigned short \*)0x9000a) // disp mem size, color mode.
8. #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。

70 #define VIDEO\_TYPE\_MDA 0x10 /\* モノクロ・テキスト表示 \*/ 71 #define VIDEO\_TYPE\_CGA 0x11 /\* CGA 表示 \*/ 72 #define VIDEO\_TYPE\_EGAM 0x20 /\* モノクロモードの EGA/VGA \*/ 73 #define VIDEO\_TYPE\_EGAC 0x21 /\* カラーモードの EGA/VGA \*/ 74

75 #define NPAR 16 // エスケープシーケンスの引数の最大数です。

76

77 int NR\_CONSOLES = 0; // システムがサポートするバーチャルコンソールの数。 78

79 extern void keyboard\_interrupt(void); // キーボード割り込みハンドラ（keyboard.S内）。

80

1. // 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。
2. static unsigned char video\_type; /\* Type of display being used \*/
3. static unsigned long video\_num\_columns; /\* Number of text columns \*/
4. static unsigned long video\_mem\_base; /\* Base of video memory \*/
5. static unsigned long video\_mem\_term; /\* End of video memory \*/ 85 static unsigned long video\_size\_row; /\* Bytes per row \*/
6. static unsigned long video\_num\_lines; /\* Number of test lines \*/
7. static unsigned char video\_page; /\* Initial video page \*/
8. static unsigned short video\_port\_reg; /\* Video register select port \*/
9. 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。

1. // vc\_video\_mem\_startおよびvc\_video\_mem\_endは、現在のバーチャルコンソールで使用されている表示メモリ領域である。
2. static struct {
3. unsigned short vc\_video\_erase\_char; // erase char attributes & char (0x0720)
4. unsigned char vc\_attr; // char attribute.
5. unsigned char vc\_def\_attr; // default attribute.
6. int vc\_bold\_attr; // bold char attribute.
7. unsigned long vc\_ques; // question mark char.
8. unsigned long vc\_state; // state of the escape or control sequence. 99 unsigned long vc\_restate; // next state of escape or control sequence.
9. unsigned long vc\_checkin;
10. unsigned long vc\_origin; /\* Used for EGA/VGA fast scroll \*/
11. unsigned long vc\_scr\_end; /\* Used for EGA/VGA fast scroll \*/
12. unsigned long vc\_pos; // The mem location of the current cursor.
13. unsigned long vc\_x,vc\_y; // current cursor column, row value.
14. unsigned long vc\_top,vc\_bottom; // The top & bottom line nr when scrolling.
15. unsigned long vc\_npar,vc\_par[NPAR]; // escape sequence param nr and array. 107 unsigned long vc\_video\_mem\_start; /\* Start of video RAM \*/ 108 unsigned long vc\_video\_mem\_end; /\* End of video RAM (sort of) \*/ 109 unsigned int vc\_saved\_x; // The saved cursor column number. 110 unsigned int vc\_saved\_y; // The saved cursor row number.
16. unsigned int vc\_iscolor; // The color display flag.
17. char \* vc\_translate; // The character set used.
18. } vc\_cons [MAX\_CONSOLES]; 114
19. // 参照を容易にするために、以下では、現在処理されているコンソールのシンボルを // 上記と同じ意味で定義する。ここで、currconsは、vc\_cons[]構造体を用いた関数の引数の中で、現在の仮想端末の番号を // 示している。
20. #define origin (vc\_cons[currcons].vc\_origin)
21. #define scr\_end (vc\_cons[currcons].vc\_scr\_end)
22. #define pos (vc\_cons[currcons].vc\_pos)
23. #define top (vc\_cons[currcons].vc\_top)
24. #define bottom (vc\_cons[currcons].vc\_bottom)
25. #define x (vc\_cons[currcons].vc\_x)
26. #define y (vc\_cons[currcons].vc\_y)
27. #define state (vc\_cons[currcons].vc\_state)
28. #define restate (vc\_cons[currcons].vc\_restate)
29. #define checkin (vc\_cons[currcons].vc\_checkin)
30. #define npar (vc\_cons[currcons].vc\_npar) 126 #define par (vc\_cons[currcons].vc\_par)
31. #define ques (vc\_cons[currcons].vc\_ques)
32. #define attr (vc\_cons[currcons].vc\_attr)
33. #define saved\_x (vc\_cons[currcons].vc\_saved\_x)
34. #define saved\_y (vc\_cons[currcons].vc\_saved\_y)
35. #define translate (vc\_cons[currcons].vc\_translate)
36. #define video\_mem\_start (vc\_cons[currcons].vc\_video\_mem\_start)
37. #define video\_mem\_end (vc\_cons[currcons].vc\_video\_mem\_end)
38. #define def\_attr (vc\_cons[currcons].vc\_def\_attr)
39. #define video\_erase\_char (vc\_cons[currcons].vc\_video\_erase\_char)
40. #define iscolor (vc\_cons[currcons].vc\_iscolor)

137

1. int blankinterval = 0; // Black screen interval.
2. int blankcount = 0; // Black screen time count.

140

141 static void sysbeep(void); // システムビープ機能。

142

1. /\*
2. \* this is what the terminal answers to a ESC-Z or csi0c 145 \* query (= vt100 response).
3. \*/ // 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

1. // 'ESC [0c') (ESC-Zも同じ働きをします)。端末は、ホストに応答するために、 // 以下のシーケンスを送信する。このシーケンス(すなわち'ESC [?1; 2c')は、端末が高度なビデオ機能を持つVT100互換の // 端末であることを示す。
2. #define RESPONSE "\033[?1;2c" 148
3. // 使用するキャラクターセットを定義します。上部は通常の7ビットのASCIIコードで、これは米国の文字セットです。下部は、VT100端末装置のライン文字、つまりチャートラインを示す文字セットに対応する。
4. static char \* translations[] = {
5. /\* normal 7-bit ascii \*/
6. " !\"#$%&'()\*+,-./0123456789:;<=>?"
7. "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^\_"
8. "`abcdefghijklmnopqrstuvwxyz{|}~ ",
9. /\* vt100 graphics \*/
10. " !\"#$%&'()\*+,-./0123456789:;<=>?"
11. "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^ "
12. "\004\261\007\007\007\007\370\361\007\007\275\267\326\323\327\304"
13. "\304\304\304\304\307\266\320\322\272\363\362\343\\007\234\007 "
14. };

160

1. #define NORM\_TRANS (translations[0])
2. #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 - 現在の仮想ターミナル; new\_x - カーソルの列番号; // new\_y - カーソルの行番号。

// 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列の番号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

1. // は、ディスプレイメモリ内の位置posに対応するように更新されます。なお，この関数内のすべての変数は， // 実際には vc\_cons[currcons] 構造体の対応するフィールドであり， // 以下の関数も同様である。
2. /\* NOTE! gotoxy thinks x==video\_num\_columns is ok \*/
3. static inline void gotoxy(int currcons, int new\_x,unsigned int new\_y) 166 {
4. if (new\_x > video\_num\_columns || new\_y >= video\_num\_lines)
5. return;
6. x = new\_x;
7. y = new\_y;
8. 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、 // パラメータで指定されたコンソールがフロントコンソールであるかどうかを判断し、 // 2つの条件が満たされない場合は終了します。

1. // 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。
2. static inline void set\_origin(int currcons)
3. {

// 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

// 直接終了してしまいます。また、フォアグラウンドコンソールでしか操作しないので、 // 現在のコンソールcurrconsがフォアグラウンドコンソールの場合に限り、 // メモリスタートの

1. // スクロール開始線に対応する位置です。
2. if (video\_type != VIDEO\_TYPE\_EGAC && video\_type != VIDEO\_TYPE\_EGAM)
3. return;
4. if (currcons != fg\_console) 179 return;

// そして、12を「ディスプレイ・レジスタ・セレクト・ポート」video\_port\_regに出力すると、ディスプレイ・コントロール・データ・レジスタr12が選択され、そこにスクロール開始アドレスの上位バイトが書き込まれる // というわけです。

// 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 バイトのメモリである。出力値は、デフォルトのディスプレイメモリの開始位置 video\_mem\_base を基準にして動作します // 例えば、EGA/VGA カラーモードの場合、viedo\_mem\_base = physical

1. // メモリアドレス0xb8000。
2. cli();
3. outb\_p(12, video\_port\_reg); // Select data register r12 to output high byte.
4. outb\_p(0xff&((origin-video\_mem\_base)>>9), video\_port\_val);
5. outb\_p(13, video\_port\_reg); // Select r13 to output the low byte.
6. outb\_p(0xff&((origin-video\_mem\_base)>>1), video\_port\_val);
7. sti();
8. }

187

//// コンテンツが1行分上にスクロールします。

// 画面スクロールウィンドウを1行下に移動し、画面スクロール領域の下部に表示される新しい行に // スペース文字を追加します。スクロール領域は1行よりも大きくなければならない。スクロール操作の原理については、10.3.3.2節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

// が最下段の行番号以上であれば、ローリング動作の条件を // 満たしていないことになります。また、EGA/VGAカードでは、スクロールするための画面上の範囲（エリア）を // 指定することができるが、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行うことができない。

1. // ディスプレイタイプがEGAとMDAの場合、別々に処理する。ディスプレイタイプがEGAの場合は、 // フルスクリーンウィンドウの移動とイントラエリアウィンドウの移動にも分けられます。ここでは、まずディスプレイカードがEGA/VGAタイプの場合を処理する。
2. if (bottom<=top)
3. return;
4. if (video\_type == VIDEO\_TYPE\_EGAC || video\_type == VIDEO\_TYPE\_EGAM)
5. {

// 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

1. // カーソルの位置と、画面端の文字ポインタscr\_endの位置も、 // 追跡される。最後に、新しいスクリーン・ウィンドウ・メモリの開始位置の原点がディスプレイ・コントローラに // 書き込まれる。
2. if (!top && bottom == video\_num\_lines) {
3. origin += video\_size\_row;
4. pos += video\_size\_row;
5. 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。そして、画面メモリのデータが移動した状況に応じて、現在の画面に対応するスタートポインタ、カーソル位置のポインタ、画面の最後に対応するメモリポインタscr\_endを再調整する。

//

// この組み込みアセンブリコードは、まず、（画面文字 // 行番号 - 1）行に対応するメモリデータを表示メモリ開始位置 video\_mem\_start に移動し、その後のメモリ位置にスペース（消去）文字データを // 1行追加します。

1. // %0 - eax (消去文字 + 属性); %1 - ecx ((画面の行数 -1) // 文字数 / 2 に対応、長文での移動); %2 - edi (ディスプレイメモリの // 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 origin). // 移動の方向。[edi] -> [esi], ecxロングワードの移動。
2. if (scr\_end > video\_mem\_end) {
3. \_\_asm\_\_("cld\n\t" // clear direction.
4. "rep\n\t" // repeat move data.
5. "movsl\n\t"
6. "movl \_video\_num\_columns,%1\n\t"
7. "rep\n\t" // fill a line of spaces.
8. "stosw"
9. ::"a" (video\_erase\_char),
10. "c" ((video\_num\_lines-1)\*video\_num\_columns>>1),
11. "D" (video\_mem\_start),
12. "S" (origin)
13. :"cx","di","si");
14. scr\_end -= origin-video\_mem\_start;
15. pos -= origin-video\_mem\_start;
16. origin = video\_mem\_start;

// 調整後の画面でのメモリポインタscr\_endが、表示メモリvideo\_mem\_endの終端を超えていない場合は、単純に消去行（スペース文字）を // 新しい行に記入する。

// %0 - eax (消去文字 + 属性); %1 - ecx (画面の行数); %2 - edi

1. //（最終行の先頭にある対応するメモリー位置）。
2. } else {
3. \_\_asm\_\_("cld\n\t"
4. "rep\n\t" // fill a line of spaces.
5. "stosw"
6. ::"a" (video\_erase\_char),
7. "c" (video\_num\_columns),
8. "D" (scr\_end-video\_size\_row)
9. :"cx","di");
10. }

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ことを意味します。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

// を消去します。このとき、指定された行頭から行末までの画面の全行に対応する表示メモリデータがそのまま1行分上に移動し、新たに現れた行に消去文字が埋められます。

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

1. // 行から下の行まで）; %2 - edi（上の行が位置するメモリの場所）; %3 // - esi（上+1行が位置するメモリの場所）。
2. } else {
3. \_\_asm\_\_("cld\n\t"
4. "rep\n\t" // repeat from top+1 to bottom.
5. "movsl\n\t"
6. "movl \_video\_num\_columns,%%ecx\n\t"
7. "rep\n\t" // fill blank char in new line.
8. "stosw"
9. ::"a" (video\_erase\_char),
10. "c" ((bottom-top-1)\*video\_num\_columns>>1),
11. "D" (origin+video\_size\_row\*top),
12. "S" (origin+video\_size\_row\*(top+1))
13. :"cx","di","si");
14. } 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

1. // 表示メモリの範囲を超えている（つまり、ポインタが自動的にスクロールする）ので、画面内容に対応するメモリを表示メモリの範囲とは別に処理しない。処理方法は EGA 非全画面移動と全く同じである。237 else /\* EGA/VGAではありません \*/
2. {
3. \_\_asm\_\_("cld\n\t"
4. "rep\n\t"
5. "movsl\n\t"
6. "movl \_video\_num\_columns,%%ecx\n\t"
7. "rep\n\t"
8. "stosw"
9. ::"a" (video\_erase\_char),
10. "c" ((bottom-top-1)\*video\_num\_columns>>1),
11. "D" (origin+video\_size\_row\*top),
12. "S" (origin+video\_size\_row\*(top+1))
13. :"cx","di","si");
14. }
15. }

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、カウントダウンの2行目の最後の文字から最後の行までをコピーし、次にカウントダウンの3行目の文字を最後の2行目にコピーする、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

1. // 画面の端に対応する表示メモリポインタscr\_endは、実際の表示メモリの端を超えていないので、ここでは通常のメモリデータの移動のみを処理すればよい。
2. if (bottom <= top)
3. return;
4. if (video\_type == VIDEO\_TYPE\_EGAC || video\_type == VIDEO\_TYPE\_EGAM)
5. {

// %0 - eax (消去文字 + 属性); %1 - ecx (対応する長い単語の数)

// 上から下への行数-1); %2 - edi (ウィンドウの右下隅にある // 最後の長い単語の位置); %3 - Esi (ウィンドウのカウントダウンの 2 行目にある // 最後の長い単語の位置).

1. // 移動の方向。[esi] -> [edi], move ecx long words.
2. \_\_asm\_\_("std\n\t" // set direction flag!!
3. "rep\n\t"
4. "movsl\n\t"
5. "addl $2,%%edi\n\t" /\* %edi has been decremented by 4 \*/
6. "movl \_video\_num\_columns,%%ecx\n\t"
7. "rep\n\t" // fill in blanks at above line.
8. "stosw"
9. ::"a" (video\_erase\_char),
10. "c" ((bottom-top-1)\*video\_num\_columns>>1),
11. "D" (origin+video\_size\_row\*bottom-4),
12. "S" (origin+video\_size\_row\*(bottom-1)-4)
13. :"ax","cx","di","si");
14. }
15. // EGAディスプレイタイプでない場合は、次のようにしてください（同上）。
16. else /\* Not EGA/VGA \*/
17. {
18. \_\_asm\_\_("std\n\t"
19. "rep\n\t"
20. "movsl\n\t"
21. "addl $2,%%edi\n\t" /\* %edi has been decremented by 4 \*/
22. "movl \_video\_num\_columns,%%ecx\n\t"
23. "rep\n\t"
24. "stosw"
25. ::"a" (video\_erase\_char),
26. "c" ((bottom-top-1)\*video\_num\_columns>>1),
27. "D" (origin+video\_size\_row\*bottom-4),
28. "S" (origin+video\_size\_row\*(bottom-1)-4)
29. :"ax","cx","di","si");
30. }
31. }

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

1. // 1行分の文字に対応する）。それ以外の場合は、画面ウィンドウの内容を // 1行上に移動させる必要があります。関数名 lf (line feed) は、処理制御文字 LF を指す。
2. static void lf(int currcons)
3. {
4. if (y+1<bottom) {
5. y++;
6. pos += video\_size\_row; // Plus the nr of bytes occupied by one line.
7. return;
8. }
9. 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。

1. // カーソルが画面の1行目にない場合は、カーソルの現在行変数y--を直接変更し、カーソルに対応する表示メモリ位置posを調整し、 // 画面上の文字の1行に対応するメモリ長のバイト数を差し引く。// そうでなければ、画面ウィンドウの内容を1行下に移動する必要がある。関数名ri(reverse index)は、制御文字RIまたはエスケープシーケンス "ESC M "を指す。
2. static void ri(int currcons)
3. {
4. if (y>top) {
5. y--;
6. pos -= video\_size\_row; // Minus the nr of bytes occupied by one line.
7. return;
8. }
9. 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

1. // \*2は、カーソルのある列までの行の0列が占めるメモリバイトの長さです。関数名cr(carriage return)は、処理される制御文字を示す。
2. static void cr(int currcons)
3. {
4. pos -= x<<1; // bytes occupied by col 0 to col of cursor.
5. x=0;
6. }

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。

1. // カーソルが0列でない場合、カーソルはメモリの位置pos（画面上の1文字に相当）に対応する2バイトでバックされ、その後、現在のカーソルの列が1だけデクリメントされ、カーソルの位置にある文字が消去される。
2. static void del(int currcons)
3. {
4. if (x) {
5. pos -= 2;
6. x--;
7. \*(unsigned short \*)pos = video\_erase\_char;
8. }
9. }

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

カーソル位置に関連付けられた // 制御シーケンスが実行され、 // 表示された文字の一部または全部が消去され、 // 文字または行が消去されてもカーソル位置は変更されません。

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 // - カーソルを画面の下まで削除することを意味し、 1 - カーソルまでの画面を削除することを意味し、 // 2 - 画面全体を削除することを意味する。

// 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。

1. // 上記の制御シーケンスを実行します。端末制御コマンドの紹介は、プログラムリストの後にある10.3.3.3節を // 参照してください。よく使われるエスケープシーケンスや制御シーケンスは、 // 本書の付録3に記載されています。
2. static void csi\_J(int currcons, int vpar)
3. {
4. long count \_\_asm\_\_("cx"); // set to use register variable.
5. long start \_\_asm\_\_("di"); 328
6. // まず、上記の3つのケースに合わせて、削除する文字数と、削除を開始するディスプレイメモリの位置を設定します。
7. switch (vpar) {
8. case 0: /\* erase from cursor to end of display \*/
9. count = (scr\_end-pos)>>1;
10. start = pos;
11. break;
12. case 1: /\* erase from start to cursor \*/
13. count = (pos-origin)>>1;
14. start = origin;
15. break;
16. case 2: /\* erase whole display \*/
17. count = video\_num\_columns \* video\_num\_lines;
18. start = origin; 341 break;
19. default:
20. return;
21. }

// その後、消去文字を使って、文字が消えた場所を埋める。

1. // %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)。
2. \_\_asm\_\_("cld\n\t"
3. "rep\n\t"
4. "stosw\n\t"
5. ::"c" (count),
6. "D" (start),"a" (video\_erase\_char)
7. :"cx","di");
8. }

352

//// カーソル位置に関連するコンテンツの一部を1行で削除します。

// この関数は，以下の条件に従って，カーソルのある行の一部または全部の文字を消去します。

// パラメータを指定します。消去操作は，他の文字に影響を与えることなく，画面上の文字を // 消去する。消去された文字は破棄されます。文字や行を消去しても、カーソルの位置は変わりません。

1. // ANSIのエスケープシーケンス：'ESC [ Ps K' (Ps = 0で行末まで，1で先頭から削除， // 2で行全体を削除)。関数のパラメータvparは、Psに対応しています。
2. static void csi\_K(int currcons, int vpar)
3. {
4. long count \_\_asm\_\_("cx");
5. long start \_\_asm\_\_("di"); 357
6. // まず、制御シーケンスの3つの条件に応じて、削除する文字数と、削除を開始するディスプレイメモリの位置を設定します。
7. switch (vpar) {
8. case 0: /\* erase from cursor to end of line \*/
9. if (x>=video\_num\_columns)
10. return;
11. count = video\_num\_columns-x;
12. start = pos;
13. break;
14. case 1: /\* erase from start of line to cursor \*/
15. start = pos - (x<<1);
16. count = (x<video\_num\_columns)?x:video\_num\_columns;
17. break;
18. case 2: /\* erase whole line \*/
19. start = pos - (x<<1);
20. count = video\_num\_columns; 372 break; 373 default:
21. return;
22. }

// その後、消去文字を使って、文字が消えた場所を埋める。

1. // %0 - ecx (delete count); %1 -edi (delete address start); %2 -eax (fill erase character).
2. \_\_asm\_\_("cld\n\t"
3. "rep\n\t"
4. "stosw\n\t"
5. ::"c" (count),
6. "D" (start),"a" (video\_erase\_char)
7. :"cx","di");
8. }

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、パラメータに従って、文字表示の属性を設定する。今後、端末に送られるすべての文字は、 // 制御シーケンスが文字表示の属性をリセットするために再び使用されるまで、 // ここで指定された属性を使用する。

// 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 - アンダーライン、5 - フラッシュ、7 - リバース、22 - ノンボールド、24 - アンダーラインなし、25 - フリッカーなし、 // 27 - ノーマル ;30--38 - 前景色の設定、39 - デフォルトの前景色（ホワイト）。

1. // 40--48 - 背景色の設定; 49 - デフォルトの背景色（黒）。
2. void csi\_m(int currcons )
3. {
4. int i; 387

// 制御シーケンスは，複数の異なるパラメータを持つことができます。パラメータは， // 配列 par[] に格納されます．以下のコードは，受信したパラメータ npar の数に応じて，各パラメータ Ps を循環的に処理します．

// Ps = 0 の場合、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は、 // デフォルトの属性 def\_attr に設定されます。初期化時、def\_attr は 0x07 // （黒地に白）に設定されている。

// Ps = 1の場合、後に表示される文字属性が太字またはハイライトに設定される。// 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、 // モノクロディスプレイの場合は、文字に下線が引かれる。

// 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

1. // 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 // 前景色が反転されます．前景色が背景色と同じ場合は， // 前景色が 1 つ増え，もう一方の色が使われます．
2. for (i=0;i<=npar;i++)
3. switch (par[i]) {
4. case 0: attr=def\_attr;break; /\* default \*/
5. case 1: attr=(iscolor?attr|0x08:attr|0x0f);break; /\* bold \*/
6. /\*case 4: attr=attr|0x01;break;\*/ /\* underline \*/
7. case 4: /\* bold \*/
8. if (!iscolor)
9. attr |= 0x01; // Mono is underlined.
10. else
11. { /\* check if forground == background \*/
12. if (vc\_cons[currcons].vc\_bold\_attr != -1)
13. attr = (vc\_cons[currcons].vc\_bold\_attr&0x0f)|(0xf0&(attr));
14. else
15. { short newattr = (attr&0xf0)|(0xf&(~attr)); 402 attr = ((newattr&0xf)==((attr>>4)&0xf)? 403 (attr&0xf0)|(((attr&0xf)+1)%0xf):
16. newattr);
17. }
18. }
19. break;

// Ps = 5の場合、現在のバーチャルコンソールに続いて表示される文字が // 点滅するように設定され、つまり属性バイトのビット7が1に設定されます。

// Ps = 7の場合、その後に表示される文字は、前景色と背景色が入れ替わるという、逆の設定になります。

// Ps = 22 の場合、後続の文字のハイライト表示をキャンセルする（太字表示をキャンセルする）。// Ps = 24 の場合、モノクロ表示では後続の文字の下線がキャンセルされ、 // カラー表示では緑がキャンセルされる。

// 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。

// Ps = 27の場合は、後続の文字の反転がキャンセルされる。

// Ps = 39 の場合、それ以降の文字の前景色は、デフォルト（白）にリセットされます。

1. // Ps = 49 の場合、それ以降の文字の背景色は、デフォルト（黒）にリセットされます。
2. case 5: attr=attr|0x80;break; /\* blinking \*/
3. case 7: attr=(attr<<4)|(attr>>4);break; /\* negative \*/
4. case 22: attr=attr&0xf7;break; /\* not bold \*/
5. case 24: attr=attr&0xfe;break; /\* not underline \*/
6. case 25: attr=attr&0x7f;break; /\* not blinking \*/ 413 case 27: attr=def\_attr;break; /\* positive image \*/
7. case 39: attr=(attr & 0xf0)|(def\_attr & 0x0f); break;
8. case 49: attr=(attr & 0x0f)|(def\_attr & 0xf0); break;
9. // 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。// Ps=30〜37の場合は前景色が設定され、Ps=40〜47の場合は背景色が設定される。色の値については、 // プログラムリストの後の説明を参照してください。
10. default:
11. if (!can\_do\_colour)
12. break;
13. iscolor = 1;
14. if ((par[i]>=30) && (par[i]<=38)) // foreground color.
15. attr = (attr & 0xf0) | (par[i]-30);
16. else /\* Background color \*/
17. if ((par[i]>=40) && (par[i]<=48)) // background color.
18. attr = (attr & 0x0f) | ((par[i]-40)<<4);
19. else
20. break;
21. }
22. }

429

//// ディスプレイカーソルを設定します。

1. // コントローラのカーソルの表示位置は、カーソルのメモリポジション // pos に応じて設定されます。
2. static inline void set\_cursor(int currcons)
3. {
4. // 表示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、 // 黒画面操作の遅延カウント値を元に戻す必要があります。また、カーソルを表示するコンソールは、 // フォアグラウンドコンソールでなければなりません。
5. blankcount = blankinterval; // Resets the count of the black screen.
6. if (currcons != fg\_console) 434 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

1. // ハイバイト（9ビット右に移動すると、ハイバイトがローバイトに移動し、 // 2で割られる）。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。その後、インデックスレジスタを使って // r15 を選択し、カーソルの現在位置のローバイトを書き込みます。
2. cli();
3. outb\_p(14, video\_port\_reg);
4. outb\_p(0xff&((pos-video\_mem\_base)>>9), video\_port\_val);
5. outb\_p(15, video\_port\_reg);
6. outb\_p(0xff&((pos-video\_mem\_base)>>1), video\_port\_val);
7. sti();
8. }

442

// カーソルを隠す。

1. // 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。
2. static inline void hide\_cursor(int currcons)
3. {

// 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

1. // ハイバイト）を選択し、カーソル位置のハイバイトを書き込みます（9ビット右に移動するということは、ハイバイトをローバイトに移動し、2で割るということです）。その後、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。
2. outb\_p(14, video\_port\_reg);
3. 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

1. 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、レスポンスシーケンスをリードバッファキューに入れ、copy\_to\_cooked()関数で処理して、 // 補助（セカンダリ）キューに入れるというものです。
2. static void respond(int currcons, struct tty\_struct \* tty)
3. {
4. char \* p = RESPONSE; // defined on line 147 ('ESC [?1; 2c').

454

1. cli();
2. while (\*p) { // put response sequence into the read queue.
3. PUTCH(\*p,tty->read\_q); // put one by one. include/linux/tty.h, line 46.
4. p++;
5. }
6. sti();
7. 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。

// カーソルの先頭にあるすべての文字を1スペース右に移動し、 // カーソルに消去文字を挿入します。

1. 464 static void insert\_char(int currcons) 465 {。
2. int i=x;
3. unsigned short tmp, old = video\_erase\_char; // erase char (with attribute).
4. unsigned short \* p = (unsigned short \*) pos; // memory position of the cursor.

469

1. while (i++<video\_num\_columns) {
2. tmp=\*p;
3. \*p=old;
4. old=tmp;
5. p++;
6. }
7. }

477

//// カーソル位置に行を挿入します。

1. // 画面のウィンドウを、カーソルのある行からウィンドウの下まで // スクロールします。カーソルは新しい空の行になります。
2. static void insert\_line(int currcons)
3. {
4. int oldtop,oldbottom; 481
5. // まず、スクリーンウィンドウのスクロール開始行「上」と最終行「下」の値を保存し、 // カーソルがある行から1行下に画面の内容をスクロールします。最後に、スクロール開始行の「上」と最終行の「下」の値を // 元に戻します。
6. oldtop=top;
7. oldbottom=bottom;
8. top=y; // set start and end lines of the scroll screen.
9. bottom = video\_num\_lines;
10. scrdown(currcons); // the screen content scrolls down one line
11. top=oldtop;
12. bottom=oldbottom;
13. }

490

//// 文字を削除します。

1. // 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 // スペース1つ分だけ左にシフトする。
2. static void delete\_char(int currcons)
3. {
4. int i;
5. unsigned short \* p = (unsigned short \*) pos;

495

1. // カーソルの現在の列位置xが画面の右端の列を超えていれば返す。// そうでなければ、カーソルの右文字から行末までのすべての文字を // 1スペースだけ左にシフトします。そして、最後の文字で消去文字を埋める。
2. if (x>=video\_num\_columns)
3. return;
4. i = x;
5. while (++i < video\_num\_columns) { // shifted to the left by one space.
6. \*p = \*(p+1);
7. p++;
8. }
9. \*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。

1. // 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。
2. static void delete\_line(int currcons)
3. {
4. int oldtop,oldbottom; 509
5. // まず、画面スクロールの開始行 'top' と最終行 'bottom' を保存し、 // カーソルのある行から 1 行分上に画面内容をスクロールします。最後に、画面スクロールの開始行 'top' と最終行 'bottom' の // 元の値を復元します。
6. oldtop=top;
7. oldbottom=bottom;
8. top=y; // set start and end lines of the scroll screen.
9. bottom = video\_num\_lines;
10. scrup(currcons); // the screen content scrolls up one line.
11. top=oldtop;
12. bottom=oldbottom;
13. }

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

1. // 現在のカーソルの位置に Pn は挿入される文字数で，デフォルトは 1 である。 // カーソルは，最初に挿入されたスペース文字の位置に留まる。カーソルと // 右の境界にある文字は右に移動し、右の境界を超えた文字は失われます。// エスケープシーケンスのパラメータ nr = Pn。
2. static void csi\_at(int currcons, unsigned int nr)
3. {
4. // 挿入文字数が1行の文字数より多い場合は、1行の文字数に // 切り捨てられます。挿入文字数nrが0の場合は、1文字が // 挿入されます。その後、周期的にnr個のスペース文字を挿入する。
5. if (nr > video\_num\_columns)
6. nr = video\_num\_columns;
7. else if (!nr)
8. nr = 1;
9. while (nr--)
10. insert\_char(currcons);
11. }

528

//// カーソル位置に nr 行を挿入します。

1. // ANSIのエスケープシーケンスを扱う：'ESC [ Pn L'. この制御シーケンスは、カーソル位置に1つ以上の空行を挿入する。操作が完了してもカーソルの位置は変わらない。// 空行が挿入されると、カーソルの下のスクロール領域の行が下に移動する。表示ページの外にスクロールしている行は // 失われる。エスケープシーケンスのパラメータ nr = Pn。
2. static void csi\_L(int currcons, unsigned int nr)
3. {

// 挿入行数が画面の最大行数より大きい場合は、画面の行数まで // 切り捨てられます。挿入行数nrが0の場合は、1行が // 挿入されます。

1. その後、周期的に nr 本の空行を挿入する。531 if (nr > video\_num\_lines)
2. nr = video\_num\_lines;
3. else if (!nr)
4. nr = 1;
5. while (nr--)
6. insert\_line(currcons);
7. }

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

1. // 最後の左シフト文字と同じになるはずですが、ここでは簡略化して、 // 文字のデフォルトのプロパティ（黒背景白前景スペース0x0720）だけを使って // スペース文字を設定しています。パラメータ nr = Pn をエスケープシーケンスで指定します。
2. static void csi\_P(int currcons, unsigned int nr)
3. {
4. // 削除文字数が1行の文字数より多い場合は、1行の文字数に // 切り捨てられ、削除文字数nrが0の場合は、1文字が削除されます。そして、カーソルで指定された文字数nrを繰り返し削除します。
5. if (nr > video\_num\_columns)
6. nr = video\_num\_columns;
7. else if (!nr)
8. nr = 1;
9. while (nr--)
10. delete\_char(currcons);
11. }

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

// スクロールエリアで削除された行の下の行が上に移動し、 // 下の行に1つの空白行が追加されます。Pn が表示ページに残っている行数よりも大きい場合、 // このシーケンスはこれらの残っている行を削除するだけで、 // スクロールエリアの外側では動作しません。

1. // エスケープシーケンスでのパラメータnr = Pn。
2. static void csi\_M(int currcons, unsigned int nr)
3. {

// 削除した行数が画面の最大行数よりも多い場合。

1. // 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ていきます。
2. if (nr > video\_num\_lines)
3. nr = video\_num\_lines;
4. else if (!nr)
5. nr=1;
6. while (nr--)
7. delete\_line(currcons);
8. }

558

1. //// 現在のカーソル位置を保存します。
2. static void save\_cur(int currcons)
3. {
4. saved\_x=x;
5. saved\_y=y;
6. }

564

1. //// 保存したカーソル位置を復元します。
2. static void restore\_cur(int currcons)
3. {
4. gotoxy(currcons,saved\_x, saved\_y);
5. }

569

570

// 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または制御シーケンスを解決するために使用されます。ESnormalは初期のエントリー状態であり、 // エスケープシーケンスや制御シーケンスが処理されたときの状態でもある。

// ESnormal - 初期のノーマル状態であることを示す。この時、通常の表示文字を受信した場合は、その文字が直接画面に表示され、 // 制御文字（キャリッジリターンなど）を受信した場合は、カーソル位置が設定されます。// エスケープシーケンスや制御シーケンスが処理されたばかりの時は、プログラムはこの状態に // 戻ります。

// 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。

//を受信しました。この状態で'['文字を受信した場合、それはエスケープシーケンスガイド

// 受け取った文字は、ESsquareにジャンプして処理されます。

// をエスケープシーケンスとして使用しています。文字セットエスケープシーケンス「ESC ('と'ESC )'」の選択については、 // 別のステートESsetgraphを使用して処理します。デバイス制御の文字列シーケンスである'ESC P'は、 // 別の状態である ESsetterm で処理される。

// 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。

// 制御シーケンスを受信したので、この状態ではゼロ初期化を行う

// パラメータ配列par[].この時に「[]」という文字が出てきたら、それは

// キーボード機能で送信されるシーケンスである「ESC [[]」シーケンスを受信した場合

// キーなので、Esfunckeyにジャンプして処理します。そうでなければ、制御シーケンスのパラメータを // 受け取る準備をする必要があるので、状態Esgetparsを設定して、シーケンスパラメータ文字を受け取って保存する状態に // 直接入ります。

// ESgetpars - この状態は、この時点で制御シーケンスのパラメータ値を // 受け取ることを示しています。引数は10進数で表現されており、 // 受信した数値文字を数値に変換して、par[]配列に保存します。

// セミコロン「;」を受信した場合は、この状態のまま、受信したパラメータ値を

は， // 次のデータ項目 par[] に保存されます。それが数字でもセミコロンでもない場合は、 // すべてのパラメータが取得されたことを示し、 // 処理するために状態 ESgotpars に移動します。

// 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

// 受信したエンディングキャラクターに応じて、対応する制御シーケンスを処理することができます。

この状態では「//」となります。しかし、処理の前に、ESsquareの状態で「？」を受け取った場合、この

// シーケンスは，ターミナルデバイスのプライベートシーケンスです．このカーネルでは

このシーケンスの // 処理で、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が実行され、シーケンスの処理後に Esnormal // 状態に復帰します。

// ESfunckey - キーボードのファンクションキーから、 // 表示する必要のないシーケンスを受信したことを示します。そのため、通常の状態ESnormalに戻ります。

// ESsetterm - DCS(Device Control String Sequence)状態であることを示す。このとき

// 時、文字'S'を受信すると、初期の表示文字属性に // 戻る。受信した文字が'L'または'l'の場合、折れ線表示モードのオン/オフを行う。

// 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。

//は、G0とG1がそれぞれ使用する文字セットを指定するために使用します。このとき、もし

1. // 文字'0'を受信した場合、グラフィックのキャラクタセットはG0とG1が選択され、 // 受信した文字が'B'の場合、通常のASCIIのキャラクタセットはG0とG1が選択される。
2. enum { ESnormal, ESesc, ESsquare, ESgetpars, ESgotpars, ESfunckey,
3. ESsetterm, ESsetgraph };

573

//// コンソールは機能を書き込みます。

// 文字は端末のttyライトバッファキューから取り出され、それぞれの

// 文字を表示します。制御文字やエスケープ、制御シーケンスであれば、カーソルの位置決めや文字の削除などの制御処理が行われ、通常の文字が // カーソルに直接表示されます。

1. // 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。
2. void con\_write(struct tty\_struct \* tty)
3. {
4. int nr;
5. char c;
6. 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

1. // シーケンスが書き込みキューに置かれている間の文字数を読み取ることができるので、 // 前回関数が終了したときには、「状態」はエスケープシーケンスや制御シーケンスを処理している別の状態になっているかもしれません。
2. currcons = tty - tty\_table;
3. if ((currcons>=MAX\_CONSOLES) || (currcons<0))
4. panic("con\_write: illegal tty");

583

1. nr = CHARS(tty->write\_q); // get nr of chars in the write queue.
2. while (nr--) {
3. if (tty->stopped)
4. break;
5. GETCH(tty->write\_q,c); // get one character.
6. if (c == 24 || c == 26) // Control chars: Cancel or Substitute.
7. 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、現在のマップされたキャラクタセットから直接取り出され、現在のカーソルがあるディスプレイメモリの位置に置かれ、つまりそのキャラクタが直接表示されます。

// その後、カーソル位置を1文字分右に移動します。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

1. // は、次の行の最初の列に移動します。そして、カーソル位置に対応するメモリポインタposを調整します。その後、文字が表示メモリposに書き込まれ、 // カーソルが1列分右に移動し、posもそれに応じて2バイト移動します。
2. case ESnormal:
3. if (c>31 && c<127) { // normal display char.
4. if (x>=video\_num\_columns) { // change line?
5. x -= video\_num\_columns;
6. pos -= video\_size\_row;
7. lf(currcons);
8. }
9. \_\_asm\_\_("movb %2,%%ah\n\t" // write char.
10. "movw %%ax,%1\n\t"
11. ::"a" (translate[c-32]),
12. "m" (\*(short \*)pos),
13. "m" (attr)
14. :"ax");
15. pos += 2;
16. 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、カーソルを先頭の列(0列目)に移動させます。// cがDEL(127)の場合、カーソルの左側の文字が消去され、カーソルは消去された文字の位置に // 移動します。

1. // cがBS（バックスペース、8）の場合、カーソルを1列分左に移動し、メモリポインタのposに対応するカーソルを // 調整する。
2. } else if (c==27) // ESC - escape char.
3. state=ESesc;
4. else if (c==10 || c==11 || c==12)
5. lf(currcons);
6. else if (c==13) // CR - carriage return.
7. cr(currcons);
8. else if (c==ERASE\_CHAR(tty))
9. del(currcons);
10. else if (c==8) { // BS - backspace.
11. if (x) {
12. x--;
13. pos -= 2;
14. }
15. // 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ます。 // カーソルの列数が画面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ます。 // cがベルBEL(7)の場合、システム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、それに応じて表示文字セットとして // G1またはG0が選択される。
16. } else if (c==9) { // HT - Horizontal Tab.
17. c=8-(x&7);
18. x += c;
19. pos += c<<1;
20. if (x>video\_num\_columns) {
21. x -= video\_num\_columns;
22. pos -= video\_size\_row;
23. lf(currcons);
24. }
25. c=9;
26. } else if (c==7) // BEL - Bell.
27. sysbeep();
28. else if (c == 14) // SO - Shift out, use G1.
29. translate = GRAF\_TRANS;
30. else if (c == 15) // SI - Shift in, use G0.
31. translate = NORM\_TRANS;
32. break;
33. // ESnormal状態でエスケープキャラクタESC(0x1b=27)を受信した場合、この // 状態の処理に入ります。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。 // 処理後は、デフォルトの状態がESnormalになります。
34. case ESesc:
35. state = ESnormal;
36. switch (c)
37. {
38. case '[': // ESC [ - CSI sequence.
39. state=ESsquare;
40. break;
41. case 'E': // ESC E - cursor next line & col 0.
42. gotoxy(currcons,0,y+1);
43. break;
44. case 'M': // ESC M - moves up one line.
45. ri(currcons);
46. break;
47. case 'D': // ESC D - moves to next line.
48. lf(currcons);
49. break;
50. case 'Z': // ESC Z - device property query.
51. respond(currcons,tty);
52. break;
53. case '7': // ESC 7 - save cursor position.
54. save\_cur(currcons);
55. break;
56. case '8': // ESC 8 - restore cursor position.
57. restore\_cur(currcons);
58. break;
59. case '(': case ')': // ESC (、ESC ) - select char set.
60. state = ESsetgraph;
61. break;
62. case 'P': // ESC P - set terminal parameters.
63. state = ESsetterm;
64. break;
65. case '#': // ESC # - modify line attributs.
66. state = -1;
67. break;
68. case 'c': // ESC c - reset to default settings.
69. tty->termios = DEF\_TERMIOS;
70. state = restate = ESnormal;
71. checkin = 0;
72. top = 0;
73. bottom = video\_num\_lines;
74. break;
75. /\* case '>': Numeric keypad \*/
76. /\* case '=': Appl. keypad \*/
77. } 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

1. // 受信されたので、次の状態はESfunckeyになります。受信した文字が '?'でない場合は、 // 直接、処理するための状態 ESgetpars に進みます。受信した文字が'?'であれば、それは // 端末装置のプライベートシーケンスであることを示し、その後に関数文字が続きます。// 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に進みます。 682 case ESsquare:
2. for(npar=0;npar<NPAR;npar++) // Initialize para array.
3. par[npar]=0;
4. npar=0;
5. state=ESgetpars;
6. if (c =='[') /\* Function key \*/ // 'ESC [['
7. { state=ESfunckey;
8. break;
9. }
10. if (ques=(c=='?'))
11. 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

1. // それが数字やセミコロンではなく，すべてのパラメータを取得したことを示す場合は，ESgotparsの状態に移行して処理を行います．
2. case ESgetpars:
3. if (c==';' && npar<NPAR-1) {
4. npar++;
5. break;
6. } else if (c>='0' && c<='9') {
7. par[npar]=10\*par[npar]+c-'0';
8. break;
9. } else state=ESgotpars;

// ESgotparsの状態は、完全な制御シーケンスを受け取ったことを示しています。このとき

// 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

1. // は端末機器のプライベートシーケンスです。このカーネルはこのシーケンスの処理をサポートしていないので、 // 直接、ESnormal状態に戻す。それ以外の場合は，対応する // 制御シーケンスに進みます。シーケンスの処理が終わると，状態をESnormalに戻す。
2. case ESgotpars:
3. state = ESnormal;
4. if (ques) // received '?'
5. { ques =0;
6. break;
7. }
8. switch(c) {

// cが'G'または'`'の場合、par[]の最初の値は列番号を表し、それが0でない場合、 // 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。パラメータが0の場合は、 // 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、 // カーソルが下に移動した行数を表します。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， // カーソルの右にある列数を表します．パラメータが 0 の場合，カーソルは 1 カラム分右に移動します．

// 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。

1. // パラメータが0の場合は、1列分左にシフトされます。
2. case 'G': case '`': // CSI Pn G -move horizontally
3. if (par[0]) par[0]--;
4. gotoxy(currcons,par[0],y);
5. break;
6. case 'A': // CSI Pn A - move up.
7. if (!par[0]) par[0]++;
8. gotoxy(currcons,x,y-par[0]);
9. break;
10. case 'B': case 'e': // CSI Pn B - move down.
11. if (!par[0]) par[0]++;
12. gotoxy(currcons,x,y+par[0]);
13. break;
14. case 'C': case 'a': // CSI Pn C - move right.
15. if (!par[0]) par[0]++;
16. gotoxy(currcons,x+par[0],y);
17. break;
18. case 'D': // CSI Pn D - move left.
19. if (!par[0]) par[0]++;
20. gotoxy(currcons,x-par[0],y);
21. break;

// cが'E'の場合、最初の値は、カーソルが下に移動して0列目に戻る行数を表します。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、 // 0列目に戻ります。パラメータが 0 の場合、1 行上に移動します。

// cが'd'の場合、最初の値は、カーソルが // 必要とするライン番号（0から数えたもの）を表します。

1. // cが'H'または'f'の場合、最初の値はカーソルが移動する行番号を表し、 // 2番目のパラメータはカーソルが移動する列番号を表します。
2. case 'E': // CSI Pn E - move down, col 0.
3. if (!par[0]) par[0]++;
4. gotoxy(currcons,0,y+par[0]);
5. break;
6. case 'F': // CSI Pn F - move up, col 0.
7. if (!par[0]) par[0]++;
8. gotoxy(currcons,0,y-par[0]);
9. break;
10. case 'd': // CSI Pn d - set cursor line no.
11. if (par[0]) par[0]--;
12. gotoxy(currcons,x,par[0]);
13. break;
14. case 'H': case 'f': // CSI Pn H -set cusor postion.
15. if (par[0]) par[0]--;
16. if (par[1]) par[1]--;
17. gotoxy(currcons,par[1],par[0]);
18. break;

// 文字cが'J'の場合（シーケンス'ESC [ Ps J'）、最初のパラメータは、カーソルの位置に関連して、 // 画面をクリアする方法を表します。

// cが'K'（'ESC [ Ps K'）の場合、最初のパラメータは、カーソルの位置に関連して行内の文字を // 削除する方法を表します。

// 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。

1. // cが'@'（'ESC [ Pn @'）の場合は、カーソル位置にn個の文字が挿入されることを意味します。
2. case 'J': // CSI Pn J - erase chars on screen.
3. csi\_J(currcons,par[0]);
4. break;
5. case 'K': // CSI Pn K - erase chars in a line. 749 csi\_K(currcons,par[0]);
6. break;
7. case 'L': // CSI Pn L - insert lines.
8. csi\_L(currcons,par[0]);
9. break;
10. case 'M': // CSI Pn M - delete lines.
11. csi\_M(currcons,par[0]);
12. break;
13. case 'P': // CSI Pn P - delete chars.
14. csi\_P(currcons,par[0]);
15. break;
16. case '@': // CSI Pn @ - insert chars.
17. csi\_at(currcons,par[0]);
18. break;

// cが'm'（'ESC [Pn m'）の場合は、太字にするなど、 // カーソルにある文字の表示属性を変更することを意味します。

// cが'r'（'ESC [Pn;Pn r'）の場合は、スクロールの開始行番号と終了行番号を // 2つのパラメータで設定することを意味します。

// 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。

1. // 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。
2. case 'm': // CSI Ps m - set char attributes.
3. csi\_m(currcons);
4. break;
5. case 'r': // CSI Pn;Pn r - set scroll range.
6. if (par[0]) par[0]--;
7. if (!par[1]) par[1] = video\_num\_lines;
8. if (par[0] < par[1] &&
9. par[1] <= video\_num\_lines) {
10. top=par[0];
11. bottom=par[1];
12. }
13. break;
14. case 's': // CSI s - saved cursor postion.
15. save\_cur(currcons);
16. break;
17. case 'u': // CSI u - restore cursor postion.
18. restore\_cur(currcons);
19. 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

1. // この条件では，cが'l'であれば，par[0]は黒画面開始時の遅延時間を表す。この条件では，cが'l'であれば，par[0]は黒画面起動時に何分遅らせるか，cが'b'であれば，par[0]の値に太字属性が設定されていることになる。
2. case 'l': /\* blank interval \*/
3. case 'b': /\* bold attribute \*/
4. if (!((npar >= 2) &&
5. ((par[1]-13) == par[0]) &&
6. ((par[2]-17) == par[0])))
7. break;
8. if ((c=='l')&&(par[0]>=0)&&(par[0]<=60))
9. {
10. blankinterval = HZ\*60\*par[0];
11. blankcount = blankinterval;
12. }
13. if (c=='b')
14. vc\_cons[currcons].vc\_bold\_attr
15. = par[0];
16. }
17. break;
18. // ステータスESfunckeyは、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しています。それを表示する必要はないので、通常の状態であるESnormalに戻ります。
19. case ESfunckey: // keyboard function key.
20. state = ESnormal;
21. break;
22. // 状態ESsettermは、DCS（Device Control String）シーケンス状態であることを示す。// このとき、文字'S'を受信すると、初期表示文字属性が // 復元される。文字'L'または'l'を受信した場合、折りたたみ表示モードのオン/オフを行う。
23. case ESsetterm: /\* Setterm functions. \*/
24. state = ESnormal;
25. if (c == 'S') {
26. def\_attr = attr;
27. video\_erase\_char = (video\_erase\_char&0x0ff) |
28. (def\_attr<<8)となっています。
29. } else if (c == 'L')
30. ; /\*linewrap on\*/
31. else if (c == 'l')
32. ; /\*linewrap off\*/
33. 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 )'を // 受信したことを示します。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。

1. // このとき、文字'0'を受信した場合は、グラフィックのキャラクタセットとして // G0とG1が選択され、受信した文字が'B'の場合は、通常のASCIIのキャラクタセットとして // G0とG1が選択されます。
2. case ESsetgraph: // 'CSI (0' or 'CSI (B' - select char set.
3. state = ESnormal;
4. if (c == '0')
5. translate = GRAF\_TRANS; 814 else if (c == 'B')
6. 815 translate = NORM\_TRANS; 816 break;
7. default:
8. state = ESnormal;
9. }
10. }
11. // 最後に、ディスプレイコントローラのカーソル位置を上記で設定した値で設定します。
12. set\_cursor(currcons);
13. }

823

1. /\*
2. \* void con\_init(void);
3. \*
4. \* This routine initalizes console interrupts, and does nothing 828 \* else. If you want the screen to clear, call tty\_write with 829 \* the appropriate escape-sequece.
5. \*
6. \* Reads the information preserved by setup.s to determine the current display
7. \* type and sets everything accordingly.
8. \*/
9. void con\_init(void)
10. {
11. register unsigned char a;
12. char \*display\_desc = "????";
13. char \*display\_ptr;
14. int currcons = 0; // current console no.
15. long base, term;
16. long video\_memory; 842
17. // まず、setup.sプログラムで得られたシステムハードウェアのパラメータに応じて、この関数に固有のいくつかの静的グローバル変数が初期化されます（60〜68行目を参照）。
18. video\_num\_columns = ORIG\_VIDEO\_COLS;
19. video\_size\_row = video\_num\_columns \* 2; // bytes used by a display row.
20. video\_num\_lines = ORIG\_VIDEO\_LINES; // lines of the display.
21. video\_page = ORIG\_VIDEO\_PAGE; // display page.
22. video\_erase\_char = 0x0720; // char: 0x20, attr: 0x07.
23. 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

// の位置と、ディスプレイ・レジスタ・インデックス・ポート番号とディスプレイ・レジスタ・データ・ポート番号をそれぞれ設定します。取得したBIOSのディスプレイモードが7に等しい場合は、 // モノクロディスプレイカードであることを意味します。

1. 850 if (ORIG\_VIDEO\_MODE == 7) /\* これはモノクロのディスプレイか？\*/ 851 {
2. video\_mem\_base = 0xb0000; // start addr of monochrome display.
3. video\_port\_reg = 0x3b4; // index register port.
4. 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の間のアドレス範囲となります。このコードでは、ディスプレイの説明文字列を「EGAm」に設定し、システムの初期化時に画面の右上隅に // 表示されるようにしています。

// 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は

// 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を意味し、これはタイプ

の // 以降のVGAです。割り込みコールの戻り値が変化していない場合、ディスプレイカードがこの機能をサポートしていないことを示す、 // 一般的な

1. // モノクロのディスプレイカードです。
2. if ((ORIG\_VIDEO\_EGA\_BX & 0xff) != 0x10)
3. {
4. video\_type = VIDEO\_TYPE\_EGAM; // video type (EGA mono).
5. video\_mem\_term = 0xb8000; // video memory end address.
6. display\_desc = "EGAm";
7. }
8. // BXレジスタの値が0x10に等しい場合、モノクロのディスプレイカードMDAを意味し、 // 8KBのディスプレイメモリしかないことを意味する。861 else
9. {
10. video\_type = VIDEO\_TYPE\_MDA; // MDA mono.
11. video\_mem\_term = 0xb2000; // memory end address.
12. display\_desc = "\*MDA";
13. } 867 }
14. // ディスプレイモードが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。この時、テキストモードで使用されるディスプレイメモリの開始アドレスは0xb8000です。
15. else /\* If not, it is color. \*/
16. {
17. can\_do\_colour = 1; // can use color flag.
18. video\_mem\_base = 0xb8000; // memory start address.
19. video\_port\_reg = 0x3d4; // index register port.
20. video\_port\_val = 0x3d5; // data register port.

// そして、ディスプレイカードのカテゴリーを判断する。BXが0x10になっていない場合は、 // EGAカードであることを意味します。このとき、合計32KBのディスプレイメモリ（0xb8000〜0xc0000）が使用可能です。

1. // それ以外の場合は、CGAカードであり、8KBのディスプレイメモリ（0xb8000-0xba000）しか使用できません。
2. if ((ORIG\_VIDEO\_EGA\_BX & 0xff) != 0x10)
3. {
4. video\_type = VIDEO\_TYPE\_EGAC; // EGA type.
5. video\_mem\_term = 0xc0000; // mem end address.
6. display\_desc = "EGAc";
7. }
8. else
9. {
10. video\_type = VIDEO\_TYPE\_CGA; // CGA type.
11. video\_mem\_term = 0xba000; // meme end address.
12. display\_desc = "\*CGA";
13. }
14. }

// 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 を各バーチャルコンソールが占有するバイト数で割ったもの。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、行数に文字の1行あたりのバイト数を掛けたものになります： 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

1. // 最後に、ビデオメモリの総数をバーチャルコンソールの数で割った値が、各バーチャルコンソール（vc）が占有するメモリのバイト数となる。
2. video\_memory = video\_mem\_term - video\_mem\_base;
3. NR\_CONSOLES = video\_memory / (video\_num\_lines \* video\_size\_row);
4. if (NR\_CONSOLES > MAX\_CONSOLES) // MAX\_CONSOLES = 8
5. NR\_CONSOLES = MAX\_CONSOLES;
6. if (!NR\_CONSOLES)
7. NR\_CONSOLES = 1;
8. video\_memory /= NR\_CONSOLES; // memory of each vc.

894

895 /\* 使用しているディスプレイドライバが何であるかをユーザに知らせます。

896

1. // そして、説明文字列を画面の右上に表示します。ここでは、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方法を用いています。// まず、ディスプレイポインタdisplay\_ptrが1行目の右端の最後の4文字を指しており（1文字に2バイト必要なので、マイナス8）、その後、文字列の文字をサイクリックにコピーし、各文字に1つの属性バイトを確保する。
2. display\_ptr = ((char \*)video\_mem\_base) + video\_size\_row - 8;
3. while (\*display\_desc)
4. {
5. \*display\_ptr++ = \*display\_desc++;
6. display\_ptr++;
7. }

903

904 /\* スクロールに使用される変数を初期化する（主にEGA/VGA） \*/ (注)

905

// このとき、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。// したがって、以下は実際には構造体vc\_cons[0]のすべてのフィールドの初期化です。

// 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。

1. // 上記の115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のend mem vc.
2. scr\_end = video\_mem\_start + video\_num\_lines \* video\_size\_row; // end mem of scroll.
3. top = 0; // top line number of scrolling.
4. bottom = video\_num\_lines; // bottom line number of scrolling.
5. attr = 0x07; // default attribute (white on black).
6. def\_attr = 0x07; // set default char attribute.
7. restate = state = ESnormal; // current and next state of escape sequence.
8. checkin = 0;
9. ques = 0; // got question mark flag.
10. iscolor = 0; // color flag.
11. translate = NORM\_TRANS; // char set used (ordinary ASCII table).
12. vc\_cons[0].vc\_bold\_attr = -1; // bold char attribute flag (-1 not used).

919

1. // コンソール0の現在のカーソル位置と、カーソルに対応するメモリの位置posを設定した後、残りの仮想コンソールの構造をループします。// それぞれが占有す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。
2. gotoxy(currcons,ORIG\_X,ORIG\_Y);
3. for (currcons = 1; currcons<NR\_CONSOLES; currcons++) {
4. vc\_cons[currcons] = vc\_cons[0]; // copy structure data of #0
5. origin = video\_mem\_start = (base += video\_memory);
6. scr\_end = origin + video\_num\_lines \* video\_size\_row;
7. video\_mem\_end = (term += video\_memory);
8. gotoxy(currcons,0,0); // cursor is at top left.
9. }

// 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

1. // キーボードからのリクエスト信号に応答します。最後にキーボードコントローラをリセットして、 // キーボードが正常に動作するようにします。
2. update\_screen(); // update foreground origin & cursor pos.
3. set\_trap\_gate(0x21,&keyboard\_interrupt); // refer to system.h, line 36.
4. outb\_p(inb\_p(0x21)&0xfd,0x21); // cancel masking of keyboard.
5. a=inb\_p(0x61); // read keyboard port 0x61 (8255A PB).
6. outb\_p(a|0x80,0x61); // disable keyboard (set bit 7).
7. 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。

1. // フォアグラウンドのコンソールを fg\_console で指定されたバーチャルコンソールに切り替えます。 fg\_console は // フォアグラウンドのバーチャルコンソール番号を設定します。
2. void update\_screen(void)
3. {
4. set\_origin(fg\_console); // set the scroll start memory address.
5. 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。

1. // 8255AのPBポートのビット1とビット0をリセットします。kernel/sched.cプログラムの後にある、タイマープログラミングの説明を参照してください。
2. void sysbeepstop(void)
3. {
4. /\* disable counter 2 \*/
5. outb(inb\_p(0x61)&0xFC, 0x61);
6. }

949

950 int beepcount = 0; // ビープ音の回数（ティック）。

951

// 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。boot/setup.sプログラムの後に、8259A割り込みコントローラの//チッププログラミング方法を参照して、プログラミング

1. // kernel/sched.cプログラムの後にタイマーの命令を出す。
2. static void sysbeep(void)
3. {
4. /\* enable counter 2 \*/
5. outb\_p(inb\_p(0x61)|3, 0x61);
6. /\* set command for counter 2, 2 byte write \*/
7. outb\_p(0xB6, 0x43); // timer control word register port.
8. /\* send 0x637 for 750 HZ \*/
9. outb\_p(0x37, 0x42); // send high & low count bytes to channel 2.
10. outb(0x06, 0x42);
11. /\* 1/8 second \*/
12. beepcount = HZ/8;
13. }

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。

1. // 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することです。
2. int do\_screendump(int arg)
3. {
4. char \*sptr, \*buf = (char \*)arg;
5. 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

1. // が適切に展開されます。そして、その先頭からのコンソール番号をカールコンします。コンソール番号が有効であることを判断した後、コンソール画面のすべてのメモリコンテンツが // ユーザーバッファにコピーされます。
2. verify\_area(buf,video\_num\_columns\*video\_num\_lines);
3. currcons = get\_fs\_byte(buf);
4. if ((currcons<1) || (currcons>NR\_CONSOLES))
5. return -EIO;
6. currcons--;
7. sptr = (char \*) origin;
8. for (l=video\_num\_lines\*video\_num\_columns; l>0 ; l--)
9. put\_fs\_byte(\*sptr++,buf++);
10. return(0);
11. }

980

// ブラックスクリーン処理。

1. // blankInterval間隔の間にユーザーが何もボタンを押さなかった場合、画面を保護するために // ブラックアウトされます。
2. void blank\_screen()
3. {
4. if (video\_type != VIDEO\_TYPE\_EGAC && video\_type != VIDEO\_TYPE\_EGAM)
5. return;
6. /\* blank here. I can't find out how to do it, though \*/
7. }

987

// 黒い画面を元に戻す。

1. // ユーザーがいずれかのボタンを押すと、ブラックスクリーン状態の画面が復元される。
2. void unblank\_screen()
3. {
4. if (video\_type != VIDEO\_TYPE\_EGAC && video\_type != VIDEO\_TYPE\_EGAM)
5. return;
6. /\* unblank here \*/
7. }

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

// 文字の特性に合わせて、バッファ内の文字を // カーソル移動を制御したり、画面に直接表示したりします。引数bは、ヌル終端の文字列 // バッファポインタです。

1. 995 void console\_print(const char \* b) 996 {。
2. int currcons = fg\_console;
3. 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。

1. // キャリッジリターン カーソルの改行操作が行われ、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行う。その後、 // 次の文字に進みます。
2. while (c = \*(b++)) {
3. if (c == 10) {
4. cr(currcons);
5. lf(currcons);
6. continue;
7. }
8. if (c == 13) {
9. cr(currcons);
10. continue; 1009 }

// 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

1. // 次の行の先頭に戻ります。そして、画面に表示されているカーソルがあるメモリの位置に、文字を // 配置します。次の文字を準備するために、カーソルを1列右に移動します。
2. if (x>=video\_num\_columns) {
3. x -= video\_num\_columns;
4. pos -= video\_size\_row;
5. lf(currcons); 1014 }
6. // レジスタALは表示されるべき文字です。ここではAHに属性バイトが入り、 // AXの内容がカーソルメモリの位置posに格納される、つまりカーソルに文字が // 表示されることになる。
7. \_\_asm\_\_("movb %2,%%ah\n\t" // attribute byte -> ah.
8. "movw %%ax,%1\n\t" // put value of ax to pos.
9. ::"a" (c),
10. "m" (\*(short \*)pos),
11. "m" (attr)
12. :"ax");
13. pos += 2;
14. x++;
15. }
16. // 最後に、ディスプレイコントローラのカーソル位置を上記で設定した値で設定します。
17. set\_cursor(currcons);
18. }

1026

### 10.3.3 Infomation

#### 10.3.3.1 VGA adapter programming

ここでは、IBM VGAとその互換グラフィックカードのポートについてのみ説明し、主にMDA、CGA、EGA、VGAのディスプレイコントロールカードの一般的なプログラミングポートについて説明します。これらのポートは、CGAで使用されているMC6845チップと互換性があります。その名称と用途を表10-5に示します。CGA/EGA/VGAポート(0x3d0-0x3df)を例に挙げて説明します。MDAのポート範囲は、0x3b0～0x3bfです。

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のデータレジスタ。レジスタr14～r17が読み出せます。  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のパレットレジスタ。使用する色を選択してください。 bit 7-6 未使用、bit 5=1 カラーセット（シアン、マゼンタ、ホワイト）を有効にしてください。  =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ステータスレジスタ。  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 r1 r2 r3 | 水平方向の合計  水平表示  水平同期位置  Horizontal Sync Paulse Width | Char  Char  Char  Char | W  W  W  W | 0x38  0x28  0x2d  0x0a | 0x71  0x50  0x5a  0x0a | 0x38  0x28  0x2d  0x0a |
| r4 r5 r6 r7 | 垂直方向の合計  垂直方向のトータルアジャスト  垂直方向の表示  Vertical Sync Position | チャーシュー  走査線  チャーシュー  Char row | W  W  W  W | 0x1f  0x06  0x19  0x1c | 0x1f  0x06  0x19  0x1c | 0x7f  0x06  0x64  0x70 |
| r8 | Interlace Mode |  | W | 0x02 | 0x02 | 0x02 |
| r9 | Maximum Scan Line Address | Scan line | W | 0x07 | 0x07 | 0x01 |
| r10 r11 | カーソルスタート  Cursor End | 走査線  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 | カーソル位置（高）  Cursor position (Low) |  | R/W  R/W | Vary |  |  |
| r16 r17 | ライトペン（高）  Light Pen (Low) |  | R  R | Vary |  |  |

#### 10.3.3.2 Principles of scrolling operation

スクロールとは、画面上の指定された開始行と終了行の文字列を上（スクロールアップ）または下（スクロールダウン）に移動させることである。画面をメモリ上の対応する画面内容を表示するウィンドウと考えると、画面内容を上に移動させることは、ウィンドウを表示メモリの下に移動させることであり、画面内容を下に移動させることは、ウィンドウを上に移動させることである。このプログラムでは、コントローラ内のビデオメモリの開始位置「原点」を再設定し、調整プログラムの対応する変数を調整します。この2つの操作には、それぞれ2つのケースがあります。

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には、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。

scr\_end

B

eyond end of memory

video\_mem\_start

video\_mem\_end

origin

scr\_end

New origin

New

scr\_end

Me

mory of

screen content

)

a

(

General case of up scrolling

Fill blanks

b)

(

Case of need to move screen data

N

ew origin

New scr\_end

Fill blanks

origin

Copy block

Memory of screen

content after

scrolling

図10-8 スクロールアップ操作の模式図

画面を下にスクロールすることは、上にスクロールすることと同様に、2つの似たような状況に遭遇する。画面のウィンドウが上に移動しただけで、画面の上部には空行が表示され、画面の内容に対応するメモリが表示メモリの範囲を超えた場合には、画面データのメモリブロックを表示メモリの終端位置まで下に移動させる必要がある。

#### 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
3. 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カーネルでは、Digital Equipment Corporation DEC（現在はCompaqやHPに組み込まれている）のVT100およびVT102端末機器との互換性しか実装されていないが、この2つの事実上の標準端末機器は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ビットの中の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 | | | | | | |  |

7ビットコード表よりもコードの列数が8個多く、合計256個のコードを含んでいます。7ビットコード表と同様に、各行は8ビットコードの下位4ビットの値を表し、各列は上位4ビットの値を表す。表の左半分（0列目〜7列目）は7ビットコード表と全く同じで、それぞれのコードの8ビット目は0なので、このビットは無視してよい。表の右半分の各コードの8ビット目は

(8列目～15列目)がすべて1であるため、これらの文字は8ビット環境でのみ使用することができます。8ビットコード表には、2つの制御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つがあります。

1. 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。
2. 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。

1. 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。
2. 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

導入部 中間部 最終部

(0または複数の文字) (1文字)

ESCは、ANSI規格で定義されているエスケープシーケンスイントロデューサである。ESCを受け取った端末は、それ以降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。

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。中間文字と終端文字を合わせて、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

エスケープシーケンスは7ビットの文字のみを使用するため、7ビットと8ビットの両方の環境で使用することができます。エスケープシーケンスや制御シーケンスを使用する際には、文字のテキスト表現ではなく、コードシーケンスを定義していることに注意してください。エスケープシーケンスの重要な用途の一つは、7ビット制御文字の機能を拡張することです。ANSI規格では、2バイトのエスケープシーケンスを7ビットのコード拡張として使用することで、C1の任意の制御文字を表現することができます。これは、7ビットの互換性を必要とするアプリケーションにおいて、非常に便利な機能です。例えば、C1の制御文字CSIとINDは、次のように7ビットコード拡張形式を使って表現することができます。

C1 char エスケープシーケンス

CSI ESC [

0x9b 0x1b 0x5b

IND ESC D

0x84 0x1b 0x44

一般的には、上記のコード拡張技術を2つの方法で使用することができます。2文字のエスケープシーケンスを使って、8ビットコード表C1の任意の制御文字を表現することができます。2文字目の値は、C1の対応する文字の値から0x40(64)を引いた値になります。また、2文字目の値が0x40から0x5fの間のエスケープシーケンスであれば、制御文字のESCを削除し、2文字目に0x40を加えることで、8ビットの制御文字に変換することができます。

5. 制御シーケンス

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

導入部 パラメータ部 中間部 最終部

(0または複数のchars) (1つのchars)

制御シーケンスイントロデューサは、8ビットの制御文字C1のCSI（0x9b）です。ただし、CSI は 7 ビットの符号拡張「ESC []」でも拡張できるため、すべての制御シーケンスは、2 文字目が左括弧「[]」であるエスケープシーケンスを用いて表現することができる。端末は、イントロデューサCSIを受信した後、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

図10-9は、すべての文字の属性をキャンセルした後、下線と反転の属性をオンにするという制御シーケンスの例です。ESC [ 0;4;7m

Final char

ESC [ 0

; 4 ; 7 m

CSI

select chars

Parameter string

seperators

図10-9 制御シーケンスの例

6. 受信した文字の端末処理

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、それを端末が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・デバイスの書き込みバッファ・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。 rs\_write()関数は、ファイルシステムでキャラクター・デバイス・ファイルが使用されているときに呼び出される。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がデータを送信した後に割り込み信号を送信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

1. /\*
2. \* serial.c
3. \*
4. \* This module implements the rs232 io functions
5. \* void rs\_write(struct tty\_struct \* queue);
6. \* void rs\_init(void);
7. \* and all interrupts pertaining to serial IO.
8. \*/

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。

// 通信です。

// <linux/sched.h> スケジューラーのヘッダーファイルは、タスク構造体task\_struct、初期タスク0のデータ、およびいくつかの組み込みアセンブリ関数マクロステートメントを定義します。

// ディスクリプタのパラメータ設定と取得について。

// <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。

// ディスクリプタ/割込みゲートなどを変更することが定義されています。

// <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。

1. // マクロの組み込みアセンブラの形式です。
2. #include <linux/tty.h>
3. #include <linux/sched.h>
4. #include <asm/system.h>
5. #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

1. // 指定されたシリアルポートの送信ボーレート（2400bps）を設定し、ライトホールドレジスタが空であることを除いて、すべての割り込みソースを許可します。また、2バイトのボーレートファクタを出力する際には、まず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。
2. static void init(int port)
3. {
4. outb\_p(0x80,port+3); /\* set DLAB of line control reg \*/
5. outb\_p(0x30,port); /\* LS of divisor (48 -> 2400 bps \*/
6. outb\_p(0x00,port+1); /\* MS of divisor \*/
7. outb\_p(0x03,port+3); /\* reset DLAB \*/
8. outb\_p(0x0b,port+4); /\* set DTR,RTS, OUT\_2 \*/
9. outb\_p(0x0d,port+1); /\* enable all intrs but writes \*/
10. (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。

1. // 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に実装されています。
2. void rs\_init(void)
3. {
4. // 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。シリアルポート1が使用する割り込みは // int 0x24で、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。
5. set\_intr\_gate(0x24,rs1\_interrupt); // set int gate descriptor of port 1 (IRQ4).
6. set\_intr\_gate(0x23,rs2\_interrupt); // set int gate descriptor of port 2 (IRQ3).
7. init(tty\_table[64].read\_q->data); // Initialize port 1 (.data is base address).
8. init(tty\_table[65].read\_q->data); // Initialize port 2
9. outb(inb\_p(0x21)&0xE7,0x21); // enable 8259A to respond to IRQ3, IRQ4. 44 }

45

1. /\*
2. \* This routine gets called when tty\_write has put something into
3. \* the write\_queue. It must check wheter the queue is empty, and
4. \* set the interrupt register accordingly
5. \*
6. \* void \_rs\_write(struct tty\_struct \* tty);
7. \*/

//// 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を立てることで、送信ホールドレジスタを空にして割り込み要求を発生させることを再び無効にします。この時点で "ループ "送信動作は終了する。

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。

1. // 送信ホールディングレジスターが空の時に送信される文字です。ただ、ここでのwrite\_q.dataは、シリアルポートのベースアドレスであることに // 注意してください。
2. cli();
3. if (!EMPTY(tty->write\_q))
4. outb(inb\_p(tty->write\_q->data+1)|0x02,tty->write\_q->data+1);
5. sti();
6. }

60

### 10.4.3 Information

#### 10.4.3.1 Universal Asynchronous Serial Communication

ユニバーサル非同期シリアル通信の伝送方式は、業界で最も一般的な従来のシリアル通信方式であり、現在でも様々な組み込みシステムで広く使用されています。専用のUART（Universal Asynchronous Receive and Transmit）コントローラチップを使用してデータ転送を行います。使用する通信フレームのフォーマットを図10-10に示します。キャラクターの転送は、スタートビット、データビット、パリティビット、1または2のストップビットで構成されます。スタートビットは同期的な役割を果たし、その値は常にゼロです。データビットは、実際に転送されるデータ、つまり1文字分のコードです。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。

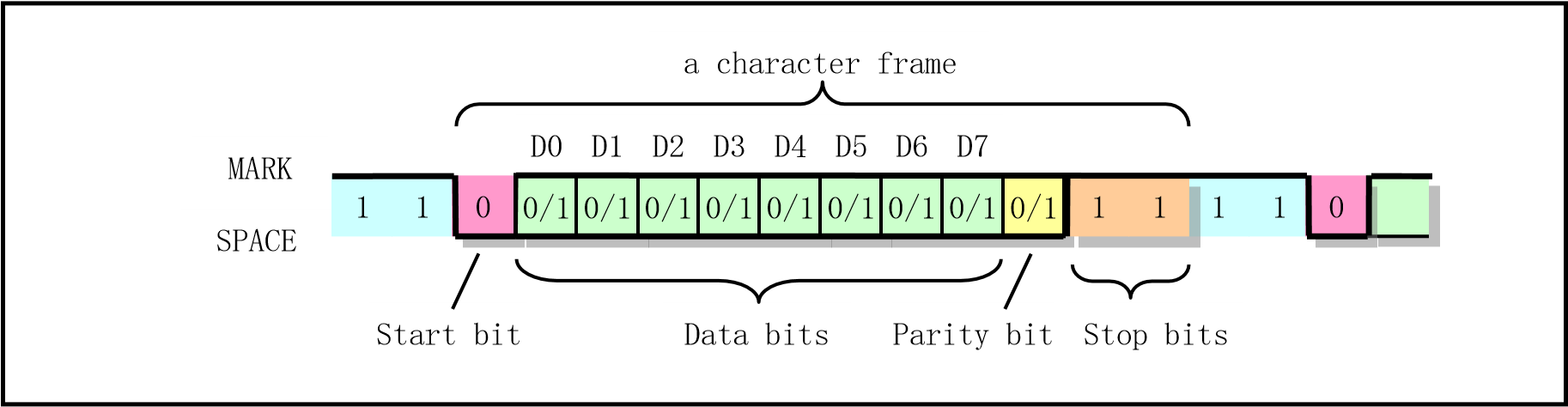


図10-10 非同期シリアル伝送のフレームフォーマット

データの送信がない場合、送信者はMARK状態となり、1を送信し続けます。データを送信する必要がある場合、送信者はまず、ビット間隔のスタートビットSPACEを送信する必要があります。SPACEビットを受信した受信者は、送信者との同期を開始し、その後、後続のデータを受信します。プログラムでパリティビットが設定されている場合は、データを送信した後にパリティビットを受信する必要があります。最後に、ストップビットです。キャラクタフレームを送信した後、すぐに次のキャラクタフレームを送信することもできますし、MARKを一時的に送信した後、再びキャラクタフレームを送信することもできます。

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、回線の干渉のほか、通信相手のフレームフォーマットの設定が異なっていることが考えられます。

1. シリアル通信インターフェースとUARTの構造

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。

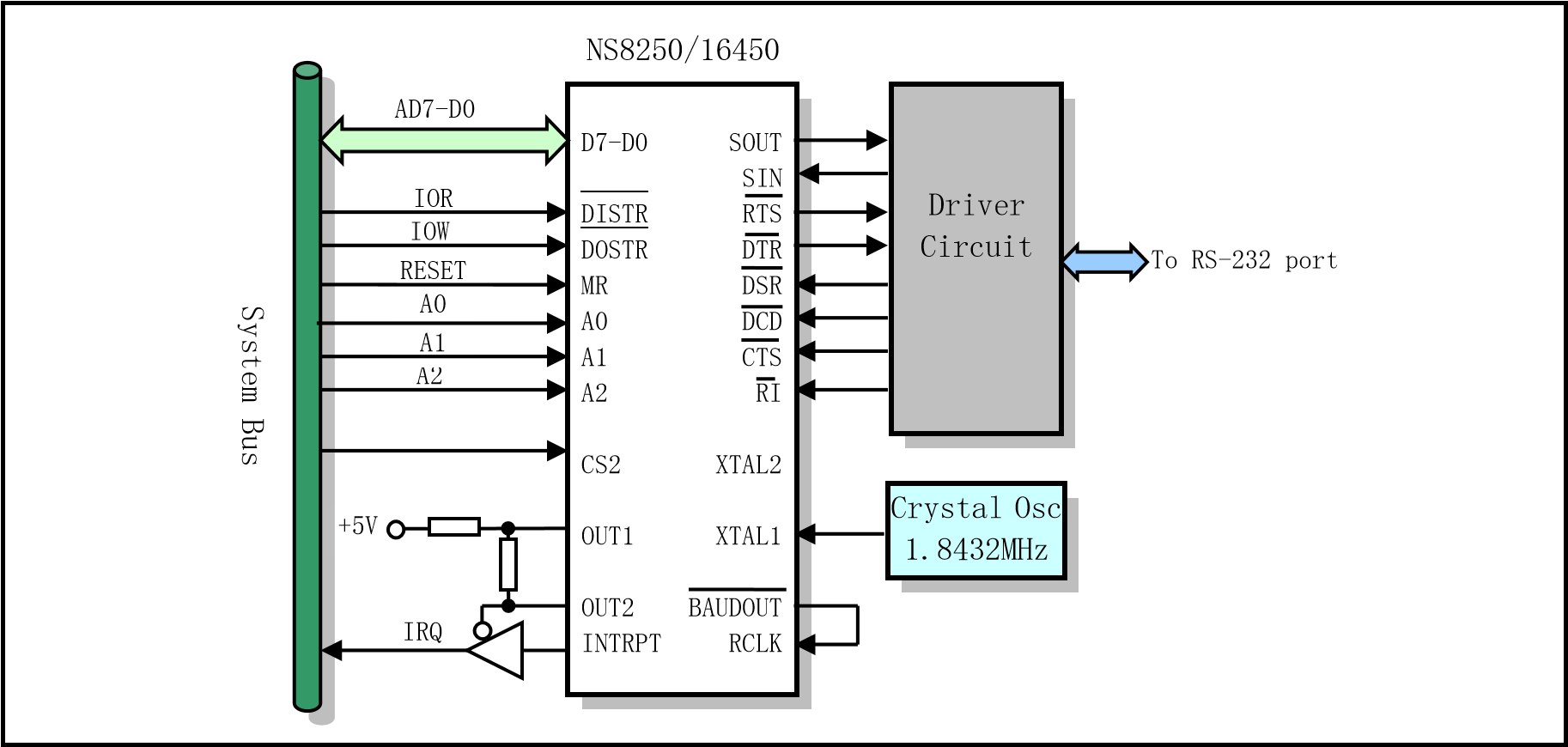


図10-11 NS8250/16450の基本的なハードウェア構成図

第2部は主に、UARTとRS-232インターフェースの間のピン部分を含みます。これらのピンは、主にシリアルデータの送受信とMODEM制御信号の生成または受信に使用されます。シリアル出力データ(SOUT)ピンはビットストリームをラインに送信し、入力データ(SIN)ピンはラインからビットストリームを受信します。データデバイスレディ(DSR)ピンは通信デバイス(MODEM)がデータの受信を開始する準備ができたことをUARTに通知するために使用され、送信要求(RTS)ピンはMODEMにコンピュータが送信モードへの切り替えを要求していることを通知するために使用されます。DCD（Device Carrier Detect）ピンはMODEMの情報を受信するために使用され、キャリア信号が受信されたことをUARTに伝えます。RI（Ring Indicator）ピンもMODEMによって使用され、通信ラインがすでに接続されていることをコンピュータに伝えます。 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生成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。

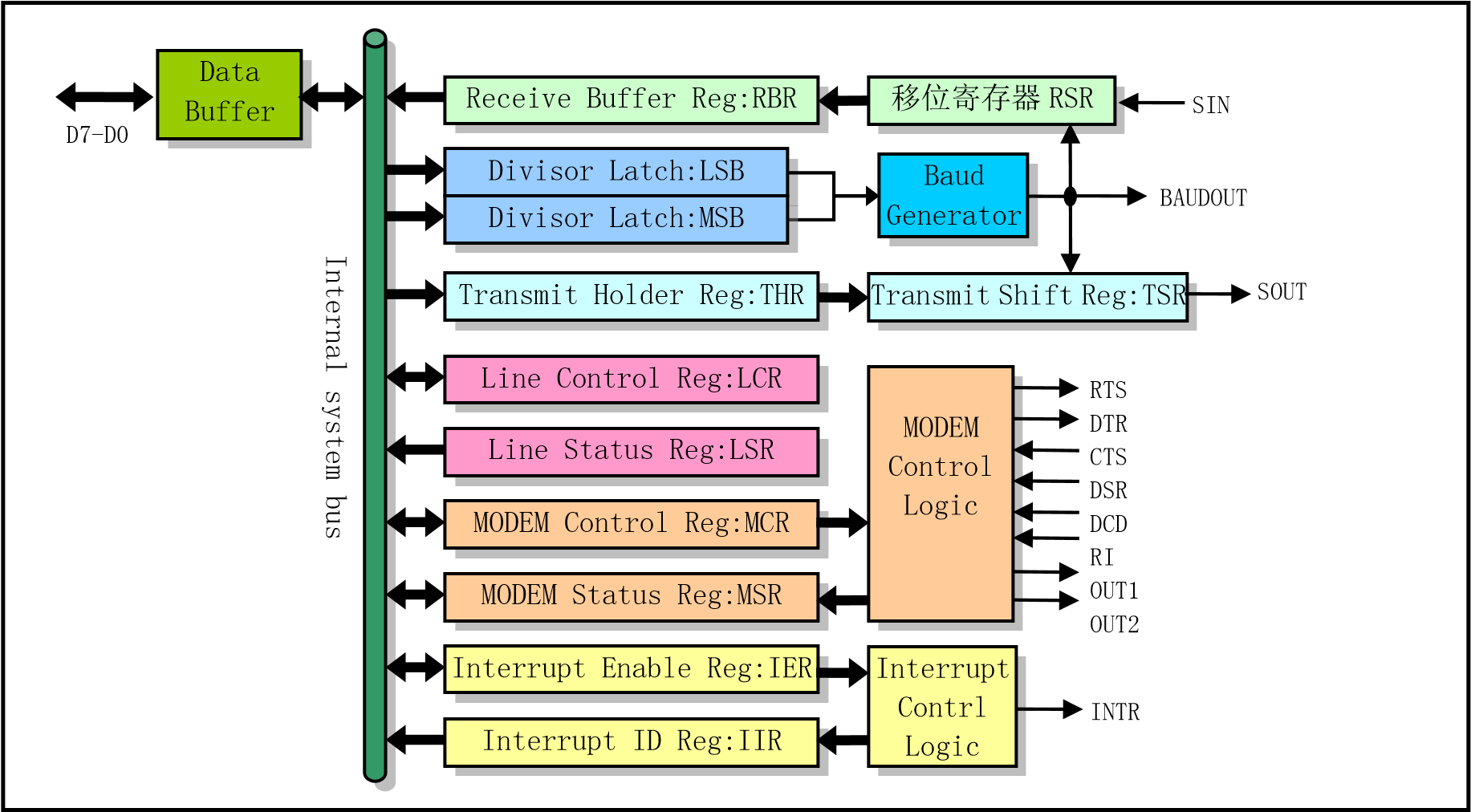


図 10-12 NS8250 UART 内部コンポーネントの基本ブロック図

NS8250の場合、CPUがアクセスできるレジスタは10本ありますが、これらのレジスタを選択するためのアドレスラインA2--A0は最大8本までしか選択できません。そこでNS8250では、ラインコントロールレジスタ（LCR）のビット（ビット7）を用いて、LSBとMSBの2つの除算器ラッチレジスタを選択しています。ビット7はDivisor Latch Access Bit (DLAB)と呼ばれています。これらのレジスタの目的とアクセスポートアドレスを表10-8に示します。

|  |  |  |  |
| --- | --- | --- | --- |
| 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 割り込みイネーブルレジスタ IER。  ビット7-4はすべて0、予約済みです。  ビット3=1 モデムステータス割り込み許可  ビット2=1 受信ラインステータス割り込み有効  Bit 1=1 Transmit Holding Register Empty Interrupt allowed; Bit 0=1 has received data interrupt enable. |
| 0x3fa (0x2fa) | R |  | 割り込み識別レジスタIIRを読み出します。割り込みハンドラは、4つのタイプのうちどのタイプの割り込みかを決定するために使用されます。  ビット7-3はすべて0（未使用）です。  ビット2-1は、割り込みの優先順位を決定します。  = 11 受信状態にエラーがあり、優先順位が最も高い。リセットするにはラインステータスを読み取ってください。  = 10 データを受信した、優先順位2。データを読むとリセットできます。  = 01 送信ホールディング・レジスタが空、優先順位3。THRを書き込むとリセットされます。  = 00 MODEM status changes, priority 4. Read MODEM state to reset it. |
|  |  |  | Bit 0 = 0 pending interrupt; =1 no interrupt. |
| 0x3fb (0x2fb) | W |  | ラインコントロールレジスタLCRの書き込み  ビット7＝1 ディバイザンラッチアクセスビット（DLAB）。  = 0 レシーバ、送信ホールド、またはインタラプトイネーブルレジスタアクセス。  ビット6=1でブレークが可能になります。  ビット5=1はパリティビットを保持します。  ビット4=1 偶数パリティ、=0 奇数パリティ。  ビット3=1はパリティあり、=0はパリティなし。  ビット2＝1になったのは、データビット長による。データビット長が5ビットの場合、ストップビットは1.5ビット、データビット長が6、7、8ビットの場合、ストップビットは2ビット、＝0ストップビットは1ビットとなります。  Bit 1-0 データビットの長さ。  = 00 5データビット  = 01 6ビットのデータビット  = 10 7-bit data bits; = 11 8-bit data bits. |
| 0x3fc (0x2fc) | W |  | MODEMコントロールレジスタMCRの書き込み。  ビット7～5はすべて0、予約済み。  ビット4=1 チップはサイクリック・フィードバック診断モードで動作しています。  ビット3=1、補助ユーザーが出力2を指定、システムへのINTRIPTを有効にする。  ビット2=1でAUXユーザーが出力1を指定し、PCは使用しません。  ビット1=1でRTS送信要求が有効になります。  Bit 0 = 1 makes the data terminal ready DTR active. |
| 0x3fd (0x2fd) | R |  | ラインステータスレジスタLSRの読み出し  Bit 7=0 reserved;  Bit 6=1 送信シフトレジスタは空です。  Bit 5=1 送信保持レジスタが空で、送信するための文字を得ることができる。  ビット4=1は、ブレーク条件を満たすビット列を受信します。  ビット3=1 フレームフォーマットエラー  ビット2=1 パリティエラー  ビット1＝1でオーバーレイエラーを超える。  Bit 0=1 Receiver data is ready and the system is readable. |
| 0x3fe (0x2fe) | R |  | MODEMステータスレジスタMSRを読む。δは、信号や状態の変化を示します。  ビット7=1 キャリアディテクト（CD）が有効である。  ビット6=1 リング表示（RI）が有効である。  ビット5=1 データ・デバイス・レディ（DSR）が有効である。  ビット4=1 クリア・トランスミット（CTS）が有効。  ビット3=1でδキャリアを検出。  ビット2=1は、リング信号のエッジを検出します。  Bit 1 = 1 δ Data Device Ready (DSR); Bit 0 = 1 δ Clear Transmit (CTS). |

2. UART初期化プログラミング方法 PCの電源を入れると、システムのRESET信号により、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がリセットされます。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。 a) 送信ボーレートを設定します。

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、ボーレート係数を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。



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

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、データ長は8ビットとなります。 c) MODEMコントロールレジスタの設定

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ビット3（OUT2）がリセットされている状態で、プログラムがUARTレジスタの内容をサイクリックにポーリングすることでシリアルデータの送受信を行うことです。MCRビット3=0の場合でも、MODEMの状態が変化した条件で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、割り込みが発生した条件に応じて割り込みフラグレジスタIIRを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。

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。

d) 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。

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が送出できるようにします。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。

3. UART 割り込みハンドラのプログラミング方法

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のかはわかりません。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。

N

N

N

N

N

Y

Y

Y

Y

Y

Read Interrupt ID Reg(

IIR

)

Serial Interrupt

Handler

Has INT(D0=0)?

Int Ret

D2D1=00?

D2D1=01?

D2D1=10?

D2D1=11?

MODEM

Status Processing

Write Character

Read Character

L

ine Status

Processing

図10-13 シリアル通信割り込み処理ブロック図

## IIRの内容を取り出した後は、まずビット0により処理すべき割り込みがあるかどうかを判断する必要があります。ビット0＝0であれば、処理すべき割込みがある。そして、ビット2とビット1に応じて、ポインタ・ジャンプテーブルを用いて、対応する割込み源タイプの処理サブルーチンを呼び出す。各サブルーチンでは、処理後にUARTの対応する割込みソースがリセットされます。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?)。この割り込みに対して他の割り込みソースがある場合、IIRのビット0は0のままなので、割り込みハンドラは対応する割り込みソースのサブルーチンを呼び出して処理を継続します。この割り込みの原因となったすべての割り込みソースが処理され、リセットされます。この時点で、UARTは自動的にIIRのビット0＝1に設定し、保留中の割り込みがないことを示すので、割り込みハンドラは終了できます。

## 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は、a.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エンプティ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、の4種類があります。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

1. /\*
2. \* rs\_io.s
3. \*
4. \* This module implements the rs232 io interrupts.
5. \*/

12

1. .text
2. .globl \_rs1\_interrupt,\_rs2\_interrupt

15

1. // size は読み書き可能なキューの長さをバイト単位で表す buff 16 size = 1024 /\* must be power of two !
2. and must match the value
3. in tty\_io.c!!! \*/

19

1. /\* these are the offsets into the read/write buffer structures \*/
2. // include/linux/tty.h // ファイル内のtty\_queue構造の各フィールドのオフセットに対応し、rs\_addrはtty\_queue構造のデータフィールドに対応します。しかし、シリアル端末のバッファキューの場合、 // このフィールドには、シリアルポートのベースアドレス（0x3f8または0x2f8）が保持されます。
3. rs\_addr = 0 // field offset of the serial ports (0x3f8 or 0x2f8).
4. head = 4 // the head pointer field offset in the buffer.
5. tail = 8 // the tail pointer field offset in the buffer.
6. proc\_list = 12 // wait for the buffered process field offset.
7. buf = 16 // buffer field offset.

26

// 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場合、 // 割り込みハンドラは待機中のプロセスを起こして、文字を書き込みキューに // 入れ続けることができます。

27 startup = 256 /\* 再起動時に書き込みキューに残っている文字数 \*/ 。

28

1. /\*
2. \* These are the actual interrupt routines. They look where 31 \* the interrupt is coming from, and take appropriate action.
3. \*/

//// 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

1. // をスタックに乗せ(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。キャラクタ // バッファのキュー構造 tty\_queue は、include/linux/tty.h の 22 行目にあります。
2. .align 2
3. \_rs1\_interrupt:
4. pushl $\_table\_list+8 // serial port 1 r/w queues pointer is pushed onto stack.
5. jmp rs\_int
6. .align 2
7. //// 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。
8. \_rs2\_interrupt:
9. 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、ポインタのジャンプテーブルを使用して、対応する割込みソースタイプの処理サブルーチンが // 呼び出されます。各サブルーチンは、処理後にUARTの対応する割込みソースをリセットします。

// 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

1. // とリセットされると、UARTは自動的にIIRビット0 =1を設定し、保留中の割り込みがないことを示すので、割り込みハンドラは終了することができます。
2. rs\_int:
3. pushl %edx
4. pushl %ecx
5. pushl %ebx
6. pushl %eax
7. push %es
8. push %ds /\* as this is an interrupt, we cannot \*/
9. pushl $0x10 /\* know that bs is ok. Load it \*/
10. pop %ds // Let ds, es point to the kernel data segment.
11. pushl $0x10
12. pop %es
13. movl 24(%esp),%edx // get serial buffer queue address.
14. movl (%edx),%edx // get read queue struct address -> edx.
15. movl rs\_addr(%edx),%edx // get serial port 1 (or port 2) base address -> edx.
16. 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。

1. // 割り込みです。ビット2とビット1で決まる割込み条件は4種類あります。モデムの状態変化、書き込み（送信）文字、読み込み（受信）文字、ラインの状態変化。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、それに応じて // 対応する割り込みを処理します。
2. rep\_int:
3. xorl %eax,%eax
4. inb %dx,%al // get interrupt identification byte. 58 testb $1,%al // bit0 = 0 ? (any interrupts ?)
5. jne end // no, jump to end.
6. cmpb $6,%al /\* this shouldn't happen, but ... \*/
7. ja end // if al > 6, jump to end.
8. movl 24(%esp),%ecx // get serial buffer queue address -> ecx.
9. pushl %edx // save the IIR port address temperoly.
10. subl $2,%edx // edx restore to port base address 0x3f8 (0x2f8).
11. 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。

// また、アドレスは4バイトなので、ここでは2倍することで、各割込みタイプのアドレスに対応するジャンプテーブル（79行目）を取得し、そこにジャンプして // 対応する処理を行うことができます。

// 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

1. // 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， // 送信ホールドレジスタ割り込みイネーブルフラグを追加することで， // システムがキャラクタを送信する必要があるときにシリアル割り込みを発生させます。
2. popl %edx // restore IIR port address 0x3fa (or 0x2fa).
3. jmp rep\_int // jump to loop processing any pending interrupts.

1. end: movb $0x20,%al // send EOI instruction to interrupt controller.
2. outb %al,$0x20 /\* EOI \*/
3. pop %ds
4. pop %es
5. popl %eax
6. popl %ebx
7. popl %ecx
8. popl %edx
9. addl $4,%esp # jump over \_table\_list entry # discard the queue address.
10. iret 78
11. // 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。割り込み条件には4種類あります。モデムの状態変化、文字の書き込み（送信）、文字の読み込み（受信）、ラインの状態変化。
12. jmp\_table:
13. .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。MODEMステータスレジスタMSRを読み込むことで、 // リセット操作が行われます。

1. 82 .align 2 83 modem\_status:
2. addl $6,%edx /\* clear intr by reading modem status reg \*/
3. inb %dx,%al // MODEM status register port: 0x3fe
4. 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。ラインステータスレジスタLSRを読み込むことで、 // リセット操作が行われます。

1. 88 .align 2 89 line\_status:
2. addl $5,%edx /\* clear intr by reading line status reg. \*/
3. inb %dx,%al // LSR port: 0x3fd
4. ret 93

// 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()）が呼ばれ、読まれた文字はcanonical modeの // バッファキュー（セカンダリバッファ）に処理されます。

1. // 現在のシリアルポート番号を求めるには、現在のシリアルポートのキューアドレスから // キューのタルベアドレス table\_list を引いて、8で割る。
2. .align 2
3. read\_char:
4. inb %dx,%al // read char in receive buffer register RBR into AL.
5. movl %ecx,%edx // serial buffer queue address -> edx
6. subl $\_table\_list,%edx // current serial no = (queue address - table\_list) / 8
7. shrl $3,%edx
8. movl (%ecx),%ecx # read-queue 101 movl head(%ecx),%ebx // get the head pointer in the read queue -> ebx.
9. movb %al,buf(%ecx,%ebx) // put the char at the head position.
10. incl %ebx // and move the head pointer forward by one.
11. andl $size-1,%ebx // modulate the head pointer by the buffer length.
12. cmpl tail(%ecx),%ebx // compare the head with the tail pointer.
13. je 1f // queue is full if equal, then jump forward.
14. movl %ebx,head(%ecx) // otherwise save the new head pointer.
15. 1: addl $63,%edx // convert serial no to tty no (63 or 64) & push into stack.
16. pushl %edx
17. call \_do\_tty\_interrupt // Call tty int handler C function (tty\_io.c, line 397).
18. addl $4,%esp // discard paras and ret.
19. ret 113

// 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たものが起こされます。そして、書き込みバッファキューの最後から1文字を取り出し、 // 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、 // ラベル write\_buffer\_empty にジャンプして処理します。

1. 114 .align 2 115 write\_char:
2. movl 4(%ecx),%ecx # write-queue // address -> ecx
3. movl head(%ecx),%ebx // get head pointer of write queue to ebx
4. subl tail(%ecx),%ebx // number of chars = head - tail.
5. andl $size-1,%ebx # nr chars in queue
6. je write\_buffer\_empty // if head == tail, the queue is empty, then jump.
7. cmpl $startup,%ebx // check the number of chars in write queue. 122 ja 1f // jump if more than 256 chars.
8. movl proc\_list(%ecx),%ebx # wake up sleeping process // get wait proc list.
9. testl %ebx,%ebx # is there any? # any process waiting to write ?
10. je 1f // none, then jump forward to label 1.
11. movl $0,(%ebx) // otherwise wake process (change state to runnable).
12. 1: movl tail(%ecx),%ebx // get a char from the tail position to AL.
13. movb buf(%ecx,%ebx),%al
14. outb %al,%dx // output to THR register (port 0x3f8 or 0x2f8).
15. incl %ebx // move the tail forward by one.
16. andl $size-1,%ebx // modulate and save the tail pointer.
17. movl %ebx,tail(%ecx)
18. cmpl head(%ecx),%ebx // tail == head ?
19. je write\_buffer\_empty // if true, it means queue is empty, then jump.
20. ret

// 次のコードでは，書き込みバッファのキュー write\_q が空の場合を処理します。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

1. // が再び書き込みキューに入ると、serial.cのrs\_write()関数は、送信保留レジスタが空になったときに // 割り込みを再び発生させることができるので、UARTは書き込みキューの文字を // 「自動的に」取得して送信します。
2. .align 2
3. write\_buffer\_empty:
4. movl proc\_list(%ecx),%ebx # wake up sleeping process
5. testl %ebx,%ebx # is there any?
6. je 1f // no process waiting, jump forward to label 1.
7. movl $0,(%ebx) // otherwise wake up the process.
8. 1: incl %edx // read int enable register IER (0x3f9 or 0x2f9).
9. inb %dx,%al
10. jmp 1f // delay for a while.
11. 1: jmp 1f // masks transmit hold register empty int (bit 1).
12. 1: andb $0xd,%al /\* disable transmit interrupt \*/
13. outb %al,%dx
14. ret

## 10.6 tty\_io.c

### 10.6.1 Function

各ttyデバイスは、tty\_struct構造体（include/linux/tty.h）で定義されている3つのバッファキュー、リードキュー（read\_q）、ライトキュー（write\_q）、セカンダリキュー（secondary）を持っています。各バッファキューにおいて、読み込み操作では、バッファキューの左端から文字を取り出し、バッファのテールポインタを右に移動させ、書き込み操作では、バッファキューの右端に文字を追加し、同じくヘッドポインタを右に移動させます。これら2つのポインタのどちらかがバッファキューの終端を超えて移動した場合は、図10-14のように左に戻してやり直します。

characters

tail

head

Read the char

and the tail

pointer moves to the right.

Write char and the head

pointer moves to the right.

End of Buffer

図10-14 ttyキャラクターバッファキューの動作

tty\_io.cというプログラムは、キャラクターデバイスの上位インターフェース関数を含んでいます。主に端末の読み書き関数tty\_read()とtty\_write()が含まれており、読み込み操作のカノニカルモード関数copy\_to\_cooked()もここに実装されています。

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()による読み込みのための補助キュー（正規モードキュー）に格納する処理を行う。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:
   1. 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);
   2. 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;
   3. If it is a stop character (^S), set the stop flag of the terminal stopped=1;
   4. 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

1. /\*
2. \* 'tty\_io.c' gives an orthogonal feeling to tty's, be they consoles 9 \* or rs-channels. It also implements echoing, cooked mode etc.
3. \*
4. \* Kill-line thanks to John T Kohl, who also corrected VMIN = VTIME = 0.
5. \*/

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。システム上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

// シグナル操作関数のプロトタイプ。

// <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

// と様々な関数が宣言されています。また、\_\_LIBRARY\_\_が定義されている場合、その中には

1. // システムコール番号とインラインアセンブリの\_syscall0()です。
2. #include <ctype.h>
3. #include <errno.h>
4. #include <signal.h>
5. #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

// 初期タスク0のデータと、組み込みアセンブリ関数のマクロ文

// ディスクリプタのパラメータ設定と取得について。

// <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。

// 通信です。

// <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。

// ディスクリプタ/割込みゲートなどを変更することが定義されています。

// <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

// セグメント・レジスタ・オペレート用

1. 21 #include <linux/sched.h> 22 #include <linux/tty.h> 3.
2. #include <asm/segment.h>
3. #include <asm/system.h> 25

// プロセスグループ関数をキルします（プロセスグループにシグナルを送ります）。パラメータ pgrp は // プロセスグループ番号を、sig はシグナルを、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

1. // 0 を返します。そうでない場合、指定されたプロセスグループ番号pgrpのプロセスが見つからないと、 // エラー番号-ESRCHが返されます。プロセスグループ番号が pgrp であるプロセスが見つかったが、 // 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。
2. int kill\_pg(int pgrp, int sig, int priv); // kernel/exit.c, line 171.
3. // 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。そうでなければ0を、そうでなければ1を返します。
4. int is\_orphaned\_pgrp(int pgrp); // kernel/exit.c, line 232.

28

1. // 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， // フラグセットにセットフラグが含まれているかどうかを判断するために使用する．
2. #define \_L\_FLAG(tty,f) ((tty)->termios.c\_lflag & f) // local mode flags.
3. #define \_I\_FLAG(tty,f) ((tty)->termios.c\_iflag & f) // input mode flags.
4. #define \_O\_FLAG(tty,f) ((tty)->termios.c\_oflag & f) // output mode flags. 32
5. // 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。
6. #define L\_CANON(tty) \_L\_FLAG((tty),ICANON) // canonical mode flag.
7. #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.
8. #define L\_ECHOE(tty) \_L\_FLAG((tty),ECHOE) // echo erase flag in canon mode.
9. #define L\_ECHOK(tty) \_L\_FLAG((tty),ECHOK) // KILL erase line flag in canon mode.
10. #define L\_ECHOCTL(tty) \_L\_FLAG((tty),ECHOCTL) // echo control char flag.
11. #define L\_ECHOKE(tty) \_L\_FLAG((tty),ECHOKE) // KILL erace line & echo flags in canon mode 40 #define L\_TOSTOP(tty) \_L\_FLAG((tty),TOSTOP) // send SIGTTOU signal for backgnd output.

41

1. // 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。
2. #define I\_UCLC(tty) \_I\_FLAG((tty),IUCLC) // get uppercase to lowercase flag.
3. #define I\_NLCR(tty) \_I\_FLAG((tty),INLCR) // Map Line feed NL to CR flag on input.
4. #define I\_CRNL(tty) \_I\_FLAG((tty),ICRNL) // Carriage return CR to NL flag.
5. #define I\_NOCR(tty) \_I\_FLAG((tty),IGNCR) // Ignore CR flag.
6. #define I\_IXON(tty) \_I\_FLAG((tty),IXON) // Input control flow flag XON.

47

1. // termios構造の出力モードフラグセットからフラグを取得します。
2. #define O\_POST(tty) \_O\_FLAG((tty),OPOST) // Post-process output.
3. #define O\_NLCR(tty) \_O\_FLAG((tty),ONLCR) // Map NL to CR-NL flag.
4. #define O\_CRNL(tty) \_O\_FLAG((tty),OCRNL) // Map CR to NL flag.
5. #define O\_NLRET(tty) \_O\_FLAG((tty),ONLRET) // NL performs CR function flag.
6. #define O\_LCUC(tty) \_O\_FLAG((tty),OLCUC) // Lowercase to uppercase flag.

53

1. // 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。
2. #define C\_SPEED(tty) ((tty)->termios.c\_cflag & CBAUD)
3. // tty端末がハングアップしているかどうか、つまり、送信ボーレートがB0（0）であるかどうかを判断する。
4. #define C\_HUP(tty) (C\_SPEED((tty)) == B0)

56

1. #ifndef MIN
2. #define MIN(a,b) ((a) < (b) ? (a) : (b))
3. #endif 60

// 以下は，tty端末が使用するバッファキュー構造体配列 tty\_queues と // tty端末テーブル構造体配列 tty\_table を定義しています。QUEUESは、tty端末が使用するバッファキューの最大数 // です。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。

1. // 各ttyターミナルは、3つのttyバッファキューを使用しています。それは、バッファリングされたキーボードまたはシリアル入力のためのreadキュー // read\_queue、バッファリングされたスクリーンまたはシリアル出力のためのwriteキュー // write\_queue、そしてcanonical mode文字を保存するための二次的な補助キューです。
2. #define QUEUES (3\*(MAX\_CONSOLES+NR\_SERIALS+2\*NR\_PTYS)) // total 54 queues.
3. static struct tty\_queue tty\_queues[QUEUES];
4. struct tty\_struct tty\_table[256];

64

// 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。

1. // 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。
2. #define con\_queues tty\_queues
3. #define rs\_queues ((3\*MAX\_CONSOLES) + tty\_queues)
4. #define mpty\_queues ((3\*(MAX\_CONSOLES+NR\_SERIALS)) + tty\_queues)
5. #define spty\_queues ((3\*(MAX\_CONSOLES+NR\_SERIALS+NR\_PTYS)) + tty\_queues) 69

// 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。

1. // 4つのメイン疑似端末では、128から始まり64項目（128 -- 191）までの項目を使用します。 // 4つのスレーブ疑似端末は192から始まる項目を最大64項目(192 -- 255)まで使用します。 70 #define con\_table tty\_table // コンソール端末のttyテーブルのシンボルを定義します。
2. #define rs\_table (64+tty\_table) // Serial terminal tty table.
3. #define mpty\_table (128+tty\_table) // The main pseudo terminal tty table.
4. #define spty\_table (192+tty\_table) // The slave pseudo terminal tty table.

74

75 int fg\_console = 0; // 現在のフォアグラウンドのコンソール番号（0～7の範囲）。

76

1. /\*
2. \* these are the tables used by the machine code handlers.
3. \* you can implement virtual consoles.
4. \*/
5. // ttyの読み書きバッファキュー構造のアドレステーブル。rs\_io.sプログラムが、 // 読み書きバッファキュー構造のアドレスを取得するために使用します。
6. struct tty\_queue \* table\_list[]={
7. con\_queues + 0, con\_queues + 1, // foreground console read/write queue address. 83 rs\_queues + 0, rs\_queues + 1, // serial terminal 1 read/write queue address. 84 rs\_queues + 3, rs\_queues + 4 // serial terminal 2 read/write queue address.

85 };

86

//// 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。

1. // Parameters: new\_console - 指定された新しいコンソール番号です。
2. void change\_console(unsigned int new\_console)
3. {

// パラメータで指定されたコンソールがすでにフォアグラウンドにある場合や、パラメータの

1. // が無効であることを示します。そうでなければ、現在のフォアグラウンド・コンソール番号が設定され、table\_list[]内のフォアグラウンド・コンソールのリードおよびライト・キュー構造のアドレスが更新されます。最後に、現在のフロントコンソール画面を // 更新します。
2. if (new\_console == fg\_console || new\_console >= NR\_CONSOLES)
3. return;
4. fg\_console = new\_console;
5. table\_list[0] = con\_queues + 0 + fg\_console\*3;
6. table\_list[1] = con\_queues + 1 + fg\_console\*3;
7. 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

1. // プロセスへのシグナルがなく、指定されたキューのバッファが空であれば、プロセスを // 割り込み可能なスリープ状態にし、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します。
2. static void sleep\_if\_empty(struct tty\_queue \* queue)
3. {
4. cli();
5. while (!(current->signal & ~current->blocked) && EMPTY(queue))
6. interruptible\_sleep\_on(&queue->proc\_list);
7. sti();
8. }

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 - 指定されたキューへのポインタ。この関数は、プロセスがキューバッファに文字を書き込む前に、 // キューの状態を判断するために呼び出される必要があります。 // キューバッファが一杯になっていない場合には、戻ります。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。

1. // そして，キューのバッファに残っている空き領域が < 128 であれば，プロセスは // 割り込み可能なスリープ状態になり，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なります。
2. static void sleep\_if\_full(struct tty\_queue \* queue)
3. {
4. if (!FULL(queue))
5. return;
6. cli();
7. while (!(current->signal & ~current->blocked) && LEFT(queue)<128)
8. interruptible\_sleep\_on(&queue->proc\_list);
9. sti();
10. }

114

//// キーが押されるのを待ちます。

1. // 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。
2. void wait\_for\_keypress(void)
3. {
4. sleep\_if\_empty(tty\_table[fg\_console].secondary);
5. }

119

//// カノニカルモードの文字列をコピーして変換します。

1. // 指定された端末の読み取りキューにある文字は、端末のtermios構造体に設定された各種フラグに従って、 // カノニカルモード（調理済みモード）の文字にコピー・変換され、補助キューに格納される。
2. void copy\_to\_cooked(struct tty\_struct \* tty)
3. {
4. signed char c; 123
5. // 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． // 3つのキューポインターがすべてNULLの場合、ttyの初期化関数に問題があります。
6. if (!(tty->read\_q || tty->write\_q || tty->secondary)) {
7. printk("copy\_to\_cooked: missing queues\n\r");
8. return; 127 }

// そうでなければ、ttyのリードキューバッファからフェッチされた各文字を適切に処理します

// terminal termios 構造体の入力フラグとローカルフラグに従って、 // 補助キュー secondary に入れます。

// 以下のループ体では、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である場合

文字の//を超えた場合、ループ本体を終了します。それ以外の場合，プログラムは読み取りキューの最後にある // ポインタから1文字を取り出し，テールポインタを1つ前に移動させる。

1. // 文字コードに応じた処理が行われる。また、\_POSIX\_VDISABLE(0) // が定義されている場合、文字コードの値が \_POSIX\_VDISABLE の値と等しいときは、対応する特殊制御文字の機能が // 禁止される。
2. while (1) {
3. if (EMPTY(tty->read\_q))
4. break;
5. if (FULL(tty->secondary))
6. break;
7. 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

1. // 他の文字に変換されます。文字が改行文字 NL(10)で，ラインフィード→キャリッジリターンフラグ NLCR が設定されている場合，CR (13)に変換される。大文字から小文字への入力フラグUCLCが設定されている場合，その文字は小文字に変換される。
2. if (c==13) {
3. if (I\_CRNL(tty))
4. c=10;
5. else if (I\_NOCR(tty))
6. continue;
7. } else if (c==10 && I\_NLCR(tty))
8. c=13;
9. if (I\_UCLC(tty))
10. 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、読み込んだ文字は // 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。

// 一行分の文字である。ttyの補助キューが空ではなく、補助キューでフェッチされた最後の文字が // 改行文字ではない場合 NL(10)、および

// 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

1. // 文字 ERASE を配置する必要があります。続いて，ttyの書き込み関数が呼び出され，書き込みキューの中のすべての // 文字が端末装置に出力される。最後に，tty補助キューの先頭の // ポインタが1バイト後退する。
2. if (L\_CANON(tty)) {
3. if ((KILL\_CHAR(tty) != \_POSIX\_VDISABLE) &&
4. (c==KILL\_CHAR(tty))) {
5. /\* deal with killing the input line \*/
6. while(!(EMPTY(tty->secondary) ||
7. (c=LAST(tty->secondary))==10 ||
8. ((EOF\_CHAR(tty) != \_POSIX\_VDISABLE) &&
9. (c==EOF\_CHAR(tty))))) {
10. if (L\_ECHO(tty)) {
11. if (c<32) // control char ?
12. PUTCH(127,tty->write\_q);
13. PUTCH(127,tty->write\_q);
14. tty->write(tty);
15. }
16. DEC(tty->secondary->head);
17. }
18. continue; // processing other chars in the line.
19. }

// 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）の場合は、追加の消去文字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，書き込みキューの中のすべての文字が端末装置に出力される。 // 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，コードは他のチャラチャラしたものを // 処理し続ける。

1. if ((ERASE\_CHAR(tty) != \_POSIX\_VDISABLE) &&
2. (c==ERASE\_CHAR(tty))) {
3. if (EMPTY(tty->secondary) ||
4. (c=LAST(tty->secondary))==10 ||
5. ((EOF\_CHAR(tty) != \_POSIX\_VDISABLE) &&
6. (c==EOF\_CHAR(tty))))
7. continue;
8. if (L\_ECHO(tty)) {
9. if (c<32)
10. PUTCH(127,tty->write\_q);
11. PUTCH(127,tty->write\_q);
12. tty->write(tty);
13. }
14. DEC(tty->secondary->head);
15. continue;
16. }
17. }

// 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

// 補助キューにある）、他の文字の処理を続けます。その文字が開始文字START(^Q)であれば， // tty停止フラグがリセットされ，tty出力が復元され， // 制御文字が廃棄され，他の文字の処理が継続される。

// 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//。疑似端末の場合は、書き込み操作が中断されます

1. // (chr\_drv/pty.c, 24行目)の理由は、 terminal stopped flagが設定されているからです。シリアル端末の場合は、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りますが、このバージョンは実装されていません。
2. if (I\_IXON(tty)) {
3. if ((STOP\_CHAR(tty) != \_POSIX\_VDISABLE) &&
4. (c==STOP\_CHAR(tty))) {
5. tty->stopped=1;
6. tty->write(tty);
7. continue;
8. }
9. if ((START\_CHAR(tty) != \_POSIX\_VDISABLE) &&
10. (c==START\_CHAR(tty))) {
11. tty->stopped=0;
12. tty->write(tty);
13. continue;
14. }
15. }

// 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(^\_^)であれば、キーボード終了信号SIGQUITがカレントプロセスのプロセスグループ内のすべてのプロセスに // 送られ、次の文字の処理が継続されることになります。

// 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

1. // 処理を行います。同様に、\_POSIX\_VDISABLE(0)が定義されている場合、文字処理中に文字コードの値が\_POSIX\_VDISABLEの値と // 等しくなると、対応する特殊制御文字の機能が禁止される。
2. if (L\_ISIG(tty)) {
3. if ((INTR\_CHAR(tty) != \_POSIX\_VDISABLE) &&
4. (c==INTR\_CHAR(tty))) {
5. kill\_pg(tty->pgrp, SIGINT, 1);
6. continue;
7. }
8. if ((QUIT\_CHAR(tty) != \_POSIX\_VDISABLE) &&
9. (c==QUIT\_CHAR(tty))) {
10. kill\_pg(tty->pgrp, SIGQUIT, 1);
11. continue;
12. }
13. if ((SUSPEND\_CHAR(tty) != \_POSIX\_VDISABLE) &&
14. (c==SUSPEND\_CHAR(tty))) {
15. if (!is\_orphaned\_pgrp(tty->pgrp))
16. kill\_pg(tty->pgrp, SIGTSTP, 1);
17. continue;
18. }
19. }

// 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

1. auxiliary queue に含まれる // を 1 つインクリメントします。関数 tty\_read()で補助キューから文字の行を取ると、行番号は1つずつデクリメントされます // 315行目を参照してください。
2. if (c==10 || (EOF\_CHAR(tty) != \_POSIX\_VDISABLE &&
3. c==EOF\_CHAR(tty)))
4. 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

1. // が設定されている場合は，文字'^'と文字c+64がtty書き込みキューに入れられ（^C，^Hなどが表示される），そうでない場合は，文字が直接tty書き込み//キューに入れられる。最後に，tty書き込み操作関数が呼び出される。
2. if (L\_ECHO(tty)) {
3. if (c==10) {
4. PUTCH(10,tty->write\_q);
5. PUTCH(13,tty->write\_q);
6. } else if (c<32) {
7. if (L\_ECHOCTL(tty)) {
8. PUTCH('^',tty->write\_q);
9. PUTCH(c+64,tty->write\_q);
10. }
11. } else
12. PUTCH(c,tty->write\_q);
13. tty->write(tty);
14. }

// 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．

1. // 最後に、ループ本体を終了した後、AUXキューを待っているプロセスを起こします。
2. PUTCH(c,tty->secondary);
3. }
4. wake\_up(&tty->secondary->proc\_list);
5. }

230

1. /\*
2. \* Called when we need to send a SIGTTIN or SIGTTOU to our process
3. \* group
4. \*
5. \* We only request that a system call be restarted if there was if the
6. \* default signal handler is being used. The reason for this is that if
7. \* a job is catching SIGTTIN or SIGTTOU, the signal handler may not want
8. \* 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
9. \* tty has been released on logout), we don't want to be in an infinite loop
10. \* while restarting the system call, and have it always generate a SIGTTIN
11. \* or SIGTTOU. The default signal handler will cause the process to stop
12. \* thus avoiding the infinite loop problem. Presumably the job-control
13. \* cognizant parent will fix things up before continuging its child process.
14. \*/

//// 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。

1. プロセスグループ内のプロセスが制御端末にアクセスしたときに、 // プロセスグループ内のプロセスが制御端末にアクセスする。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か無視したかに関わらず、 // 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リターンする。
2. int tty\_signal(int sig, struct tty\_struct \*tty)
3. {

// 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

1. ファイル kernel/exit.c の // 232）。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、 // エラーが返されます。そうでない場合は、指定されたシグナルsigが、 // 現在のプロセスグループのすべてのプロセスに送信されます。
2. if (is\_orphaned\_pgrp(current->pgrp))
3. return -EIO; /\* don't stop an orphaned pgrp \*/ 250 (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。

1. // それ以外の場合は、現在のプロセスがシグナルsigの新しいハンドラを設定していれば、 // 割り込み可能な情報を返します。それ以外の場合は、システムコールの再起動後に // 実行可能な情報を返します。
2. if ((current->blocked & (1<<(sig-1))) ||
3. ((int) current->sigaction[sig-1].sa\_handler == 1))
4. return -EIO; /\* Our signal will be ignored \*/
5. else if (current->sigaction[sig-1].sa\_handler)
6. return -EINTR; /\* We \_will\_ be interrupted :-) \*/
7. else
8. return -ERESTARTSYS; /\* We \_will\_ be interrupted :-) \*/
9. /\* (but restart after we continue) \*/ 259 }

260

//// tty read機能。

1. // 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、 // ユーザーが指定したバッファに配置します。パラメータ: channel - サブデバイスの番号, buf - ユーザ // バッファポインタ, nr - 読み込むバイト数。nr - 読み込んだバイト数を返す。
2. int tty\_read(unsigned channel, char \* buf, int nr)
3. {
4. struct tty\_struct \* tty;
5. struct tty\_struct \* other\_tty = NULL;
6. char c, \* b=buf;
7. int minimum,time;

267

// 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

1. // 端末の tty端末の3つのキューポインタがともにNULLの場合は， // EIOのエラーメッセージが返される。tty 端末が疑似端末の場合，疑似端末に対応する別の tty 構造体 other\_tty // を取得する必要がある。
2. if (channel > 255)
3. return -EIO;
4. tty = TTY\_TABLE(channel);
5. if (!(tty->write\_q || tty->read\_q || tty->secondary))
6. 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シグナルを送ってリターンし、 // フォアグラウンドのプロセスグループになるのを待ってから、読み取り操作を行う必要があります。

1. // また、現在の端末が擬似端末の場合、対応する他の擬似端末は、 // other\_ttyとなります。ここでのttyがプライマリ疑似端末であれば、other\_ttyが対応するスレーブ疑似端末であり、 // その逆もまた然りです。
2. if ((current->tty == channel) && (tty->pgrp != current->pgrp))
3. return(tty\_signal(SIGTTIN, tty));
4. if (channel & 0x80)
5. 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 // と VMIN 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 // を決定します。非正規モードでは、これら2つはタイムアウトのタイミング値である。VMINは、読み出し操作を満足させるために必要な、 // 最小限の文字数を表す。

1. // VTIMEは、1/10秒カウントのタイミング値です。
2. time = 10L\*tty->termios.c\_cc[VTIME]; // set read timeout timing value.
3. 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る場合は

// 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

// 最小値よりも小さい場合、次のコードは、指定されたタイムアウト値の時間に従ってプロセスの読み取りタイムアウト値を設定し、残りの文字が読み取られるのを待ちます（328行目を参照）。

// 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

1. // は、カノニカルモードでの読み出し動作では、VTIMEとVMINの対応する制御文字値の制約と制御を受けず、 // 非カノニカルモード（Rawモード）の動作でのみ機能します。
2. if (L\_CANON(tty)) {
3. minimum = nr;
4. current->timeout = 0xffffffff;
5. time = 0;
6. } else if (minimum)
7. current->timeout = 0xffffffff;
8. else {
9. minimum = nr;
10. if (time)
11. current->timeout = time + jiffies;
12. time = 0;
13. }
14. if (minimum>nr)
15. 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。

// 対応する他の擬似端末の書き込み操作機能により、他の擬似端末が // 現在の擬似端末の補助キューに文字を書き込むことができる。

1. // バッファにコピーする。すなわち、他の端末は、書き込みキューバッファの文字を、 // 現在の擬似端末の読み出しキューバッファにコピーし、 // カノニカルモード関数で変換された後、現在の擬似端末の // 補助キューに変換します。
2. while (nr>0) {
3. if (other\_tty)
4. 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場合や、現在のプロセスが信号を受信した場合は、 // 最初にループボディを終了する。そうでない場合、その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ていた場合にも、 // ループボディを終了する。

// 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

1. canonicalモードでユーザーに文字行単位で // データを提供する場合、このモードでは補助キューに少なくとも1行の文字が // ある必要があります。つまり、secondary.data // の最小値は1です。
2. cli();
3. if (EMPTY(tty->secondary) || (L\_CANON(tty) && 298 !FULL(tty->read\_q) && !tty->secondary->data)) {
4. if (!current->timeout ||
5. (current->signal & ~current->blocked)) {
6. sti();
7. break;
8. }
9. if (IS\_A\_PTY\_SLAVE(channel) && C\_HUP(other\_tty))
10. break;
11. interruptible\_sleep\_on(&tty->secondary->proc\_list);
12. sti();
13. continue;
14. }
15. sti();

// 以下は、文字のフェッチ操作の正式な実行を開始する。読み込まれる文字数nrは、nr=0になるか、補助バッファのキューが空になるまで、 // 連続的にデクリメントされる。

// 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 をデクリメントして、読み取るべき文字数を 1 つ減らします。このとき、読み込まれる文字数がすでに0であれば、 // ループは中断される。また、端末が // カノニカルモードであり、読み込んだ文字が改行 NL(10)であった場合も、ループを終了する。

1. // さらに、nr文字が取られておらず、補助キューが空でない限り、 // キュー内の文字は継続して読み込まれる。
2. do {
3. GETCH(tty->secondary,c);
4. if ((EOF\_CHAR(tty) != \_POSIX\_VDISABLE &&
5. c==EOF\_CHAR(tty)) || c==10)
6. tty->secondary->data--;
7. if ((EOF\_CHAR(tty) != \_POSIX\_VDISABLE &&
8. c==EOF\_CHAR(tty)) && L\_CANON(tty))
9. break;
10. else {
11. put\_fs\_byte(c,b++);
12. if (!--nr)
13. break;
14. }
15. if (c==10 && L\_CANON(tty))
16. break;
17. } 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

1. // 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になっている場合や、 // nr文字が読み込まれた場合には、この大きなループを直接終了することができます。
2. wake\_up(&tty->read\_q->proc\_list);
3. if (time)
4. current->timeout = time+jiffies;
5. if (L\_CANON(tty) || b-buf >= minimum)
6. break;
7. }

// 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

1. // 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、まだ文字が読み込まれていない場合は、システムコール番号を再起動することで返され、そうでない場合は、読み込まれた文字数(b-buf)が返される。
2. current->timeout = 0;
3. if ((current->signal & ~current->blocked) && !(b-buf))
4. return -ERESTARTSYS;
5. return (b-buf);
6. }

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ます。

1. // 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 - 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。
2. int tty\_write(unsigned channel, char \* buf, int nr)
3. {
4. static cr\_flag=0;
5. struct tty\_struct \* tty;
6. char c, \*b=buf; 344
7. // まず，パラメータの有効性を判断し，端末の tty 構造体ポインタを取る。tty端末の3つのキューポインタがともにNULLの場合は、 // EIOエラーメッセージが返されます。
8. if (channel > 255)
9. return -EIO;
10. tty = TTY\_TABLE(channel);
11. if (!(tty->write\_q || tty->read\_q || tty->secondary))
12. 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

1. // 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、現在のプロセスグループにSIGTTOU // シグナルを送ってリターンし、フォアグラウンドプロセスグループになるのを待ってから、書き込み操作を行う必要があります。
2. if (L\_TOSTOP(tty) &&
3. (current->tty == channel) && (tty->pgrp != current->pgrp)) 352 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

1. // ループ操作を行います。ループ中、この時点でttyの書き込みキューが満杯の場合、 // 現在のプロセスは割り込み可能なスリープ状態に入ります。現在のプロセスが処理すべき信号を // 持っている場合、ループ本体を終了する。
2. while (nr>0) {
3. sleep\_if\_full(tty->write\_q);
4. if (current->signal & ~current->blocked)
5. break;

// 書き込まれる文字数nrがまだ0より大きく、tty write

1. // キューのバッファがいっぱいにならないと，以下の操作が周期的に行われる。最初にユーザバッファから1バイトを取る。端末の出力モードフラグセットの実行出力処理フラグOPOSTが設定されている場合、その文字に対する後処理動作を行う。
2. while (nr>0 && !FULL(tty->write\_q)) {
3. c=get\_fs\_byte(b);
4. 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。

1. // そうでなければ、その文字が改行文字であり、改行復帰機能フラグONLRETが設定されている場合、 // その文字はキャリッジリターン文字'r'に置き換えられる。
2. if (c=='\r' && O\_CRNL(tty))
3. c='\n';
4. else if (c=='\n' && O\_NLRET(tty))
5. c='\r';
6. // その文字が改行文字'\n'であり、キャリッジリターンフラグcr\_flagが設定されておらず、 // カーニューラインフラグONLCRへのラインフィードが設定されている場合、cr\_flagフラグが設定され、 // CRがライトキューに置かれることになります。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がセットされていれば、 // その文字は大文字に変換される。
7. if (c=='\n' && !cr\_flag && O\_NLCR(tty)) {
8. cr\_flag = 1;
9. PUTCH(13,tty->write\_q);
10. continue;
11. }
12. if (O\_LCUC(tty))
13. c=toupper(c);
14. }

// 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。

1. // cr\_flagフラグがリセットされ、そのバイトがtty write // queueに置かれます。
2. b++; nr--;
3. cr\_flag = 0;
4. PUTCH(c,tty->write\_q);
5. }

// 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。

// ループに入ります。この時点で、対応するttyの書き込み関数が呼び出され、書き込みキューの中の文字を // コンソール画面に表示するか、シリアルポートを通じて送信します。

// 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

1. // tty はシリアルターミナルで、tty->write() は rs\_write() 関数です。まだ書き込むべきバイトが // ある場合は、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。なので、ここでスケジューラーを呼び出して、 // 他のタスクを先に進めます。
2. tty->write(tty);
3. if (nr>0)
4. schedule();
5. }
6. return (b-buf); // finally returns the number of bytes written. 381 }

382

1. /\*
2. \* Jeh, sometimes I really like the 386.
3. \* This routine is called from an interrupt,
4. \* and there should be absolutely no problem 387 \* with sleeping even in an interrupt (I hope).
5. \* Of course, if somebody proves me wrong, I'll
6. \* hate intel for all time :-). We'll have to
7. \* be careful and see to reinstating the interrupt 391 \* chips before calling this, though.
8. \*
9. \* I don't think we sleep here under normal circumstances 394 \* anyway, which is good, as the task sleeping might be 395 \* totally innocent.
10. \*/

//// 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。

1. // 指定されたttyターミナルキューの文字をカノニカル（クックド）モードの文字にコピーまたは変換して、 // 補助キューに格納します。この関数は、シリアルポートの文字読み取り割り込み(rs\_io.s, 110)とキーボード割り込み(kerboard.S, 76)で呼び出されます。
2. void do\_tty\_interrupt(int tty)
3. {
4. copy\_to\_cooked(TTY\_TABLE(tty));
5. }

401

1. //// 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。
2. void chr\_dev\_init(void)
3. {
4. }

405

//// tty端末の初期化機能。

1. // 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。
2. void tty\_init(void)
3. {
4. int i;

409

// まず，すべての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。シリアル端末のリード/ライトバッファキューについては、 // そのデータフィールドにシリアルポートのベースアドレスを設定する。

1. // シリアルポート1は0x3f8、シリアルポート2は0x2f8です。次に、すべての端末のtty構造を // 初期設定します。特殊文字配列c\_cc[]の初期値は、 // include/linux/tty.hファイルで定義されています。
2. for (i=0 ; i < QUEUES ; i++)
3. tty\_queues[i] = (struct tty\_queue) {0,0,0,0,""};
4. rs\_queues[0] = (struct tty\_queue) {0x3f8,0,0,0,""}; // read queue.
5. rs\_queues[1] = (struct tty\_queue) {0x3f8,0,0,0,""}; // write queue.
6. rs\_queues[3] = (struct tty\_queue) {0x2f8,0,0,0,""};
7. rs\_queues[4] = (struct tty\_queue) {0x2f8,0,0,0,""};
8. for (i=0 ; i<256 ; i++) {
9. tty\_table[i] = (struct tty\_struct) {
10. {0, 0, 0, 0, 0, INIT\_C\_CC},
11. 0, 0, 0, NULL, NULL, NULL, NULL
12. }; 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

1. // 435行目で初期化され，それぞれがttyキュー構造体配列tty\_table[]の対応する構造体項目を指しています // 61～73行目の説明を参照してください．
2. con\_init();
3. for (i = 0 ; i<NR\_CONSOLES ; i++) {
4. con\_table[i] = (struct tty\_struct) {
5. {ICRNL, /\* change incoming CR to NL \*/
6. OPOST|ONLCR, /\* change outgoing NL to CRNL \*/
7. 0, // control mode flag set.
8. IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, // local flag set. 429 0, /\* console termio \*/ // 0 -- TTY.
9. INIT\_C\_CC}, // control char array c\_cc[]
10. 0, /\* initial pgrp \*/
11. 0, /\* initial session \*/
12. 0, /\* initial stopped \*/
13. con\_write, // console write function.
14. con\_queues+0+i\*3,con\_queues+1+i\*3,con\_queues+2+i\*3
15. }; 437 }

// そして、シリアルターミナルのtty構造のフィールドを初期化します。450行目の初期化は

1. // シリアルターミナルのtty構造体の読み書きおよび補助バッファキュー構造体は， // ttyバッファキュー構造体配列の対応する構造体項目を指しています // tty\_table[]. 61--73行目の記述を参照。438 for (i = 0 ; i<NR\_SERIALS ; i++) {。
2. rs\_table[i] = (struct tty\_struct) {
3. {0, /\* no translation \*/ // input mode flag set
4. 0, /\* no translation \*/ // output mode flag set.
5. B2400 | CS8, // control mode flag set.2400bps,8bits.
6. 0, // local mode flag set.
7. 0, // line procedure, 0 -- TTY.
8. INIT\_C\_CC}, // control character array.
9. 0, // initial process group.
10. 0, // init session.
11. 0, // initial stopped flag.
12. rs\_write, // serial port write function.
13. rs\_queues+0+i\*3,rs\_queues+1+i\*3,rs\_queues+2+i\*3 // three queues.
14. }; 452 }

// そして，擬似ターミナルが使用する tty 構造体を再初期化する．疑似端末は

// ペアになっている、つまり、マスター疑似端末にスレーブ疑似端末が搭載されているのです。そのため

1. // 初期化する必要があります。以下のループでは，まず各マスター疑似端末の tty 構造体を初期化し， // 各スレーブ疑似端末の対応する tty 構造体を初期化している。
2. for (i = 0 ; i<NR\_PTYS ; i++) {
3. mpty\_table[i] = (struct tty\_struct) {
4. {0, /\* no translation \*/ // input mode flag set.
5. 0, /\* no translation \*/ // output mode flag set.
6. B9600 | CS8, // control mode flag set. 9600bps, 8bits.
7. 0, // local mode flag set.
8. 0, // line procedure, 0--TTY.
9. INIT\_C\_CC}, // control character array.
10. 0, // initial process group.
11. 0, // initial session.
12. 0, // initial stopped flag.
13. mpty\_write, // master pseudo write function.
14. mpty\_queues+0+i\*3,mpty\_queues+1+i\*3,mpty\_queues+2+i\*3
15. };
16. spty\_table[i] = (struct tty\_struct) {
17. {0, /\* no translation \*/
18. 0, /\* no translation \*/
19. B9600 | CS8,
20. IXON | ISIG | ICANON, // local mode flag set.
21. 0,
22. INIT\_C\_CC},
23. 0,
24. 0,
25. 0,
26. spty\_write, // slave pseudo write function.
27. spty\_queues+0+i\*3,spty\_queues+1+i\*3,spty\_queues+2+i\*3
28. }; 480 }
29. // 最後に、シリアル割り込みハンドラとシリアルインターフェース1、2（serial.c、37行目）が // 初期化され、システムに含まれる仮想コンソールの数NR\_CONSOLESと擬似端末の数 // NR\_PTYSが表示されます。
30. rs\_init();
31. printk("%d virtual consoles\n\r",NR\_CONSOLES);
32. printk("%d pty's\n\r",NR\_PTYS);
33. }

485

### 10.6.3 Information

#### 10.6.3.1 Control characters VTIME, VMIN

## 非正規モードでは、この2つの値は、タイムアウトのタイミング値と最小読み取り文字数です。VMINは、読み出し動作を満足させるために必要な最小文字数を示す。VTIMEは、10分の1秒カウントのタイムアウト値です。両方が設定されている場合、読み出し動作は少なくとも1文字が読み込まれるまで待機します。タイムアウトになる前にVMIN文字が受信された場合、読み出し動作は満足されます。VMIN文字を受信する前にタイムアウトが経過した場合、この時点で受信していた文字がユーザーに返されます。VMIN のみが設定されている場合、VMIN 文字が読み込まれるまで読み出し動作は戻りません。VTIME のみが設定されている場合は、少なくとも 1 文字を読み込んだ後、またはタイムアウトした後、直ちにリードが復帰します。どちらも設定されていない場合は、現在読み込まれているバイト数のみを表示して、直ちに復帰します。詳細は、termios.hファイルを参照してください。

## 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

1. /\*
2. \* linux/kernel/chr\_drv/tty\_ioctl.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が含まれています。 // <termios.h> 端末入出力機能のヘッダファイルです。主に非同期通信ポートを制御するターミナル // インターフェースを定義しています。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <linux/tty.h> ttyヘッダーファイルは、tty\_ioというシリアル通信のためのパラメータや定数を定義しています。

// <asm/io.h> Io のヘッダーファイルです。ioポートを操作する関数を、 // マクロの組み込みアセンブラの形で定義する。

// <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

// セグメント・レジスタ・オペレーションのための

// <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。

1. // のディスクリプター/割り込みゲートなどが定義されています。
2. #include <errno.h>
3. #include <termios.h>

9

1. #include <linux/sched.h>
2. #include <linux/kernel.h>
3. #include <linux/tty.h>

13

1. #include <asm/io.h>
2. #include <asm/segment.h>
3. #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, 161 行目で定義されている) に従って所属する．2 番目の関数 tty\_signal()は，プロセスグループ内のすべてのプロセスで， // 指定された tty 端末を使用している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ます）．

// 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。

1. // ボーレートとボーレートファクターを設定します。例えば、ボーレートが2400bpsの場合、対応する // ファクターは48（0x30）、9600bpsのファクターは12（0x1c）となります。プログラムリストの後にある命令を参照してください。
2. extern int session\_of\_pgrp(int pgrp);
3. extern int tty\_signal(int sig, struct tty\_struct \*tty);

20

1. static unsigned short quotient[] = {
2. 0, 2304, 1536, 1047, 857,
3. 768, 576, 384, 192, 96,
4. 64, 48, 24, 12, 6, 3
5. };

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

1. // 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、書き込み後にDLABビットがリセットされます。シリアルポート2の場合、2つのポートは0x2f8と0x2f9です。
2. static void change\_speed(struct tty\_struct \* tty)
3. {
4. 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フィールドを格納する

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

1. 端末のtermios構造体の制御モードフラグセットから // が設定されており，ボーレート係数配列 quotient[]から対応するボーレート係数値を // 得る。CBAUD // は，制御モードフラグセットのボーレートビットマスクである。
2. if (!(port = tty->read\_q->data))
3. return;
4. 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。書き込みを行う前に，まず，除算器のラッチアクセスを

1. // ラインコントロールレジスタLCRのDLABビット（ビット7）をリセットしてから、16ビットボーレートファクターのローバイトとハイバイトをそれぞれポート0x3f8, 0x3f9に書き込みます。最後にLCRのDLABフラグビットを // リセットします。
2. cli();
3. outb\_p(0x80,port+3); /\* set DLAB \*/
4. outb\_p(quot & 0xff,port); /\* LS of divisor \*/
5. outb\_p(quot >> 8,port+1); /\* MS of divisor \*/
6. outb(0x03,port+3); /\* reset DLAB \*/
7. sti();
8. }

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。

1. // 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。
2. static void flush(struct tty\_queue \* queue)
3. {
4. cli();
5. queue->head = queue->tail;
6. sti();
7. }

48

1. static void wait\_until\_sent(struct tty\_struct \* tty)
2. {
3. /\* do nothing - not implemented \*/
4. }

53

1. static void send\_break(struct tty\_struct \* tty)
2. {
3. /\* do nothing - not implemented \*/
4. }

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構造体のポインタを指定する; termios - termios構造体を保持するために使用された // ユーザーバッファ。

59 static int get\_termios(struct tty\_struct \* tty, struct termios \* termios) 60 {。

61 int i; 62

1. // まず，ユーザバッファポインタが示すメモリ領域が十分かどうかを確認します。そうでなければ， // メモリを確保する。次に，指定された端末のtermios構造情報を // ユーザバッファにコピーする。最後に0を返す。
2. verify\_area(termios, sizeof (\*termios)); // kernel/fork.c, line 24.
3. for (i=0 ; i< (sizeof (\*termios)) ; i++)
4. put\_fs\_byte( ((char \*)&tty->termios)[i] , i+(char \*)termios );
5. return 0;
6. }

68

//// 端末のtermios構造に情報を設定します。

1. // パラメータ： tty - 端末のtty構造体ポインタを指定する; termios - ユーザーデータエリアの // termios構造体ポインタを指定する。
2. static int set\_termios(struct tty\_struct \* tty, struct termios \* termios,
3. int channel)
4. {
5. int i, retsig;

73

1. 74 /\* ターミナルの状態を設定しようとしたときに、自分がフォアグラウンドにいない場合は、SIGTTOUを送信します。 もしシグナルがブロックされたり
2. 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

1. // 実行を停止して、先にtermios構造体を修正できるようにします。ただし、送信関数tty\_signal()がERESTARTSYSまたはEINTRの値を返した場合には、 // 後で再び操作が行われることになります。
2. if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
3. retsig = tty\_signal(SIGTTOU, tty); // chr\_drv/tty\_io.c, line 246.
4. if (retsig == -ERESTARTSYS || retsig == -EINTR)
5. return retsig;
6. }

// 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

// シリアルポートの送信ボーレートは、シリアルUARTチップ内のボーレートがtermiosの制御モードフラグc\_flagのボーレート情報に従って、 // 再度変更されます。

1. // 構造体を作成し、最後に0.を返します。
2. for (i=0 ; i< (sizeof (\*termios)) ; i++)
3. ((char \*)&tty->termios)[i]=get\_fs\_byte(i+(char \*)termios);
4. change\_speed(tty);
5. return 0;
6. }

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 - termio構造体の情報を保持する // ユーザーバッファ。

1. 88 static int get\_termio(struct tty\_struct \* tty, struct termio \* termio) 89 {。
2. int i;
3. 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

1. // 後者は短い。したがって、一時termio構造体へのコピーの目的は、 // データ型の変換のためである。最後に，一時termio構造体の情報をバイトごとにユーザバッファにコピーして // 0を返す。
2. verify\_area(termio, sizeof (\*termio));
3. tmp\_termio.c\_iflag = tty->termios.c\_iflag;
4. tmp\_termio.c\_oflag = tty->termios.c\_oflag;
5. tmp\_termio.c\_cflag = tty->termios.c\_cflag;
6. tmp\_termio.c\_lflag = tty->termios.c\_lflag;
7. tmp\_termio.c\_line = tty->termios.c\_line;
8. for(i=0 ; i < NCC ; i++)
9. tmp\_termio.c\_cc[i] = tty->termios.c\_cc[i];
10. for (i=0 ; i< (sizeof (\*termio)) ; i++)
11. put\_fs\_byte( ((char \*)&tmp\_termio)[i] , i+(char \*)termio );
12. return 0;
13. }

105

1. /\*
2. \* This only works as the 386 is low-byt-first
3. \*/

//// 端末のtermio構造情報を設定します。

1. // パラメータ: tty - 端末のtty構造体ポインタを指定する; termio - ユーザデータ領域の termio // 構造体。この関数は、ユーザーバッファ termio の情報を、 // 端末の termios 構造体にコピーして、0 を返す。
2. static int set\_termio(struct tty\_struct \* tty, struct termio \* termio,
3. int channel)
4. {
5. int i, retsig;  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、このターミナル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。

1. // 実行を一時的に停止することで、先にtermios構造体を修正することができます。ただし、 // 送信関数 tty\_signal() が ERESTARTSYS または EINTR の値を返した場合は、 // 後で再び操作が行われることになります。
2. if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
3. retsig = tty\_signal(SIGTTOU, tty);
4. if (retsig == -ERESTARTSYS || retsig == -EINTR)
5. return retsig;
6. }

// 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場合，つまり， // termio の短整数型から termios の長整数型に変換するためのものである。しかし、2つのフィールドのc\_lineとc\_cc[]は

1. // の構造は同じです。
2. for (i=0 ; i< (sizeof (\*termio)) ; i++)
3. ((char \*)&tmp\_termio)[i]=get\_fs\_byte(i+(char \*)termio);
4. \*(unsigned short \*)&tty->termios.c\_iflag = tmp\_termio.c\_iflag;
5. \*(unsigned short \*)&tty->termios.c\_oflag = tmp\_termio.c\_oflag;
6. \*(unsigned short \*)&tty->termios.c\_cflag = tmp\_termio.c\_cflag;
7. \*(unsigned short \*)&tty->termios.c\_lflag = tmp\_termio.c\_lflag;
8. tty->termios.c\_line = tmp\_termio.c\_line;
9. for(i=0 ; i < NCC ; i++)
10. tty->termios.c\_cc[i] = tmp\_termio.c\_cc[i];
11. // 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、シリアルUARTチップのボーレートは、termios構造体の制御モードフラグc\_flagのボーレート情報に従って変更され、0が返されます。
12. change\_speed(tty);
13. return 0;
14. }

132

//// tty端末装置の入出力制御機能。

1. // パラメータ： dev - デバイス番号， cmd - ioctlコマンド， arg - 操作パラメータポインタ。 // この関数は，まず，デバイス番号に応じて，対応する端末の // tty 構造体を見つけ，次に，制御コマンド cmd に応じて処理を行う。
2. int tty\_ioctl(int dev, int cmd, int arg)
3. {
4. struct tty\_struct \* tty;
5. 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

1. // マシンが停止します。メジャーデバイス番号が5ではなく4の場合は、デバイス番号からサブデバイス番号を // 得ることができます。サブデバイス番号は、0（コンソール端末）、1（シリアル1端末）、2（シリアル2端末）のいずれかです。
2. if (MAJOR(dev) == 5) {
3. dev=current->tty;
4. if (dev<0)
5. panic("tty\_ioctl: dev<0");
6. } else
7. 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]となります。dev = 1の場合、対応する端末のtty構造は、 // tty\_table[0]となります。tty\_io.cプログラムの70〜73行目をご覧ください。

144 tty = tty\_table + (dev ? ((dev < 64)? dev-1:dev) : fg\_console) となります。

// TCGETS: 対応する端末のtermios構造情報を取得します。この時点では， // パラメータargは，ユーザバッファポインタである。

// TCSETSF: termios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 // 待ち、入力キューをフラッシュする必要があります。その後、ターミオスを設定する操作を行います。

// TCSETSW：termiosを設定する前に、出力キュー内のすべてのデータが処理されるのを // 待つ必要があります。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。// TCSETSです。対応するターミナルのtermios構造情報を設定する。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ある。

// TCGETA: 対応する端末のtermio構造体の情報を取得する。この時点では、 // パラメータargは、ユーザバッファポインタである。

// 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、 // 入力キューがフラッシュされるのを待つ必要があります。その後、端末のtermio設定操作を行います。

TCSETAW：termiosを設定する前に、出力キュー内のすべてのデータが処理されるのを // 待つ必要があります。この形式は、paramsを修正すると出力に影響する状況で必要となる。 // TCSETA: 対応する端末のターミオ構造情報を設定する。このとき、 // 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ある。

// TCSBRK：パラメータargの値が0の場合、出力キューの処理を待って // ブレークを送ります。

145 switch (cmd) { 146 case TCGETS:

147 return get\_termios(tty,(struct termios \*) arg); 148 case TCSETSF:

149 flush(tty->read\_q); /\* フォールスルー \*/ 150 case TCSETSW:

1. 151 wait\_until\_sent(tty); /\* fallthrough \*/ 152 case TCSETS:
2. return set\_termios(tty,(struct termios \*) arg, dev);
3. case TCGETA:
4. return get\_termio(tty,(struct termio \*) arg); 156 case TCSETAF:

157 flush(tty->read\_q); /\* フォールスルー \*/ 158 case TCSETAW:

1. 159 wait\_until\_sent(tty); /\* fallthrough \*/ 160 case TCSETA:
2. return set\_termio(tty,(struct termio \*) arg, dev);
3. case TCSBRK:
4. if (!arg) {
5. wait\_until\_sent(tty);
6. send\_break(tty);
7. }
8. 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

1. // 出力を中断または再開すると、ユーザのインタラクションの反応を速めるために、 // 書き込みキューの中の文字を出力する必要がある。argがTCIOFF(Terminal Control Input ON)であれば、 // 入力を一時停止し、TCIONであれば、保留中の入力を再開する。
2. case TCXONC:
3. switch (arg) {
4. case TCOOFF:
5. tty->stopped = 1; // stop the terminal output.
6. tty->write(tty); // write queue output. 173 return 0;
7. case TCOON:
8. tty->stopped = 0; // restore the terminal output.
9. tty->write(tty);
10. 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、端末がその文字を受け取ると // 入力が中断されることを意味する。パラメータがTCIONの場合は、START文字を送信して // 端末に送信を再開させることを意味する。

// 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

1. を端末の書き込みキューに入れることができます。以下も同様である。
2. case TCIOFF:
3. if (STOP\_CHAR(tty))
4. PUTCH(STOP\_CHAR(tty),tty->write\_q); 181 return 0;
5. case TCION:
6. if (START\_CHAR(tty))
7. PUTCH(START\_CHAR(tty),tty->write\_q);
8. return 0;
9. }
10. return -EINVAL; /\* not implemented \*/

// 書き込まれたがまだ送信されていないデータや、受信したがまだ読まれていないデータをフラッシュします。もし、そのような

1. // パラメータ arg が 0 の場合は、入力キューがフラッシュされ、1 の場合は、出力キューがフラッシュされ、 // 2 の場合は、入力キューと出力キューの両方がフラッシュされます。
2. case TCFLSH:
3. if (arg==0)
4. flush(tty->read\_q);
5. else if (arg==1)
6. flush(tty->write\_q);
7. else if (arg==2) {
8. flush(tty->read\_q);
9. flush(tty->write\_q);
10. } else
11. return -EINVAL;
12. return 0;

// TIOCEXCL：端末のシリアルラインの専用モードを設定します。

// TIOCNXCL：端末のシリアルライン専用モードをリセットします。 // TIOCSCTTY: tty を制御端末として設定します。(TIOCNOTTY - 制御端末なし)。

1. // TIOCGPGRP: 端末のプロセスグループ番号を読み取る（つまり、フォアグラウンドのプロセスグループ // 番号を読み取る）。まずユーザバッファの長さを確認し、次に端末ttyのpgrpフィールドを // ユーザバッファにコピーする。この時、パラメータ arg はユーザバッファポインタである。
2. case TIOCEXCL:
3. return -EINVAL; /\* not implemented \*/ 201 case TIOCNXCL:
4. 202 return -EINVAL; /\* not implemented \*/ 203 case TIOCSCTTY:
5. return -EINVAL; /\* set controlling term NI \*/
6. case TIOCGPGRP: // implement function tcgetpgrp().
7. verify\_area((void \*) arg,4);
8. 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。このコマンドを実行するための // 前提条件は、プロセスが制御端末を持っていることである。

// 現在のプロセスが制御端末を持っていないか、devがその制御端末ではないか、 // 制御端末は現在処理中のdev端末であるが、プロセスのセッション番号がdev端末のセッション番号と // 異なっている場合、terminalless // エラーメッセージが返される。

// 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。

1. をprgpに変更します。この時点で、prgpがフォアグラウンドのプロセスグループになります。
2. case TIOCSPGRP: // implement function tcsetpgrp().
3. if ((current->tty < 0) ||
4. (current->tty != dev) ||
5. (tty->session != current->session))
6. return -ENOTTY;
7. pgrp=get\_fs\_long((unsigned long \*) arg); // get from user buffer.
8. if (pgrp < 0)
9. return -EINVAL;
10. if (session\_of\_pgrp(pgrp) != current->session)
11. return -EPERM;
12. tty->pgrp = pgrp;
13. return 0;

// TIOCOUTQ: 出力キューに残っている文字数を返します。最初にユーザバッファの長さを確認し、次にキューの中の文字数をユーザにコピーします。この時点では、 // パラメータargはユーザバッファポインタです。

// 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

1. // ユーザーバッファの長さがまず検証され、その後、補助キューの文字数が // ユーザーにコピーされる。この時点で、パラメータ arg はユーザバッファポインタである。
2. case TIOCOUTQ:
3. verify\_area((void \*) arg,4);
4. put\_fs\_long(CHARS(tty->write\_q),(unsigned long \*) arg); 224 return 0;
5. case TIOCINQ:
6. verify\_area((void \*) arg,4);
7. put\_fs\_long(CHARS(tty->secondary), 228 (unsigned long \*) arg);
8. 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，パラメータとして文字へのポインタをとり， // その文字が端末上で入力されたものと仮定する。ユーザは制御端末に対して， // スーパーユーザ権限または読み取り権限を持っていなければならない。

// 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）。

1. // 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）。
2. case TIOCSTI:
3. return -EINVAL; /\* not implemented \*/ 232 case TIOCGWINSZ:
4. 233 return -EINVAL; /\* not implemented \*/ 234 case TIOCSWINSZ:
5. 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。

1. // 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ると，対応するステータスピンがアサートされます。
2. case TIOCMGET:
3. return -EINVAL; /\* not implemented \*/ 238 case TIOCMBIS:

239 return -EINVAL; /\* not implemented \*/ 240 case TIOCMBIC:

1. 241 return -EINVAL; /\* not implemented \*/ 242 case TIOCMSET:
2. return -EINVAL; /\* not implemented \*/

10.8 まとめ

TIOCGSOFTCARです。ソフトウェアキャリア検出フラグを読み取る（1 - オン、0 - オフ）。

1. // tiocssoftcar: 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）。
2. case TIOCGSOFTCAR:
3. return -EINVAL; /\* not implemented \*/ 246 case TIOCSSOFTCAR:
4. 247 return -EINVAL; /\* not implemented \*/ 248 default:
5. return -EINVAL;
6. }
7. }

252

### 10.7.3 Information

#### 10.7.3.1 Baud Rate and Baud Rate Factor

ボーレートは、ボーレート = 1.8432MHz / (16 X ボーレート係数)で計算されます。

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

## 10.8 Summary

本章では、キャラクタ・デバイスの実装について詳細に説明し、シリアル通信に基づくローカル・ターミナル・デバイスとダム・ターミナルの動作原理と実装コードを紹介する。まず、UNIXオペレーティングシステムのキャラクターデバイスにアクセスするモジュールの階層関係と、いくつかの一般的な端末の種類を示す。続いて、端末が使用するtermiosデータ構造と3つのバッファキュー構造を組み合わせ、正規モード（cooked mode）と非正規モード（raw mode）の定義とそのプログラム実装を詳細に説明し、コンソール端末とシリアル端末についても説明した。そして、各プログラムをファイルの順に詳しく紹介し、注釈をつけました。

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、読者がカーネルの実装を完全に理解することを妨げることはありません。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

-

17:15:34

01

convert.c

4348

bytes

1992

-

01

-

01

19:07:43

div.c

2099

bytes

1992

-

01

-

01:41:43

01

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

-

01:38:13

01

math\_emulate.c

11540

bytes

1992

-

01

-

21:12:05

07

mul.c

bytes

1517

1992

-

01

-

01

01:42:33



## 11.1 Function Description

コンピュータで計算量の多い演算を行うには、通常、3つの方法があります。1つは、CPUの通常の命令を使って直接計算を行う方法。CPUの通常の命令は汎用命令の一種であるため、複雑で大規模な計算を行うためには、複雑な計算サブルーチンが必要となり、一般的には数学やコンピュータに精通した人でなければ、このサブルーチンをプログラムすることはできないと言われている。また、CPUに専用の数学コプロセッサチップを構成する方法もあります。コプロセッサチップを使用することで、数学的プログラミングの難易度を大幅に下げることができ、演算の速度や効率も2倍になりますが、追加のハードウェア投資が必要です。また、システムのカーネルレベルでエミュレータを使い、コプロセッサの演算機能をシミュレートする方法もあります。この方法は、速度と効率が最も低い方法かもしれませんが、コプロセッサを使って計算プログラムを組むのと同じくらい簡単で、コプロセッサを搭載したマシン上でプログラムを変更することなく実行できるので、コードの互換性があります。

Linux 0.1x、さらには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の

(またはその互換チップ）は高価であり、一般のPCでは常に贅沢品であった。そのため、科学的な計算を大量に行う場合や、特に必要とされる場所でない限り、一般のPCに80387チップが搭載されることはない。現在のインテル・プロセッサーには数学コプロセッサー機能が組み込まれているが、現在のOSではコプロセッサー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。

### 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。

### 11.1.1 Floating point data type

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。

1. 整数データタイプ

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。たとえば、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

10進数の139に対応しています。通常、重みが最も小さいもの（2^0）をLSB（Least Significant Bit）と呼び、重みが最も大きいもの（2^7）をMSB（Most Significant Bit）と呼びます。

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 |

2の補数表記は、CPUの符号なし数字の単純な加算が、この形式のデータ演算にも適用できることから、現在、ほとんどのコンピュータのCPUが使用している整数表現である。この表記法では、負の数は、各ビットを反転させた後の数に1を加えたものとなる。MSBビットは、数値の符号ビットです。正の数の場合はMSB=0、負の数の場合はMSB=1となります。80386 CPUは、8ビット（1バイト）、16ビット（1ワード）、32ビット（2ワード）の2の補数データタイプを持っています。それぞれが表現できるデータ範囲は、-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。

Byte

1



S

Not used

Sign

Bit

Number

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

1. 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の圧縮BCDコード表現は、0x01011001となります。非圧縮BCDコードでは、各バイトは下位4ビットのみで1桁の10進数を表す。

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。

Byte

1



d17 d16



d15 d14



d13 d12



d11 d10



d9 d8



d7 d6



d5 d4



d3 d2



d1 d0

S

X

Sign

図11-2 80387がサポートするBCDコードデータタイプ

1. 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」ことがあるため、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。

Value =

(

-

1)

S

\* Significand

\* 2

Exp

Sign

Exponent

Significand

S

図11-3 浮動小数点の一般的なフォーマット

Single Real

)

Short Real

(

Double Real

)

(

Long Real

79

64

63

62

0

63

52

51

0

1

b

yte



S



Exp.

(8)



S

Exp.

(11)

S

Exp.

(15)

1

Fraction

(64)

Fraction

(52)

F

raction(23)

s

how 1 after normalized

Hidden 1

Hidden 1

23

31

22

0

Extended Real

Temporary Real

)

(

図11-4 80387コプロセッサの実数フォーマット

図11-4において、Sは1ビットの符号ビットである。S=1は負の実数を示し、S=0は正の実数を示す。Significandは、実数の有効桁または仮数を示します。指数形式を使うと、実数をさまざまな形で表現することができます。例えば、10進数の実数10.34は、1034.0×10^-2、10.34×10^0、1.034×10^1、0.1034×10^2などと表すことができます。計算で最大の精度値を得るためには、必ず実数の正規化、つまり2進数の最上位値が常に1で、小数点が右側になるように実数の指数値を調整します。したがって、上の例の正しい正規化の結果は1.034×10^1となります。2進数の場合は、1.XXXX \* 2^N（Xは1または0）となります。実数を表すのに常にこの形式を使用するならば、小数点の左が1でなければなりません。そのため、80387の短実数（単精度）や長実数（倍精度）の形式では、この「1」を明示的に表現する必要はない。したがって、短実数や長実数の2進数では、0x0111...010は、実際には0x1.0111...010となります。

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に明示的に1が置かれます。

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。

1. typedef struct {
2. long a,b; // a is lower 32 bits, b is upper 32 bits (including 1 fixed bit).
3. short exponent;
4. } temp\_real;

中でも64ビットの仮数は、2つの長い変数で表される。変数aは下位32ビット、bは上位32ビット（1固定ビットを含む）である。なお、当時のデータ構造では、gccコンパイラのバイトアライメントの問題を解決するために、同じ機能を持つ別の構造体が定義されています。

1. typedef struct {
2. short m0,m1,m2,m3;
3. short exponent;
4. } temp\_real\_unaligned;

4. 特殊な実数

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ことになる。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  Quiet NaNs |
| 0/1 | 11...11 | 1 | ... |
| 0/1 | 11...11 | 1 | 10...00 | Indefinite |
| 0/1 | 11...11 | 1 | 01...11 | Signalling NaNs - SNaNs |
| 0/1 | 11...11 | 1 | ... |
| 0/1 | 11...11 | 1 | 00...01 |
| 0/1 | 11...11 | 1 | 00...00 | Infinite |
| 0/1 | 11...10 | 1 | 11...11 | Normals |
| 0/1 | ... | 1 | ... |
| 0/1 | 00...01 | 1 | 00...00 |
| 0/1 | 00...00 | 1 | 11...11 | Pseudo-Denormals |
| 0/1 | 00...00 | 1 | ... |
| 0/1 | 00...00 | 1 | 00...00 |
| 0/1 | 00...00 | 0 | 11...11 | Denormals |
| 0/1 | 00...00 | 0 | ... |
| 0/1 | 00...00 | 0 | 00...01 |
| 0/1 | 00...00 | 0 | 00...00 | Zero |

ゼロとは、指数値と符号値がともに0である値で、指数値が0の残りの値は予約されています。つまり、指数値が0の値は、通常の実数値を表すことができません。無限とは、指数の値がすべて1、シニフィカントの値がすべて0の値で、残りの指数の値が0x11...11の値もすべて予約されている値です。

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、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。

### 11.1.2 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。

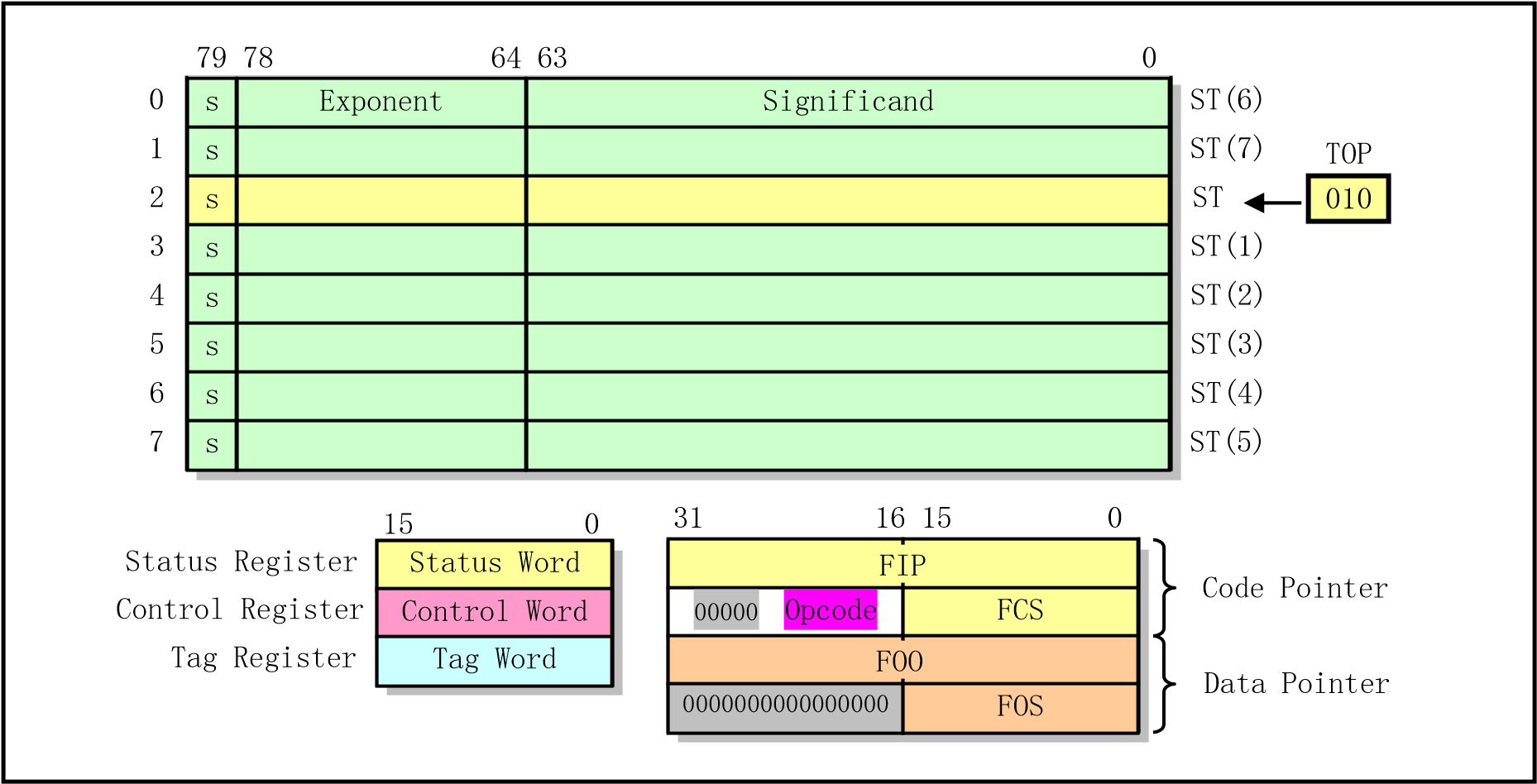


図11-5 80387のレジスタ

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。

1. 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んでスタックにプッシュし、浮動小数点ストア命令は、スタックの現在の先頭から値を取り出してメモリに書き込みます。スタック内の値が不要になった場合には、同時にポップ演算を行うこともできます。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。
2. 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。

15 8 7 0

Control Word 0 0 0 0 RC PC 0 1 PM UM OM ZM DM IM

Rounding Control Field Precision Control Field Exception Masks

RC Rounding Mode PC Precision Stat Mask Condition

* 1. Round to nearest(even) 00 24-bit Sigle Precision IE IM Invalid Operation
  2. Round down(toward -∞) 01 Reserved DE DM Denormal OPed

10 切り上げ（+∞方向） 10 53ビット（Long Real） ZE ZM Zero Divide 11 0方向への切り上げ（Truncat） 11 64ビット（Temp Real） OE OM Overflow

UE UM Underflow

PE PM Precision

15 8 7 0

ステータスワード B C3 TOP C2 C1 C0 ES SF PE UE OE ZE DE IE IE

Day Word Day7 Day6 Day5 Day4 Day3 Day2 Day1 Day0

Status Word Field Tag Word coding

Field Description Tag Accumulator

PE,UE,OE, These bits indicate exceptions that occurred while 00 Valid

ZE,DE,IE executing fp instruction. Must be reset by program. 01 Zero

Stack error flag. This flag is set if the invalid Special:invalid (NaN,

SF operation is due to the overflow of the accumulator, 10 non-numeric, Denormal)

otherwise it is automatically reset. 11 Empty

FPU Busy, Error Summary Statusの略。マスクされていないものがあれば

B,ES exception in bit 5-0 occurs, these 2 bits are set,

それ以外の場合はすべて0です。

C3,C2, CPUのEFLAGSと同様の条件コードが使用される C1,C0は浮動小数点比較命令に使用される。

Top of Stack Pointerのことで、どの80ビットを指すかを示します。

現在、スタックの最上位にあるTOPレジスタ、すなわちアキュムレータSTに対応する物理レジスタはどれか。

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

1. 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

機能説明

1. 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。
2. Status Word

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つの例外すべてが、CPUに同じ例外int16を発生させるからです。C. タグ・ワード

1. 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。
2. Error-Pointer Register
3. 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個のポインターを含む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。
4. 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 |  |  |

* 1. 表中のフィールドの意味は以下のとおりです（これらのフィールドの具体的な意味や詳細な説明については、80X86プロセッサのマニュアルを参照してください）。
  2. OP (Operation opcode) is the instruction opcode. In some instructions, it is divided into two parts: OPA and OPB.
  3. 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.
  4. P(Pop) indicates whether to perform a popup process after the operation: 0 - no need; 1 - pop up the stack after the operation.
  5. d (destination) indicates the accumulator that saves the result of the operation: 0 - ST(0); 1 - ST(i).
  6. MOD (Mode) and R/M (Register/Memory) are the operation mode field and the operand position field.
  7. SIB (Scale Index Base) and DISP (Displacement) are optional follow-up fields with MOD and R/M field instructions.
  8. 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。
  9. FI All instructions for operating integer data start with FI, such as FIADD, FILD, etc.
  10. FB All instructions for operating BCD type data start with FB, such as FBLD, FBST, etc.
  11. FxxP All instructions that perform a pop operation are terminated with the letter P, such as FSTP, FADDP, etc.
  12. FxxPP All instructions that perform two pop operations are terminated with the letter PP, such as
  13. FCOMPP」「FUCOMPP」など。
  14. 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が浮動小数点命令に遭遇したときに例外int7が発生し、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。

1. struct info {
2. long \_\_\_math\_ret; // caller (int7) return address.
3. long \_\_\_orig\_eip; // a place where the original EIP is temporarily saved.
4. long \_\_\_edi; // registers pushed on stack during int7 processing.
5. long \_\_\_esi;
6. long \_\_\_ebp;
7. long \_\_\_sys\_call\_ret; // process system-call's return code.
8. long \_\_\_eax; // below are the same as stacks of the system-call.
9. long \_\_\_ebx;
10. long \_\_\_ecx;
11. long \_\_\_edx;
12. long \_\_\_orig\_eax; // its -1 if not a system-call.
13. long \_\_\_fs;
14. long \_\_\_es;
15. long \_\_\_ds;
16. long \_\_\_eip; // pushed by CPU automatically.
17. long \_\_\_cs;
18. long \_\_\_eflags;
19. long \_\_\_esp;
20. long \_\_\_ss;
21. };

do\_emu()関数(52行目)では，まずステータスワードに基づいて，シミュレーションのためのコプロセッサ内部例外があるかどうかを判断します．そして、コプロセッサ例外を発生させる2バイトの浮動小数点命令コードを上記info構造のEIPフィールドから取得し、そこに含まれるESCコード（バイナリ11011）のビット部分をマスクした上で、ソフトウェアのシミュレーション演算処理を行うために使用します。本機能では、処理を容易にするために、5つのスイッチ文を用いて5種類の浮動小数点命令コードを処理します。例えば、最初のスイッチ・ステートメント（75行目）は、メモリ・オペランドのアドレス指定を伴わない浮動小数点命令を処理するために使用され、最後の2つのスイッチ・ステートメント（419行目、432行目）は、メモリに関連するオペランドを持つ命令を処理するために専用に使用されます。後者のタイプの命令の場合、基本的な処理の流れは、まず命令コードのアドレッシング・モード・バイトに従ってメモリ・オペランドの実効アドレスを取得し、次にその実効アドレスから対応するデータ（整数、実数、またはBCDコード値）を読み取る。読み出された値は、80387の内部処理で使用される一時的な実数形式に変換されます。演算終了後、一時的な実数形式の値は元のデータ型に変換され、最終的にユーザーデータ領域に保存されます。

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、カレント・プロセスに指定されたシグナルを送り、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

1. /\*
2. \* Limited emulation 27.12.91 - mostly loads/stores, which gcc wants
3. \* even for soft-float, unless you use bruce evans' patches. The patches
4. \* 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.
5. \*
6. \* 28.12.91 - loads/stores work, even BCD. I'll have to start thinking
7. \* about add/sub/mul/div. Urgel. I should find some good source, but I'll 16 \* just fake up something.
8. \*
9. \* 30.12.91 - add/sub/mul/div/com seem to work mostly. I should really 19 \* test every possible combination.

20 \*/

21

1. /\*
2. \* This file is full of ugly macros etc: one problem was that gcc simply
3. \* didn't want to make the structures as they should be: it has to try to
4. \* 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.
5. \*
6. \* 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

// シグナル操作の関数のプロトタイプ。

// <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

// 浮動小数点表現構造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。のプロトタイプ定義が含まれています。

// カーネルのよく使う機能

// <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

// セグメントレジスタ操作のための 33 #include <signal.h> (英語)

34

1. #define \_\_ALIGNED\_TEMP\_REAL 1
2. #include <linux/math\_emu.h>
3. #include <linux/kernel.h>
4. #include <asm/segment.h>

39

1. #define bswapw(x) \_\_asm\_\_("xchgb %%al,%%ah":"=a" (x):"" ((short)x)) // exchange 2 bytes.
2. #define ST(x) (\*\_\_st((x))) // get simulated ST(x) value.
3. #define PST(x) ((const temp\_real \*) \_\_st((x))) // get pointr to the simulated ST(x).

43

1. /\*
2. \* We don't want these inlined - it gets too messy in the machine-code.
3. \*/
4. // 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。
5. static void fpop(void);
6. static void fpush(void);
7. static void fxchg(temp\_real\_unaligned \* a, temp\_real\_unaligned \* b);
8. 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、コードを解析し、その意味に応じた処理を行います。異なるコードタイプに対して、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ます。

1. // 引数は、info構造体へのポインタです。
2. static void do\_emu(struct info \* info)
3. {
4. unsigned short code;
5. temp\_real tmp;
6. 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。

1. // カーネル内に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。そして、浮動小数点命令のCS、EIP、「Needs Mathematical Simulation in the Kernel」の情報を表示した後、 // カーネルはシャットダウンします。
2. if (I387.cwd & I387.swd & 0x3f)
3. I387.swd |= 0x8000; // sets the busy flag B.
4. else
5. I387.swd &= 0x7fff; // clear the flag B.
6. ORIG\_EIP = EIP; // save the original EIP.
7. /\* 0x0007 means user code space \*/
8. if (CS != 0x000F) { // if not in user code space, STOP.
9. printk("math\_emulate: %04x:%08x\n\r",CS,EIP);
10. panic("Math emulation needed in kernel");
11. }

// 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はデータを "Little endien "形式で格納しますが、この時にフェッチされるコードは、命令の1バイト目と2バイト目が // 正確に逆になっています。そこで、「コード」の2バイトの順序を // 入れ替える必要があります。そして、コード1バイト目（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

1. // 模擬プロセッサの例外が発生したときに、本物のコプロセッサのようにプログラムを処理できるように // 保存されています。最後に、EIPに後続の浮動小数点演算命令 // またはオペランドを指定させます。
2. code = get\_fs\_word((unsigned short \*) EIP); // get 2 bytes floating-point code.
3. bswapw(code); // exchange bytes.
4. code &= 0x7ff; // Mask the ESC part in the code.
5. I387.fip = EIP; // save EIP, code selector and code.
6. \*(unsigned short \*) &I387.fcs = CS;
7. \*(1+(unsigned short \*) &I387.fcs) = code;
8. EIP += 2; // point to next instruction code.

// その後、コードを解析し、その意味に応じて処理します。異なるコードタイプの//値に対して、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（レジスタなどに依存しない）の場合は、以下の処理を行います。

1. // linux/math\_emu.hの52行目で定義されているマクロ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ます。実際の実装コードは、 // プログラムの 488 行目にあります。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。
2. switch (code) {
3. case 0x1d0: /\* fnop \*/ // like nop.
4. return;
5. case 0x1d1: case 0x1d2: case 0x1d3: // invalid code, send signal and exit.
6. case 0x1d4: case 0x1d5: case 0x1d6: case 0x1d7:
7. math\_abort(info,1<<(SIGILL-1));
8. case 0x1e0: // FCHS - Change the ST sign bit: ST = -ST.
9. ST(0).exponent ^= 0x8000;
10. return;
11. case 0x1e1: // FABS - get absolute value. ST = |ST|.
12. ST(0).exponent &= 0x7fff;
13. return;
14. case 0x1e2: case 0x1e3: // invalid code. send signal and exit.
15. math\_abort(info,1<<(SIGILL-1));
16. case 0x1e4: // FTST - Test TS and set Cn in status word.
17. ftst(PST(0));
18. return;
19. case 0x1e5: // FXAM - Check TS and modify Cn in status word.
20. printk("fxam not implemented\n\r");
21. math\_abort(info,1<<(SIGILL-1));
22. case 0x1e6: case 0x1e7: // invalid code. send signal and exit.
23. math\_abort(info,1<<(SIGILL-1));
24. case 0x1e8: // FLD1 - Load constant 1.0 to accumulator ST.
25. fpush();
26. ST(0) = CONST1;
27. return;
28. case 0x1e9: // FLDL2T - Load constant Log2(10) to ST.
29. fpush();
30. ST(0) = CONSTL2T;
31. return;
32. case 0x1ea: // FLDL2E - Load constant Log2(e) to ST.
33. fpush();
34. ST(0) = CONSTL2E;
35. return;
36. case 0x1eb: // FLDPI - Load constant Pi to ST.
37. fpush();
38. ST(0) = CONSTPI;
39. return;
40. case 0x1ec: // FLDLG2 - Load constant Log10(2) to ST.
41. fpush();
42. ST(0) = CONSTLG2;
43. return;
44. case 0x1ed: // FLDLN2 - Load constant Loge(2) to ST.
45. fpush();
46. ST(0) = CONSTLN2;
47. return;
48. case 0x1ee: // FLDZ - Load constant 0.0 to ST.
49. fpush();
50. ST(0) = CONSTZ;
51. return;
52. case 0x1ef: // invalid code. send signal and exit.
53. math\_abort(info,1<<(SIGILL-1));
54. case 0x1f0: case 0x1f1: case 0x1f2: case 0x1f3:
55. case 0x1f4: case 0x1f5: case 0x1f6: case 0x1f7:
56. case 0x1f8: case 0x1f9: case 0x1fa: case 0x1fb:
57. case 0x1fc: case 0x1fd: case 0x1fe: case 0x1ff:
58. printk("%04x fxxx not implemented\n\r",code + 0xc800);
59. math\_abort(info,1<<(SIGILL-1));
60. case 0x2e9: // FUCOMPP - no order comparison.
61. fucom(PST(1),PST(0));
62. fpop(); fpop();
63. return;
64. case 0x3d0: case 0x3d1: // FNOP - on 387. !! should be 0x3e0, 0x3e1.
65. return;
66. case 0x3e2: // FCLEX - Clears exception flag in status word.
67. I387.swd &= 0x7f00;
68. return;
69. case 0x3e3: // FINIT - Initializes the coprocessor.
70. I387.cwd = 0x037f;
71. I387.swd = 0x0000;
72. I387.twd = 0x0000;
73. return;
74. case 0x3e4: // FNOP - on 80387.
75. return;
76. case 0x6d9: // FCOMPP - compares ST(1) with ST, pops twice.
77. fcom(PST(1),PST(0));
78. fpop(); fpop();
79. return;
80. case 0x7e0: // FSTSW AX - Saves status word to AX register.
81. \*(short \*) &EAX = I387.swd;
82. return;
83. }
84. // (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あり、プレフィックスの "11011 "はすでにクリアされている。 // ファイルlinux/math\_emu.hの72行目で定義されているマクロreal\_to\_real(a, b)は、 // 2つの一時的なリアル間の代入に使用されます。
85. switch (code >> 3) {
86. case 0x18: // FADD ST, ST(i)
87. fadd(PST(0),PST(code & 7),&tmp);
88. real\_to\_real(&tmp,&ST(0));
89. return;
90. case 0x19: // FMUL ST, ST(i)
91. fmul(PST(0),PST(code & 7),&tmp);
92. real\_to\_real(&tmp,&ST(0));
93. return;
94. case 0x1a: // FCOM ST(i)
95. fcom(PST(code & 7),&tmp);
96. real\_to\_real(&tmp,&ST(0));
97. return;
98. case 0x1b: // FCOMP ST(i)
99. fcom(PST(code & 7),&tmp);
100. real\_to\_real(&tmp,&ST(0));
101. fpop();
102. return;
103. case 0x1c: // FSUB ST, ST(i)
104. real\_to\_real(&ST(code & 7),&tmp);
105. tmp.exponent ^= 0x8000;
106. fadd(PST(0),&tmp,&tmp);
107. real\_to\_real(&tmp,&ST(0));
108. return;
109. case 0x1d: // FSUBR ST, ST(i)
110. ST(0).exponent ^= 0x8000;
111. fadd(PST(0),PST(code & 7),&tmp);
112. real\_to\_real(&tmp,&ST(0));
113. return;
114. case 0x1e: // FDIV ST, ST(i)
115. fdiv(PST(0),PST(code & 7),&tmp);
116. real\_to\_real(&tmp,&ST(0));
117. return;
118. case 0x1f: // FDIVR ST, ST(i)
119. fdiv(PST(code & 7),PST(0),&tmp);
120. real\_to\_real(&tmp,&ST(0));
121. return;
122. case 0x38: // FLD ST(i)
123. fpush();
124. ST(0) = ST((code & 7)+1);
125. return;
126. case 0x39: // FXCH ST(i) 199 fxchg(&ST(0),&ST(code & 7));
127. return;
128. case 0x3b: // FSTP ST(i)
129. ST(code & 7) = ST(0);
130. fpop();
131. return;
132. case 0x98: // FADD ST(i), ST
133. fadd(PST(0),PST(code & 7),&tmp);
134. real\_to\_real(&tmp,&ST(code & 7));
135. return;
136. case 0x99: // FMUL ST(i), ST
137. fmul(PST(0),PST(code & 7),&tmp);
138. real\_to\_real(&tmp,&ST(code & 7));
139. return;
140. case 0x9a: // FCOM ST(i)
141. fcom(PST(code & 7),PST(0));
142. return;
143. case 0x9b: // FCOMP ST(i)
144. fcom(PST(code & 7),PST(0));
145. fpop();
146. return;
147. case 0x9c: // FSUBR ST(i), ST
148. ST(code & 7).exponent ^= 0x8000;
149. fadd(PST(0),PST(code & 7),&tmp);
150. real\_to\_real(&tmp,&ST(code & 7));
151. return;
152. case 0x9d: // FSUB ST(i), ST
153. real\_to\_real(&ST(0),&tmp);
154. tmp.exponent ^= 0x8000;
155. fadd(PST(code & 7),&tmp,&tmp);
156. real\_to\_real(&tmp,&ST(code & 7));
157. return;
158. case 0x9e: // FDIVR ST(i), ST
159. fdiv(PST(0),PST(code & 7),&tmp);
160. real\_to\_real(&tmp,&ST(code & 7));
161. return;
162. case 0x9f: // FDIV ST(i), ST
163. fdiv(PST(code & 7),PST(0),&tmp);
164. real\_to\_real(&tmp,&ST(code & 7));
165. return;
166. case 0xb8: // FFREE ST(i)
167. printk("ffree not implemented\n\r");
168. math\_abort(info,1<<(SIGILL-1));
169. case 0xb9: // FXCH ST(i) 243 fxchg(&ST(0),&ST(code & 7));
170. return;
171. case 0xba: // FST ST(i)
172. ST(code & 7) = ST(0);
173. return;
174. case 0xbb: // FSTP ST(i)
175. ST(code & 7) = ST(0);
176. fpop();
177. return;
178. case 0xbc: // FUCOM ST(i)
179. fucom(PST(code & 7),PST(0));
180. return;
181. case 0xbd: // FUCOMP ST(i)
182. fucom(PST(code & 7),PST(0));
183. fpop();
184. return;
185. case 0xd8: // FADDP ST(i), ST
186. fadd(PST(code & 7),PST(0),&tmp);
187. real\_to\_real(&tmp,&ST(code & 7));
188. fpop();
189. return;
190. case 0xd9: // FMULP ST(i), ST
191. fmul(PST(code & 7),PST(0),&tmp);
192. real\_to\_real(&tmp,&ST(code & 7));
193. fpop();
194. return;
195. case 0xda: // FCOMP ST(i)
196. fcom(PST(code & 7),PST(0));
197. fpop();
198. return;
199. case 0xdc: // FSUBRP ST(i), ST
200. ST(code & 7).exponent ^= 0x8000;
201. fadd(PST(0),PST(code & 7),&tmp);
202. real\_to\_real(&tmp,&ST(code & 7));
203. fpop();
204. return;
205. case 0xdd: // FSUBP ST(i), ST
206. real\_to\_real(&ST(0),&tmp);
207. tmp.exponent ^= 0x8000;
208. fadd(PST(code & 7),&tmp,&tmp);
209. real\_to\_real(&tmp,&ST(code & 7));
210. fpop();
211. return;
212. case 0xde: // FDIVRP ST(i), ST
213. fdiv(PST(0),PST(code & 7),&tmp);
214. real\_to\_real(&tmp,&ST(code & 7));
215. fpop();
216. return;
217. case 0xdf: // FDIVP ST(i), ST
218. fdiv(PST(code & 7),PST(0),&tmp);
219. real\_to\_real(&tmp,&ST(code & 7));
220. fpop();
221. return;
222. case 0xf8: // FFREE ST(i)
223. printk("ffree not implemented\n\r");
224. math\_abort(info,1<<(SIGILL-1));
225. fpop();
226. return;
227. case 0xf9: // FXCH ST(i) 302 fxchg(&ST(0),&ST(code & 7));
228. return;
229. case 0xfa: // FSTP ST(i)
230. case 0xfb: // FSTP ST(i)
231. ST(code & 7) = ST(0);
232. fpop();
233. return;
234. }
235. // (3) 次に、2バイト目の7--6がMOD、ビット2--0がR/Mという形でコードを処理します。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るので、 // 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。
236. switch ((code>>3) & 0xe7) {
237. case 0x22: // FST - Saves single precision real (short real)
238. put\_short\_real(PST(0),info,code);
239. return;
240. case 0x23: // FSTP - Saves single precision real (short real)
241. put\_short\_real(PST(0),info,code);
242. fpop();
243. return;
244. case 0x24: // FLDENV - Load status and control registers, etc.
245. address = ea(info,code); // get efficient address.
246. for (code = 0 ; code < 7 ; code++) {
247. ((long \*) & I387)[code] =
248. get\_fs\_long((unsigned long \*) address);
249. address += 4;
250. }
251. return;
252. case 0x25: // FLDCW - Load control word.
253. address = ea(info,code);
254. \*(unsigned short \*) &I387.cwd =
255. get\_fs\_word((unsigned short \*) address);
256. return;
257. case 0x26: // FSTENV - Store status and control registers, etc.
258. address = ea(info,code);
259. verify\_area(address,28);
260. for (code = 0 ; code < 7 ; code++) {
261. put\_fs\_long( ((long \*) & I387)[code],
262. (unsigned long \*) address);
263. address += 4;
264. }
265. return;
266. case 0x27: // FSTCW - Store control word.
267. address = ea(info,code);
268. verify\_area(address,2);
269. put\_fs\_word(I387.cwd,(short \*) address);
270. return;
271. case 0x62: // FIST - Stores short integer.
272. put\_long\_int(PST(0),info,code);
273. return;
274. case 0x63: // FISTP - Stores short integer.
275. put\_long\_int(PST(0),info,code);
276. fpop();
277. return;
278. case 0x65: // FLD - Load an extended (temporary) real number.
279. fpush();
280. get\_temp\_real(&tmp,info,code);
281. real\_to\_real(&tmp,&ST(0));
282. return;
283. case 0x67: // FSTP - Store the temporary real.
284. put\_temp\_real(PST(0),info,code);
285. fpop();
286. return;
287. case 0xa2: // FST - Stores double precision (long) real.
288. put\_long\_real(PST(0),info,code);
289. return;
290. case 0xa3: // FSTP - Stores double precision (long) real.
291. put\_long\_real(PST(0),info,code);
292. fpop();
293. return;
294. case 0xa4: // FRSTOR - Restores all 108 bytes register contents.
295. address = ea(info,code);
296. for (code = 0 ; code < 27 ; code++) {
297. ((long \*) & I387)[code] =
298. get\_fs\_long((unsigned long \*) address);
299. address += 4;
300. }
301. return;
302. case 0xa6: // FSAVE - Saves all 108 bytes register contents.
303. address = ea(info,code);
304. verify\_area(address,108);
305. for (code = 0 ; code < 27 ; code++) {
306. put\_fs\_long( ((long \*) & I387)[code],
307. (unsigned long \*) address);
308. address += 4;
309. }
310. I387.cwd = 0x037f;
311. I387.swd = 0x0000;
312. I387.twd = 0x0000;
313. return;
314. case 0xa7: // FSTSW - Store status word.
315. address = ea(info,code);
316. verify\_area(address,2);
317. put\_fs\_word(I387.swd,(short \*) address);
318. return;
319. case 0xe2: // FIST - Stores short integer.
320. put\_short\_int(PST(0),info,code);
321. return;
322. case 0xe3: // FISTP - Stores short integer.
323. put\_short\_int(PST(0),info,code);
324. fpop();
325. return;
326. case 0xe4: // FBLD - Loads the number of BCD type.
327. fpush();
328. get\_BCD(&tmp,info,code);
329. real\_to\_real(&tmp,&ST(0));
330. return;
331. case 0xe5: // FILD - Loads a long integer.
332. fpush();
333. get\_longlong\_int(&tmp,info,code);
334. real\_to\_real(&tmp,&ST(0));
335. return;
336. case 0xe6: // FBSTP - Stores number of BCD type.
337. put\_BCD(PST(0),info,code);
338. fpop();
339. return;
340. case 0xe7: // BISTP - Stores a long integer. 415 put\_longlong\_int(PST(0),info,code);
341. fpop();
342. return;
343. }

// (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

1. // 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 // を取り、OPAとOPBの合成値に応じて個別に処理します。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ことになります。
2. switch (code >> 9) {
3. case 0: // MF = 00, short real (32-bit real).
4. get\_short\_real(&tmp,info,code);
5. break;
6. case 1: // MF = 01, short integer (32-bit integer).
7. get\_long\_int(&tmp,info,code);
8. break;
9. case 2: // MF = 10, long real (64-bit real).
10. get\_long\_real(&tmp,info,code);
11. break;
12. case 4: // MF = 11, a long integer (64-bit). should be 'case 3'!
13. get\_short\_int(&tmp,info,code);
14. }
15. // (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。
16. switch ((code>>3) & 0x27) {
17. case 0: // FADD
18. fadd(&tmp,PST(0),&tmp);
19. real\_to\_real(&tmp,&ST(0));
20. return;
21. case 1: // FMUL
22. fmul(&tmp,PST(0),&tmp);
23. real\_to\_real(&tmp,&ST(0));
24. return;
25. case 2: // FCOM
26. fcom(&tmp,PST(0));
27. return;
28. case 3: // FCOMP
29. fcom(&tmp,PST(0));
30. fpop();
31. return;
32. case 4: // FSUB
33. tmp.exponent ^= 0x8000;
34. fadd(&tmp,PST(0),&tmp);
35. real\_to\_real(&tmp,&ST(0));
36. return;
37. case 5: // FSUBR
38. ST(0).exponent ^= 0x8000;
39. fadd(&tmp,PST(0),&tmp);
40. real\_to\_real(&tmp,&ST(0));
41. return;
42. case 6: // FDIV
43. fdiv(PST(0),&tmp,&tmp);
44. real\_to\_real(&tmp,&ST(0));
45. return;
46. case 7: // FDIVR
47. fdiv(&tmp,PST(0),&tmp);
48. real\_to\_real(&tmp,&ST(0));
49. return;
50. }
51. // 「11011, XX, 1, XX, 000, R/M」の形式の命令コードを処理します。
52. if ((code & 0x138) == 0x100) { // FLD, FILD
53. fpush();
54. real\_to\_real(&tmp,&ST(0));
55. return;
56. }
57. // 残りは無効な命令です。
58. printk("Unknown math-insns: %04x:%08x %04x\n\r",CS,EIP,code);
59. math\_abort(info,1<<(SIGFPE-1));
60. }

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

1. // used\_mathを使用し、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、浮動小数点命令エミュレーション関数do\_emu()を呼び出すためのパラメータとして、int 7がこの関数を呼び出したときのリターンアドレスを // 使用します。パラメータ \_\_\_false は \_orig\_eip です。
2. void math\_emulate(long \_\_\_false)
3. {
4. if (!current->used\_math) {
5. current->used\_math = 1;
6. I387.cwd = 0x037f;
7. I387.swd = 0x0000;
8. I387.twd = 0x0000;
9. }
10. /\* &\_\_\_false points to info->\_\_\_orig\_eip, so subtract 1 to get info \*/
11. do\_emu((struct info \*) ((&\_\_\_false) - 1));
12. }

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

1. // 処理を行います。最後に，int7 がこの関数を呼び出すと， // スタックポインタはリターンアドレスに向けられ， // 直接，割込みに戻ります。この関数は、マクロ // math\_abort() で使用されます。詳しい説明は、linux/math\_emu.hファイルの50行目を参照してください。
2. void \_\_math\_abort(struct info \* info, unsigned int signal)
3. {
4. EIP = ORIG\_EIP;
5. current->signal |= signal;
6. \_\_asm\_\_("movl %0,%%esp ; ret"::"g" ((long) info));
7. }

494

// アキュムレータ・スタック・ポップアップ・オペレーション

1. // ステータスワードのTOPフィールドを1増やし、モジュロ7とする。
2. static void fpop(void)
3. {
4. unsigned long tmp;

498

1. tmp = I387.swd & 0xffffc7ff;
2. I387.swd += 0x00000800;
3. I387.swd &= 0x00003800;
4. 11.3 error.c
5. I387.swd |= tmp;
6. }

504

// アキュムレータ・スタック プッシュ操作。

1. // 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。
2. static void fpush(void)
3. {
4. unsigned long tmp;

508

1. tmp = I387.swd & 0xffffc7ff;
2. I387.swd += 0x00003800;
3. I387.swd &= 0x00003800;
4. I387.swd |= tmp;
5. }

514

1. // 2つのアキュムレータ・レジスタの値を入れ替えます。
2. static void fxchg(temp\_real\_unaligned \* a, temp\_real\_unaligned \* b)
3. {
4. temp\_real\_unaligned c;

518

1. c = \*a;
2. \*a = \*b;
3. \*b = c;
4. }

523

// I387構造体のST(i)メモリポインタを取得する。

1. // 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、 // モジュロ7をかけて、最後にST(i)に対応するポインタを返す。
2. static temp\_real\_unaligned \* \_\_st(int i)
3. {
4. i += I387.swd >> 11; // Get the TOP field in the status word.
5. i &= 7;
6. return (temp\_real\_unaligned \*) (i\*10 + (char \*)(I387.st\_space)); 529 }

530

## 11.3 error.c

### 11.3.1 Function

### コプロセッサがエラーを検出すると、80387チップのERROR端子を介してCPUに通知します。error.cプログラムは、コプロセッサから送られてきたエラー信号を処理するためのもので、主にmath\_error()関数を実行します。

### 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

// シグナル操作の関数のプロトタイプ。

// <linux/sched.h> スケジューラーのヘッダーファイルは、タスク構造体task\_struct、初期タスク0のデータ、およびいくつかの組み込みアセンブリ関数マクロステートメントを定義します。

// ディスクリプタのパラメータ設定と取得について。

7 #include <signal.h> (英語)

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する。最後のタスクがコプロセッサを使用していた場合は、 // コプロセッサのエラー信号フラグを設定します。リターン後、この関数はret\_from\_sys\_callのシステムコール割込みのリターンプレースに // ジャンプして実行を継続します。

1. 11 void math\_error(void) 12 {。
2. \_\_asm\_\_("fnclex"); // clear all exception flags and busy bit in status word.
3. if (last\_task\_used\_math) // if used coprocessor, set signal flag.
4. last\_task\_used\_math->signal |= 1<<(SIGFPE-1);
5. }

17

## 11.4 ea.c

### 11.4.1 Function

ea.cプログラムは、浮動小数点演算命令をシミュレートする際に、オペランドが使用する実効アドレスを計算するために使用します。命令の実効アドレス情報を解析するためには、命令のエンコード方式を理解しておく必要があります。インテル・プロセッサ命令の一般的な符号化形式を図11-7に示します。

Prefix Byt

e

(0

--

4)

Op

c

ode

(1

--

2)

d

MODRM Operan

(0

--

2)

Addr Offset

(0

--

4)

I

mmediate constant

(0

--

4)

Low addr

High addr

図11-7 一般命令の符号化形式

図からわかるように、各命令は最大5つのフィールドを持つことができます。プレフィックスフィールドは、0から

4バイトで、次の命令の修正に使用されます。opcodeフィールドは、命令の動作を示すメインフィールドです。各命令には少なくとも1バイトのオペレーションコードが必要です。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。

|  |  |  |
| --- | --- | --- |
|  | | |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | O  P  Code(s)    O  P  Code(s)    O  P  Code(s) | |  |  |  | | --- | --- | --- | | 00 | OPC | 100 | |  |  |  | | 01 | REG | 100 | |  |  |  | | 10 | REG | 100 | | |  |  |  | | --- | --- | --- | | S | Idx | 101 | |  |  |  | | S | Idx | Base | |  |  |  | | S | Idx | Base | | 32  -  bit Offset    32  -  bit Offset    8  -  bit Offset | | |  |  | | --- | --- | | スケールフィールド  S | Size | | 00 | 1 | | 01 | 2 | | 10 | 4 | | 11 | 8 | |  |

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

### 本プログラムのea()関数は、命令中のアドレッシング・モード・バイトに基づいて、実効アドレスを計算するために使用されます。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ります。MOD=0b11であれば、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でない場合は、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。

### 11.4.2 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

1. /\*
2. \* Calculate the effective address.
3. \*/ 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用ヘッダファイルです。コプロセッサのデータ構造と

// 浮動小数点表現構造が定義されています。

// <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

// セグメント・レジスタ・オペレーションのための

11 #include <stddef.h>.

12

1. #include <linux/math\_emu.h>
2. #include <asm/segment.h> 15
3. // info構造体の各レジスタのオフセット位置です。 offsetof()は、構造体の指定されたフィールドの // オフセット位置を見つけるために使用されます。include/stddef.hファイルを参照してください。
4. static int \_\_regoffset[] = {
5. offsetof(struct info,\_\_\_eax),
6. offsetof(struct info,\_\_\_ecx),
7. offsetof(struct info,\_\_\_edx),
8. offsetof(struct info,\_\_\_ebx),
9. offsetof(struct info,\_\_\_esp),
10. offsetof(struct info,\_\_\_ebp),
11. offsetof(struct info,\_\_\_esi),
12. offsetof(struct info,\_\_\_edi)
13. };

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

1. // 第2オペランド指示バイトSIB(Scale, Index, Base)の値を取得します。
2. static char \* sib(struct info \* info, int mod)
3. {
4. unsigned char ss,index,base;
5. long offset = 0; 33
6. // まずユーザーコードセグメントからSIBバイトを取得し、次にフィールドビット値を取得します。 34 base = get\_fs\_byte((char \*) EIP);
7. EIP++;
8. ss = base >> 6; // scale size.
9. index = (base >> 3) & 7;
10. base &= 7;
11. // 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 // 値 offset = register content \* scale factor.
12. if (index == 4)
13. offset = 0;
14. else
15. offset = REG(index);
16. 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

1. base で指定されたレジスタにオフセット値があることを // 示します。したがって，オフセットはベースに対応するレジスタの内容に // 追加される必要がある。MOD=1の場合，オフセット値は // 1バイト(8ビット)となる。そうでない場合、MOD=2、またはbase=0b101であれば、オフセット値は4バイトになります。
2. if (mod || base != 5)
3. offset += REG(base);
4. if (mod == 1) {
5. offset += (signed char) get\_fs\_byte((char \*) EIP);
6. EIP++;
7. } else if (mod == 2 || base == 5) {
8. offset += (signed) get\_fs\_long((unsigned long \*) EIP);
9. EIP += 4;
10. }
11. // 最後にオフセット値を保存して返します。
12. I387.foo = offset;
13. I387.fos = 0x17;
14. return (char \*) offset;
15. } 57
16. // 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。
17. char \* ea(struct info \* info, unsigned short code)
18. {
19. unsigned char mod,rm;
20. long \* tmp = &EAX;
21. 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

1. // 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でない場合は、2バイトアドレスモードのアドレッシングを意味するので、sib()を呼んでオフセット値を調べて // 返します。
2. mod = (code >> 6) & 3; // MOD field.
3. rm = code & 7; // R/M field.
4. if (rm == 4 && mod != 3)
5. return sib(info,mod);
6. // R/Mフィールドが0b101でMODが0の場合は、1バイトのアドレスモードエンコーディングに // 32バイトのオフセット値が続いていることを示します。そして、ユーザーコードの4バイトのオフセット値を取り、それを保存して // 返す。
7. if (rm == 5 && !mod) {
8. offset = get\_fs\_long((unsigned long \*) EIP);
9. EIP += 4;
10. I387.foo = offset;
11. I387.fos = 0x17;
12. return (char \*) offset;
13. }

// 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

1. R/Mコードの対応するレジスタの//がポインタtmpとして取り出される。MOD=0の場合，オフセット値はない。MOD=1の場合，コードの後に1バイトのオフセット値がある。MOD=2の場合， // コードの後に4バイトのオフセット値がある。最後に，有効なアドレス値を保存して返します。
2. tmp = & REG(rm);
3. switch (mod) {
4. case 0: offset = 0; break;
5. case 1:
6. offset = (signed char) get\_fs\_byte((char \*) EIP);
7. EIP++;
8. break;
9. case 2:
10. offset = (signed) get\_fs\_long((unsigned long \*) EIP);
11. EIP += 4; 85 break;
12. case 3:
13. math\_abort(info,1<<(SIGILL-1));
14. }
15. I387.foo = offset;
16. I387.fos = 0x17;
17. return offset + (char \*) \*tmp;
18. }

93

## 11.5 convert.c

### 11.5.1 Function

convert.cプログラムには、80387エミュレーション動作中のデータ型変換機能が含まれています。シミュレーション計算を行う前に、ユーザーから提供された整数型や実数型を、シミュレーションで使用する一時的な実数形式に変換し、シミュレーションが終了した後に元の形式に戻す必要があります。例えば、図11-9は、短い実数を一時的な実数形式に変換する模式図です。

(

)

Short Real

79

0

63 62

64

S

Exp(8)

S

E

xp(15)

1

Fraction

(23)

After normalized, it will explicitly contain 1

.

Hide

1

23

0

31

)

(

Temporary Real

Fraction(64)

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

1. /\*
2. \* 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
3. \* know what's going on. They are the adding of one in the rounding: the
4. \* overflow bit is also used for adding one into the exponent. Thus it
5. \* looks like the overflow would be incorrectly handled, but due to the 15 \* way the IEEE numbers work, things are correct.
6. \*
7. \* 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。

1. // 短い実数は32ビット、その有効数字（仮数）は23ビット、 // 指数は8ビット、そして1つの符号ビットがあります。
2. void short\_to\_temp(const short\_real \* a, temp\_real \* b)
3. {
4. // 0であれば，一時的な実数bの符号ビットは0に設定され，一時的な実数の符号ビットは， // 短い実数の符号ビット，すなわち指数の最上位ビットに従って設定されます．
5. if (!(\*a & 0x7fffffff)) {
6. b->a = b->b = 0; // set significand of the temp real = 0.
7. if (\*a)
8. b->exponent = 0x8000; // set sign bit.
9. else
10. b->exponent = 0;
11. 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

1. 仮の実数のビット62に仮の実数の//のビットがあること。一時的な実数の仮数のビット63は1に設定する必要があり、これにはOR 0x80000000の演算が必要です。最後に、一時的な実数の下位32ビットの有効数字がクリアされます。
2. b->exponent = ((\*a>>23) & 0xff)-127+16383; // change biased value to 16383.
3. if (\*a<0)
4. b->exponent |= 0x8000; // add negative sign if need.
5. b->b = (\*a<<8) | 0x80000000; // put mantissa, set 63th bit.
6. b->a = 0;
7. }

37

// long realをtemporary realに変換します。

1. // このメソッドは、short\_to\_temp()と全く同じです。ただし、長実数は64ビットで、 // その有効数字（仮数）は52ビット、指数は11ビット、そして // 1つの符号ビットがあります。また、long real の指数の偏り値は 1023 です。
2. void long\_to\_temp(const long\_real \* a, temp\_real \* b)
3. {
4. if (!a->a && !(a->b & 0x7fffffff)) {
5. b->a = b->b = 0;
6. if (a->b)
7. b->exponent = 0x8000; // set sign bit.
8. else
9. b->exponent = 0;
10. return;
11. }
12. b->exponent = ((a->b >> 20) & 0x7ff)-1023+16383; // change biased value to 16383.
13. if (a->b<0)
14. b->exponent |= 0x8000;
15. b->b = 0x80000000 | (a->b<<11) | (((unsigned long)a->a)>>21); // place 1.
16. b->a = a->a<<11;
17. }

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。

1. // 丸めの問題。
2. void temp\_to\_short(const temp\_real \* a, short\_real \* b)
3. {
4. // 指数部が0の場合，符号ビットの有無に応じて，短実数は-0または0になる．
5. if (!(a->exponent & 0x7fff)) {
6. \*b = (a->exponent)?0x80000000:0;
7. return; 60 }
8. // まず、指数部分の処理、つまり指数の偏りの量（16383）を // 短い実数の偏りの量127に置き換え、 // 負の数であれば符号ビットを設定します。
9. \*b = ((((long) a->exponent)-16383+127) << 23) & 0x7f800000;
10. if (a->exponent < 0) // set sign if negative.
11. \*b |= 0x80000000;
12. // その後、一時的な実数からシニフカントの最上位23ビットを取得し、 // コントロールワードの丸め設定に従って丸め演算を行います。
13. \*b |= (a->b >> 8) & 0x007fffff; // get higher 23bits of the temp real.
14. switch (ROUNDING) {
15. case ROUND\_NEAREST:
16. if ((a->b & 0xff) > 0x80)
17. ++\*b;
18. break;
19. case ROUND\_DOWN:
20. if ((a->exponent & 0x8000) && (a->b & 0xff))
21. ++\*b;
22. break; 74 case ROUND\_UP:
23. if (!(a->exponent & 0x8000) && (a->b & 0xff))
24. ++\*b;
25. break;
26. }
27. }

80

// 一時的な実数を長い実数に変換します。

1. // 長い実数は64ビットで、その有効数字（仮数）は52ビット、指数は11ビットで、符号ビットが1つあり、指数の偏り値は1023となります。
2. void temp\_to\_long(const temp\_real \* a, long\_real \* b)
3. {
4. if (!(a->exponent & 0x7fff)) {
5. b->a = 0;
6. b->b = (a->exponent)?0x80000000:0;
7. return;
8. }
9. b->b = (((0x7fff & (long) a->exponent)-16383+1023) << 20) & 0x7ff00000;
10. if (a->exponent < 0)
11. b->b |= 0x80000000;
12. b->b |= (a->b >> 11) & 0x000fffff; 92 b->a = a->b << 21;
13. b->a |= (a->a >> 11) & 0x001fffff;
14. switch (ROUNDING) {
15. case ROUND\_NEAREST:
16. if ((a->a & 0x7ff) > 0x400)
17. \_\_asm\_\_("addl $1,%0 ; adcl $0,%1"
18. :"=r" (b->a),"=r" (b->b)
19. :"" (b->a),"1" (b->b));
20. break; 101 case ROUND\_DOWN:
21. if ((a->exponent & 0x8000) && (a->b & 0xff))
22. \_\_asm\_\_("addl $1,%0 ; adcl $0,%1"
23. :"=r" (b->a),"=r" (b->b)
24. :"" (b->a),"1" (b->b));
25. break; 107 case ROUND\_UP:
26. if (!(a->exponent & 0x8000) && (a->b & 0xff))
27. \_\_asm\_\_("addl $1,%0 ; adcl $0,%1"
28. :"=r" (b->a),"=r" (b->b)
29. :"" (b->a),"1" (b->b));
30. break;
31. }
32. }

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

1. // 上位2バイトのうち、ビットが1の場合は負の数を示し、ビットが0の場合は正の数を示します。
2. void real\_to\_int(const temp\_real \* a, temp\_int \* b)
3. {
4. int shift = 16383 + 63 - (a->exponent & 0x7fff);
5. unsigned long underflow;

120

1. b->a = b->b = underflow = 0;
2. b->sign = (a->exponent < 0);
3. if (shift < 0) {
4. set\_OE();
5. return;
6. }
7. if (shift < 32) {
8. b->b = a->b; b->a = a->a;
9. } else if (shift < 64) {
10. b->a = a->b; underflow = a->a;
11. shift -= 32;
12. } else if (shift < 96) {
13. underflow = a->b;
14. shift -= 64;
15. } else
16. return;
17. \_\_asm\_\_("shrdl %2,%1,%0" // 32-bit shift right.
18. :"=r" (underflow),"=r" (b->a)
19. :"c" ((char) shift),"" (underflow),"1" (b->a));
20. \_\_asm\_\_("shrdl %2,%1,%0"
21. :"=r" (b->a),"=r" (b->b)
22. :"c" ((char) shift),"" (b->a),"1" (b->b));
23. \_\_asm\_\_("shrl %1,%0"
24. :"=r" (b->b)
25. :"c" ((char) shift),"" (b->b));
26. switch (ROUNDING) {
27. case ROUND\_NEAREST:
28. \_\_asm\_\_("addl %4,%5 ; adcl $0,%0 ; adcl $0,%1"
29. :"=r" (b->a),"=r" (b->b)
30. :"" (b->a),"1" (b->b)
31. ,"r" (0x7fffffff + (b->a & 1))
32. ,"m" (\*&underflow));
33. break; 154 case ROUND\_UP:
34. if (!b->sign && underflow)
35. \_\_asm\_\_("addl $1,%0 ; adcl $0,%1"
36. :"=r" (b->a),"=r" (b->b)
37. :"" (b->a),"1" (b->b));
38. break; 160 case ROUND\_DOWN:
39. if (b->sign && underflow)
40. \_\_asm\_\_("addl $1,%0 ; adcl $0,%1"
41. :"=r" (b->a),"=r" (b->b)
42. :"" (b->a),"1" (b->b));
43. break;
44. }
45. }

168

1. // 一時的な整数を一時的な実数形式に変換します。
2. void int\_to\_real(const temp\_int \* a, temp\_real \* b)
3. {
4. // 元の値が整数なので、仮の実数に変換する際に、指数は // 偏った量の16383に加えて63を加えます。これは、シニフィカントも整数値であるため、 // 2^63 を掛ける必要があることを意味しています。
5. b->a = a->a;
6. b->b = a->b;
7. if (b->a || b->b)
8. b->exponent = 16383 + 63 + (a->sign? 0x8000:0);
9. else {
10. b->exponent = 0;
11. return; 178 }
12. // 変換形式が正規化された後の正常な実数，つまり，符号の上位ビットが0ではない // 。
13. while (b->b >= 0) {
14. b->exponent--;
15. \_\_asm\_\_("addl %0,%0 ; adcl %1,%1"
16. :"=r" (b->a),"=r" (b->b)
17. :"" (b->a),"1" (b->b));
18. }
19. }

186

## 11.6 add.c

### 11.6.1 Function

add.c プログラムは、シミュレーション中の加算処理に使用されます。浮動小数点数の仮数を計算するためには、まず仮数を記号化し、計算後に非記号化を行い、仮実数形式で浮動小数点数を表現することを再開する必要があります。浮動小数点数の仮数の記号化と非記号化のフォーマット変換の模式図を図11-10に示します。

signify()

long b

long a

Short Exponent

Exponent

Fraction

mantissa

)

(

S

Exponent

Fraction

0

S

Temp Real

Simu

l.

Format

unsignify()

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

1. /\*
2. \* temporary real addition routine.
3. \*
4. \* NOTE! These aren't exact: they are only 62 bits wide, and don't do
5. \* correct rounding. Fast hack. The reason is that we shift right the
6. \* values by two, in order not to have overflow (1 bit), and to be able
7. \* to move the sign into the mantissa (1 bit). Much simpler algorithms,
8. \* and 62 bits (61 really - no rounding) accuracy is usually enough. The
9. \* 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。

1. // 演算内容は，一時的な実数の仮数（significand）を反転させ， // 1を加えるというものです。パラメータ 'a' は，一時的な実数構造体へのポインタです．そのフィールド a と b の組み合わせが，一時的な実数のシニフィカントです．
2. #define NEGINT(a) \
3. \_\_asm\_\_("notl %0 ; notl %1 ; addl $1,%0 ; adcl $0,%1" \
4. :"=r" (a->a),"=r" (a->b) \
5. :"" (a->a),"1" (a->b))

26

// 仮数を符号化しました。

// つまり、シミュレーション操作を容易にするために、一時的な実数を指数や整数の表現に変換します。なので、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

1. // 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、仮数を補数で表して // （負を取る）、指数を正の値にします。このとき、仮数部には // 2ビットシフトしたシニフィカントだけでなく、値の符号ビットも含まれています。
2. static void signify(temp\_real \* a)
3. {

// 30 行目: %0 は a->a; %1 は a->b です。アセンブリ命令 "shldl $2, %1, %0" は以下を実行します。

// 倍精度（64ビット）の右シフトで，結合された仮数 <b, a> を

1. // 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、 // 2 ビットだけ右にシフトする必要があります。
2. a->exponent += 2; // increases exponent by 2.
3. \_\_asm\_\_("shrdl $2,%1,%0 ; shrl $2,%1" // mantissa is shifted to right by 2 bits.
4. :"=r" (a->a),"=r" (a->b)
5. :"" (a->a),"1" (a->b));
6. if (a->exponent < 0)
7. NEGINT(a); // be negative.
8. a->exponent &= 0x7fff; // remove the sign bit (if any). 36 }

37

// 仮数を無符号化します。

1. // 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、 // 指数と整数で表される実数を一時的な実数形式に変換します。
2. static void unsignify(temp\_real \* a)
3. {

// 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。もしそうならば，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

1. // 指数値はそれに応じてデクリメントされます。つまり，仮数の最上位ビットが0にならないように // 左シフト操作を行います（最後のa->bの値が // 負の値のように見える）．
2. if (!(a->a || a->b)) { // ret if zero.
3. a->exponent = 0;
4. return;
5. }
6. a->exponent &= 0x7fff; // reset the sign bit.
7. if (a->b < 0) { // if negative, let mantissa be a positive.
8. NEGINT(a);
9. a->exponent |= 0x8000; // add a sign bit.
10. }
11. while (a->b >= 0) {
12. a->exponent--; // the mantissa is normalized.
13. \_\_asm\_\_("addl %0,%0 ; adcl %1,%1"
14. :"=r" (a->a),"=r" (a->b)
15. :"" (a->a),"1" (a->b));
16. }
17. }

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.

1. 57 void fadd(const temp\_real \* src1, const temp\_real \* src2, temp\_real \* result) 58 {...
2. temp\_real a,b;
3. int x1,x2,shift; 61
4. // まず、2つの数字の指数値x1, x2を（符号ビットを取り除いて）取り、 // 変数aを最大値に、変数'shift'を指数差に // 等しくします（つまり、差の2の倍数）。
5. x1 = src1->exponent & 0x7fff;
6. x2 = src2->exponent & 0x7fff;
7. if (x1 > x2) {
8. a = \*src1;
9. b = \*src2;
10. shift = x1-x2;
11. } else {
12. a = \*src2;
13. 11.7 compare.c
14. b = \*src1;
15. shift = x2-x1;
16. }

// 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に入れ、次にbの指数を32倍にします。つまり、指数の差を // 32倍にして引きます。この調整の後、追加された2つの数字のマンタは、次のように下がります。

1. // 実質的に同じ範囲である。
2. if (shift >= 64) {
3. \*result = a;
4. return;
5. }
6. if (shift >= 32) {
7. b.a = b.b;
8. b.b = 0;
9. shift -= 32;
10. }

// 次に、両者の指数が同じ値になるように微調整します。調整方法は、 // 小さな値bの仮数を'shift'ビットだけ右にシフトします。

1. // 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，この2つのマンタを // 加算することができます．加算する前に、シミュレーション形式に変換し、 // 加算演算後に一時的な実数形式に戻す必要があります。
2. \_\_asm\_\_("shrdl %4,%1,%0 ; shrl %4,%1" // Double precision (64 bit) right shift.
3. :"=r" (b.a),"=r" (b.b)
4. :"" (b.a),"1" (b.b),"c" ((char) shift));
5. signify(&a); // change format.
6. signify(&b);
7. \_\_asm\_\_("addl %4,%0 ; adcl %5,%1" // adding by using normal instructions.
8. :"=r" (a.a),"=r" (a.b)
9. :"" (a.a),"1" (a.b),"g" (b.a),"g" (b.b));
10. unsignify(&a); // change back to temp real format.
11. \*result = a;
12. }

93

## 11.7 compare.c

### 11.7.1 Function

### compare.cプログラムは，シミュレーション中にアキュムレータ内の2つの一時的な実数のサイズを比較するために使用されます．

### 11.7.2 Code annotation

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

1. /\*
2. \* linux/kernel/math/compare.c
3. 11.7 compare.c
4. \*
5. \* (C) 1991 Linus Torvalds
6. \*/

6

1. /\*
2. \* temporary real comparison routines
3. \*/

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。

1. // 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。
2. static void normalize(temp\_real \* a)
3. {
4. int i = a->exponent & 0x7fff; // get the exponent (ignore sign bit).
5. int sign = a->exponent & 0x8000; // get sign.

19

1. // 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0と等しい。
2. if (!(a->a || a->b)) {
3. a->exponent = 0;
4. return; 23 }
5. // aの仮数部の左端に0値のビットがある場合は、仮数部を左にシフトして // 指数値を調整（デクリメント）します。b // フィールドのMSB（最上位ビット）が1になるまで（bが負の値として現れる場合）。最後に符号ビットを加えます。
6. while (i && a->b >= 0) {
7. i--;
8. \_\_asm\_\_("addl %0,%0 ; adcl %1,%1"
9. :"=r" (a->a),"=r" (a->b)
10. :"" (a->a),"1" (a->b));
11. }
12. a->exponent = i | sign;
13. }

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

1. 比較結果に応じて， // ステータスワードが設定される。ST > 0.0の場合，C3，C2，C0はそれぞれ // 000となり，ST < 0.0の場合，条件ビットは001となり，ST == 0.0の場合，条件ビットは // 100となり，そうでない場合，条件ビットは111となります。
2. void ftst(const temp\_real \* a)
3. {
4. temp\_real b; 36

// まず、ステータスワードの条件フラグがクリアされ、比較値b（ST）が

1. // は正規化されます。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， // 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がセットされる．
2. clear\_Cx();
3. b = \*a;
4. normalize(&b);
5. if (b.a || b.b || b.exponent) {
6. if (b.exponent < 0)
7. set\_C0();
8. } else
9. set\_C3();
10. }

46

// 浮動小数点演算命令FCOM（Floating-point Compare）をシミュレートします。

1. // 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します。src1 > src2 の場合、C3, C2, C0 はそれぞれ 000 となります。src1 < src2 の場合、条件ビットは 001 となり、2つが等しい場合、条件ビットは 100 となります。
2. void fcom(const temp\_real \* src1, const temp\_real \* src2)
3. {
4. temp\_real a;

50

1. a = \*src1;
2. a.exponent ^= 0x8000; // invert the sign bit.
3. fadd(&a,src2,&a); // add the two (ie subtract).
4. 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。

1. // 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。
2. void fucom(const temp\_real \* src1, const temp\_real \* src2)
3. {
4. fcom(src1,src2);
5. }

61

## 11.8 get\_put.c

### 11.8.1 Function

### get\_put.cプログラムは、ユーザーメモリへのすべてのアクセス（フェッチおよびストア命令／実数値／BCD値）を処理します。他のデータ形式に関わるのはこの部分だけです。シミュレーションプロセスにおける他のすべての操作は、一時的な実数フォーマットを使用します。

### 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

1. /\*
2. \* This file handles all accesses to user memory: getting and putting
3. \* ints/reals/BCD etc. This is the only part that concerns itself with
4. \* other than temporary real format. All other cals are strictly temp\_real.
5. \*/

// <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

// シグナル操作の関数のプロトタイプ。

// <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

// 浮動小数点表現構造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。のプロトタイプ定義が含まれています。

// カーネルのよく使う機能

// <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

1. // セグメント・レジスタ・オペレーションのための
2. #include <signal.h>

13

1. #include <linux/math\_emu.h>
2. #include <linux/kernel.h>
3. #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

// 短い実数が配置されている // （ファイル math/ea.c 参照）が取得され、その後、対応する実数値がユーザデータ領域から読み込まれます。最後に，ユーザの短い実数が一時的な実数に変換され（math/convert.c），返されます。 // パラメータ：tmp - 一時的な実数へのポインタ，info - 情報構造体ポインタ， // code - 命令コード。

1. 18 void get\_short\_real(temp\_real \* tmp, 19 struct info \* info, unsigned short code)
2. {
3. char \* addr;
4. short\_real sr;

23

24 addr = ea(info,code); // 有効なアドレスを計算する。 25 sr = get\_fs\_long((unsigned long \*) addr); // ユーザーデータ領域から取得する。

26 short\_to\_temp(&sr,tmp); // tempの実数形式に変換する。 27 }

28

// ユーザーメモリのlong real（倍精度の実数）を取得します。

1. // この関数はget\_short\_real()と同じように処理されます。
2. void get\_long\_real(temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;
6. long\_real lr;

34

1. addr = ea(info,code); // get effective address.
2. lr.a = get\_fs\_long((unsigned long \*) addr); // get long real.
3. lr.b = get\_fs\_long(1 + (unsigned long \*) addr);
4. 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。

1. // Parameters: 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; // code - 命令コード。
2. void get\_temp\_real(temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;

45

1. addr = ea(info,code); // get effective address.
2. tmp->a = get\_fs\_long((unsigned long \*) addr);
3. tmp->b = get\_fs\_long(1 + (unsigned long \*) addr);
4. tmp->exponent = get\_fs\_word(4 + (unsigned short \*) addr);
5. }

51

// 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 // カレントレジスタから、対応する整数値をユーザデータ領域から // 読み出し、一時的な整数形式として保存します。最後に、一時的な整数値を一時的な実数に // 変換します。

// 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す。上位2バイトの最上位ビットが1の場合は負の数を，最上位ビットが0の場合は正の数を意味します。

1. // Parameters: 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; // code - 命令コード。
2. void get\_short\_int(temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;
6. temp\_int ti;

57

1. addr = ea(info,code); // get effective in the instruction code.
2. ti.a = (signed short) get\_fs\_word((unsigned short \*) addr);
3. ti.b = 0;
4. if (ti.sign = (ti.a < 0)) // set sign bit if it is a negative.
5. ti.a = - ti.a; // and unsignify the mantissa part.
6. int\_to\_real(&ti,tmp); // change to temp real. 64 }

65

// 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。

1. // この関数は、上記のget\_short\_int()と同じように処理されます。
2. void get\_long\_int(temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;
6. temp\_int ti;

71

1. addr = ea(info,code);
2. ti.a = get\_fs\_long((unsigned long \*) addr);
3. ti.b = 0;
4. if (ti.sign = (ti.a < 0))
5. ti.a = - ti.a;
6. int\_to\_real(&ti,tmp);
7. }

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。

ユーザーデータ領域から // 一時的な整数値として保存されます。最後に、一時的な整数値を一時的な実数に // 変換します。

1. // 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。
2. void get\_longlong\_int(temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;
6. temp\_int ti;

85

1. addr = ea(info,code); // get effective address.
2. ti.a = get\_fs\_long((unsigned long \*) addr); // get 64-bit longlong integer.
3. ti.b = get\_fs\_long(1 + (unsigned long \*) addr);
4. if (ti.sign = (ti.b < 0)) // set sign bit if it is a negative, 90 \_\_asm\_\_("notl %0 ; notl %1\n\t" // and complement, carry adjustment.
5. "addl $1,%0 ; adcl $0,%1"
6. :"=r" (ti.a),"=r" (ti.b)
7. :"" (ti.a),"1" (ti.b));
8. 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。

1. // このマクロは、以下のBCDコード値を一時的な実数形式に変換する際に使用します。その方法は，N<<1 + N<<3 である。
2. #define MUL10(low,high) \
3. \_\_asm\_\_("addl %0,%0 ; adcl %1,%1\n\t" \
4. "movl %0,%%ecx ; movl %1,%%ebx\n\t" \
5. "addl %0,%0 ; adcl %1,%1\n\t" \
6. "addl %0,%0 ; adcl %1,%1\n\t" \
7. "addl %%ecx,%0 ; adcl %%ebx,%1" \
8. :"=a" (low),"=d" (high) \
9. :"" (low),"1" (high):"cx","bx")

105

1. // 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。
2. #define ADD64(val,low,high) \
3. \_\_asm\_\_("addl %4,%0 ; adcl $0,%1":"=r" (low),"=r" (high) \
4. :"" (low),"1" (high),"r" ((unsigned long) (val)))

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

1. // その後，ユーザデータ領域から対応する10バイトのBCDコード（1バイトは符号に使用）を読み出し，一時的な整数値に変換します。最後に，一時的な整数値を一時的な実数に変換します。 // パラメータ：tmp - 一時的な実数へのポインタ，info - 情報構造体ポインタ， // code - 命令コード。
2. void get\_BCD(temp\_real \* tmp, struct info \* info, unsigned short code)
3. {
4. int k;
5. char \* addr;
6. temp\_int i;
7. unsigned char c; 116

// BCDコード値のメモリ有効アドレスを取得して、から処理を開始する。

// 最後のBCDコードバイト（最上位ビット）。最初にBCDコード値の符号ビットを取得する

1. // 一時的な整数の符号ビットを設定します。その後、9バイトのBCDコード値を一時的な整数形式に変換し、最後に一時的な整数値を一時的な実数に変換します。
2. addr = ea(info,code); // get effective address.
3. addr += 9; // point to the last (10th) byte.
4. i.sign = 0x80 & get\_fs\_byte(addr--); // get sign bit.
5. i.a = i.b = 0;
6. for (k = 0; k < 9; k++) { // change to temporary integer.
7. c = get\_fs\_byte(addr--); 123 MUL10(i.a, i.b);
8. ADD64((c>>4), i.a, i.b);
9. MUL10(i.a, i.b);
10. ADD64((c&0xf), i.a, i.b);
11. }
12. int\_to\_real(&i,tmp); // change to temporary real. 129 }

130

// 結果をショート（単精度）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。

// この関数は，まず命令コードの実効アドレス addr を取得し， // 一時実数の結果を短実数形式に変換して addr の位置に格納します。

1. // パラメータ：tmp - 一時的なリアルフォーマットの結果値、info - 情報構造体ポインタ、 // code - 命令コード。
2. void put\_short\_real(const temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;
6. short\_real sr;

136

1. 137 addr = ea(info,code); // 有効なアドレスを取得します。138 verify\_area(addr,4); // データ領域の確認。
2. temp\_to\_short(tmp,&sr); // convert to short real.
3. 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。

1. // この関数は、上記のput\_real\_real()と同じように処理されます。
2. void put\_long\_real(const temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;
6. long\_real lr;

148

1. addr = ea(info,code);
2. verify\_area(addr,8);
3. temp\_to\_long(tmp,&lr);
4. put\_fs\_long(lr.a, (unsigned long \*) addr);
5. put\_fs\_long(lr.b, 1 + (unsigned long \*) addr);
6. }

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， // そのアドレスに十分な（10 バイトの）ユーザメモリがあることを確認した後， // 一時的な実数を addr に保存します．

1. // パラメータ：tmp - 一時的なリアルフォーマットの結果値、info - 情報構造体ポインタ、 // code - 命令コード。
2. void put\_temp\_real(const temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;

160

1. addr = ea(info,code); // get effective address.
2. verify\_area(addr,10); // verify there is enough uesr memory.
3. put\_fs\_long(tmp->a, (unsigned long \*) addr); // store temp real to user area.
4. put\_fs\_long(tmp->b, 1 + (unsigned long \*) addr);
5. put\_fs\_word(tmp->exponent, 4 + (short \*) addr);
6. }

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

1. // 一時的な実数形式の結果を一時的な整数形式にします。それが負の数であれば、 // 整数の符号ビットを設定します。最後に，その整数をユーザメモリに保存します。 // パラメータ：tmp - 一時的実数フォーマットの結果値，info - 情報構造体ポインタ， // code - 命令コード。
2. void put\_short\_int(const temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;
6. temp\_int ti;

173

1. addr = ea(info,code); // get effective address.
2. real\_to\_int(tmp,&ti); // change to temp integer.
3. verify\_area(addr,2); // verify user area (need 2 bytes)
4. if (ti.sign)
5. ti.a = -ti.a; // negative the result.
6. put\_fs\_word(ti.a,(short \*) addr); // store to user data area. 180 }

181

// 結果を長整数形式でユーザーデータ領域に保存します。

1. // この関数は、上記のput\_short\_int()と同じように処理されます。
2. void put\_long\_int(const temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;
6. temp\_int ti;

187

1. addr = ea(info,code); // get effective address.
2. real\_to\_int(tmp,&ti); // change to temp integer.
3. verify\_area(addr,4); // verify user area (4 bytes).
4. if (ti.sign) // if signed, negative the result.
5. ti.a = -ti.a;
6. 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

1. // 一時的な実数形式の結果を一時的な整数形式にします。それが負の数であれば、 // 整数の符号ビットを設定します。最後に，その整数をユーザメモリに保存します。 // パラメータ：tmp - 一時的実数フォーマットの結果値，info - 情報構造体ポインタ， // code - 命令コード。
2. void put\_longlong\_int(const temp\_real \* tmp,
3. struct info \* info, unsigned short code)
4. {
5. char \* addr;
6. temp\_int ti;

201

1. addr = ea(info,code); // get effective address.
2. real\_to\_int(tmp,&ti); // change to temporary integer.
3. verify\_area(addr,8); // verify that 8 bytes are available.
4. if (ti.sign) // if signed, change to negative value.
5. \_\_asm\_\_("notl %0 ; notl %1\n\t" // invert and add one.
6. "addl $1,%0 ; adcl $0,%1"
7. :"=r" (ti.a),"=r" (ti.b)
8. :"" (ti.a),"1" (ti.b));
9. put\_fs\_long(ti.a,(unsigned long \*) addr); // store to the user data area.
10. put\_fs\_long(ti.b,1 + (unsigned long \*) addr);
11. }

213

1. // 符号なしの数値<high, low>を10で割り、その余りをremに入れる。
2. #define DIV10(low,high,rem) \
3. \_\_asm\_\_("divl %6 ; xchgl %1,%2 ; divl %6" \
4. :"=d" (rem),"=a" (low),"=b" (high) \
5. :"" (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コードを保存するためのスペースです。一時的な実数形式の結果は、その後、BCDコード形式のデータに変換され、 // ユーザーメモリに保存されます。負の値の場合は、 // メモリの最上位バイトの最上位ビットがセットされます。

// パラメータ：tmp - 一時的なリアルフォーマットの結果値、info - 情報構造体ポインタ、 // code - 命令コード。

1. 219 void put\_BCD(const temp\_real \* tmp,struct info \* info, unsigned short code) 220 {
2. int k,rem;
3. char \* addr;
4. temp\_int i;
5. unsigned char c; 225
6. addr = ea(info,code); // get the address for storing result.
7. verify\_area(addr,10); // verify memory space.
8. real\_to\_int(tmp,&i); // change to temporary integer.
9. 11.9 mul.c
10. if (i.sign) // if signed, set MSB bit of high byte.
11. put\_fs\_byte(0x80, addr+9);
12. else // otherwise reset the MSB bit.
13. put\_fs\_byte(0, addr+9);
14. for (k = 0; k < 9; k++) { // change to BCD code and stored.
15. DIV10(i.a,i.b,rem); 235 c = rem;
16. DIV10(i.a,i.b,rem);
17. c += rem<<4;
18. put\_fs\_byte(c,addr++);
19. }
20. }

241

## 11.9 mul.c

### 11.9.1 Function

### mul.cプログラムの関数は、CPUの通常の演算命令を使用して80387の乗算をシミュレートしています。

### 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

1. /\*
2. \* temporary real multiplication routine.
3. \*/

10

11 #include <linux/math\_emu.h>を追加します。

12

1. // パラメータcのポインタの16バイトの値を1ビット（2倍）だけ左にシフトします。
2. static void shift(int \* c)
3. {
4. \_\_asm\_\_("movl (%0),%%eax ; addl %%eax,(%0)\n\t"
5. "movl 4(%0),%%eax ; adcl %%eax,4(%0)\n\t"
6. "movl 8(%0),%%eax ; adcl %%eax,8(%0)\n\t"
7. "movl 12(%0),%%eax ; adcl %%eax,12(%0)"
8. ::"r" ((long) c):"ax");
9. }

21

1. // 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。
2. static void mul64(const temp\_real \* a, const temp\_real \* b, int \* c)
3. {
4. 11.9 mul.c
5. \_\_asm\_\_("movl (%0),%%eax\n\t"
6. "mull (%1)\n\t"
7. "movl %%eax,(%2)\n\t"
8. "movl %%edx,4(%2)\n\t"
9. "movl 4(%0),%%eax\n\t" 29 "mull 4(%1)\n\t"
10. "movl %%eax,8(%2)\n\t"
11. "movl %%edx,12(%2)\n\t"
12. "movl (%0),%%eax\n\t"
13. "mull 4(%1)\n\t"
14. "addl %%eax,4(%2)\n\t"
15. "adcl %%edx,8(%2)\n\t" 36 "adcl $0,12(%2)\n\t"
16. 37 "movl 4(%0),%%eax\\t" 38 "mull (%1)୨୧"
17. "addl %%eax,4(%2)\n\t"
18. "adcl %%edx,8(%2)\n\t"
19. "adcl $0,12(%2)"
20. ::"b" ((long) a),"c" ((long) b),"D" ((long) c)
21. :"ax","dx");
22. }

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.

1. 46 void fmul(const temp\_real \* src1, const temp\_real \* src2, temp\_real \* result) 47 {...
2. int i,sign;
3. 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

1. は， // 乗算時に加算されます。しかし，指数が偏った数の形式で格納されているため， // 2つの数の指数を加算するときに偏った量が2回加算されるので， // 偏った数を減算する必要があります（仮実数の偏った数は16383）．
2. sign = (src1->exponent ^ src2->exponent) & 0x8000;
3. i = (src1->exponent & 0x7fff) + (src2->exponent & 0x7fff) - 16383 + 1;

// 結果の指数が負になった場合、2つの数値が掛け合わされて // アンダーフローが発生したことを意味します。そこで，符号付きのゼロの値を直接返します。

1. // 結果の指数が 0x7fff より大きい場合、オーバーフローが発生したことを示し、 // ステータスワードのオーバーフロー例外フラグが設定され、返されます。
2. if (i<0) {
3. result->exponent = sign;
4. result->a = result->b = 0;
5. return; 57 }
6. if (i>0x7fff) {
7. set\_OE();
8. 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

11.10 div.c

1. // 2つの数値の仮数を掛け合わせた結果は0で、指数も0に設定されます。
2. mul64(src1,src2,tmp);
3. if (tmp[0] || tmp[1] || tmp[2] || tmp[3])
4. while (i && tmp[3] >= 0) {
5. i--;
6. shift(tmp);
7. }
8. else
9. i = 0;
10. result->exponent = i | sign;
11. result->a = tmp[2];
12. result->b = tmp[3];
13. }

74

## 11.10 div.c

### 11.10.1 Function

### div.cプログラムは、CPUの通常の計算命令を使って、80387コプロセッサの除算をシミュレートしています。

### 11.10.2 Code annotation

プログラム 11-9 linux/kernel/math/div.c

1. /\*
2. \* linux/kernel/math/div.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/

6

1. /\*
2. \* temporary real division routine.
3. \*/

10

11 #include <linux/math\_emu.h>を追加します。

12

1. // 4バイトの内容を1ビットだけ左にシフト（2倍）します。
2. static void shift\_left(int \* c)
3. {
4. \_\_asm\_\_ \_\_volatile\_\_("movl (%0),%%eax ; addl %%eax,(%0)\n\t"
5. "movl 4(%0),%%eax ; adcl %%eax,4(%0)\n\t"
6. "movl 8(%0),%%eax ; adcl %%eax,8(%0)\n\t"
7. "movl 12(%0),%%eax ; adcl %%eax,12(%0)"
8. ::"r" ((long) c):"ax");
9. }

21

// 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。

1. 11.10 div.c
2. static void shift\_right(int \* c)
3. {
4. \_\_asm\_\_("shrl $1,12(%0) ; rcrl $1,8(%0) ; rcrl $1,4(%0) ; rcrl $1,(%0)"
5. ::"r" ((long) c));
6. }

27

// 16バイトの減算関数。

1. // 最後に，借用フラグ(CF=1)があるかどうかで，okが設定される。借入がない場合（CF=0）は ok = 1，そうでない場合は ok = 0 となります。
2. static int try\_sub(int \* a, int \* b)
3. {
4. char ok;

31

1. \_\_asm\_\_ \_\_volatile\_\_("movl (%1),%%eax ; subl %%eax,(%2)\n\t"
2. "movl 4(%1),%%eax ; sbbl %%eax,4(%2)\n\t"
3. "movl 8(%1),%%eax ; sbbl %%eax,8(%2)\n\t"
4. "movl 12(%1),%%eax ; sbbl %%eax,12(%2)\n\t"
5. "setae %%al":"=a" (ok):"c" ((long) a),"d" ((long) b));
6. return ok;
7. }

39

// 16バイトの分割機能。

1. // 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。
2. static void div64(int \* a, int \* b, int \* c)
3. {
4. int tmp[4];
5. int i;
6. unsigned int mask = 0;

45

1. c += 4;
2. for (i = 0 ; i<64 ; i++) {
3. if (!(mask >>= 1)) {
4. c--;
5. mask = 0x80000000;
6. }
7. tmp[0] = a[0]; tmp[1] = a[1];
8. tmp[2] = a[2]; tmp[3] = a[3];
9. if (try\_sub(b,tmp)) {
10. \*c |= mask;
11. a[0] = tmp[0]; a[1] = tmp[1];
12. a[2] = tmp[2]; a[3] = tmp[3];
13. }
14. shift\_right(b);
15. }
16. }

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。

1. 63 void fdiv(const temp\_real \* src1, const temp\_real \* src2, temp\_real \* result) 64 {。
2. int i,sign;
3. int a[4],b[4],tmp[4] = {0,0,0,0}; 67
4. 11.11 まとめ
5. sign = (src1->exponent ^ src2->exponent) & 0x8000;
6. if (!(src2->a || src2->b)) {
7. set\_ZE();
8. return;
9. }
10. i = (src1->exponent & 0x7fff) - (src2->exponent & 0x7fff) + 16383;
11. if (i<0) {
12. set\_UE();
13. result->exponent = sign;
14. result->a = result->b = 0;
15. return;
16. }
17. a[0] = a[1] = 0;
18. a[2] = src1->a;
19. a[3] = src1->b;
20. b[0] = b[1] = 0;
21. b[2] = src2->a;
22. b[3] = src2->b;
23. while (b[3] >= 0) {
24. i++;
25. shift\_left(b);
26. }
27. div64(a,b,tmp);
28. if (tmp[0] || tmp[1] || tmp[2] || tmp[3]) {
29. while (i && tmp[3] >= 0) {
30. i--;
31. shift\_left(tmp);
32. }
33. if (tmp[3] >= 0)
34. set\_DE();
35. } else
36. i = 0;
37. if (i>0x7fff) {
38. set\_OE();
39. return;
40. }
41. if (tmp[0] || tmp[1])
42. set\_PE();
43. result->exponent = i | sign;
44. result->a = tmp[2];
45. result->b = tmp[3];
46. }

110

## 11.11 Summary

本章では、80387 math coprocessor chipをエミュレートするためのLinuxカーネルの手法とコードの実装について説明します。まず、一般的に使用されているいくつかの整数型と浮動小数点型を紹介し、80387のランタイムとソフトウェア・シミュレーションで使用される一時的な実数型について説明し、それらの具体的な表現形式を示しました。続いて、演算コプロセッサの構成と動作原理を紹介し、さらに

11.11 まとめ

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

|  |  |  |
| --- | --- | --- |
| ファイル名 サイズ 最終更新時刻(GMT) Desc. Makefile 7176バイト 1992-01-12 19:49:06 bitmap.c 4007バイト 1992-01-11 19:57:29 block\_dev.c 1763バイト 1991-12-09 21:11:23 buffer.c 9072バイト 1991-12-06 20:21:00 char\_dev. c 2103 バイト 1991-11-19 09:10:22 exec.c 9908 バイト 1992-01-13 23:36:33 fcntl.c 1455 バイト 1991-10-02 14:16:29 file\_dev.c 1852 バイト 1991-12-01 19:02:43 file\_table. c 122バイト 1991-10-02 14:16:29 inode.c 7166バイト 1992-01-10 22:27:26 ioctl.c 1136バイト 1991-12-21 01:58:35 namei.c 18958バイト 1992-01-12 04:09:58 open. c 4862 バイト 1992-01-08 20:01:36 pipe.c 2834 バイト 1992-01-10 22:18:11 read\_write.c 2802 バイト 1991-11-25 15:47:20 select.c 6381 バイト 1992-01-13 22:25:23 stat.c 1875 バイト 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

ファイルシステムは、オペレーティングシステムの重要な部分であり、オペレーティングシステムが大量のプログラムやデータを長期間保存しておく場所である。システムが実行プログラムを読み込む際には、ファイルシステムからメモリに素早く読み込んで実行する必要がある。また、システムの動作中に生成される一部の一時ファイルも、ファイルシステムに動的に保存する必要があります。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。

### 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。

### 12.1.1 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。

a disk block

data

zone

i

-

nodes

boot block

super block

i

-

node bitmap logic block bitmap

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

図では、ブロック装置全体を1KB単位でディスクブロックに分割しているので、360KBのフロッピーディスク装置の場合、上の図では360個のディスクブロックがあり、それぞれの四角が1つのディスクブロックを表していることになります。後述するように、MINIX 1.0のファイルシステムでは、ディスクブロックのサイズは論理ブロックのサイズと全く同じで、同じく1KBである。したがって、360KBのディスクには360個の論理ブロックが含まれています。この後の説明では、この2つの名称を混在させることがあります。

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。しかし、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、どのディスクブロック装置にもブートブロックのスペースが必要です。つまり、ファイルシステムは、ブロックデバイスにブートブロックを格納するためのスペースを残すだけです。カーネルイメージファイルをファイルシステムに置いておけば、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ロードさ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。

FAT32 FS

NTFS

EXT2

F

S

MINIX F

S

master boot sector

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

スーパーブロックは、ディスク装置上のファイルシステムの構造情報を格納するためのもので、各部のサイズが記述されている。その構造を図12-3に示す。ここで、s\_ninodesはデバイス上のi-nodeの総数、s\_nzonesはデバイス上の論理ブロックの総数、s\_imap\_blocksとs\_zmap\_blocksはそれぞれi-nodeのビットマップと論理ブロックのビットマップが占有するディスクブロック数、s\_firstdatazoneはデバイス上のデータ領域の先頭が占有する最初の論理ブロック番号、S\_log\_zone\_sizeはベース2の対数で表される各論理ブロックに含まれるディスクブロック数である。MINIX 1.0ファイルシステムの場合、この値は0であるため、その論理ブロックのサイズはディスクブロックのサイズである1KBに等しい。 s\_max\_sizeは、4GBを超えない最大ファイル長をバイトで表したものである。もちろん、この長さの値は、ディスクの容量によって制限される。S\_magicは、ファイルシステムの種類を示すためのファイルシステムマジックナンバーである。MINIX 1.0ファイルシステムの場合、マジックナンバーは0x137fである。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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。  で利用可能です。   |  |  |  | | --- | --- | --- | | ディスク＆メモリー  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 | Log2(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) |   メモリ上でしか利用できません。 |

図12-3 MINIXファイルシステムのスーパーブロック構造

論理ブロックビットマップは、ディスク上の各データブロックの使用状況を表すのに使用されます。論理ブロックビットマップの各ビットは、最初のビット(ビット0)に加えて、ディスク上のデータ領域の論理ブロックを順に表します。したがって、論理ブロックビットマップのビット1は、ディスク上の最初のディスクブロック（ブートブロック）ではなく、ディスク上のデータ領域における最初のデータディスクブロックを表している。データディスクブロックが占有されると，論理ブロックビットマップの対応するビットがセットされる。すべてのディスクデータブロックが占有されている場合、空きディスクブロックを見つける関数は0の値を返すため、論理ブロックビットマップの最下位ビット（ビット0）はアイドル状態であり、ファイルシステムの作成時に1に設定される。

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ますので、MINIXファイルシステム1.0がサポートでき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。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| f  ields in disk  and memory    total 32 bytes    fields  available    only in memory | |  |  |  | | --- | --- | --- | | 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 | ファイルが占有している論理ブロック番号：zone[0]-zone[6] 直接ブロック番号 zone[7] 間接ブロック番号  ゾーン[8] 第2間接ブロック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

i\_modeフィールドは、ファイルタイプとアクセス権のプロパティを保存するためのフィールドです。図12-5に示すように，ビット15～12はファイルタイプを，ビット11～9はファイル実行時に設定される情報を，ビット8～0はファイルのアクセス権を保存するのに使用される。詳細は，ファイルinclude/sys/stat.hの20行目～50行目およびファイルinclude/fcntl.hを参照のこと。

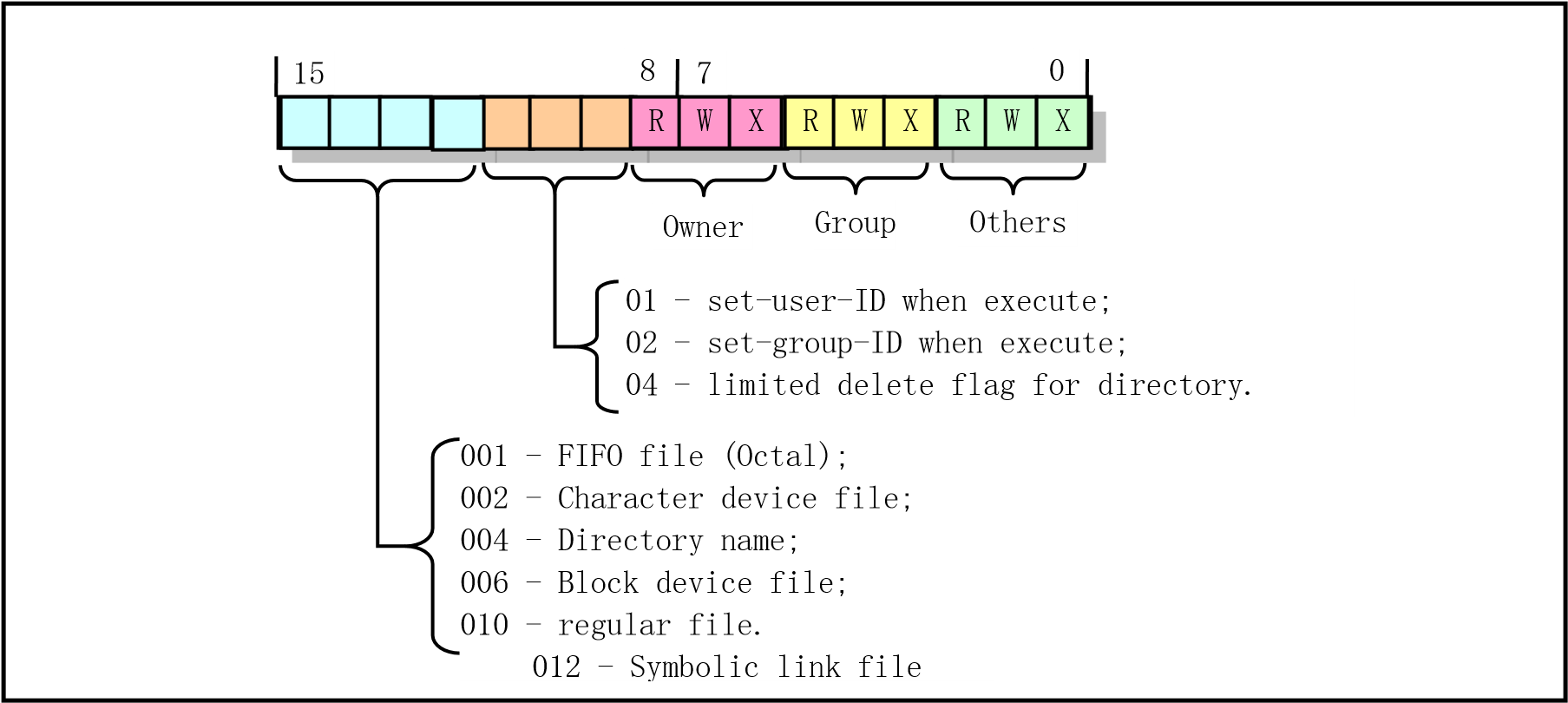


図12-5 i-nodeモードフィールドの内容

ファイルのデータはディスクブロックのデータ領域に格納され、ファイル名は対応するi-nodeを通じて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i\_zone[]に格納されている。i\_zone[]配列は、i-nodeに対応するファイルのディスク・ブロック番号を格納するために使用される。i\_zone[0]～i\_zone[6]は、ダイレクト・ブロックと呼ばれるファイルの先頭にある7つのディスク・ブロック番号を格納するために使用される。ファイルの長さが7Kバイト以下の場合は、i-nodeに応じて使用するディスクブロックをすぐに見つけることができる。ファイルのサイズが大きい場合は、追加のブロック番号を保持する間接ブロック（i\_zone[7]）を使用する必要があります。MINIXファイルシステムの場合、512個のディスクブロック番号を格納できるので、512個のディスクブロックを指定することができます。ファイルのサイズが大きい場合は、セカンダリ間接ディスクブロック（i\_zone[8]）を使用する必要があります。二次間接ブロックの一次ディスクブロックは、一回限りの間接ディスクブロックと同様に機能するので、図12-6に示すように、二次間接ディスクブロックを使って512\*512のディスクブロックをアドレスすることができる。したがって、MINIXファイルシステムのバージョン1.0の場合、1つのファイルの最大長は（7 + 512 + 512 \* 512） = 262,663KBとなります。

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。

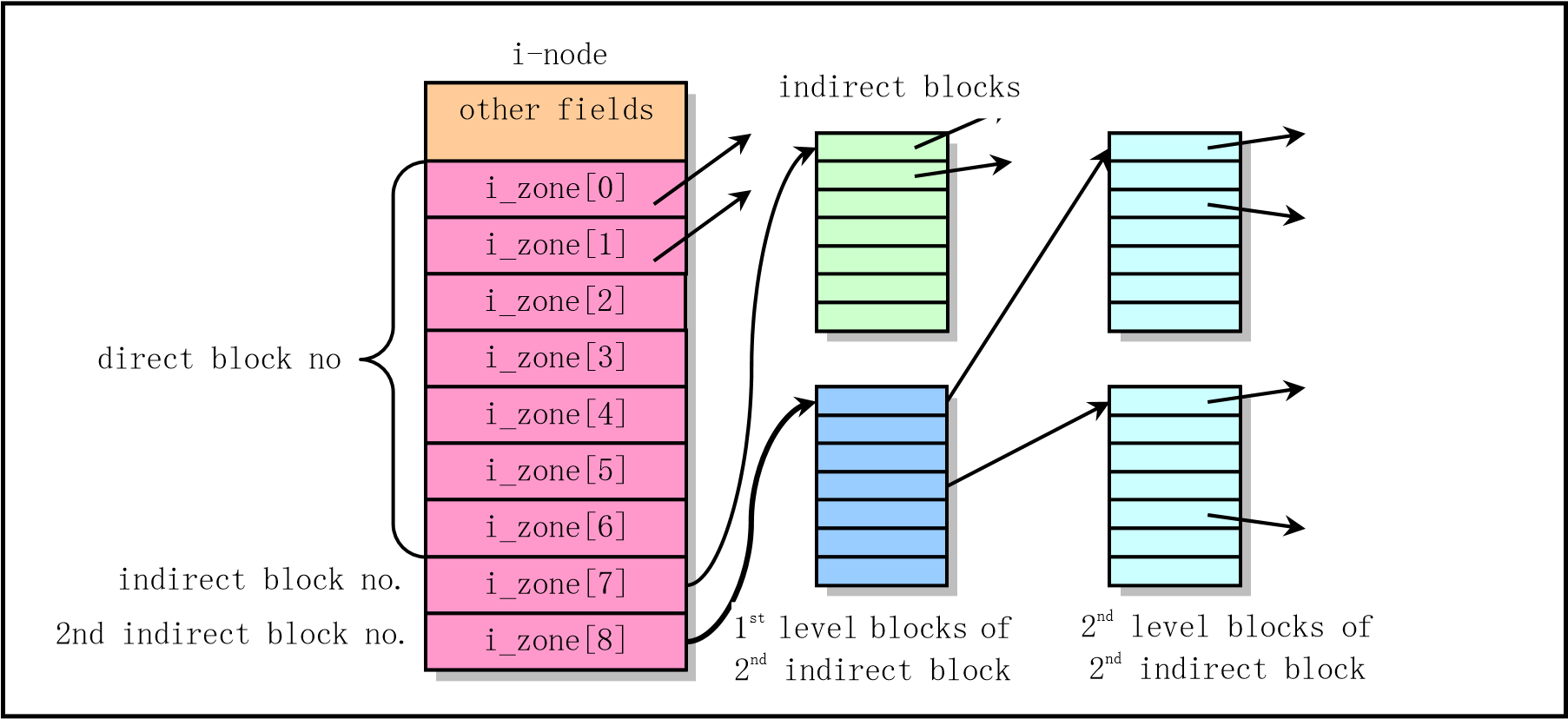


図12-6 i-nodeの論理ブロック配列の機能

ここでも、すべてのi-nodeが使用されている場合、アイドルのi-nodeを探す関数は0の値を返します。したがって、i-nodeビットマップの最下位ビットとi-node 0は使用されません。i-node 0の構造は全て0に初期化され、ファイルシステムの作成時にi-nodeビットマップの0のビット位置が設定される。 PCの場合、ブロックデバイスのデータブロック長としては、1セクタ（512バイト）の長さが一般的に用いられる。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。

### 12.1.2 File Types, Attributes, and Directory Items

#### 12.1.2.1 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。

L

ast modified time

owner

others

p

ermissions

-

rwxr

-

xr

-

x 1 ftpadm ftp 479

Sep

26

README

17:28

File

types

'

-

' Regular file

'd' Directory name

's' Symbolic link

'p' Named pipe

'c' Character device file

'b' Block device file

group

file name

File size

group name

owner

name

Link

count

図12-7 コマンド「ls -l」で表示されるファイル情報

図では、表示されたファイル情報の最初の文字が、リストアップされたファイルの種類を示しています。例えば、図中の「-」は、そのファイルがレギュラー（通常）ファイルであることを示しています。

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ており、上位4ビットで表される。ファイルシステムを操作する際、システムはS\_ISBLK、S\_ISDIRなどのファイルタイプを決定するいくつかのマクロを使用します。これらのマクロは，include/sys/stat.hファイルで定義されている。 図では，ファイルタイプ文字の後に，1セットの3文字で構成される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のに使用されている。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 0.12のソースコードでは、ファイルパーミッションの情報は、i-nodeのi\_modeフィールドにも格納されており、その下位9ビットは3組のパーミッションを表すのに使われ、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。

#### 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。

#### 12.1.2.2 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 // 最大の名前の長さ。

#define ROOT\_INO 1 // ルートのi-nodeです。

// struct dir\_entry { unsigned short inode; // i-node number.

char name[NAME\_LEN]; // ファイル名。};

各ファイルのディレクトリエントリは、長さ14バイトのファイル名の文字列と、ファイル名に対応する2バイトのi-node番号のみを含む。したがって、1つの論理ディスクブロックには、1024/16=64個のディレクトリエントリを格納することができる。ファイルに関するその他の情報は、i-node番号で指定されるi-node構造体に格納される。構造体には主に、ファイルのアクセス属性、ホスト、長さ、アクセスセーブタイム、ディスクブロックなどの情報が格納される。各inode番号のi-nodeは、ディスク上の固定された場所に配置されている。

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番号を知ることができるので、ディスクからi-node構造の情報を得ることができます。

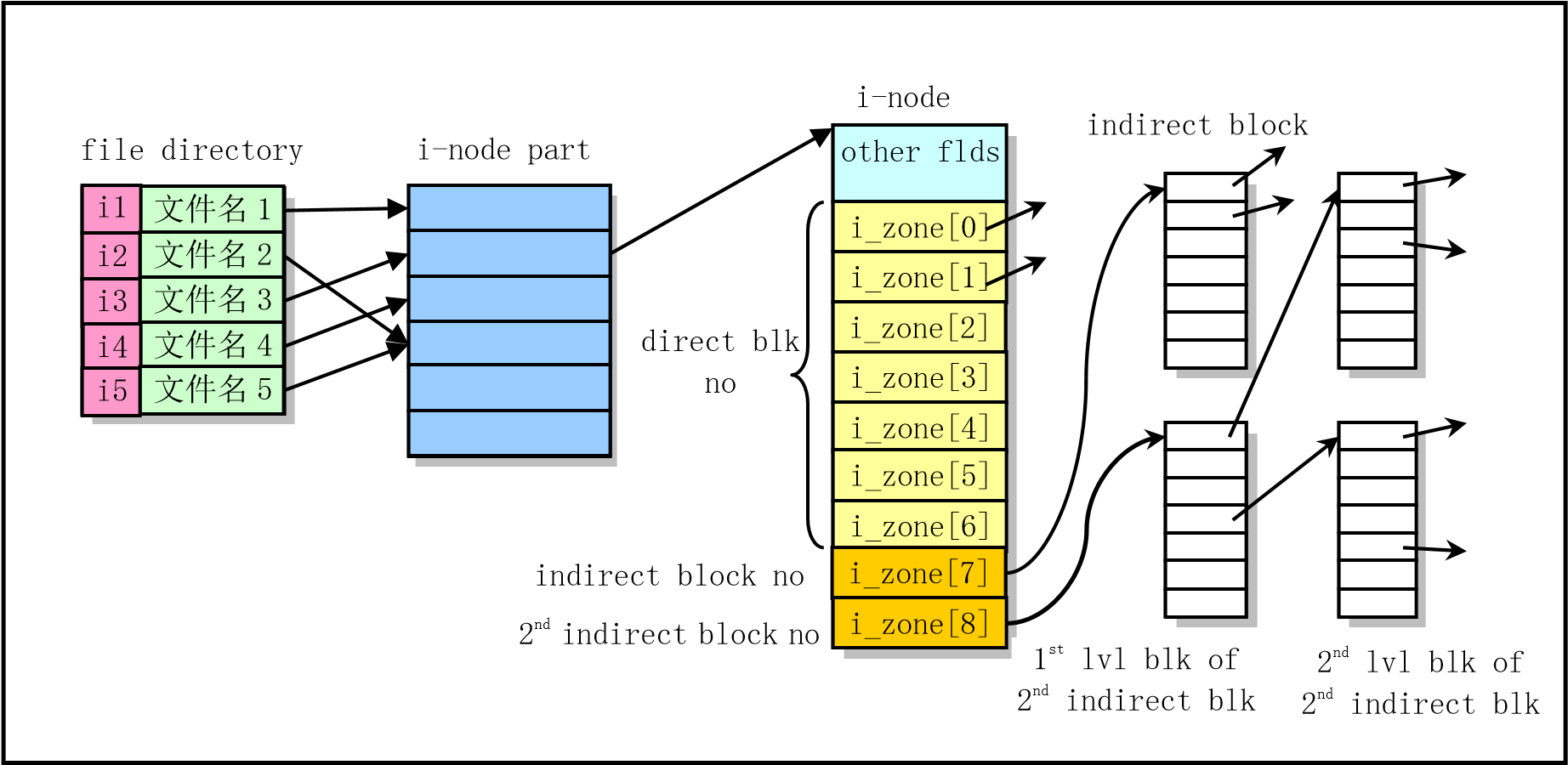


図12-8 ファイル名からファイルのディスクブロックの位置を探す

ディスク上のファイルの分布を見ると、ファイルのブロック情報を検索するプロセスは、図12-9で表すことができます（ブートブロック、スーパーブロック、i-node、論理ブロックのビットマップは表示されていません）。

Data blocks

i

-

node

table

Directory block

i

-

nod

i

-

nod

i

-

nod

i

-

nod

i

-

node no

Filename

1

Directory block

i

-

node no

Filename2

①

②

③

図12-9 ファイル名からそのデータブロックを取得する

ユーザープログラムで指定されたファイル名により、対応するディレクトリエントリを見つけることができる。ディレクトリ・エントリのi-node番号に応じて、i-nodeテーブルの対応するi-node構造を見つけることができる。i-node構造には、ファイルデータのブロック番号情報が含まれているので、最終的にファイル名に対応するデータ情報を得ることができる。図では、同じi-nodeを指す2つのディレクトリエントリがあるので、2つのファイル名に基づいて、ディスク上の同じデータを得ることができる。各i-node構造体にはリンクカウントフィールドi\_nlinksがあり、そのi-nodeを指しているディレクトリエントリの数、つまりファイルのハードリンクカウント値が記録されています。ファイルの削除操作を行う際，カーネルはi-nodeのリンクカウント値が0の場合にのみ，実際にファイルのデータをディスク上に削除する。また，ディレクトリ・エントリのi-node番号は，現在のファイル・システムでしか使用できないため，あるファイル・システムのディレクトリ・エントリを使用して，他のファイル・システムのi-nodeを指すことはできない，つまり，ハード・リンクはファイル・システムを横断することはできない。

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のです。 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。

56

..

123

.

i

-

node no

..

56

.

123

mydir

i

-

node table

Current directory block

0

56

123

Sub

-

directory block

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

この図では，iノード番号56のディレクトリに「mydir」サブディレクトリを作成し，このサブディレクトリのiノード番号を123としています．mydir」サブディレクトリの「...」ディレクトリ・エントリは自身のiノード123を指し、「...」ディレクトリ・エントリは親ディレクトリのiノード56を指しています。このように、ディレクトリのディレクトリ・エントリは常に2つのリンクを持っているので、サブディレクトリがある場合、親ディレクトリのi-nodeのリンク数は2+サブディレクトリの数に等しいことがわかります。

#### 12.1.2.3 Directory Structure Example

Linux 0.12システムを例にして、そのルートディレクトリ構造を見てみましょう。bochsシミュレーションシステムでLinux 0.12システムを実行した後、まず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。これ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 336 Mar 21 2004 dev drwxr-xr-x 2 root 224 Mar 21 2004 etc drwxr-xr-x 8 root 128 Mar 21 2004 image drwxr-。 xr-x 2 root 32 Mar 21 2004 mnt drwxr-xr-x 2 root 64 Mar 21 2004 tmp drwxr-xr-x 10 root 192 Mar 29 2004 usr drwxr-xr-x 2 root 32 Mar 21 2004 var

[/]# hexdump .

0000000 0001 002e 0000 0000 0000 0000 0000 0000 // .

0000010 0001 2e2e 0000 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 // デバイス

0000040 0004 7465 0063 0000 0000 0000 0000 0000 0000 // その他

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 // idle, not used.

0000090 0052 6d69 6761 0065 0000 0000 0000 0000 // イメージ

00000a0 007b 6176 0072 0000 0000 0000 0000 0000 // var.

00000b0

[/]#

hexdump.コマンドを実行すると、i-nodeのデータブロックに含まれるすべてのディレクトリ・エントリがリストアップされます。各行がディレクトリ・エントリに対応する。各行の最初の2バイトがi-node番号、次の14バイトがファイル名またはディレクトリ名の文字列である。ディレクトリ・エントリのi-node番号が0の場合は、そのディレクトリ・エントリが使用されていないか、または対応するファイルが削除されたか除去されたことを意味する。最初の2つのディレクトリ・エントリ（'.'と'.'）のi-node番号はすべて1番である。これは、ファイルシステムのルートディレクトリ構造の特殊性であり、他のサブディレクトリ構造とは異なるものである。

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 total 32 drwxr-xr-x 2 root 224 Mar 21 2004 . drwxr-xr-x 10 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-- 1 root root 142 Mar 5 2004 mtools

-rw-r--r-- 1 root root 266 Mar 4 2004 passwd

-rw-r--r-- 1 root 147 Mar 4 2004 profile -rw-r--r-- 1 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 0000 // ...

0000020 0007 6372 0000 0000 0000 0000 0000 0000 // rc

0000030 000b 7075 6164 6574 0000 0000 0000 0000 // アップデート

0000040 0113 6574 6d72 6163 0070 0000 0000 // タームキャップ

0000050 00ee 746d 6261 0000 0000 0000 0000 // mtab 0000060 0000 746d 6261 007e 0000 0000 0000 // 使用していません。

0000070 007C 616D 6967 0063 0000 0000 0000 // magic

0000080 0016 7270 666f 6c69 0065 0000 0000 // プロファイル

0000090 007E 6170 7373 6477 0000 0000 // passwd 00000a0 0081 7267 756f 0070 0000 0000 // group

00000b0 01ee 746d 6f6f 736c 0000 0000 0000 0000 // mtools

00000c0

[/]#

### この時点で、etc/ディレクトリ名のi-nodeに対応するデータブロックには、サブディレクトリ内のすべてのファイルのディレクトリエントリ情報が含まれていることがわかります。ディレクトリ・エントリ「...」のi-nodeは、etc/ディレクトリ・エントリのi-node番号4であり、「...」のi-nodeは、etc/親ディレクトリのi-node番号1である。

### 12.1.3 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。

### 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。

### 12.1.4 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。

namei

iget iput bmap

get\_super

put\_super

new\_block

free\_block

new\_inode

free\_inode

truncate

図12-11 ファイルシステムの低レベル機能階層

### 12.1.5 File Data Access Operations

ファイル内のデータに対するアクセス操作コードは、主にblock\_dev.c、file\_dev.c、char\_dev.c、pipe.c、read\_write.cの5つのファイルが関係しており、関係する読み書き機能を図12-12に示します。最初の4つのファイルは、ブロック・デバイス、キャラクタ・デバイス、パイプライン・デバイス、通常のファイルとファイルの読み書きシステム・コールのためのインターフェース・プログラムと考えることができます。これらはまとめてread\_write.cのread()とwrite()のシステムコールを実装しています。操作されるファイルの属性を判断して、2つのシステムコールはこれらのファイル内の関連する処理関数を呼び出して動作させます。

system

-

calls read() write()

block\_read()

block

\_write()

file\_read()

file\_write()

read\_pipe()

write\_pipe()

rw\_char()

図12-12 ファイルデータアクセス機能

block\_dev.cプログラムのblock\_read()およびblock\_write()関数は、ブロック・デバイス・スペシャル・ファイルからデータを読み書きするために使用されます。使用するパラメータは、アクセスするデバイス番号、読み出しと書き込みの開始位置と長さを指定します。

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。

pipe.cファイルには、パイプの読み書き関数read\_pipe()とwrite\_pipe()が実装されています。また、匿名のパイプを作成するシステムコールpipe()も実装されています。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全く同じです。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・キャラクター・デバイスなどがありますが、キャラクター・デバイス・ファイルでは、システム・コールのread()とwrite()がchar\_dev.cのrw\_char()関数を呼び出します。 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、ファイルへの操作上のアクセスを管理している。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; // ファイルの動作モード(RWビット) unsigned short f\_flags; // ファイルのオープンフラグとコントロールフラグ unsigned short f\_count; // ファイルハンドラのカウント struct m\_inode \* f\_inode; // メモリ上のi-node(v-node)を指す off\_t f\_pos; // 現在の読み書きの位置。

}; struct file file\_table[NR\_FILE] // ファイルテーブルの配列、全64項目。

ファイルハンドルとメモリi-nodeテーブルのi-nodeエントリとの関係を確立するために使用される。ファイルタイプおよびアクセス属性フィールドf\_modeは，前述のファイルi-node構造体のi\_modeフィールドと同じ意味を持つ。f\_flagsフィールドは，ファイルinclude/fcntl.hで定義されているオープンファイル関数open()および制御関数fcntl()のパラメータフラグによって与えられる，いくつかのオープン操作制御フラグを組み合わせたもので，以下のフラグがある。

1. // 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。
2. #define O\_RDONLY 00 // open file in read only mode.
3. #define O\_WRONLY 01 // open file in write only mode.
4. #define O\_RDWR 02 // open file with read/write mode.
5. // 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。
6. #define O\_CREAT 00100 // create if not exist (not used by fcntl).
7. #define O\_EXCL 00200 // exclude using file flag.
8. #define O\_NOCTTY 00400 // no control tty.
9. #define O\_TRUNC 01000 // truncate to 0 if file exist and in write mode.
10. #define O\_APPEND 02000 // open in append mode, point to eof.
11. #define O\_NONBLOCK 04000 // open in nonblack mode.
12. #define O\_NDELAY O\_NONBLOCK

ファイル構造体のファイル参照回数フィールドf\_countは，ファイルハンドルによるファイルの参照回数を示す。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テーブルのメモリi-node構造体項目を指す。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、開いている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）。

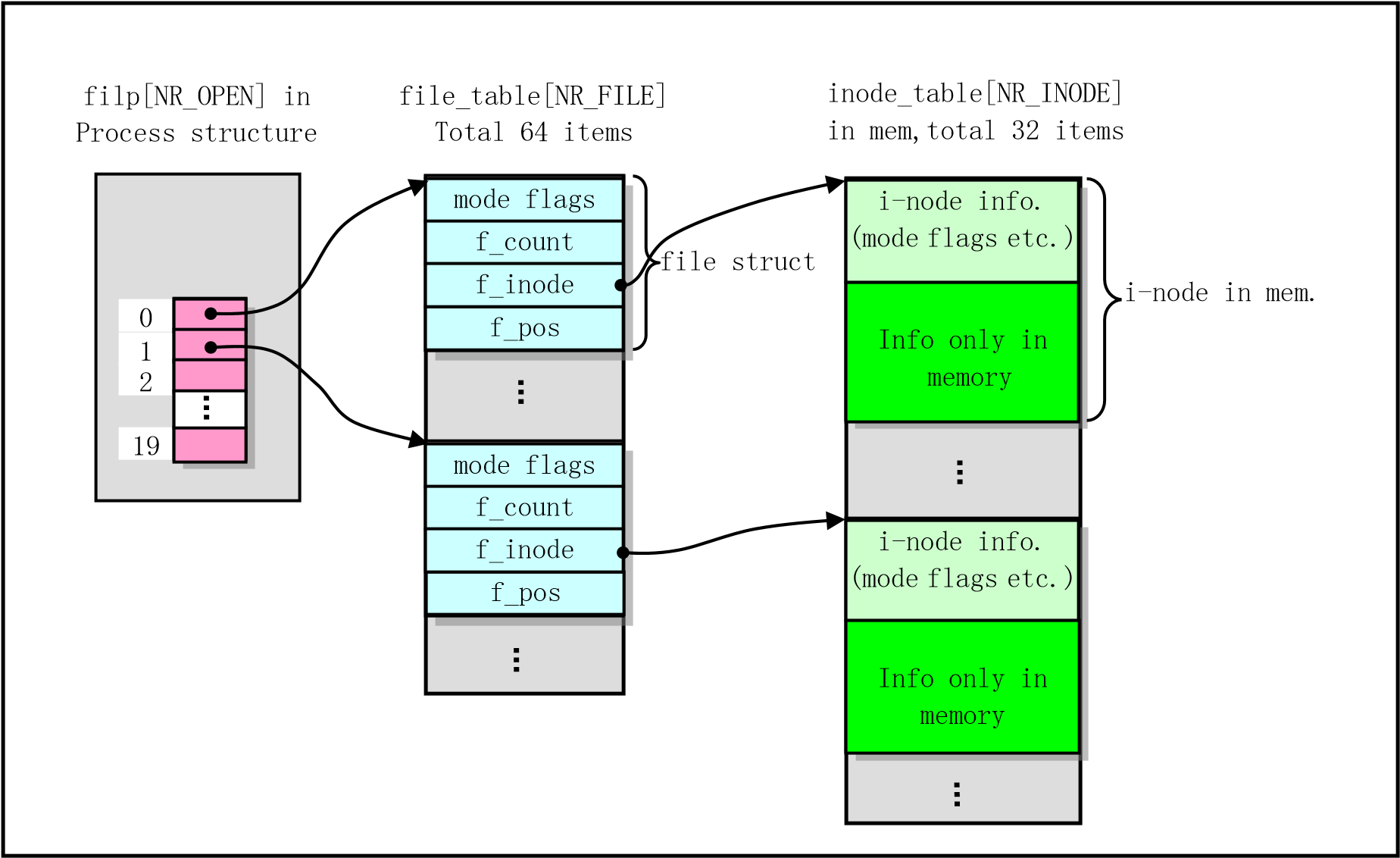


図12-13 ファイルを開くためにプロセスが使用する構造体

### 12.1.6 File and Directory Management System-Calls

ユーザーによるファイルシステムへのアクセスは、カーネルが提供するシステムコールによって実現されます。ファイルのシステムコールの上位層の実装は、基本的に図12-14に示す5つのファイルで構成されています。

open.c

exec.c

fcntl.c

stat.c

ioctl.c

図12-14 ファイルシステム上位層の動作プログラム

open.cファイルは、ファイル操作に関するシステムコールの実装に使用されます。主に、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、ファイルの時刻の変更、システムファイルのルートの変更などがあります。

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)の関数番号\_\_NR\_execve()のCハンドラであり、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。

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

図12-1のファイルシステム構造の理解を深めるために、Linux 0.12システムを使って、360KBのフロッピーイメージの中に、hello.cというファイルだけを格納したMINIX 1.0のファイルシステムを作成しました。

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 // 2nd fdに360KBのファイルシステムを作成。

120 inode // 120のi-nodeと360のディスク（ロジック）ブロック。

360ブロック

Firstdatazone=8 (8) // データゾーンの開始ブロック番号は8です。

Zonesize=1024 // ブロックサイズは1024バイト。

Maxsize=268966912 // ファイルの最大サイズ(明らかに間違っています)

[/usr/root]# mount /dev/fd1 /mnt // /mntディレクトリにマウントして、ファイルをコピーする。

[/usr/root]# cp hello.c /mnt

[/usr/root]# ll -a /mnt // 3つのディレクトリエントリがあります。 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 // ファイルシステムをアンマウントします。

[/usr/root]#

上記の一連の操作では、第2フロッピードライブのフロッピーディスク（イメージファイル）に対してmkfsコマンドを実行した後、ディスク上にMINIXファイルシステムが作成されます。コマンド実行後に表示される内容から、このファイルシステムには、合計120個のi-nodeと360個のディスクブロックが含まれていることがわかる。ディスクのデータゾーンの最初のブロック番号は8で、論理ブロックのサイズはディスクブロックと同じ1024バイトで、保存できるファイルの最大サイズは268,966,912バイトである（明らかにおかしい）。

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/root]# hexdump /dev/fd1 | more

0000000 44eb 4d90 6f74 6c6f 2073 0020 0102 0001 // 0x0000～0x03ff（1KB）がブートブロック。

0000010 E002 4000 F00B 0009 0012 0002 0000 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）はスーパーブロックです。

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 0000 0000 ff00 // 0x0800～0x0bff（1KB）はi-nodeのビットマップです。

* 0000810 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),論理的なBlkビットマップ。

0000c10 0000 0000 0000 0000 0000 0000 0000 0000

0000C20 0000 0000 0000 0000 0000 0000 0000 FFFE FFFE

0000c30 ffff ffff ffff ffff ffff ffff

\*

0001000 41ed 0000 0030 0000 c200 421c 0200 0008 // 0x1000 - 0x1fff (4KB)は120個のi-nodeです。

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）、ルートi-nodeデータ

0002010 0001 2e2e 0000 0000 0000 0000 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)、hello.cファイルです。

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

--もっと--。

それでは、上にあげた内容を一つずつ分析してみましょう。図12-1によると、MINIX1.0ファイルシステムの最初のディスクブロックはブートブロックなので、ディスクブロック0（0x0000～0x03ff、1KB）がブートブロックの内容となります。ディスクがシステムの起動に使われるかどうかにかかわらず、新しく作成されたファイルシステムには、それぞれブート・ディスク・ブロックが保持されます。新しく作成されたディスクイメージファイルでは、ブートディスクブロックの内容はすべて0にします。上記の表示データのうち、ブート・ディスク・ブロックの内容は、元のイメージ・ファイルに残されているデータであり、つまり、mkfsコマンドは、ファイル・システムの作成時にブート・ディスク・ブロックの内容を変更していません。

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 |

ディスクブロック2(0x0800 - 0x0bff, 1 KB)には、i-nodeのビットマップ情報が含まれている。ファイルシステムには合計120個のi-nodeがあり、各ビットがi-node構造を表しているので、実際には1KBサイズのディスクブロックの中で120/8=15バイトを占めている。ビット値の0は対応するi-node構造が占有されていないことを示し、1は占有または予約されていることを示す。ディスク・ブロック内の残りの未使用のバイト・ビットは、mkfsコマンドによって1に初期化されます。

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ビットはすでに占有されている。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. |

ルートiノードのデータブロックは1ブロックしかなく、その論理ブロック番号は8で、ディスクデータゾーンの最初のブロックに位置し、長さは30バイトであることがわかる。前節でディレクトリエントリの長さを16(0x10)バイトとしたので、この論理ブロックには3つのディレクトリエントリ(0x30バイト)が共存していることになる。ディレクトリであるため、リンク数は2である。

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） |

## ディスクブロック9（0x2400～0x27ff、1KB）は、hello.cファイルの内容です。74バイトのテキスト情報が含まれています。

## 12.2 buffer.c

### 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。

### 12.2.1 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について説明します。
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。

end

0

M

4

4.5

M

M

16

M

1

K

640

ernel

K

Module

Buffer Cache

RAMDisk

Main Memory

Area

Video Mem &

BIOS ROM

図12-15 物理メモリ内のバッファキャッシュの位置

1. 図中のキャッシュの開始位置は、カーネルモジュールの終了位置「end」から始まっており、「end」はカーネルモジュールのリンク時にリンカーldが設定する外部変数である。この記号はカーネルコードでは定義されていません。リンカがシステムモジュールを生成する際、ldプログラムは'end'のアドレスを設定するが、このアドレスは'data\_start + datasize + bss\_size'に等しく、カーネルモジュールの終わりであるbssセグメントの終わりの後の最初の有効アドレスとなる。さらにリンカーは、コードセグメントの後の最初のアドレスとデータセグメントの後の最初のアドレスを表す2つの外部変数'etext'と'edata'も設定する。
2. 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。

Block

Low end

High end

Buffer head struct

L

ess than one block,

discarded

図12-16 バッファキャッシュの初期化

1. 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; // バッファブロックのデータ領域(1024バイト)を指す unsigned long b\_blocknr; // ブロックNo. unsigned short b\_dev; // データソースのデバイスNo.を示す(0 = free) unsigned char b\_uptodate; // データが更新されたかどうかを示す。

unsigned char b\_dirt; // モディファイドフラグ：0 -クリーン、1 -モディファイド（ダーティ）。

unsigned char b\_count; // ブロックを使用しているユーザーの数。

unsigned char b\_lock; // そのブロックがロックされているかどうか。0-ok, 1-lock struct task\_struct \* b\_wait; // バッファを待っているタスクを指す struct buffer\_head \* b\_prev; // ハッシュキューの前のブロック。

struct buffer\_head \* b\_next; // ハッシュキュー上の次のブロック struct buffer\_head \* b\_prev\_free; // フリーリスト上の前のブロック struct buffer\_head \* b\_next\_free; // フリーリスト上の次のブロック。};

フィールドb\_lockは、ドライバがバッファブロックの内容を変更しているため、バッファブロックがビジー状態であり、ロックされていることを示すロックフラグです。このフラグは、バッファブロックの他のフラグとは独立しており、主にblk\_drv/ll\_rw\_block.cプログラムにおいて、バッファブロック内のデータ情報を更新する際に、バッファブロックをロックするために使用されます。なぜなら、バッファブロック内のデータを更新する際には、現在のプロセスが自発的にスリープ待機に入るため、他のプロセスがバッファブロックにアクセスする機会があるからです。そのため、他のプロセスにデータを利用されないようにするためには、スリープする前にバッファブロックをロックする必要があります。

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によって使用（参照）されている回数を示しています。したがって、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。

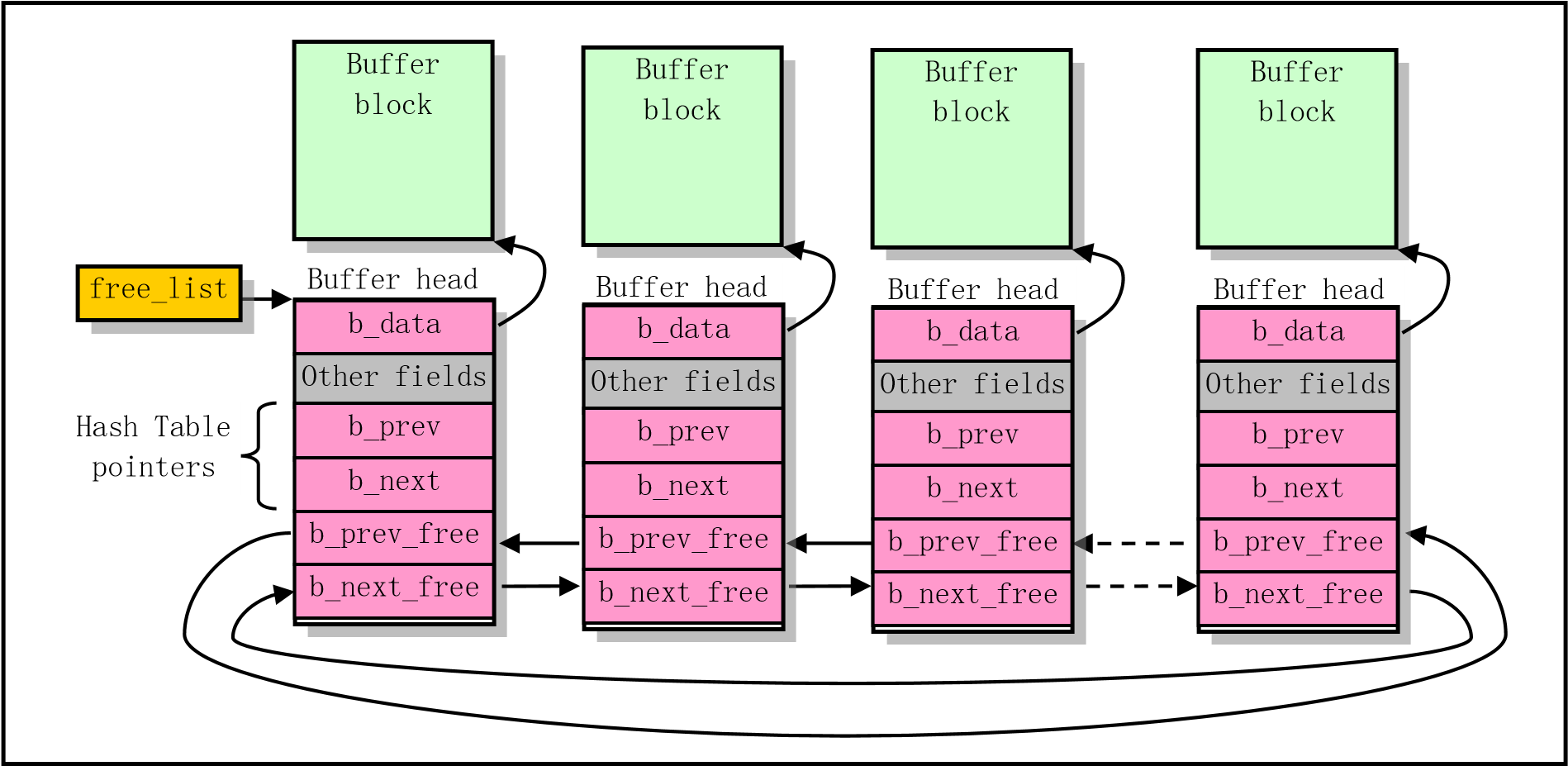


図12-17 全バッファブロックの双方向サーキュラーリンクされたフリーリスト

図のバッファヘッダ構造の「その他のフィールド」には、ブロックデバイス番号とバッファリングされたデータの論理ブロック番号があり、この2つのフィールドにより、バッファブロック内のデータに対応するブロックデバイスとそのデータが一意に決定されます。また、「データ有効（更新）フラグ」、「モディファイドフラグ」、「データの使用プロセス数」、「バッファブロックがロックされているかどうか」などのステータスフラグがあります。

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。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | bread, breada, bread\_page | |  | |  |  |  | | ( getblk ) | ブレス |  | | get\_hash\_table, find\_buffer等 | |  | | |

図12-18 バッファーマネジメント機能の階層的な関係

1. 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る。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は，リンクリスト上の同じ配列アイテムにリンクされることになる。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章の記述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を参照のこと。

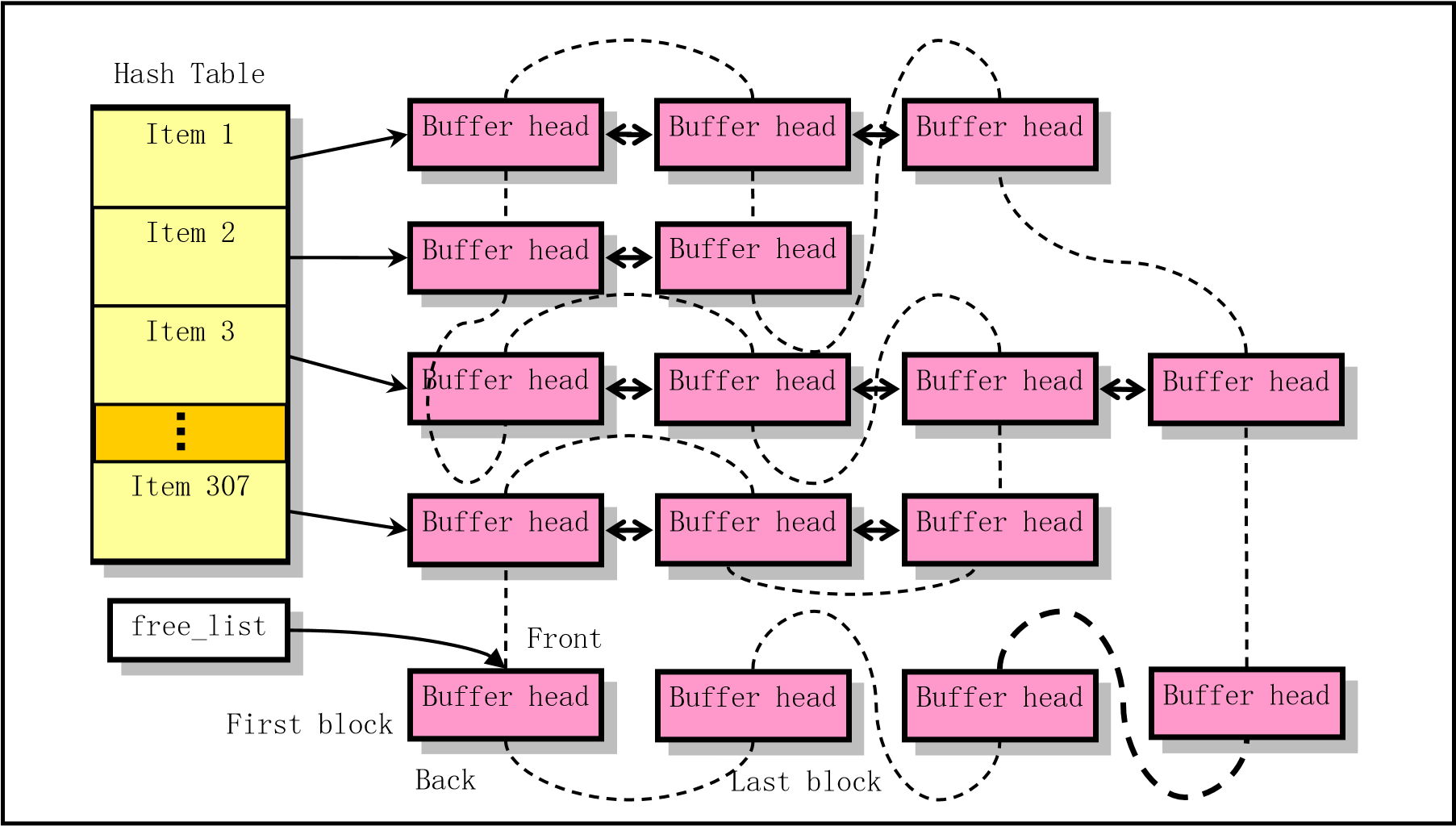


図12-19 ある瞬間のバッファブロックのハッシュキューの模式図

1. ここで、二重矢印の水平線は、同一ハッシュエントリ内のバッファブロックヘッダ構造間でハッシュ化された双方向リンクポインタを示しています。点線は、バッファキャッシュ内の全てのバッファブロックの双方向円形リンクリスト（いわゆるフリーリンクリスト）を示す。free\_listは、リストの中で最も空いているバッファブロックのヘッドポインタです。実際、この二重リンクリストは、LRU（Lest Recently Used）リンクリストである。以下では、バッファブロック検索関数getblk()について詳しく説明します。
2. 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。このとき、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。

つまり、我々が眠っている間に、他のプロセスが我々の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可能性があるので、最後にここでハッシュキューを検索する必要があります。もし、本当に必要なバッファブロックがハッシュキューにあった場合は、見つけ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ます。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()の処理全体を図12-20に示します。

* + 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。

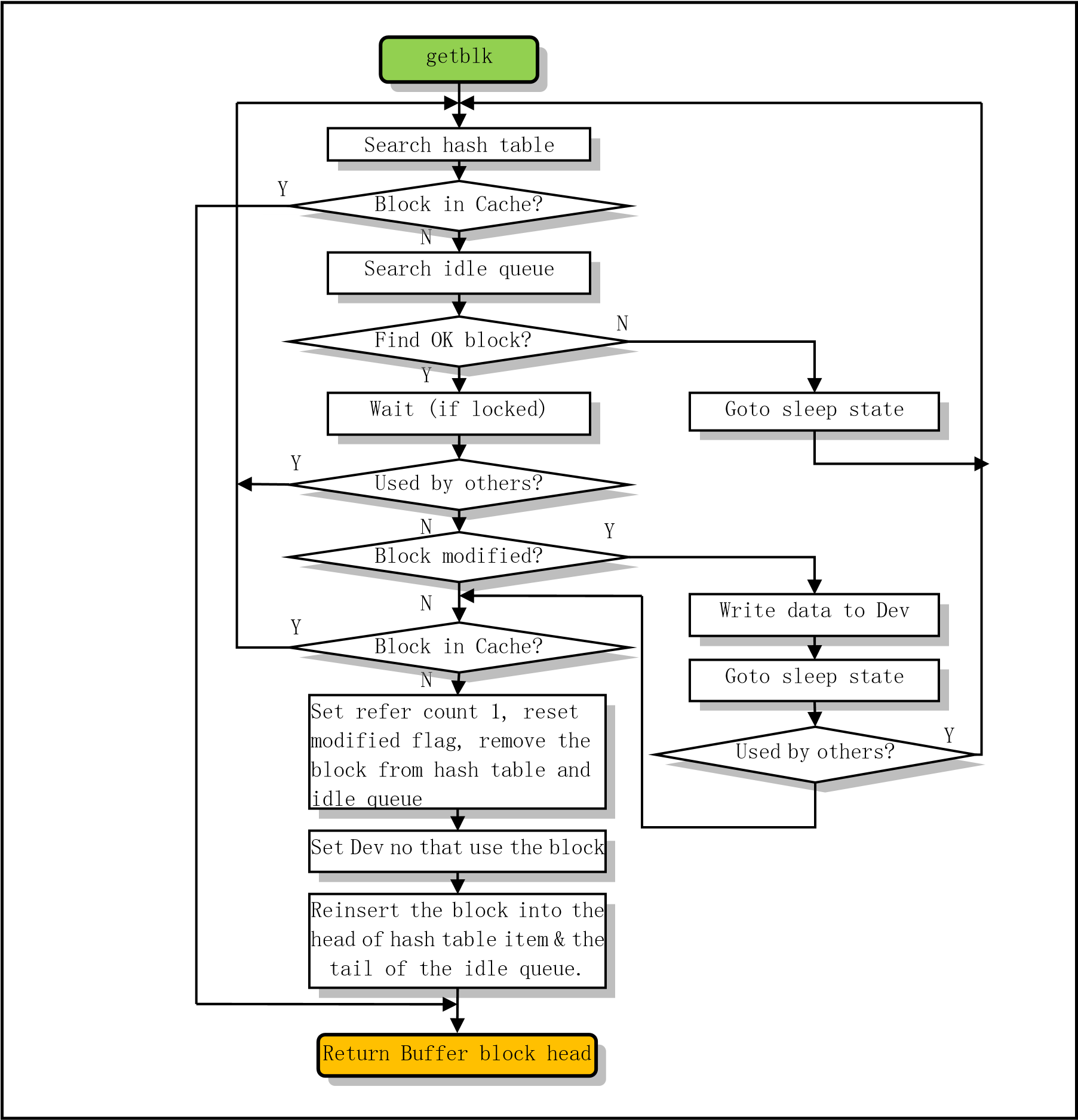


図12-20 Getblk()関数の実行フローチャート

1. 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。

N

Y

bread

get buffer block(getblk)

Data in block?

Call

low

-

level block

R/W

function ll\_rw\_block()

Enter sleep wait state

N

Y

Data

in block?

Release buffer block

Return NULL

Return buffer head

図 12-21 Bread()機能の実行フロー図

1. プログラムがバッファブロックのデータを使用する必要がなくなったときには、brelse()関数を呼び出してバッファブロックを解放し、バッファブロックを待っていたために眠っていたプロセスを起こすことができます。なお、フリーリストに登録されているバッファブロックはすべてフリーではありません。そのため、ディスクがリフレッシュされ、ロックが解除され、他のプロセスが参照していない（参照カウント＝0）場合にのみ使用することができます。
2. 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、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。

bread

out

/

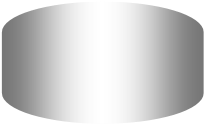
in

ll\_rw\_block

Buffer Cache

R/W

operation



Block Device

Kernel up code

Controller

Dev Driver

図12-22 カーネル・ブロック・デバイス・アクセス動作

* 1. 更新や同期の操作において、その主な機能は、メモリ上のいくつかのバッファブロックの内容を、ディスクなどのブロックデバイス上の情報と一致させることです。sync\_inodes()の主な機能は、inode\_tableのi-node情報をディスク上の情報と一致させることですが、システムバッファキャッシュの中間リンクを経由する必要があります。実際、あらゆる同期操作は2つのフェーズに分かれています。
  2. The data structure information is synchronized with the buffer block in the cache buffer and is independently responsible for the driver;
  3. 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。

### 12.2.2 Code annotation

プログラム 12-1 linux/fs/buffer.c

1. /\*
2. \* linux/fs/buffer.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/

6

1. /\*
2. \* 'buffer.c' implements the buffer-cache functions. Race-conditions have
3. \* been avoided by NEVER letting a interrupt change a buffer (except for the
4. \* data, of course), but instead letting the caller do it. NOTE! As interrupts 11 \* can wake up a caller, some cli-sti sequences are needed to check for 12 \* sleep-on-calls. These should be extremely quick, though (I hope).

13 \*/

14

1. /\*
2. \* NOTE! There is one discordant note here: checking floppies for
3. \* disk change. This is where it fits best, I think, as it should 18 \* invalidate changed floppy-disk-caches.

19 \*/

20

// <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。

//形式のマクロです。主に、1つの型（va\_list）と3つのマクロ（va\_start,

vsprintf, vprintf, vfprintf 関数の // va\_arg と va\_end）を使用しています。

// <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

// ディスクタイプ（HD\_TYPE）のオプションです。

// <linux/sched.h> スケジューラーのヘッダーファイルは、タスク構造体task\_struct、初期タスク0のデータ、およびいくつかの組み込みアセンブリ関数マクロステートメントを定義します。

// ディスクリプタのパラメータ設定と取得について。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <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。

// ディスクリプタ/割込みゲートなどを変更することが定義されています。

// <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。

// マクロの組み込みアセンブラの形式です。

21 #include <stdarg.h>.

22

1. #include <linux/config.h>
2. #include <linux/sched.h>
3. #include <linux/kernel.h>
4. #include <asm/system.h>
5. #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ファイルを指します。ここでは、バッファキャッシュがカーネルコードの最後から始まることを示すために使用されています。

// 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

1. // 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 // 待機キューに追加されます。b\_wait は、指定されたバッファブロック（すなわち、b\_wait に対応するバッファブロック）を待つタスク専用の待ち行列ヘッダです。
2. extern int end;
3. struct buffer\_head \* start\_buffer = (struct buffer\_head \*) &end;
4. struct buffer\_head \* hash\_table[NR\_HASH]; // NR\_HASH = 307
5. static struct buffer\_head \* free\_list; // free block link list head pointer.
6. 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

1. // ファイルを作成しました。Linus氏は、nr\_buffersがカーネルバッファの初期化後も変化しない // 「定数」であることを暗に示すようにコードを書いています。これは初期化関数 buffer\_init() (371行目)で // 設定されます。
2. 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

1. // 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ますが、プロセスが切り替わると CPU の現在の EFLAGS の値が変化します。sleep\_on()を使ってスリープ状態になったプロセスは、wake\_up()で明示的にウェイクアップする必要があります。 36 static inline void wait\_on\_buffer(struct buffer\_head \* bh)
2. {
3. // 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。
4. cli(); // disable int.
5. while (bh->b\_lock)
6. sleep\_on(&bh->b\_wait);
7. sti(); // enable int.
8. }

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。

1. // そして、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生成され、 // 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デバイスと // 同期します。
2. int sys\_sync(void)
3. {
4. int i;
5. struct buffer\_head \* bh;

48

1. 49 sync\_inodes(); /\* inodeをバッファに書き出す \*/ 50 bh = start\_buffer; // キャッシュの先頭を指す。
2. for (i=0 ; i<NR\_BUFFERS ; i++,bh++) {
3. wait\_on\_buffer(bh); // Wait for the buffer to be unlocked (if locked).
4. if (bh->b\_dirt)
5. ll\_rw\_block(WRITE,bh); // Generate a write device block request.
6. }
7. return 0;
8. }

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

1. // 指定されたデバイスデブのバッファのデータが変更された場合には、 // 同期操作を行う。その後，インメモリのi-nodeテーブルデータがバッファキャッシュに書き込まれる。その後，指定されたデバイス dev に対して，上記と同じ書き込み操作を // 再度行う。
2. int sync\_dev(int dev)
3. {
4. int i;
5. struct buffer\_head \* bh;

63

// まず，パラメータで指定されたデバイスに対してデータ同期操作を行い，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、キャッシュ内の全てのバッファブロックをスキャンし、指定されたデバイスdevのバッファブロックが // ロックされているかどうかを確認し、ロックされている場合は、スリープしてロックが解除されるのを待つ。

// 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

1. // の上に置かれています。スリープ中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ため、先に進む前に、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を再度確認する必要があります。
2. bh = start\_buffer; // points to the beginning of cache.
3. for (i=0 ; i<NR\_BUFFERS ; i++,bh++) {
4. if (bh->b\_dev != dev) // ignore blocks that doesn't belong to dev.
5. continue;
6. wait\_on\_buffer(bh); // wait for the buffer to be unlocked.
7. if (bh->b\_dev == dev && bh->b\_dirt)
8. ll\_rw\_block(WRITE,bh);
9. }

// 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 バッファ同期の最初のパスでは， // 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，同期操作のための

1. // i-nodeを効率的に実行することができます。この2回目のバッファ同期は、i-nodeの同期動作によってダーティになったバッファブロックを、デバイス内のデータと同期させます。
2. sync\_inodes();
3. bh = start\_buffer;
4. for (i=0 ; i<NR\_BUFFERS ; i++,bh++) {
5. if (bh->b\_dev != dev)
6. continue;
7. wait\_on\_buffer(bh);
8. if (bh->b\_dev == dev && bh->b\_dirt)
9. ll\_rw\_block(WRITE,bh);
10. }
11. return 0;
12. }

83

//// バッファキャッシュの指定されたデバイスのデータを無効にする。

1. // キャッシュ内のすべてのバッファブロックをスキャンする。指定されたデバイスのバッファブロックについて、 // その有効（更新）フラグと修正フラグをリセットします。
2. void inline invalidate\_buffers(int dev)
3. {
4. int i;
5. struct buffer\_head \* bh;

88

1. bh = start\_buffer;
2. for (i=0 ; i<NR\_BUFFERS ; i++,bh++) {
3. if (bh->b\_dev != dev) // Ignore blocks that doesn't belong to dev.
4. continue;
5. wait\_on\_buffer(bh); // wait for the buffer to be unlocked. // Since the process has performed a sleep wait, it is necessary to determine whether the buffer // still belongs to the specified device.
6. if (bh->b\_dev == dev)
7. bh->b\_uptodate = bh->b\_dirt = 0;
8. }
9. }

98

1. /\*
2. \* This routine checks whether a floppy has been changed, and
3. \* invalidates all buffer-cache-entries in that case. This
4. \* is a relatively slow routine, so we have to try to minimize using 103 \* it. Thus it is called only upon a 'mount' or 'open'. This
5. \* is the best way of combining speed and utility, I think.
6. \* People changing diskettes in the middle of an operation deserve
7. \* to loose :-)
8. \*
9. \* NOTE! Although currently this is only for floppies, the idea is
10. \* that any additional removable block-device will use this routine, 110 \* and that mount/open needn't know that floppies/whatever are 111 \* special. 112 \*/
11. //// ディスクが変更されたかどうかをチェックし、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)
12. {
13. int i; 116
14. // 最初に、それがフロッピーデバイスであるかどうかをチェックします。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、交換されていなければ終了します。関数 floppy\_change() は blk\_drv/floppy.c の 139 行目にあります。
15. if (MAJOR(dev) != 2)
16. return;
17. if (!floppy\_change(dev & 0x03))
18. return;
19. // フロッピーディスクが交換されたので、対応するデバイスのi-nodeビットマップと論理ブロックビットマップが占有していたバッファブロックが解放され、デバイスのi-nodeとデータブロック情報が占有していたキャッシュブロックが無効化されたことを示します。
20. for (i=0 ; i<NR\_SUPER ; i++)
21. if (super\_block[i].s\_dev == dev)
22. put\_super(super\_block[i].s\_dev);
23. invalidate\_inodes(dev);
24. invalidate\_buffers(dev);
25. } 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、要素を探す時間を短縮することです。要素の格納場所とキーワード（ハッシュ関数）の間に // 対応関係を確立することで、 // 指定された要素を関数の計算によって直接すぐに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、 // キーの値を計算する1つの方法に過ぎません。キーとなる値にモジュロ演算(%)を行うことで、その値が

1. 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．
2. #define \_hashfn(dev,block) (((unsigned)(dev^block))%NR\_HASH)
3. #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

1. // 双方向のサーキュラーリンクリスト構造です。
2. static inline void remove\_from\_queues(struct buffer\_head \* bh)
3. {
4. /\* remove from hash-queue \*/
5. if (bh->b\_next)
6. bh->b\_next->b\_prev = bh->b\_prev; 136 if (bh->b\_prev)

137 bh->b\_prev->b\_next = bh->b\_next;

1. // バッファブロックがキューの最初のブロックであった場合、 // ハッシュテーブルの対応するエントリはキューの次のバッファブロックを指します。138 if (hash(bh->b\_dev,bh->b\_blocknr) == bh)
2. hash(bh->b\_dev,bh->b\_blocknr) = bh->b\_next;
3. /\* remove from free list \*/
4. if (!(bh->b\_prev\_free) || !(bh->b\_next\_free))
5. panic("Free block list corrupted");
6. bh->b\_prev\_free->b\_next\_free = bh->b\_next\_free;
7. bh->b\_next\_free->b\_prev\_free = bh->b\_prev\_free;
8. // 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。
9. if (free\_list == bh)
10. free\_list = bh->b\_next\_free;
11. }

148

1. //// 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。
2. static inline void insert\_into\_queues(struct buffer\_head \* bh)
3. {
4. /\* put at end of free list \*/
5. bh->b\_next\_free = free\_list;
6. bh->b\_prev\_free = free\_list->b\_prev\_free;
7. free\_list->b\_prev\_free->b\_next\_free = bh;
8. free\_list->b\_prev\_free = bh;
9. /\* 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

1. // bh->b\_nextがNULLでない場合にのみ、b\_prevにbhの値を割り当てるようにします。つまり，163行目の // 前に，「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 // 0.96 バージョン以降に修正されました．
2. bh->b\_prev = NULL;
3. bh->b\_next = NULL;
4. if (!bh->b\_dev)
5. return;
6. bh->b\_next = hash(bh->b\_dev,bh->b\_blocknr);
7. hash(bh->b\_dev,bh->b\_blocknr) = bh;
8. bh->b\_next->b\_prev = bh; // "if (bh->b\_next)" should be added before this. 164 } 165
9. //// 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 // バッファキャッシュの中から探し出します．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．
10. static struct buffer\_head \* find\_buffer(int dev, int block)
11. {
12. struct buffer\_head \* tmp;

169

1. // 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。
2. for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b\_next)
3. if (tmp->b\_dev==dev && tmp->b\_blocknr==block)
4. return tmp;
5. return NULL;
6. }

175

1. /\*
2. \* Why like this, I hear you say... The reason is race-conditions.
3. \* As we don't lock buffers (unless we are readint them, that is), 179 \* something might happen to it while we sleep (ie a read-error
4. 180 \* は強制的に悪くなります）。これは現在、実際には起こるべきではありませんが、181 \* コードの準備はできています。
5. \*/

//// 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。

1. // 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、そのブロックはロックされ、 // ブロックヘッダポインタが返されます。
2. struct buffer\_head \* get\_hash\_table(int dev, int block) 184 {

185 struct buffer\_head \* bh;

186

1. 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。

1. // 見つからなかった場合は、NULLを返して終了します。
2. if (!(bh=find\_buffer(dev,block)))
3. return NULL;
4. // 目的のブロックが見つかった場合は、その参照カウントを1つ増やし、 // ブロックのロックが解除されるのを待ちます（ロックされていた場合）。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る。 // 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無効になり、再度検索が行われます。
5. bh->b\_count++;
6. wait\_on\_buffer(bh); // Wait for the buffer to be unlocked (if locked).
7. if (bh->b\_dev == dev && bh->b\_blocknr == block)
8. return bh;
9. bh->b\_count--;
10. }
11. }

197

1. /\*
2. \* Ok, this is getblk, and it isn't very clear, again to hinder
3. \* race-conditions. Most of the code is seldom used, (ie repeating), 201 \* so it should be much more efficient than it looks.
4. \*
5. \* The algoritm is changed: hopefully better, and an elusive bug removed.
6. \*/

// 以下のマクロは、ブロックのモディファイドフラグとロックフラグを同時にチェックするためのもので、モディファイドフラグの重みは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。205 #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，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る。

1. // 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。
2. struct buffer\_head \* getblk(int dev,int block)
3. {
4. struct buffer\_head \* tmp, \* bh; 209
5. repeat:
6. // ハッシュテーブルを検索し、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、 // 対応するブロックヘッドポインタを返して終了します。
7. if (bh = get\_hash\_table(dev,block))
8. return bh;
9. // 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'にフリーリストの // 最初のフリーブロックのヘッダを指させ，その後ループして後続の処理を行う。
10. tmp = free\_list; 214 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

// キャッシュの中では、必ずしもクリーン(b\_dirt=0)やアンロック(b\_lock=0)ではありません。そのため、 // まだ次のような判断や選択を続ける必要があります。例えば、次のようなブロックに対して

// 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

1. // タスクが複数のブロックを先読みするためにbreada()を実行すると、ll\_rw\_block()が呼ばれると同時にb\_countがデクリメントされます。しかし、このとき、 // ハードディスクのアクセス操作がまだ行われている可能性があるので、 // b\_lock=1であるが、b\_count=0である。
2. if (tmp->b\_count)
3. 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()

1. // = 0）であれば，指定されたデバイス上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， // このループを終了します．それ以外の場合は、このループを実行し続け、 // BADNESS()の値が最も小さいバッファを見つけられるかどうかを確認します。
2. if (!bh || BADNESS(tmp)<BADNESS(bh)) {
3. bh = tmp;
4. if (!BADNESS(tmp))
5. break;
6. }
7. /\* and repeat until we find something good \*/
8. } 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

1. // カウントが >0）、sleep は空きバッファブロックが利用可能になるのを待ちます。空きバッファブロックが利用可能になると、プロセスは明示的に // 起こされ、その後、空きバッファブロックを再度探すために // 関数の先頭にジャンプします。
2. if (!bh) {
3. sleep\_on(&buffer\_wait);
4. goto repeat; // jump to line 210.
5. }
6. // 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（ロックされていた場合）。スリープ期間中にバッファが他のタスクに使用された場合は、 // 上記の検索プロセスを再び繰り返さなければなりません。
7. wait\_on\_buffer(bh);
8. if (bh->b\_count) // occupied again ??
9. goto repeat;
10. // 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、再びバッファのロックが解除されるのを // 待ちます。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、上記の検索処理が // 繰り返されます。
11. while (bh->b\_dirt) {
12. sync\_dev(bh->b\_dev);
13. wait\_on\_buffer(bh);
14. if (bh->b\_count) // occupied again ??
15. goto repeat;
16. }
17. /\* NOTE!! While we slept waiting for this block, somebody else might \*/
18. /\* 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。

1. // そうであれば、上記の検索プロセスをもう一度繰り返します。
2. if (find\_buffer(dev,block))
3. goto repeat;
4. /\* OK, FINALLY we know that this buffer is the only one of it's kind, \*/
5. /\* 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。

1. // そして、有効（更新）フラグ。
2. bh->b\_count=1;
3. bh->b\_dirt=0;
4. 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

1. // 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。その後，バッファブロックは， // フリーリストとハッシュキューの新しい位置に再挿入され，最後に // バッファブロックヘッダが返される。
2. remove\_from\_queues(bh);
3. bh->b\_dev=dev;
4. bh->b\_blocknr=block;
5. insert\_into\_queues(bh);
6. return bh;
7. }

252

//// 指定したバッファブロックを解放します。

1. // 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、 // フリーの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されます。
2. void brelse(struct buffer\_head \* buf)
3. {
4. if (!buf) // Returns if the block head pointer is invalid.
5. return;
6. wait\_on\_buffer(buf);
7. if (!(buf->b\_count--))
8. panic("Trying to free free buffer");
9. wake\_up(&buffer\_wait);
10. }

262

1. /\*
2. \* bread() reads a specified block and returns the buffer that contains 265 \* it. It returns NULL if the block was unreadable.
3. \*/

//// 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。

1. // デバイスとブロック番号のブロックを指定します。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 // バッファブロックポインタを返します。そうでない場合は、指定されたデータブロックがデバイスから // バッファブロックに読み込まれ、バッファブロックポインタが返されます。
2. struct buffer\_head \* bread(int dev,int block)
3. {
4. 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

1. //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。バッファブロック内のデータが有効（更新されている）で、 // 直接使用できるかどうかを返します。
2. if (!(bh=getblk(dev,block)))
3. panic("bread: getblk returned NULL\n");
4. if (bh->b\_uptodate)
5. 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

1. // 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， // デバイスの読み込み操作に失敗したことを示すので，バッファを解放し，NULL を返して終了する。
2. ll\_rw\_block(READ,bh);
3. wait\_on\_buffer(bh);
4. if (bh->b\_uptodate)
5. return bh;
6. brelse(bh);
7. return NULL;
8. }

282

//// メモリブロックをコピーします。

1. // 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。
2. #define COPYBLK(from,to) \
3. \_\_asm\_\_("cld\n\t" \
4. "rep\n\t" \
5. "movsl\n\t" \
6. ::"c" (BLOCK\_SIZE/4),"S" (from),"D" (to) \
7. :"cx","di","si")

289

1. /\*
2. \* bread\_page reads four buffers into memory at the desired address. It's
3. \* a function of its own, as there is some speed to be got by reading them 293 \* all at the same time, not waiting for one to be read, and then another 294 \* etc. 295 \*/

//// 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。

1. // パラメーターの'address'はページデータが保存されるアドレス、'dev'は指定された // デバイス番号、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、mm/memory.cファイルのdo\_no\_page()関数（428行目）で // 使用されている。
2. void bread\_page(unsigned long address,int dev,int b[4])
3. {
4. struct buffer\_head \* bh[4];
5. int i; 300

// この関数は4回実行されます。配列b[]に入れられた4つのブロック番号に従って、 // デバイスdevからページが読み込まれ、指定されたメモリ位置'address'に置かれます。

// パラメータb[i]で与えられた有効なブロック番号に対して，この関数はまず，バッファ

// 指定されたデバイスとブロック番号のブロックをバッファキャッシュから削除します。のデータが消えてしまうと

// バッファブロックが無効（更新されていない）の場合は、デバイスの読み取り要求が行われて

// デバイスからの対応するデータブロック。無効なブロック番号b[i]については

1. // に対応しています。そのため，この関数は，指定されたb[]のブロック番号に従って， // 1〜4個のデータブロックを自由に読み取ることができる。
2. for (i=0 ; i<4 ; i++)
3. if (b[i]) { // If the block number is valid.
4. if (bh[i] = getblk(dev,b[i]))
5. if (!bh[i]->b\_uptodate)
6. ll\_rw\_block(READ,bh[i]);
7. } else
8. bh[i] = NULL;

// 4つのバッファブロックの内容を、指定されたアドレスに順次コピーします。

// バッファブロックをコピー（使用）する前に、スリープして待つ必要がある

1. // をロック解除する必要があります（ロックされている場合）。さらに，スリープしている可能性があるので， // コピーする前にバッファブロックのデータが有効かどうかをチェックする必要があります。また、コピー後にはバッファブロックを解放する必要があります。
2. for (i=0 ; i<4 ; i++,address += BLOCK\_SIZE)
3. if (bh[i]) {
4. wait\_on\_buffer(bh[i]); // Wait for block to unlock (if it is locked).
5. if (bh[i]->b\_uptodate) // If the data is valid, copy it.
6. COPYBLK((unsigned long) bh[i]->b\_data,address);
7. brelse(bh[i]); // Release the buffer block.
8. }
9. }

316

1. /\*
2. \* 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.
3. \*/

//// 指定したデバイスから指定したブロックの一部を読み出す。

1. // パラメータの数は可変で，指定されたブロック番号の連続である。この関数が成功すると， // 最初のブロックのブロックヘッダが返され，そうでなければ NULL が返されます。
2. struct buffer\_head \* breada(int dev,int first, ...)
3. {
4. va\_list args;
5. 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

1. // キャッシュから指定されたデバイスとブロック番号のブロックを読み出します。バッファブロックのデータが無効な場合（更新フラグが設定されていない場合）、デバイスデータブロックの読み込み要求が発行されます。
2. va\_start(args,first);
3. if (!(bh=getblk(dev,first)))
4. panic("bread: getblk returned NULL\n");
5. if (!bh->b\_uptodate)
6. 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

1. // は事前に読み込まれた後続のデータブロックであり、キャッシュに読み込まれるだけですぐには使用されないため、337行目のステートメントはブロックを解放するためにその参照カウントをデクリメントする必要がある（getblk()関数はブロックの参照カウントを増加させるため）。
2. while ((first=va\_arg(args,int))>=0) {
3. tmp=getblk(dev,first);
4. if (tmp) {
5. if (!tmp->b\_uptodate)
6. ll\_rw\_block(READA,bh); // here 'bh' should be 'tmp'.
7. tmp->b\_count--; // release the pre-read block.
8. } 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。その後、最初のバッファブロックのロックが解除されるのを // 待ちます（ロックされていた場合）。待ち時間が終わった後、 // 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。

1. // そうでなければ、バッファは解放され、NULLを返します。
2. va\_end(args);
3. wait\_on\_buffer(bh);
4. if (bh->b\_uptodate)
5. return bh;
6. brelse(bh);
7. return (NULL);
8. }

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

1. // は2MBに設定されます。この関数は，バッファキャッシュのメモリがすべて割り当てられるまで，ブロックヘッダ構造と対応するデータブロックを，それぞれバッファキャッシュの開始位置である start\_buffer とバッファの終了位置である buffer\_end // から設定（初期化）する。
2. void buffer\_init(long buffer\_end)
3. {
4. struct buffer\_head \* h = start\_buffer;
5. void \* b;
6. 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。

1. // ビデオメモリや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はハイエンドで640KBである必要があります。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よりも大きくなければなりません。
2. if (buffer\_end == 1<<20)
3. b = (void \*) (640\*1024);
4. else
5. 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

バッファの上端から1KBの大きさの // ブロックを取り出し、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た // buffer\_head を構築し、buffer\_head を // 二重にリンクされたリストに形成することができる。

// '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。

1. // hのブロックヘッダです。ブロックヘッダ構造を格納するのに十分なメモリがあることを保証するために、'b'が指すメモリブロックのアドレスは、'h'のブロックヘッダの終わりよりも大きいか // 等しくなければならない、つまり、'b >= h+1'が必要であるということです。
2. while ( (b -= BLOCK\_SIZE) >= ((void \*) (h+1)) ) { 359 h->b\_dev = 0; // device no.
3. h->b\_dirt = 0; // dirt flag (block modified flag).
4. h->b\_count = 0; // block reference count.
5. h->b\_lock = 0; // block lock flag.
6. h->b\_uptodate = 0; // Block update flag (or data valid flag). 364 h->b\_wait = NULL; // block waiting queue.
7. h->b\_next = NULL; // points to next header with same hash value.
8. h->b\_prev = NULL; // points to previous header with same hash value.
9. h->b\_data = (char \*) b; // block data pointer.
10. 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.
11. h++; // h points to next new buffer header location.
12. NR\_BUFFERS++; // counting buffer blocks.
13. // bが1MBにデクリメントされた場合、384KB（ビデオメモリ）をスキップし、bが0xA0000（640KB）を指すようにします。
14. if (b == (void \*) 0x100000)
15. b = (void \*) 0xA0000;
16. }

// 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

1. // 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，テーブル内のすべての // ポインタを NULL に設定します．
2. h--; // h points to the last block header.
3. free\_list = start\_buffer; // free list header points to first block header.
4. free\_list->b\_prev\_free = h;
5. h->b\_next\_free = free\_list;
6. for (i=0;i<NR\_HASH;i++)
7. hash\_table[i]=NULL;
8. }

382

## 12.3 bitmap.c

このプログラムを皮切りに、ファイルシステムの構成の2つ目の部分である、ファイルシステムの基礎となる運用機能の部分を探っていきます。この部分は、super.c、bitmap.c、truncate.c、inode.c、namei.cという5つのファイルで構成されています。

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。

### 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に説明します。

### 12.3.1 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します。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。見つからない場合は、ディスクデバイスの容量を使い果たし、0を返します。それ以外の場合は、最初に見つかった0ビットの位置が1に設定され、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。

### 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。

### 12.3.2 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
6. /\* 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は、タスク構造体task\_struct、初期タスク0のデータ、およびいくつかの組み込みアセンブリ関数マクロステートメントを定義します。

// ディスクリプタのパラメータ設定と取得について。

// <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。

1. // カーネルのよく使う機能
2. #include <string.h> // The memset() function is used here.

9

1. #include <linux/sched.h>
2. #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。

1. // 入力： eax = 0; ecx = ブロックの長さをロングワードで指定 (BLOCK\_SIZE/4); edi = // 開始アドレスを指定 addr.
2. #define clear\_block(addr) \
3. \_\_asm\_\_("cld\n\t" \ // clear direction.
4. "rep\n\t" \ // repeated execution of stored data (0).
5. "stosl" \
6. ::"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

効率的なアクセスと操作のために // EAX レジスタを使用します。このような変数の定義方法は、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あり、その値は // 最後の「res」の値である。

// 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。

1. // ベースアドレス(%3)とビットオフセット(%2)で // キャリーフラグ CF を設定し、 // ビットを 1 にします。 SETB 命令は、キャリーフラグ CF に従ってオペランド(%al)を設定するために使用されます。CF=1 の場合は %al =1、そうでない場合は %al =0 となります。
2. #define set\_bit(nr,addr) ({\
3. register int res \_\_asm\_\_("ax"); \
4. \_\_asm\_\_ \_\_volatile\_\_("btsl %2,%3\n\tsetb %%al": \
5. "=a" (res):"" (0),"r" (nr),"m" (\*(addr))); \
6. 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。

1. // 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。SETNB命令は、キャリーフラグCFに応じてオペランド（%al）を設定するために // 使用されます。CF = 1 の場合は %al = 0、そうでない場合は %al // = 1 となります。
2. #define clear\_bit(nr,addr) ({\
3. register int res \_\_asm\_\_("ax"); \
4. \_\_asm\_\_ \_\_volatile\_\_("btrl %2,%3\n\tsetnb %%al": \
5. "=a" (res):"" (0),"r" (nr),"m" (\*(addr))); \
6. 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， // 0である最初のビットを探し， // アドレスadrからのビットオフセットを返す．addr'はバッファブロックのデータ領域のアドレスで、 // スキャンの範囲は1024バイト（8192ビット）です。

1. // 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して、 // そのオフセットをEDXに配置するために使用されます。
2. #define find\_first\_zero(addr) ({ \
3. int \_\_res; \
4. \_\_asm\_\_("cld\n" \ // clear direction flag.
5. "1:\tlodsl\n\t" \ // obtain [esi] -> eax
6. "notl %%eax\n\t" \ // inverse each bits in eax
7. "bsfl %%eax,%%edx\n\t" \ // offset of first non-zero bit -> edx 37 "je 2f\n\t" \ // jump forward to label 2 if eax = 0.
8. "addl %%edx,%%ecx\n\t" \ // offset is added to ecx (first 0 bit in bitmap).
9. "jmp 3f\n" \ // jump forward to label 3 (end).
10. "2:\taddl $32,%%ecx\n\t" \ // ecx adds 32 bit offset if bit 0 not found.
11. "cmpl $8192,%%ecx\n\t" \ // scanned 8192 bits (1024 bytes)?
12. "jl 1b\n" \ // jump backward to label 1 if not yet.
13. "3:" \ // end. At this time, ecx contains 0 bit offset.
14. :"=c" (\_\_res):"c" (0),"S" (addr):"ax","dx","si"); \
15. \_\_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。

1. // 成功すれば1を、そうでなければ0を返します。
2. int free\_block(int dev, int block)
3. {
4. struct super\_block \* sb;
5. 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、データブロックの開始論理ブロック番号と、ファイルシステム内の論理ブロックの総数に基づいてチェックされます。もし、指定されたデバイス

// 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

1. // ディスクのデータゾーンの最初の論理ブロックのブロック番号が、 // デバイス上の論理ブロックの総数よりも大きい場合は、システムも停止します。
2. if (!(sb = get\_super(dev))) // in fs/super.c, line 56.
3. panic("trying to free block on nonexistent device");
4. if (block < sb->s\_firstdatazone || block >= sb->s\_nzones)
5. panic("trying to free block not in datazone"); 56 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。

1. // を1だけ増やして終了します。そうでなければ，modified フラグと updated フラグをクリアして，データブロックを // 解放します。このコードの主な目的は、ロジックブロックが現在バッファキャッシュに // 存在する場合、対応するバッファブロックが解放されることを検出することです。
2. if (bh) {
3. if (bh->b\_count > 1) {
4. brelse(bh); // ret after b\_count--, still used by others.
5. return 0;
6. }
7. bh->b\_dirt=0;
8. bh->b\_uptodate=0;
9. if (bh->b\_count) // if b\_count=1, call brelse().
10. brelse(bh);
11. }

// 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。

1. // そのため，'block / 8192'は論理ビットマップのどのブロックが指定されたブロックであるかを計算し， // 'block & 8191'は論理ブロックビットマップの現在のブロックにおけるブロックのビットオフセット位置を得ることができる。
2. block -= sb->s\_firstdatazone - 1 ; // block = block - (s\_firstdatazone -1);
3. if (clear\_bit(block&8191,sb->s\_zmap[block/8192]->b\_data)) {
4. printk("block (%04x:%d) ",dev,block+sb->s\_firstdatazone-1);
5. printk("free\_block: bit already cleared\n");
6. }
7. // 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。
8. sb->s\_zmap[block/8192]->b\_dirt = 1;
9. return 1;
10. }

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。実行が成功した場合，この関数は，論理ブロック番号（ディスクブロック

1. // 数）を、そうでなければ0を返します。
2. int new\_block(int dev)
3. {
4. struct buffer\_head \* bh;
5. struct super\_block \* sb;
6. 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。

1. // 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すべてのビットをスキャンして // (i >= 8 or j >= 8192)、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 // 無効(bh = NULL)であれば、0で終了する(空き論理ブロックがない)。
2. if (!(sb = get\_super(dev)))
3. panic("trying to get new block from nonexistant device");
4. j = 8192;
5. for (i=0 ; i<8 ; i++)
6. if (bh=sb->s\_zmap[i])
7. if ((j=find\_first\_zero(bh->b\_data))<8192)
8. break;
9. if (i>=8 || !bh || j>=8192)
10. 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

// つまり、論理ブロックビットマップのビットオフセット値は、データゾーンの先頭からのブロック番号を示しているので、データゾーンの最初のブロックの論理ブロック番号をここに追加して、jを論理ブロック番号に変換する必要があります。

1. // 新しい論理ブロック番号が、デバイス上の論理ブロックの総数よりも大きい場合、 // そのブロックはデバイス上に存在しません。アプリケーションは失敗したので、0を返して終了します。
2. if (set\_bit(j,bh->b\_data))
3. panic("new\_block: bit already set");
4. bh->b\_dirt = 1;
5. j += i\*8192 + sb->s\_firstdatazone-1;
6. if (j >= sb->s\_nzones)
7. 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、バッファブロックヘッダポインタを返します。なぜなら，論理ブロック

1. 取得したばかりの//は、参照カウントが1（getblk()で設定）でなければならず、1でなければ//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 // 設定されます。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。
2. if (!(bh=getblk(dev,j)))
3. panic("new\_block: cannot get block");
4. if (bh->b\_count != 1)
5. panic("new block: count is != 1");
6. clear\_block(bh->b\_data);
7. bh->b\_uptodate = 1;
8. bh->b\_dirt = 1;
9. brelse(bh);
10. return j;
11. }

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

1. // スーパーブロック情報を用いて // 操作された場合，i-node番号に対応するi-nodeビットマップのビットがリセットされ，i-node構造がクリアされる。
2. void free\_inode(struct m\_inode \* inode)
3. {
4. struct super\_block \* sb;
5. struct buffer\_head \* bh;

112

1. // まず、解放する必要のあるi-nodeの有効性や正当性を判断します。i-node pointer = 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の場合、そのノードは // 使用されていません。その後、対応するi-nodeが占有していたメモリ領域をクリアして戻ります。 // Memset()はファイルinclude/string.hの395行目で定義されています。これは、inodeポインタで指定されたメモリ領域が0で埋められ、sizeof(\*inode)バイトが埋められることを意味しています。
2. if (!inode)
3. return;
4. if (!inode->i\_dev) {
5. memset(inode,0,sizeof(\*inode));
6. 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。

1. // 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ことを意味します。ファイルリンクの数が0でない場合は， // ノードを使用している他のファイルディレクトリエントリがあることを意味するので， // 解放するのではなく，元に戻すべきである。
2. if (inode->i\_count>1) {
3. printk("trying to free inode with count=%d\n",inode->i\_count);
4. panic("free\_inode");
5. }
6. if (inode->i\_nlinks)
7. 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

1. i-node に対応する // ビットマップが存在しない場合はエラーになります。バッファブロックの i-node ビットマップは 8192 ビットなので、 // i\_num>>13 (すなわち i\_num/8192) は、現在の i-node 番号、つまりディスクブロックの s\_imap[] エントリを得ることができる。
2. if (!(sb = get\_super(inode->i\_dev)))
3. panic("trying to free inode on nonexistent device");
4. if (inode->i\_num < 1 || inode->i\_num > sb->s\_ninodes)
5. panic("trying to free inode 0 or nonexistant inode");
6. if (!(bh=sb->s\_imap[inode->i\_num>>13]))
7. 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

1. // が0になると，エラー警告メッセージが表示される。最後に，i-node // ビットマップが配置されているバッファが変更され，i-node 構造体が占めているメモリ領域が // クリアされる。
2. if (clear\_bit(inode->i\_num&8191,bh->b\_data))
3. printk("free\_inode: bit already cleared.\n\r");
4. bh->b\_dirt = 1;
5. memset(inode,0,sizeof(\*inode));
6. }

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。

1. // 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ビットマップからフリーのi-nodeを見つけます。
2. struct m\_inode \* new\_inode(int dev)
3. {
4. struct m\_inode \* inode;
5. struct super\_block \* sb;
6. struct buffer\_head \* bh;
7. int i,j; 143

// まず，メモリ内のi-nodeテーブル（inode\_table）から空いているi-nodeエントリを取得し，read

// 指定されたデバイスのスーパーブロック構造を調べます。次に、8ブロックのi-nodeビットマップをスキャンして

// スーパーブロックの最初の0ビットを探して（フリーノードを探して）、ノード番号を取得します。

// i-nodeが置かれる場所。スキャン後にi-nodeが見つからない場合，あるいは，バッファブロックが

1. // ビットマップが配置されている場所が無効（bh = NULL）であれば，以前に要求された // i-node テーブルの i-node が返され，NULL ポインタが返されて終了します（利用可能な Idle // i-node がないことを示す）。
2. if (!(inode=get\_empty\_inode())) // fs/inode.c, line 197.
3. return NULL;
4. if (!(sb = get\_super(dev))) // fs/super.c, line 56.
5. panic("new\_inode with unknown device");
6. j = 8192;
7. for (i=0 ; i<8 ; i++)
8. if (bh=sb->s\_imap[i])
9. if ((j=find\_first\_zero(bh->b\_data))<8192)
10. break;
11. if (!bh || j >= 8192 || j+i\*8192 > sb->s\_ninodes) {
12. iput(inode);
13. return NULL;
14. }

// 使用されていないi-node番号jが見つかったので、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

1. // 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モディファイドフラグを設定する。最後に，i-node構造体を初期化する（i\_ctimeは，i-nodeの内容が変更された時の時間）．
2. if (set\_bit(j,bh->b\_data))
3. panic("new\_inode: bit already set");
4. bh->b\_dirt = 1;
5. inode->i\_count=1; // reference count.
6. inode->i\_nlinks=1; // number of file directory entry links.
7. inode->i\_dev=dev; // device number the i-node is located.
8. inode->i\_uid=current->euid; // user id of the i-node. 164 inode->i\_gid=current->egid; // group id.
9. inode->i\_dirt=1; // set the modified flag.
10. inode->i\_num = j + i\*8192; // Corresponds to i-node nr in the device.
11. inode->i\_mtime = inode->i\_atime = inode->i\_ctime = CURRENT\_TIME;
12. return inode;
13. }

170

## 12.4 truncate.c

### 12.4.1 Function

truncate.cプログラムは、デバイス上の指定されたi-nodeが占有しているすべての論理ブロック（直接ブロック、一次間接ブロック、二次間接ブロックを含む）を解放するために使用されます。そのため，ファイルのノードに対応するファイル長が0にカットされ，占有していたデバイス空間が解放される。読みやすくするために，図12-23に示すように，i-nodeにおける直接ブロックと間接ブロックの組織構造の模式図を再度示す。

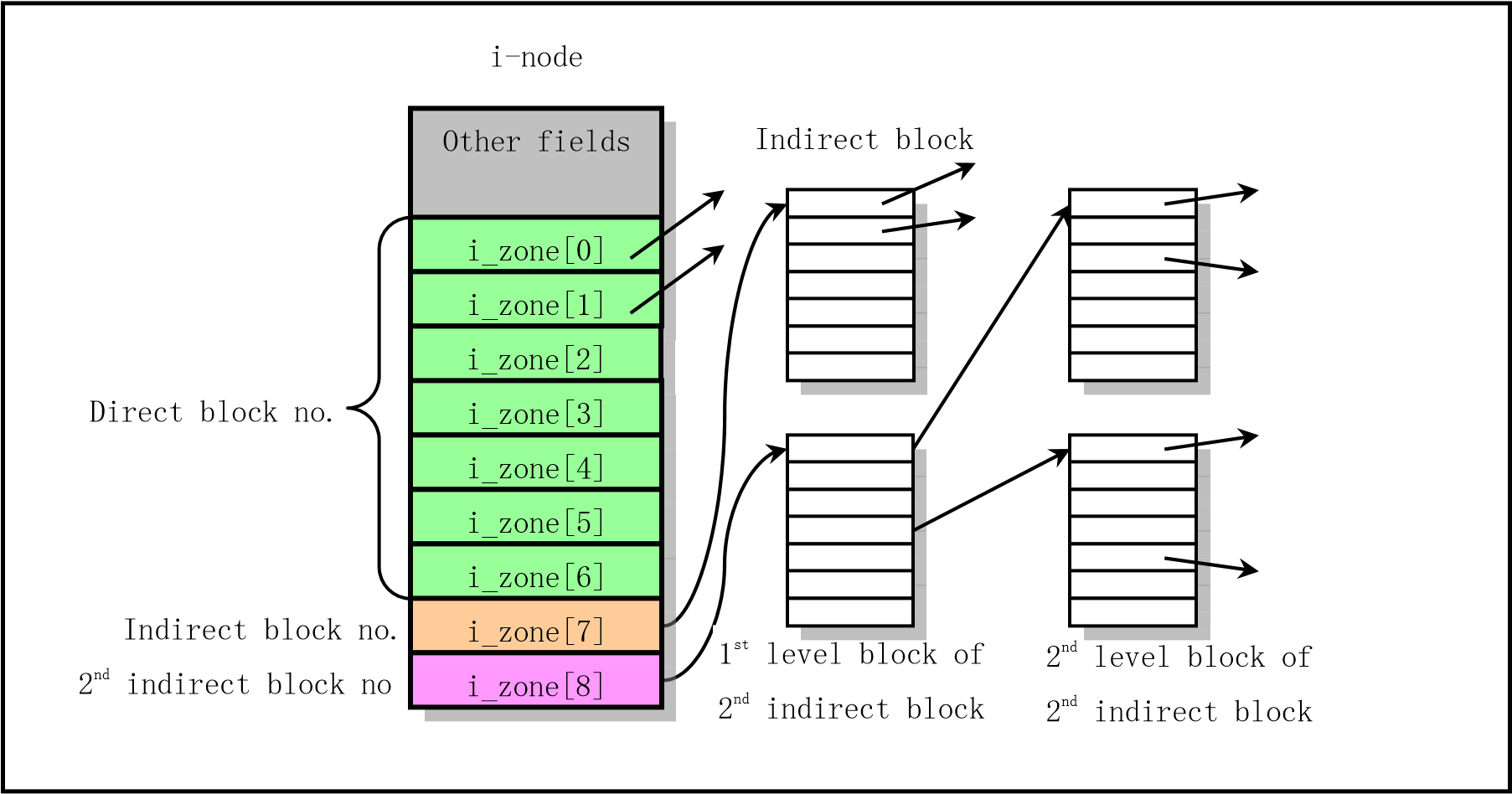


図12-23 i-nodeの論理ブロック接続の模式図

i-nodeのi\_zone[]配列には、デバイス上の論理ブロックのディスクブロック番号が格納されています。配列の最初の7項目（i\_zone[0]～i\_zone[6]）には、該当ファイルの最初の7つのデータブロックの番号が直接格納されています。 i\_zone[7]には、一次間接ブロックのブロック番号が格納されています。ディスクサイズが1024バイトであることから、各ディスクブロックには（1024÷2）＝512個のディスクブロック番号が格納され、1つの間接ブロック番号で最大512個のデバイスディスクブロックをアドレスすることができます。したがって，2次間接ブロック番号i\_zone[8]は，(512 \* 512) = 261, 144個のディスクブロックをアドレスすることができる。

### 12.4.2 Code annotation

プログラム 12-3 linux/fs/truncate.c

1. / 1 /\*
2. \* linux/fs/truncate.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/ 6

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <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 stat{} があります。 // と定数が含まれています。

7 #include <linux/sched.h>.

8

9 #include <sys/stat.h> (日本語)

10

//// 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。

1. // 成功すれば1を、そうでなければ0を返します。
2. static int free\_ind(int dev,int block)
3. {
4. struct buffer\_head \* bh;
5. unsigned short \* p;
6. int i;
7. 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

1. // この一次間接ブロックのバッファブロックを解放する。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 // 47行目)。
2. if (!block)
3. return 1;
4. block\_busy = 0;
5. if (bh=bread(dev,block)) {
6. p = (unsigned short \*) bh->b\_data; // points to the block data area.
7. for (i=0;i<512;i++,p++) // a block can contains 512 blcok nr.
8. if (\*p)
9. if (free\_block(dev,\*p)) {
10. \*p = 0;
11. bh->b\_dirt = 1; // set the modified flag. 28 } else
12. block\_busy = 1; // block is not released.
13. brelse(bh); // release the buffer block occupied by the indirect block.
14. }
15. // 最後にデバイス上の間接ブロックを解放します。ただし、解放されていない論理ブロックが // ある場合は、0（失敗）を返します。
16. if (block\_busy)
17. return 0;
18. else
19. 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。

1. // パラメータ dev はデバイス番号、'block' はセカンダリ // 間接ブロックの論理ブロック番号です。
2. static int free\_dind(int dev,int block)
3. {
4. struct buffer\_head \* bh;
5. unsigned short \* p;
6. int i;
7. 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。

1. // 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、使用済みであることを示すその上のすべての論理ブロックを解放し、さらに第1レベルのブロックのバッファブロックを解放します。
2. if (!block)
3. return 1;
4. block\_busy = 0;
5. if (bh=bread(dev,block)) {
6. p = (unsigned short \*) bh->b\_data; // points to the block data area.
7. for (i=0;i<512;i++,p++) // 512 second-level blocks
8. if (\*p)
9. if (free\_ind(dev,\*p)) { // release primary indirect block.
10. \*p = 0; 54 bh->b\_dirt = 1; // set modified flag.
11. } else
12. block\_busy = 1; // block not released.
13. brelse(bh); // release buffer block occupied by secondary indirect block.
14. }
15. // 最後に、デバイス上の2次間接ブロックを解放します。しかし、解放されていない論理ブロックが // ある場合は、0（失敗）を返します。
16. if (block\_busy)
17. return 0;
18. else
19. return free\_block(dev,block);
20. }

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

1. // をリリースしました。
2. void truncate(struct m\_inode \* inode)
3. {
4. int i;
5. int block\_busy; 69
6. // 最初に指定されたi-nodeの有効性を判断します。通常のファイル、 // ディレクトリ、またはリンクエントリでない場合に返します。次に、i-nodeの7つのダイレクトブロックを解放し、7つの論理ブロック項目をすべて0にします。ブロックがビジーで解放されていない場合は、block\_busyフラグが設定されます。
7. if (!(S\_ISREG(inode->i\_mode) || S\_ISDIR(inode->i\_mode) ||
8. S\_ISLNK(inode->i\_mode))) 72 return;
9. repeat:
10. block\_busy = 0;
11. for (i=0;i<7;i++)
12. if (inode->i\_zone[i]) { // release if block nr is not zero.
13. if (free\_block(inode->i\_dev,inode->i\_zone[i]))
14. inode->i\_zone[i]=0;
15. else
16. block\_busy = 1; // set if not released.
17. }
18. if (free\_ind(inode->i\_dev,inode->i\_zone[7])) // release primory indirect blocks.
19. inode->i\_zone[7] = 0;
20. else
21. block\_busy = 1; // set if not released.
22. if (free\_dind(inode->i\_dev,inode->i\_zone[8])) // release secondary indirect blocks
23. inode->i\_zone[8] = 0;
24. else
25. block\_busy = 1;

1. // その後、i-node変更フラグを設定し、「ビジー」のために解放されていない論理ブロックがまだある場合は、現在のプロセス実行時間スライスを0に設定して、他のプロセスに切り替えて実行し、しばらく待ってから解放操作を再実行します。 // 最後に、ファイルの変更時間とi-nodeの変更時間を現在の時間に設定します。 // 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、 // 1970:0:0:0からの秒数値を(startup\_time + jiffies/HZ)と定義しています。
2. inode->i\_dirt = 1;
3. if (block\_busy) {
4. current->counter = 0; // current process time slice is set to 0.
5. schedule();
6. goto repeat;
7. }
8. inode->i\_size = 0; // the file size is set to zero.
9. inode->i\_mtime = inode->i\_ctime = CURRENT\_TIME;
10. }

99

100

## 12.5 inode.c

### 12.5.1 Function

inode.cプログラムには、i-nodeを処理する関数iget()、iput()、ブロックマッピング関数bmap()のほか、その他の補助関数が含まれている。iget()、iput()、bmap()の各関数は、主にnamei.cプログラムのnamei()マッピング関数で使用され、ファイルパス名から対応するi-nodeを探し出す。

1. iget()関数

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、その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。

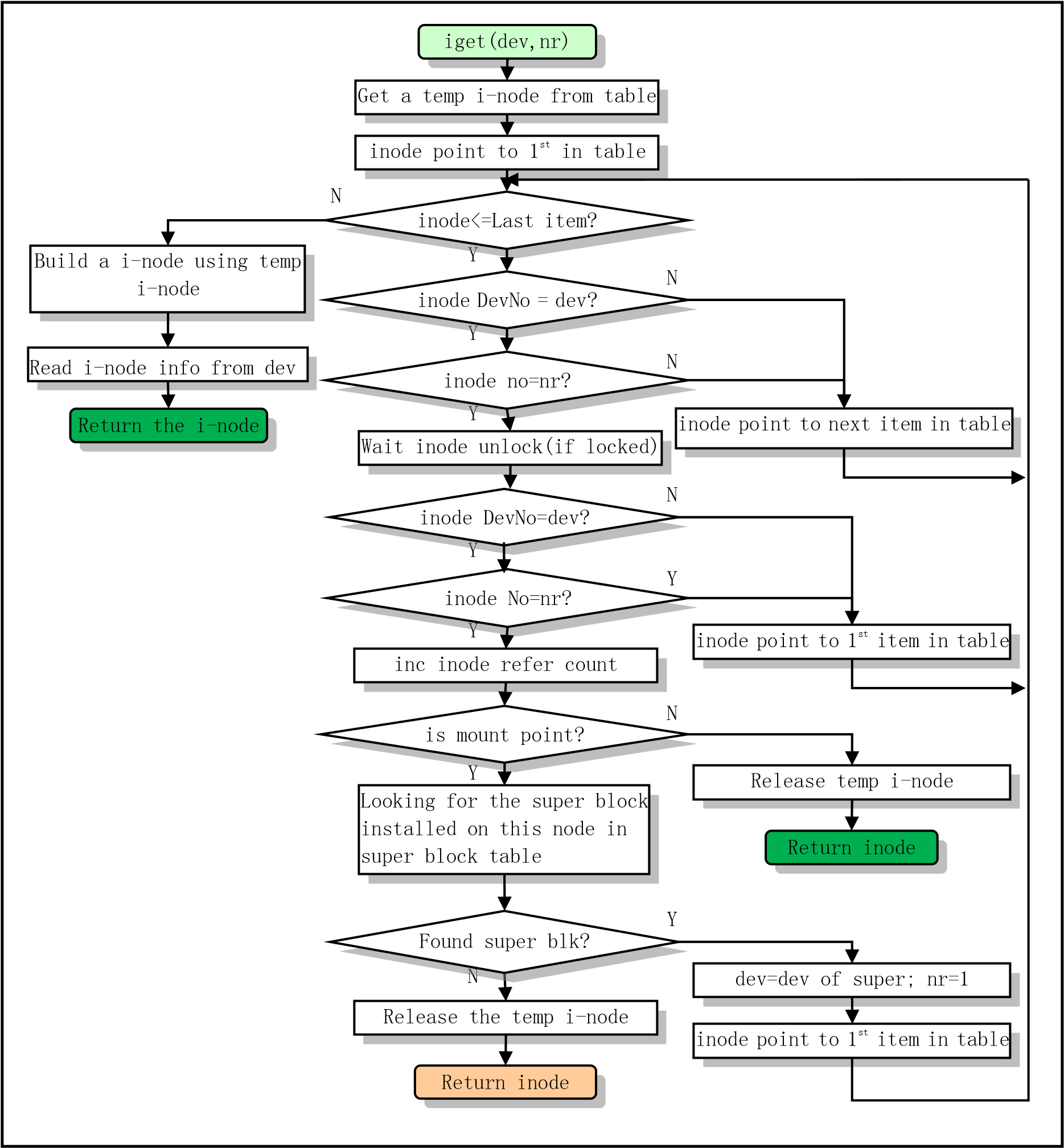


図12-24 iget()関数の動作フローチャート

i-nodeがファイルシステムのマウントポイントである場合は、スーパーブロックテーブルを検索し、i-nodeにインストールされているスーパーブロックを探す。対応するスーパーブロックが見つからない場合は、エラーメッセージを表示し、関数が取得を開始したフリーノードを解放して、i-nodeポインタを返します。対応するスーパーブロックが見つかった場合は、i-nodeをディスクに書き込む。次に、i-nodeのファイルシステムにインストールされているスーパーブロックからデバイス番号を取得し、i-node番号を1とする。その後、i-nodeテーブル全体を再度スキャンして、マウントされたファイルシステムのルートノードを取り出す。

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はない場合、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の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、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情報を読み込んで、i-nodeポインタを返します。

2. iput()関数

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。

i-node reference count field i\_countの値を1つ増やす。

namei(), dir\_namei(), open\_namei()関数を呼び出した。

iget()、new\_inode()、get\_empty\_inode()のいずれかの関数を呼び出した。

ファイルを閉じるとき、他のプロセスがそのファイルを使用していない場合。

ファイルシステムをアンマウントする場合（デバイスファイルのi-nodeの交換など）。

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。

3. bmap()関数

\_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、一次間接ブロック項目i\_zone[7]を使用する必要があり、ファイルが大きい場合は、二次間接ブロック項目i\_zone[8]が必要となる。したがって、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 stat{} // と定数が含まれています。 // 定数が含まれています。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <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

// 関数のプロトタイプをリリースします。

// <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。

1. // のディスクリプター/割り込みゲートなどが定義されています。
2. #include <string.h>
3. #include <sys/stat.h>

9

1. #include <linux/sched.h>
2. #include <linux/kernel.h>
3. #include <linux/mm.h>
4. #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

// ブロックアレイの総数は、サブデバイス番号で決まるサブデバイスが所有するデータブロックの総数に対応しています。

15 extern int \*blk\_size[];

16

17 struct m\_inode inode\_table[NR\_INODE]={{0,},}; // インメモリのi-nodeテーブル(NR\_INODE=32)。

18

1. static void read\_inode(struct m\_inode \* inode); // read i-node information, line 297.
2. 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。

1. // i-nodeがロックされている場合、i-nodeがロックを解除して明示的にタスクを起こすまで、現在のタスクは中断できない待機状態になり、 // i-waitキューに追加されます。
2. static inline void wait\_on\_inode(struct m\_inode \* inode)
3. {
4. cli();
5. while (inode->i\_lock)
6. sleep\_on(&inode->i\_wait); // kernel/sched.c, line 199.
7. sti();
8. }

29

//// i-nodeをロックする（与えられたi-nodeをロックする）。

1. // i-nodeがロックされている場合、現在のタスクは中断できない待機状態になり、 // i-waitキューi\_waitに追加されます。i-nodeがロックを解除して明示的にタスクを起こすまで、 // その後、ロックします。
2. static inline void lock\_inode(struct m\_inode \* inode)
3. {
4. cli();
5. while (inode->i\_lock)
6. sleep\_on(&inode->i\_wait);
7. inode->i\_lock=1; // set locked flag.
8. sti();
9. }

38

//// 指定された i ノードのロックを解除します。

1. // i-nodeのロックフラグをリセットして、i-waitキュー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 i\_wait.
2. static inline void unlock\_inode(struct m\_inode \* inode)
3. {
4. inode->i\_lock=0;
5. 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。

1. // メモリ上のi-nodeテーブル配列をスキャンし、指定されたデバイスでアイテムが使用されている場合は、 // リリースします。
2. void invalidate\_inodes(int dev)
3. {
4. int i;
5. 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

1. // が解放され、つまり、i-nodeのデバイス番号フィールドi\_devが0になり、モディファイドフラグもリセットされます。その間、まだ使用されているかどうか、つまり // その参照カウントが0ではないかどうかもチェックされ、そうであれば警告メッセージが表示されます。 // 50行目のポインタ代入文「0+inode\_table」は、「inode\_table」、 // 「&inode\_table[0]」と同等ですが、こちらの方がわかりやすいかもしれません。
2. inode = 0+inode\_table; // points to the first item of i-node table.
3. for(i=0 ; i<NR\_INODE ; i++,inode++) {
4. wait\_on\_inode(inode); // wait for i-node to be available(unlocked).
5. if (inode->i\_dev == dev) {
6. if (inode->i\_count) // if references is not 0, error warning.
7. printk("inode in use on removed disk\n\r"); 56 inode->i\_dev = inode->i\_dirt = 0; // release i-node.
8. }
9. }
10. }

60

//// すべてのi-nodeを同期させます。

1. // 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。
2. void sync\_inodes(void)
3. {
4. int i;
5. 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），次に i-node が変更されていて // パイプノードではないかどうかをチェックする。もしそうであれば，i-nodeはバッファキャッシュに書き込まれる。バッファマネージャプログラムは，適切なタイミングでディスクに書き込む。

1. 66 inode = 0+inode\_table; // テーブルの最初のアイテムを指します。 67 for(i=0 ; i<NR\_INODE ; i++,inode++) {。
2. wait\_on\_inode(inode); // wait for i-node to be available.
3. if (inode->i\_dirt && !inode->i\_pipe) // modified and not a pipe node.
4. write\_inode(inode); // write to the buffer cache.
5. }
6. }

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。

// デバイス上のファイルに対応する論理ブロック番号（ディスクブロック番号）と

1. // 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)直接ブロック処理、(3)一次間接ブロック処理、(4)二次間接ブロック処理。
2. static int \_bmap(struct m\_inode \* inode,int block,int create)
3. {
4. struct buffer\_head \* bh;
5. int i; 78
6. // (1) まず、パラメータの有効性を確認します。ファイルデータのブロック番号が // 0より小さい場合、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 + 2 番目の間接ブロック nr) より大きい場合は、ファイルシステムの範囲外です。
7. if (block<0)
8. panic("\_bmap: block<0");
9. if (block >= 7+512+512\*512)
10. 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。

1. // そして、ディスク上の論理ブロック番号を論理ブロックフィールドに記入します。その後、 // i-node change timeとmodified flagを設定し、最後に論理ブロック番号を返します。関数 // new\_block()は、bitmap.cプログラムの76行目で定義されています。
2. if (block<7) {
3. if (create && !inode->i\_zone[block])
4. if (inode->i\_zone[block]=new\_block(inode->i\_dev)) {
5. inode->i\_ctime=CURRENT\_TIME; // ctime - change time
6. inode->i\_dirt=1; // set modified flag.
7. }
8. return inode->i\_zone[block];
9. }

// (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

1. // 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の // ブロックは失敗し、0を返して終了します。
2. block -= 7;
3. if (block<512) {
4. if (create && !inode->i\_zone[7])
5. if (inode->i\_zone[7]=new\_block(inode->i\_dev)) {
6. inode->i\_dirt=1;
7. inode->i\_ctime=CURRENT\_TIME;
8. }
9. if (!inode->i\_zone[7])
10. 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

1. // 操作を行った場合、'i'はマッピング（検索）が必要な論理ブロック番号となります。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， // 新しいアプリケーションの論理ブロック番号またはディスク上の対応するブロックが返される。
2. if (!(bh = bread(inode->i\_dev,inode->i\_zone[7])))
3. return 0;
4. i = ((unsigned short \*) (bh->b\_data))[block];
5. if (create && !i)
6. if (i=new\_block(inode->i\_dev)) {
7. ((unsigned short \*) (bh->b\_data))[block]=i;
8. bh->b\_dirt=1;
9. }
10. brelse(bh);
11. return i;
12. }

// (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

1. // であれば、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。または、作成操作ではないが、i\_zone[8]が0であることが判明し、i-nodeに間接ブロックがないことを示すので、マッピングされたディスクブロックは失敗し、0を返して終了する。
2. block -= 512;
3. if (create && !inode->i\_zone[8])
4. if (inode->i\_zone[8]=new\_block(inode->i\_dev)) {
5. inode->i\_dirt=1;
6. inode->i\_ctime=CURRENT\_TIME;
7. }
8. if (!inode->i\_zone[8])
9. 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）の項目が等しいこと。

// 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

1. // セカンダリ間接ブロックの // この第一階層のブロックを解放する。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ある。
2. if (!(bh=bread(inode->i\_dev,inode->i\_zone[8])))
3. return 0;
4. i = ((unsigned short \*)bh->b\_data)[block>>9];
5. if (create && !i)
6. if (i=new\_block(inode->i\_dev)) {
7. ((unsigned short \*) (bh->b\_data))[block>>9]=i;
8. bh->b\_dirt=1;
9. }
10. 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

1. // を返します。それ以外の場合は、2次間接ブロックの第2レベルのブロックがデバイスから読み込まれ、第2レベルのブロックのブロックアイテムの論理ブロック番号が取得されます。
2. if (!i)
3. return 0;
4. if (!(bh=bread(inode->i\_dev,i)))
5. return 0;
6. 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，最終的にデータ情報を格納するブロックとしてディスクブロックを適用し， // 第2階層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」に一致させる。

1. // そして，第2レベルのブロックの修正フラグを設定する。最後に，二次間接ブロックの第2レベルのブロックが // 解放され，ディスク上で新たに適用された // または元の対応するブロックの論理ブロック番号が返される。
2. if (create && !i)
3. if (i=new\_block(inode->i\_dev)) {
4. ((unsigned short \*) (bh->b\_data))[block&511]=i;
5. bh->b\_dirt=1;
6. }
7. brelse(bh);
8. return i;
9. }

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 -ファイル内のデータブロック番号。 block -ファイル内のデータブロック番号 // 操作が成功した場合は，論理ブロック番号が返され，そうでない場合は 0 が返されます。

1. 142 int bmap(struct m\_inode \* inode,int block) 143 {。
2. return \_bmap(inode,block,0); // create flag = 0.
3. }

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。

1. // 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。 block -ファイル内のデータブロック番号 // 操作が成功した場合は，論理ブロック番号が返され，そうでない場合は 0 が返されます。
2. int create\_block(struct m\_inode \* inode, int block)
3. {
4. return \_bmap(inode,block,1);
5. }

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

1. // パイプのi-nodeは、待機中のプロセスを起こします。ブロックデバイスファイルのi-nodeであれば、デバイスを更新し、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を解放します。
2. void iput(struct m\_inode \* inode)
3. {

// 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

1. // はすでにアイドル状態になっています。カーネルはそれに対して put back 操作を要求し、カーネル内のコードに問題があることを // 示します。その後，エラーメッセージが表示され，マシンが // シャットダウンされる。
2. if (!inode)
3. return;
4. wait\_on\_inode(inode);
5. if (!inode->i\_count)
6. 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ウェイクアップし、参照数を1デクリメントし、まだ参照数が0でなければ、関数を返します。

1. // そうでない場合は，パイプラインが占有していたメモリページが解放され，ノードの参照カウント， // モディファイドフラグ，パイプフラグがリセットされた後，関数が返される。パイプの i ノードでは，inode->i\_size にメモリページの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。
2. if (inode->i\_pipe) {
3. wake\_up(&inode->i\_wait);
4. wake\_up(&inode->i\_wait2); //
5. if (--inode->i\_count)
6. return;
7. free\_page(inode->i\_size);
8. inode->i\_count=0;
9. inode->i\_dirt=0;
10. inode->i\_pipe=0;
11. 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

1. i-nodeであれば、 // 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、 // 論理ブロックフィールドの0（i\_zone[0]）がデバイス番号であるため、デバイスをリフレッシュして、 // i-nodeがロック解除するのを待ちます。
2. if (!inode->i\_dev) {
3. inode->i\_count--;
4. return;
5. }
6. if (S\_ISBLK(inode->i\_mode)) {
7. sync\_dev(inode->i\_zone[0]);
8. wait\_on\_inode(inode);
9. }

// 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

1. // i-nodeの参照カウント値は1です（157行目でチェックされていて0だったため）。i-nodeのリンク数が0の場合、対応するファイルが削除されたことを示します。その後、i-nodeのすべての論理ブロックが解放され、i-nodeが解放されます。 // 関数 free\_inode() は、実際に i-node の操作を解除するために使用されます。つまり、 // i-node のビットマップ・ビットをリセットし、i-node 構造体のコンテンツをクリアします。
2. repeat:
3. if (inode->i\_count>1) {
4. inode->i\_count--;
5. return;
6. }
7. if (!inode->i\_nlinks) { // at this point, i\_count = 1.
8. truncate(inode);
9. free\_inode(inode); // bitmap.c, line 108.
10. 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

1. // 解除する（ロックされている場合）。i-nodeの書き込み時にスリープすることがあるので、この時に他のプロセスがi-nodeを変更する可能性があり、上述のチェック手順（ラベルリピート）はプロセスがウェイクアップした後に // 再度繰り返す必要があります。
2. if (inode->i\_dirt) {
3. write\_inode(inode); /\* we can sleep - so do again \*/
4. wait\_on\_inode(inode);
5. goto repeat;
6. }

// 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

1. // が0ではなく，コンテンツが変更されていないことを示します。そのため、i-nodeの参照カウントが1だけデクリメントされる限り、 // 返すことができます。このとき、i\_count=0は、 // i-nodeが解放されたことを示しています。
2. inode->i\_count--;
3. return;
4. }

196

//// 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。

1. // 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、 // そのポインタを返します。これで、参照カウントは1になりました。
2. struct m\_inode \* get\_empty\_inode(void)
3. {
4. struct m\_inode \* inode;
5. static struct m\_inode \* last\_inode = inode\_table; // point to the first item.
6. int i; 202

// 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

// 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。last\_inodeが指すi-nodeの参照カウントが0であれば、それは

1. // 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。i-nodeのmodified // flagとlock flagが共に0であれば、そのi-nodeを使用することができるので、 // for()ループを終了します。
2. do {
3. inode = NULL;
4. for (i = NR\_INODE; i ; i--) { // NR\_INODE = 32。
5. if (++last\_inode >= inode\_table + NR\_INODE)
6. last\_inode = inode\_table;
7. if (!last\_inode->i\_count) {
8. inode = last\_inode;
9. if (!inode->i\_dirt && !inode->i\_lock)
10. break;
11. }
12. }
13. // 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テーブルを表示し、 // システムを停止します。
14. if (!inode) {
15. for (i=0 ; i<NR\_INODE ; i++)
16. printk("%04x: %6d\t",inode\_table[i].i\_dev,
17. inode\_table[i].i\_num);
18. panic("No free inodes in mem");
19. }
20. // 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 modified flag // がセットされている場合、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のロック解除を待つためには、再度サイクルする必要があります。
21. wait\_on\_inode(inode);
22. while (inode->i\_dirt) {
23. write\_inode(inode);
24. wait\_on\_inode(inode);
25. }
26. } 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）場合には

1. // アイドルi-nodeを再度検索する必要があります。そうでなければ，条件を満たす // アイドルi-nodeアイテムが見つかったことを示す。そして，i-node itemの内容をクリアし， // 参照カウントを1にして，i-nodeポインタを返します。
2. memset(inode,0,sizeof(\*inode));
3. inode->i\_count = 1;
4. return inode;
5. }

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。

1. // パイプラインが使用するためのものです。そして，取得したi-nodeの参照カウントを2（リーダ，ライタ）にし，パイプのヘッドとテールを初期化し，i-nodeのパイプタイプ表現を // 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。
2. struct m\_inode \* get\_pipe\_inode(void)
3. {
4. 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. // の1つです。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 // ページを指すようにします。空きメモリがない場合，i-nodeは解放され，NULLが返される。
2. if (!(inode = get\_empty\_inode()))
3. return NULL;
4. if (!(inode->i\_size=get\_free\_page())) { // i\_size points to a buffer page.
5. inode->i\_count = 0;
6. return NULL;
7. }

// 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。

1. // 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 // パイプヘッダとパイプテールポインタが格納されます。最後に，i-nodeフラグをパイプのi-nodeタイプになるように // 設定し，i-nodeを返します．
2. inode->i\_count = 2; /\* sum of readers/writers \*/
3. PIPE\_HEAD(\*inode) = PIPE\_TAIL(\*inode) = 0; // Reset the pipe head and tail.
4. inode->i\_pipe = 1; // pipe i-node.
5. return inode;
6. }

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。

// 指定されたi-node番号のi-node構造体のコンテンツ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

1. // 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，i-nodeがデバイスから読み込まれ， // i-nodeテーブルに置かれ，i-nodeポインタが返される。
2. struct m\_inode \* iget(int dev,int nr)
3. {
4. 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

// カーネルコードの問題で、エラーメッセージを表示して停止します。それ以外の場合は、バックアップのためにi-nodeテーブルからあらかじめアイドルのi-nodeを // 取っておく。その後、i-nodeテーブル全体をスキャンし、指定されたノード番号nrを持つi-nodeを // 探し、参照番号を1つインクリメントする。

1. // ただし，現在スキャンしている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， // スキャンを継続する。
2. if (!dev)
3. panic("iget with dev==0");
4. empty = get\_empty\_inode();
5. inode = inode\_table; // points to the first item of table.
6. while (inode < NR\_INODE+inode\_table) {
7. if (inode->i\_dev != dev || inode->i\_num != nr) {
8. inode++;
9. continue;
10. }

// 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

1. // 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、 // コードの実行を続ける際には、上記と同じチェックが再度必要となります。変更があった場合は、 // i-nodeテーブル全体が再度スキャンされます。260 wait\_on\_inode(inode);
2. if (inode->i\_dev != dev || inode->i\_num != nr) {
3. inode = inode\_table;
4. 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。

1. // このi-nodeにインストールされている。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、 // 関数の最初に取得したアイドル・ノード・エンプティが戻され、i-node // ポインタが返されます。
2. inode->i\_count++;
3. if (inode->i\_mount) {
4. int i;

268

1. for (i = 0 ; i<NR\_SUPER ; i++)
2. if (super\_block[i].s\_imount==inode)
3. break;
4. if (i >= NR\_SUPER) {
5. printk("Mounted inode hasn't got sb\n");
6. if (empty)
7. iput(empty);
8. return inode;
9. }

// 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

1. // 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 // デバイス番号を比較し、必要なi-node番号をROOT\_INO（1）に再設定します。
2. iput(inode);
3. dev = super\_block[i].s\_dev;
4. nr = ROOT\_INO;
5. inode = inode\_table;
6. continue;
7. }
8. // 最後に対応するi-nodeが見つかりました。したがって、この関数の最初に一時的に適用されたアイドルi-nodeを // 捨てて、見つかったi-nodeを返すことができます。
9. if (empty)
10. iput(empty);
11. return inode;
12. }
13. // テーブルに指定したi-nodeが見つからない場合、前のアプリケーションのアイドルi-node 'empty'を使用してテーブルにi-nodeが作成され、対応するデバイスからi-node情報が読み込まれ、i-nodeポインタが返されます。
14. if (!empty)
15. return (NULL);
16. inode=empty;
17. inode->i\_dev = dev; // set the device of the new i-node.
18. inode->i\_num = nr; // set the i-node number.
19. read\_inode(inode);
20. return inode;
21. }

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を最初に読み込んで、論理的な

// ブロック番号を算出します。i-nodeが配置されている論理ブロック番号を計算した後、 // 論理ブロックをバッファブロックに読み込んで、 // i-nodeのコンテンツを

1. // 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。
2. static void read\_inode(struct m\_inode \* inode)
3. {
4. struct super\_block \* sb;
5. struct buffer\_head \* bh;
6. int block;

302

1. // 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。
2. lock\_inode(inode);
3. if (!(sb=get\_super(inode->i\_dev)))
4. 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。

1. // 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、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み、i-nodeの内容をinodeポインタが示す場所に // コピーします。
2. block = 2 + sb->s\_imap\_blocks + sb->s\_zmap\_blocks +
3. (inode->i\_num-1)/INODES\_PER\_BLOCK;
4. if (!(bh=bread(inode->i\_dev,block)))
5. panic("unable to read i-node block");
6. \*(struct d\_inode \*)inode =
7. ((struct d\_inode \*)bh->b\_data)
8. [(inode->i\_num-1)%INODES\_PER\_BLOCK];
9. // 最後に、読み込んだバッファブロックを解放し、i-nodeをアンロックします。ブロックデバイス // ファイルの場合は、i-nodeにファイルの最大長を設定する必要もあります。
10. brelse(bh);
11. if (S\_ISBLK(inode->i\_mode)) {
12. int i = inode->i\_zone[0]; // i\_zone[0] contains dev nr for device file.
13. if (blk\_size[MAJOR(i)])
14. inode->i\_size = 1024\*blk\_size[MAJOR(i)][MINOR(i)];
15. else
16. inode->i\_size = 0x7fffffff;
17. }
18. unlock\_inode(inode);
19. }

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み， // キャッシュが更新されたときにディスクに書き込みます。i-nodeが配置されているデバイスの論理ブロック番号を // 決定するためには，まずデバイス上のスーパーブロックを読む必要がある。

1. // i-nodeが配置されている論理ブロック番号を計算した後、その論理ブロックをバッファブロックに読み込み、i-nodeの内容をバッファブロックの対応する位置にコピーします。
2. static void write\_inode(struct m\_inode \* inode)
3. {
4. struct super\_block \* sb;
5. struct buffer\_head \* bh;
6. 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

1. // 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。変更されていないi-nodeの場合、そのコンテンツはバッファやデバイス内のものと同じです。次に、i-nodeのスーパーブロックを // 取得します。
2. lock\_inode(inode);
3. if (!inode->i\_dirt || !inode->i\_dev) {
4. unlock\_inode(inode);
5. return;
6. }
7. if (!(sb=get\_super(inode->i\_dev)))
8. panic("trying to write inode without device");
9. // そして、デバイスからi-nodeが配置されている論理ブロックを読み込み、 // i-nodeの情報をi-nodeのアイテムの位置に対応する論理ブロックにコピーします。
10. block = 2 + sb->s\_imap\_blocks + sb->s\_zmap\_blocks +
11. (inode->i\_num-1)/INODES\_PER\_BLOCK;
12. if (!(bh=bread(inode->i\_dev,block)))
13. panic("unable to read i-node block"); 341 ((struct d\_inode \*)bh->b\_data)
14. [(inode->i\_num-1)%INODES\_PER\_BLOCK] =
15. \*(struct d\_inode \*)inode;
16. // その後、バッファモディファイドフラグを設定し、i-nodeのコンテンツは既にバッファと一致しているので、i-nodeダーティフラグは0に設定されます。最後に、i-nodeを含むバッファが // リリースされ、i-nodeがアンロックされます。
17. bh->b\_dirt=1;
18. inode->i\_dirt=0;
19. brelse(bh);
20. unlock\_inode(inode);
21. }

349

12.5.3 情報

## なし :)

## 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。

スーパーブロックには、主にファイルシステム全体の情報が格納されています。その構造については、図12-3の

セクション12.1「全体的な機能の説明」に記載されています。

get\_super()関数は、指定されたデバイスの条件下で、メモリのスーパーブロック配列の中から対応するスーパーブロックを検索し、そのポインタを返す関数である。したがって、この関数を呼び出すときには、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[]の対応する操作ブロック項目を解放する。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。

mount\_root

Disk i

-

node struct mus

t be 32 bytes

Initialize file table 64 items

If root

FS

is a floppy disk, prompt to

insert

t

he root disk

Init super block array

(8

ite

ms)

Read super block and root i

-

node

struct info on the root device

.

Set root i

-

node ref

count to 4,

use it

as current working dir

of process

1

and the i

-

node of the root directory.

Count nr of free disk blocks according

to the logic block bitmap

Count the nr of idle i

-

nodes according

to the i

-

node bitmap and display

.

Return

図12-25 mount\_root()関数の目的

この関数は、ルートファイルシステムのインストールに加え、ファイルシステムを使用するカーネルの初期化を行います。メモリ内スーパーブロック配列の初期化、ファイル記述子配列テーブル file\_table[]の初期化、ルートファイルシステムの空きディスクブロック数とアイドルi-nodeの表示を行います。

### 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。

### 12.6.2 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

1. /\*
2. \* super.c contains code to handle the super-block tables.
3. \*/

// <linux/config.h> カーネル設定用のヘッダーファイルです。キーボード言語とハードディスクタイプ（HD\_TYPE）のオプションを定義します。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <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。

// のディスクリプター/割り込みゲートなどが定義されています。

// <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が書かれています。 // <sys/stat.h> ファイルの状態を表す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 stat{} が含まれています。 // 定数を含みます。

1. 10 #include <linux/config.h> 11 #include <linux/sched.h>
2. #include <linux/kernel.h>
3. #include <asm/system.h>

14

1. #include <errno.h>
2. #include <sys/stat.h> 17
3. // 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)
4. int sync\_dev(int dev);
5. void wait\_for\_keypress(void); // kernel/chr\_drv/tty\_io.c, line 140.

20

21 /\* set\_bit は、ガスが setc を認識しないため、setb を使用します \*/ 。

//// 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、このマクロはビット設定をテストして返すだけで、ビットに変更を加えていません（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」の値である式です。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

1. // ビットオフセット bitnr (%2) をキャリーフラグ CF に入れる。SETB 命令はキャリーフラグ CF に従って // オペランド %al を設定するために使用されます。CF = 1 の場合は %al = 1、そうでない場合は %al = 0 となります。
2. #define set\_bit(bitnr,addr) ({ \
3. register int \_\_res \_\_asm\_\_("ax"); \
4. \_\_asm\_\_("bt %2,%3;setb %%al":"=a" (\_\_res):"a" (0),"r" (bitnr),"m" (\*(addr))); \
5. \_\_res; })

26

1. struct super\_block super\_block[NR\_SUPER]; // Super block table array (NR\_SUPER = 8).
2. /\* this is initialized in init/main.c \*/
3. 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

// スーパーブロックのロックが解除され、 // タスクが明示的に起動されるまで、s\_wait を行います。その後、ロックします。

1. // 次の3つの関数（lock\_super()、free\_super()、wait\_on\_super()）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、操作されるオブジェクトがスーパーブロックに置き換えられています。
2. static void lock\_super(struct super\_block \* sb)
3. {
4. cli(); // disable interrupt
5. while (sb->s\_lock) // sleep if super block is already locked.
6. sleep\_on(&(sb->s\_wait)); // kernel/sched.c, line 199.
7. sb->s\_lock = 1; // lock it!
8. sti(); // enable interrupt
9. }

39

//// 指定したスーパーブロックのロックを解除します。

1. // スーパーブロックのロックフラグをリセットし、この待ち行列 s\_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します（unlock\_super() という名前を使えば、より適切かもしれません）。
2. static void free\_super(struct super\_block \* sb)
3. {
4. cli();
5. sb->s\_lock = 0; // reset the lock.
6. wake\_up(&(sb->s\_wait)); // kernel/sched.c, line 188.
7. sti();
8. }

47

//// スーパーブロックが解除されるのを待つ眠り。

1. // スーパーブロックがロックされていた場合、現在のタスクは中断されない待機状態になり、 // スーパーブロックの待機キューs\_waitに追加されます。スーパーブロックがロックを解除し、 // 明示的にタスクを目覚めさせるまで。
2. static void wait\_on\_super(struct super\_block \* sb)
3. {
4. cli();
5. while (sb->s\_lock) // sleep if super block is already locked.
6. sleep\_on(&(sb->s\_wait));
7. sti();
8. }

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。

1. // スーパーブロックテーブル（配列）の中から，指定されたデバイスデブのスーパーブロック構造を検索します．見つかった場合はスーパーブロックへのポインタを返し、見つからない場合はNULLポインタを返します。
2. struct super\_block \* get\_super(int dev)
3. {
4. 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

1. // 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て、 // スーパーブロック配列全体の検索を開始し、 // 指定されたデバイスdevのスーパーブロックを見つけます。
2. if (!dev)
3. return NULL;
4. s = 0+super\_block;
5. 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

1. // 指定されたデバイスのスーパーブロックであるかどうかを， // スリープが戻った後に再度確認します．もしそうであれば，スーパーブロックのポインタを返すが，そうでなければ， // スーパーブロックの配列を再び検索するので，sはスーパーブロックの配列の先頭を再び指す必要がある．
2. if (s->s\_dev == dev) {
3. wait\_on\_super(s);
4. if (s->s\_dev == dev)
5. return s;
6. s = 0+super\_block;
7. // 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、指定されたスーパーブロックが // 見つからない場合は、NULLポインタが返されます。
8. } else
9. s++;
10. return NULL;
11. }

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

1. 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です。スーパーブロックに対応するファイルシステムがルートファイルシステムであったり， // そのiノードのいずれかに別のファイルシステムが既にインストールされていたりすると， // スーパーブロックを解放することはできない。
2. void put\_super(int dev)
3. {
4. struct super\_block \* sb;
5. 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

1. // 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 // 警告メッセージを表示して返します。ファイルシステムのアンマウント処理では、この関数を呼び出す前にs\_imountが // nullに設定される（192行目を参照）。
2. if (dev == ROOT\_DEV) {
3. printk("root diskette changed: prepare for armageddon\n\r");
4. return;
5. }
6. if (!(sb = get\_super(dev)))
7. return;
8. if (sb->s\_imount) {
9. printk("Mounted disk changed - tssk, tssk\n\r");
10. 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、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。

// 次に、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。

// 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

1. // が変更された場合、バッファブロックのデータをデバイスに書き込むための同期操作が必要になります。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。
2. lock\_super(sb);
3. sb->s\_dev = 0; // Set the super block to be idle.
4. for(i=0;i<I\_MAP\_SLOTS;i++)
5. brelse(sb->s\_imap[i]); 93 for(i=0;i<Z\_MAP\_SLOTS;i++)
6. brelse(sb->s\_zmap[i]);
7. free\_super(sb);
8. return;
9. }

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。

1. // 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。そうでなければ，スーパーブロックはデバイスデブから // バッファブロックに読み込まれ，スーパーブロックテーブルにコピーされる。最後に // スーパーブロックのポインタを返します。
2. static struct super\_block \* read\_super(int dev)
3. {
4. struct super\_block \* s;
5. struct buffer\_head \* bh;
6. int i,block; 104

// 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

1. // （つまり、フロッピーデバイスであるかどうか）を確認します。ディスクが交換されると、バッファキャッシュ内のすべてのバッファブロックが無効になるため、 // 無効化する必要がある、つまり、元々読み込まれていたファイルシステムが // 解放される。
2. if (!dev)
3. return NULL;
4. check\_disk\_change(dev);
5. // 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。そうでなければ、スーパーブロックの配列の中で空いているアイテム（つまり、 // フィールド s\_dev=0 のアイテム）を探します。配列が既に一杯の場合は、NULLポインタを返します。
6. if (s = get\_super(dev))
7. return s;
8. for (s = 0+super\_block ;; s++) {
9. if (s >= NR\_SUPER+super\_block)
10. return NULL;
11. if (!s->s\_dev)
12. break;
13. }
14. // スーパーブロック配列に空きエントリが見つかると，それを使ってデバイスdevのファイルシステムを // 指定する。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る。
15. s->s\_dev = dev;
16. s->s\_isup = NULL;
17. s->s\_imount = NULL;
18. s->s\_time = 0;
19. s->s\_rd\_only = 0;
20. 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

1. //の情報を公開しています。
2. lock\_super(s);
3. if (!(bh = bread(dev,1))) {
4. s->s\_dev=0;
5. free\_super(s);
6. return NULL;
7. }
8. \*((struct d\_super\_block \*) s) =
9. \*((struct d\_super\_block \*) bh->b\_data);
10. 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

1. // 上記で選択されたスーパーブロックの配列が解放され，そのアイテムのロックが解除され， // 空のポインタが返される。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 1.0 のみがサポートされており， // マジックナンバーは 0x137f である。
2. if (s->s\_magic != SUPER\_MAGIC) {
3. s->s\_dev = 0;
4. free\_super(s);
5. return NULL;
6. }

// 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

1. // スーパーブロックの対応するフィールドです。i-nodeビットマップは、デバイス上のブロック2から始まる論理ブロックに格納され、 // 合計s\_imap\_blocksブロックを占有する。論理ブロックのビットマップは，それ以降のブロックではs\_zmap\_blocksブロックを占有する。
2. for (i=0;i<I\_MAP\_SLOTS;i++) // initialize.
3. s->s\_imap[i] = NULL; 138 for (i=0;i<Z\_MAP\_SLOTS;i++)
4. s->s\_zmap[i] = NULL;
5. block=2;
6. for (i=0 ; i < s->s\_imap\_blocks ; i++) // read i-node bitmap in the device.
7. if (s->s\_imap[i]=bread(dev,block))
8. block++;
9. else
10. break;
11. for (i=0 ; i < s->s\_zmap\_blocks ; i++) // Read logic block bitmap in the device.
12. if (s->s\_zmap[i]=bread(dev,block))
13. block++;
14. else
15. 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

1. とoccupiedを解除することができます。つまり，i-nodeビットマップと // 論理ブロックビットマップによって占有されたキャッシュブロックが解除され， // 上記で選択されたスーパーブロックの配列項目が解除され， // スーパーブロックの項目がロック解除され，ヌルポインタが返されます。.
2. if (block != 2+s->s\_imap\_blocks+s->s\_zmap\_blocks) {
3. for(i=0;i<I\_MAP\_SLOTS;i++) // release buffer blocks used by bitmap.
4. brelse(s->s\_imap[i]);
5. for(i=0;i<Z\_MAP\_SLOTS;i++)
6. brelse(s->s\_zmap[i]);
7. s->s\_dev=0; // releases the selected super block item.
8. free\_super(s); // unlock the super block item.
9. return NULL;
10. }

// それ以外はすべてOKです。また、アイドル状態のi-nodeを要求する関数では、 // デバイス上のすべてのi-nodeが使用されている場合、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

1. 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 に設定されます．同じ理由で，論理ブロックビットマップの最下位ビットも // 1に設定されます． 最後の関数は，スーパーブロックのロックを解除し， // スーパーブロックポインタを返します．
2. s->s\_imap[0]->b\_data[0] |= 1;
3. s->s\_zmap[0]->b\_data[0] |= 1;
4. free\_super(s);
5. return s;
6. }

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はデバイス上のデータと // 同期します。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 // エラーコードを返します。

1. 166 int sys\_umount(char \* dev\_name) 167 {。
2. struct m\_inode \* inode;
3. struct super\_block \* sb;
4. 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

1. // fs/namei.cプログラムのシステムコールsys\_mknod()のうちの1つである。また、ファイルシステムはブロックデバイスに格納する必要があるため、ブロックデバイスのファイルでない場合は、先ほどのi-node dev\_iを戻して、エラーコードを返す。
2. if (!(inode=namei(dev\_name)))
3. return -ENOENT;
4. dev = inode->i\_zone[0];
5. if (!S\_ISBLK(inode->i\_mode)) {
6. iput(inode); // fs/inode.c, line 150.
7. return -ENOTBLK;
8. }

// 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、 // ここでi-nodeを戻します。次に、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

1. // マウントされている相手がフラグ i\_mount を設定していない場合は，警告メッセージを表示する。その後、 // 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、 // ある場合はビジーエラーコードを返します。
2. iput(inode);
3. if (dev==ROOT\_DEV)
4. return -EBUSY;
5. if (!(sb=get\_super(dev)) || !(sb->s\_imount))
6. return -ENOENT;
7. if (!sb->s\_imount->i\_mount)
8. printk("Mounted inode has i\_mount=0\n");
9. for (inode=inode\_table+0 ; inode<inode\_table+NR\_INODE ; inode++)
10. if (inode->i\_dev==dev && inode->i\_count)
11. 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。

1. // マウントされている場合は、i-nodeを解放し、スーパーブロックのマウントされたi-nodeフィールドをNULLにして、デバイスファイルシステムのルートi-nodeを戻します。その後，スーパーブロック内のマウントされたファイルシステムのルートiノードのポインタを // NULLに設定する。
2. sb->s\_imount->i\_mount=0;
3. iput(sb->s\_imount);
4. sb->s\_imount = NULL;
5. iput(sb->s\_isup);
6. sb->s\_isup = NULL;
7. // 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， // キャッシュとデバイス上のデータの間で同期操作を行い， // 0（アンマウント成功）を返します。
8. put\_super(dev);
9. sync\_dev(dev);
10. return 0;
11. }

198

//// マウント（新規）ファイルシステム system-call.

1. // パラメータdev\_nameはデバイスファイル名，dir\_nameはマウントするディレクトリ名， // rw\_flagはマウントするファイルシステムの読み取り/書き込みフラグである。マウントされる場所はディレクトリ名でなければならず， // 対応するi-nodeが他のプログラムによって占有されていてはならない。 // マウントに成功した場合は0を，そうでない場合はエラー番号を返す。
2. int sys\_mount(char \* dev\_name, char \* dir\_name, int rw\_flag)
3. {
4. struct m\_inode \* dev\_i, \* dir\_i;
5. struct super\_block \* sb;
6. int dev; 204

// まず、デバイスファイル名から対応するi-nodeを探して、デバイス番号を取得します。ブロックスペシャルデバイスファイルの場合，デバイス番号はそのi-nodeのi\_zone[0]にあります．

1. // また、ファイルシステムはブロックデバイス内になければならないので、そうでない場合は、先ほど取得したi-node dev\_iを戻して、エラーコードを返します。
2. if (!(dev\_i=namei(dev\_name)))
3. return -ENOENT;
4. dev = dev\_i->i\_zone[0];
5. if (!S\_ISBLK(dev\_i->i\_mode)) {
6. iput(dev\_i);
7. return -EPERM;
8. }

// 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

1. // を参照），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， // iノードを戻してエラーコードを返す。また，ノードがディレクトリファイルノードでない場合は， // 新しいファイルシステムはディレクトリ名にしかマウントできないので， // iノードも戻してエラーコードを返す。212 iput(dev\_i);
2. if (!(dir\_i=namei(dir\_name)))
3. return -ENOENT;
4. if (dir\_i->i\_count != 1 || dir\_i->i\_num == ROOT\_INO) {
5. iput(dir\_i);
6. return -EBUSY;
7. }
8. if (!S\_ISDIR(dir\_i->i\_mode)) { // mount point needs to be a directory entry.
9. iput(dir\_i);
10. return -EPERM;
11. }

// 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

1. // システムをチェックしています。新しいファイルシステムが他の場所でマウントされていないこと、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を確認したい、さもなければ // エラーが発生して返される。
2. if (!(sb=read\_super(dev))) {
3. iput(dir\_i);
4. return -EBUSY;
5. }
6. if (sb->s\_imount) { // if new fs has been mounted elsewhere.
7. iput(dir\_i);
8. return -EBUSY;
9. }
10. if (dir\_i->i\_mount) { // if the mount point is occupied.
11. iput(dir\_i);
12. return -EPERM;
13. }
14. // 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「s\_imount」に、マウント先のディレクトリのi-nodeを指すように設定し、マウントフラグとマウント先のi-nodeのノードモディファイドフラグを設定して、0を返します（インストールが成功した場合）。
15. sb->s\_imount=dir\_i;
16. dir\_i->i\_mount=1;
17. dir\_i->i\_dirt=1; /\* NOTE! we don't iput(dir\_i) \*/ 238 return 0; /\* we do that in umount \*/ 239 }

240

//// ルートファイルシステムをマウントします。

// この関数は、システムの初期化操作の一部です。この関数は，まずファイル

// file\_table[]テーブルとスーパーブロックテーブルを使用して、ルートファイルシステムのスーパーブロックを読み込みます。

1. // し、ルートiノードを取得します。最後に、ルートファイルシステム上の利用可能なリソース(空きブロック数と // アイドルiノード数)をカウントして表示します。この関数は、システムの初期化（ファイルblk\_drv/hd.cの157行目のsys\_setup()）で // 呼び出されます。
2. void mount\_root(void)
3. {
4. int i,free;
5. struct super\_block \* p;
6. struct m\_inode \* mi;

246

// まず、コードを修正したときに不整合が起きないように、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ます。次にファイルテーブルの初期化（64項目）

1. // とスーパーブロックテーブル(8項目)がある。ここで，すべてのファイル構造体の参照カウントは // 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（同じくアイドルを示す）に初期化される。ルートファイルシステムが置かれているデバイスが // フロッピーディスクの場合、「Insert root floppy and press ENTER」とプロンプトを出し、キーを待つ。
2. if (32 != sizeof (struct d\_inode))
3. panic("bad i-node size");
4. for(i=0;i<NR\_FILE;i++) // initialize file table (64 items).
5. file\_table[i].f\_count=0;
6. if (MAJOR(ROOT\_DEV) == 2) {
7. printk("Insert root floppy and press ENTER");
8. wait\_for\_keypress();
9. }
10. for(p = &super\_block[0] ; p < &super\_block[NR\_SUPER] ; p++) {
11. p->s\_dev = 0; // initialize super block table (8).
12. p->s\_lock = 0;
13. p->s\_wait = NULL;
14. }

// 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。

1. // その後，ルートデバイスからファイルシステムのスーパーブロックが読み込まれ，メモリのi-nodeテーブルのファイルシステムのルートi-node // （ノード1）へのポインタが取得される。デバイス上のスーパーブロックが故障したり，ルートノードが故障したりすると， // メッセージが表示され，デバイスがダウンする。
2. if (!(p=read\_super(ROOT\_DEV)))
3. panic("Unable to mount root");
4. if (!(mi=iget(ROOT\_DEV,ROOT\_INO))) // ROOT\_INO is defined as 1 in fs.h.
5. 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、 // ルートのi-nodeの参照番号を3回インクリメントします。i-nodeは次の数行のコードでも参照されるので、 // さらに、iget()関数の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

1. 現在のプロセスの // root i-node も設定されます。この時点では、カレントプロセスは、 // ナンバー1プロセス（initプロセス）です。
2. mi->i\_count += 3 ; /\* NOTE! it is logically used 4 times, not 1 \*/
3. p->s\_isup = p->s\_imount = mi;
4. current->pwd = mi;
5. 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の数を示す。まず、'i'をスーパーブロックで示されたデバイスの // 論理ブロックの総数とし、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 // 空きブロックの数をカウントする。

1. // 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ます。"i&8191 "は、現在のビットマップブロック内のiノード番号の対応するビットオフセット値を // 取得するために使用される。"i>>13" // iをディスクブロックに含まれるビット数である8192で割る。
2. free=0;
3. i=p->s\_nzones;
4. while (-- i >= 0)
5. if (!set\_bit(i&8191,p->s\_zmap[i>>13]->b\_data))
6. 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

1. // i-nodeビットマップの対応するビットの占有率に応じて、アイドルi-nodeの数を計算します。最後に、空いているi-nodeの数と、デバイス上で利用可能なi-nodeの // 合計数が表示されます。
2. printk("%d/%d free blocks\n\r",free,p->s\_nzones);
3. free=0;
4. i=p->s\_ninodes+1;
5. while (-- i >= 0)
6. if (!set\_bit(i&8191,p->s\_imap[i>>13]->b\_data))
7. free++;
8. printk("%d/%d free inodes\n\r",free,p->s\_ninodes);
9. }

281

## 12.7 namei.c

### 12.7.1 Function

namei.cプログラムは、Linux 0.12カーネルの中で最も長いファイルですが、コードの行数は約900行しかありません :)。このファイルは主に、ディレクトリ名やファイル名に応じて対応するi-nodeを見つける関数namei()と、ディレクトリの作成や削除、ディレクトリエントリの作成や削除に関するいくつかの操作関数やシステムコールを実装しています。

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。

36 #define NAME\_LEN 14 // ファイル名の最大長 36 #define NAME\_LEN 14 // ファイル名の最大長です。

37 #define ROOT\_INO 1 // ルートのi-nodeの番号。

1. // ファイルディレクトリのエントリ構造
2. struct dir\_entry {
3. unsigned short inode; // i-node number.
4. char name[NAME\_LEN]; // filename string 160 };

ファイルに関するその他の情報は、i-node番号で指定されたi-node構造に保存される。i-node構造には主に、ファイルのアクセス属性、ホスト、長さ、アクセスセーブタイム、ディスクブロックなどの情報が含まれる。各inode番号のi-nodeは、ディスク上の固定された場所に配置されている。

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番号を見つけ出し、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出す。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、それによってファイル「/usr」のi-node番号を取得するのである。i-node numberの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ます。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・エントリに従って親ディレクトリの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うことができる。 なお，プログラム中のいくつかの主要な機能については，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い方も明確であるため，ここでは説明を省略する。

### 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

1. /\*
2. \* Some corrections by tytso.
3. \*/ 10

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <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

// セグメント・レジスタ・オペレーションのための

// <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。ファイルとそのディスクリプターに使用される // 操作制御の定数記号の定義です。

// <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。

1. // <const.h> i-nodeのi\_modeフィールドのフラグを定義している定数シンボルファイルです。 // <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stat{}が含まれています。 // 定数を含む。
2. #include <linux/sched.h>
3. #include <linux/kernel.h>
4. #include <asm/segment.h>

14

1. #include <string.h>
2. #include <fcntl.h>
3. #include <errno.h>
4. #include <const.h>
5. #include <sys/stat.h>

20

1. // ファイル名で対応するi-nodeを探す内部関数です。
2. static struct m\_inode \* \_namei(const char \* filename, struct m\_inode \* base,
3. 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。したがって、表示できない文字（例えば、ASCIIコードが0x00--0x1fの範囲にある制御文字）は、そのASCIIコードの値で // 表現することができる。

//

// 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，このマクロは， // 読み取り可能性と書き込み可能性（rw）を示す006という8進数の値を返します。また，O\_ACCMODE = 00003 の場合は， // インデックス値 x のマスクです。

/ #define ACC\_MODE(x) ("˶‾᷄⌓‾᷅˵"[(x)&O\_ACCMODE])

25

1. /\*
2. \* comment out this line if you want names > NAME\_LEN chars to be 28 \* truncated. Else they will be disallowed.
3. \*/
4. /\* #define NO\_TRUNCATE \*/

31

1. #define MAY\_EXEC 1
2. #define MAY\_WRITE 2
3. #define MAY\_READ 4

35

1. /\*
2. \* permission()
3. \*
4. \* is used to check for read/write/execute permissions on a file.
5. \* I don't know if we should look at just the euid or both euid and 41 \* uid, but that should be easily changed.
6. \*/

//// 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．

1. // 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。
2. static int permission(struct m\_inode \* inode,int mask)
3. {
4. int mode = inode->i\_mode; // File access mode.

46

1. /\* 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

1. // プロセスがi-nodeのユーザIDと同じであれば、ファイル所有者のアクセス権が取られます。そうでない場合、プロセスの有効なグループID（egid）がi-nodeのグループIDと同じであれば、グループ・ユーザーのアクセス権が取られます。
2. if (inode->i\_dev && !inode->i\_nlinks) 49 return 0;
3. else if (current->euid==inode->i\_uid)
4. mode >>= 6;
5. else if (in\_group\_p(inode->i\_gid))
6. mode >>= 3;
7. // 最後に、アクセス権がマスクコードと同じか、スーパーユーザーであれば1を、そうでなければ0を返すように判断しています。
8. if (((mode & mask & 0007) == mask) || suser())
9. return 1;
10. return 0;
11. }

58

1. /\*
2. \* ok, we cannot use strncmp, as the name is not in our data space. 61 \* Thus we'll have to use match. No big problem. Match also makes 62 \* some sanity tests.
3. \*
4. \* NOTE! unlike strncmp, match returns 1 for success, 0 for failure.
5. \*/

//// 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。

1. // を返します。同じであれば1を、そうでなければ0を返します。
2. static int match(int len,const char \* name,struct dir\_entry \* de)
3. {
4. 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を超えているかのいずれかである。比較する長さが 0 であっても、 // ディレクトリエントリのファイル名が '.'であれば、同じと判断して、1（一致）を返します。 // 比較する長さ 'len' が NAME\_LEN より小さくても、ファイル名の長さが // len を超える場合も、0（一致しない）を返します。

1. // 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は， // name[len] が NULL かどうかを検出することです。長さが len を超えていれば， name[len] は NULL ではない通常の文字である // ということになる。長さが 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。
2. if (!de || !de->inode || len > NAME\_LEN)
3. return 0;
4. /\* "" means "." ---> so paths like "/usr/lib//libc.a" work \*/
5. if (!len && (de->name[0]=='.') && (de->name[1]=='\0'))
6. return 1;
7. if (len < NAME\_LEN && de->name[len]) // the length of the given name > len.
8. return 0;

// その後、埋め込みアセンブリステートメントを使用して、迅速な比較を行います。これは、ユーザーデータ空間（fsセグメント）で文字列の比較を行います。

// %0 - eax (比較結果、同じ); %1 - eax (eaxの初期値0); %2 - esi (名前)。

1. // %3 - edi (ディレクトリ名ポインタ); %4 - ecx (比較されたバイト) len)。
2. \_\_asm\_\_("cld\n\t" // clear direction.
3. "fs ; repe ; cmpsb\n\t" // compare [esi++] and [edi++] in user space.
4. "setz %%al" // set al = 1 if identical (same = eax).
5. :"=a" (same)
6. :"" (0),"S" ((long) name),"D" ((long) de->name),"c" (len)
7. :"cx","di","si");
8. return same; // return the result.
9. }

85

1. /\*
2. \* find\_entry()
3. \*
4. \* finds an entry in the specified directory with the wanted name. It
5. \* returns the cache buffer in which the entry was found, and the entry
6. \* 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.
7. \*
8. \* This also takes care of the few special cases due to '..'-traversal 95 \* over a pseudo-root and a mount point.
9. \*/

//// 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のうち、ファイル名が '...' である場合には、現在の関連する設定に従って // 特別な処理を行います。関数の引数におけるポインターの役割については、 // linux/sched.c の 151 行目以前のコメントを参照してください。

1. // 戻り値：成功した場合はキャッシュブロックポインタ、 \*res\_dir で返されたディレクトリエントリポインタ // を返します。失敗した場合は null ポインタを返します。
2. static struct buffer\_head \* find\_entry(struct m\_inode \*\* dir,
3. const char \* name, int namelen, struct dir\_entry \*\* res\_dir)
4. {
5. int entries;
6. int block,i;
7. 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

1. // パラメータを指定します。上記30行目でシンボル定数NO\_TRUNCATEを定義した場合、ファイル名の長さが最大長NAME\_LENを超えた場合、 // 処理されません。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NAME\_LENを超えると、 // 切り捨てられます。
2. #ifdef NO\_TRUNCATE
3. if (namelen > NAME\_LEN)
4. return NULL;
5. #else
6. if (namelen > NAME\_LEN)
7. namelen = NAME\_LEN;
8. #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

1. // 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るので、 // それをディレクトリエントリのサイズ（16バイト）で割って、ディレクトリエントリの数を求めます。 // 次に，返すべきディレクトリエントリへのポインタをnullします。
2. entries = (\*dir)->i\_size / (sizeof (struct dir\_entry)); 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 が関数パラメータで与えられたのと同じディレクトリである場合、 // このプロセスにとって、このディレクトリは擬似的なルートディレクトリであることを意味します。 // つまり、このプロセスにとって、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たということです。そこで // 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。

1. // コラム」プロジェクト :)
2. /\* check for '..', as we might have to do some "magic" for it \*/
3. if (namelen==2 && get\_fs\_byte(name)=='.' && get\_fs\_byte(name+1)=='.') {
4. /\* '..' in a pseudo-root results in a faked '.' (just change namelen) \*/
5. if ((\*dir) == current->root)
6. namelen=1;
7. else if ((\*dir)->i\_num == ROOT\_INO) {
8. /\* '..' over a mount-point results in 'dir' being exchanged for the mounted
9. directory-inode. NOTE! We set mounted, so that we can iput the new dir \*/
10. sb=get\_super((\*dir)->i\_dev);
11. if (sb->s\_imount) { // mounted point.
12. iput(\*dir);
13. (\*dir)=sb->s\_imount;
14. (\*dir)->i\_count++;
15. }
16. }
17. }

// 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。

1. // したがって，ディレクトリのデータを読む，つまり，ディレクトリのi-nodeに対応するブロックデバイスのデータゾーンにあるデータブロック（論理ブロック）を取り出す必要があります。これらの論理ブロックのブロック番号は， // i-node構造体のi\_zone[]配列に格納されている． // まず、そこに保存されている最初のダイレクトブロック番号を取り出し、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読み出します。
2. if (!(block = (\*dir)->i\_zone[0]))
3. return NULL;
4. if (!(bh = bread((\*dir)->i\_dev,block)))
5. 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

1. // データブロックを指定します。まず'de'にバッファブロックのデータブロック部分を指させ、ディレクトリのエントリ数を超えないように // 検索をループさせます。ここで，iはディレクトリエントリの // インデックスである。
2. i = 0;
3. de = (struct dir\_entry \*) bh->b\_data;
4. 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。

1. を // 入れることができます。141行目のi/DIR\_ENTRIES\_PER\_BLOCKでは、ディレクトリアイテムが現在検索されているディレクトリファイル内のデータブロック番号を取得することができ、bmap()関数（inode.c、142行目）では、デバイス上の対応する論理ブロック番号を算出することができます。
2. if ((char \*)de >= BLOCK\_SIZE+bh->b\_data) {
3. brelse(bh);
4. bh = NULL;
5. if (!(block = bmap(\*dir,i/DIR\_ENTRIES\_PER\_BLOCK)) ||
6. !(bh = bread((\*dir)->i\_dev,block))) {
7. i += DIR\_ENTRIES\_PER\_BLOCK;
8. continue;
9. }
10. de = (struct dir\_entry \*) bh->b\_data;
11. }

// 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

1. // 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・データ・ブロック・ポインタ 'bh' が // 返され、この関数は終了します。それ以外の場合は，ディレクトリ・エントリ・データ・ブロック内の次のディレクトリ・エントリの // 比較を続ける。
2. if (match(namelen,name,de)) {
3. \*res\_dir = de;
4. return bh;
5. }
6. de++;
7. i++;
8. }
9. // 指定されたディレクトリのすべてのディレクトリエントリを検索しても // 対応するディレクトリエントリが見つからない場合は、ディレクトリのデータブロックが解放され、 // 最後にNULLが返されます（失敗）。
10. brelse(bh);
11. return NULL;
12. }

158

1. /\*
2. \* add\_entry()
3. \*
4. \* adds a file entry to the specified directory, using the same 163 \* semantics as find\_entry(). It returns NULL if it failed.
5. \*
6. \* 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.
7. \*/

//// 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．

1. // 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。
2. static struct buffer\_head \* add\_entry(struct m\_inode \* dir,
3. const char \* name, int namelen, struct dir\_entry \*\* res\_dir)
4. {
5. int block,i;
6. struct buffer\_head \* bh;
7. 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

1. // パラメータを指定します。上記30行目でシンボル定数NO\_TRUNCATEを定義した場合、ファイル名の長さが最大長NAME\_LENを超えた場合、 // 処理されません。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NAME\_LENを超えると、 // 切り捨てられます。
2. \*res\_dir = NULL; // the result directory entry pointer.
3. #ifdef NO\_TRUNCATE
4. if (namelen > NAME\_LEN)
5. return NULL;
6. #else
7. if (namelen > NAME\_LEN)
8. namelen = NAME\_LEN;
9. #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。

1. // i-nodeの構造です。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、 // 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読み込みます。また，パラメータで指定されたファイル名の長さが // 0になった場合は，NULLを返します。
2. if (!namelen)
3. return NULL;
4. if (!(block = dir->i\_zone[0]))
5. return NULL;
6. if (!(bh = bread(dir->i\_dev,block)))
7. 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

1. // 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は、バッファブロックのデータ // ブロック部分、つまり最初のディレクトリエントリを指すようにします。ここで，i はディレクトリ内のディレクトリエントリの // インデックスである。
2. i = 0;
3. de = (struct dir\_entry \*) bh->b\_data;
4. 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

// 論理ブロックが保持できることを示し、ブロックをスキップして検索を続ける。そうでない場合は、新しく読み込まれたブロックがディレクトリエントリデータを持っているので、ディレクトリエントリ構造のポインタがブロックの // バッファブロックデータ部分を指し、そこで検索を続けます。

1. // 196行目のi/DIR\_ENTRIES\_PER\_BLOCKは、現在検索されているディレクトリエントリiがあるディレクトリファイル内のブロック番号を計算するために使用でき、関数 // create\_block()(inode.c, line 147)は、デバイス上のロジックブロックを読み込んだり、作成したりするために使用できます。
2. if ((char \*)de >= BLOCK\_SIZE+bh->b\_data) {
3. brelse(bh);
4. bh = NULL;
5. block = create\_block(dir,i/DIR\_ENTRIES\_PER\_BLOCK);
6. if (!block)
7. return NULL;
8. if (!(bh = bread(dir->i\_dev,block))) { // skip if empty.
9. i += DIR\_ENTRIES\_PER\_BLOCK;
10. continue;
11. }
12. de = (struct dir\_entry \*) bh->b\_data;
13. }

// 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

1. // エントリを空にして、ディレクトリファイルのサイズ（＋ディレクトリエントリの長さ）を更新して、ディレクトリのi-nodeを変更するように設定して、このディレクトリの変更時間を現在の時間に更新する。
2. if (i\*sizeof(struct dir\_entry) >= dir->i\_size) {
3. de->inode=0;
4. dir->i\_size = (i+1)\*sizeof(struct dir\_entry);
5. dir->i\_dirt = 1;
6. dir->i\_ctime = CURRENT\_TIME;
7. }

// 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、 // 未使用または新規に追加されたフリーのディレクトリ・エントリが見つかったことを意味します。

// 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

1. ユーザーデータ空間からディレクトリエントリのファイル名フィールドに // 名前を入力し，ディレクトリエントリの対応するキャッシュバッファブロックの // 変更フラグを設定します。ディレクトリエントリへのポインターとキャッシュバッファブロックへのポインターを // 返します。
2. if (!de->inode) {
3. dir->i\_mtime = CURRENT\_TIME;
4. for (i=0; i < NAME\_LEN ; i++)
5. de->name[i]=(i<namelen)?get\_fs\_byte(name+i):0;
6. bh->b\_dirt = 1;
7. \*res\_dir = de;
8. return bh;
9. }
10. de++; // if entry is in use, continue to check the next entry.
11. i++; 221 }
12. // この関数はここでは実行できません。これは、Linus氏が上記のfind\_entry()関数のコードをコピーし、それをこの関数に変更したからかもしれません :)。
13. brelse(bh);
14. return NULL;
15. }

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。

1. 226 static struct m\_inode \* follow\_link(struct m\_inode \* dir, struct m\_inode \* inode) 227 {。
2. unsigned short fs; // Used to temporarily save FS segment register.
3. 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、 // プロセスのタスク構造に設定されたルートのi-nodeを使用し、リンクの数を増やします

1. // を1つずつ入力します。ディレクトリエントリのi-nodeが指定されていない場合は、ディレクトリのi-nodeを戻して、 // NULLを返します。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， // そのディレクトリエントリに対応する i-node を直接返します。
2. if (!dir) {
3. dir = current->root;
4. dir->i\_count++;
5. }
6. if (!inode) {
7. iput(dir);
8. return NULL;
9. }
10. if (!S\_ISLNK(inode->i\_mode)) {
11. iput(dir);
12. return inode;
13. }

// 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ている。FSがユーザデータセグメントを指していない場合，あるいは，与えられたディレクトリエントリのi-nodeの最初のダイレクトブロック番号が0に等しい場合，あるいは，最初のダイレクトブロックを読み込む際にエラーが発生した場合には，dirおよびinode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。

1. // 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は， // bh が指すバッファブロックデータ領域にすでに存在している。実際には，このバッファブロックデータ領域には， // リンクが指すファイルパス名文字列が1つだけ含まれている。
2. \_\_asm\_\_("mov %%fs,%0":"=r" (fs)); // get FS contents.
3. if (fs != 0x17 || !inode->i\_zone[0] ||
4. !(bh = bread(inode->i\_dev, inode->i\_zone[0]))) {
5. iput(dir);
6. iput(inode);
7. return NULL;
8. }

// この時点では、シンボリックリンクファイルのデータ内容を使って、i-nodeの

// 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

1. // FS = 0x10とし，関数namei()が呼ばれた後に， // 元のFSに戻す。最後に，バッファブロックが解放され，\_namei()で得られたシンボルリンクが指し示すファイルのi-nodeが // 返される．
2. iput(inode);
3. \_\_asm\_\_("mov %0,%%fs"::"r" ((unsigned short) 0x10));
4. inode = \_namei(bh->b\_data,dir,0);
5. \_\_asm\_\_("mov %0,%%fs"::"r" (fs));
6. brelse(bh);
7. return inode;
8. }

257

1. /\*
2. \* get\_dir()
3. \*
4. \* Getdir traverses the pathname until it hits the topmost directory.
5. \* It returns NULL on failure.
6. \*/

//// 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。

1. // 戻り値：ディレクトリのi-nodeポインタ、失敗した場合はNULL。
2. static struct m\_inode \* get\_dir(const char \* pathname, struct m\_inode \* inode) 265 {
3. char c;
4. const char \* thisname;
5. struct buffer\_head \* bh;
6. int namelen,inr;
7. struct dir\_entry \* de;
8. struct m\_inode \* dir;

272

// まず、パラメータの有効性を判断します。与えられたディレクトリのi-nodeポインタinodeが空の場合、 // 現在のプロセスのカレントワーキングディレクトリの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。

1. i-nodeの // 参照カウントを削除し、パス名の最初の文字「/」を削除する。これにより、現在のプロセスは、 // 検索の開始点として設定したルートのi-nodeのみを使用できるようになります。
2. if (!inode) {
3. inode = current->pwd; // i-node of the working directory.
4. inode->i\_count++;
5. }
6. if ((c=get\_fs\_byte(pathname))=='/') {
7. iput(inode); // put back the original i-node.
8. inode = current->root; // root i-node specified for the process.
9. pathname++;
10. inode->i\_count++;
11. }

// 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

1. // カレントディレクトリの // 'inode' は，プロセスルートの i-node，カレントワーキングディレクトリの i-node，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。
2. while (1) {
3. thisname = pathname;
4. if (!S\_ISDIR(inode->i\_mode) || !permission(inode,MAY\_EXEC)) {
5. iput(inode);
6. return NULL;
7. }

// 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。この時点では、変数'namelen'は現在処理されているディレクトリ名部分の長さにぴったりで、変数'thisname'はディレクトリ名部分の先頭を指しています。このとき、終了文字 // NULL であれば、パス名の最後まで検索し、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ことを示し、i-node ポインタを返します。

1. // 注意：パス名の最後の名前がディレクトリ名であっても、'/'文字が付加されていない場合、この関数は最後のディレクトリ名のi-nodeを返しません。 // 例えば、以下のようになります。パス名が '/usr/src/linux' の場合、この関数は、 // 'src/' ディレクトリ名の i-node を返すだけです。
2. for(namelen=0;(c=get\_fs\_byte(pathname++))&&(c!='/');namelen++)
3. /\* nothing \*/ ;
4. if (!c)
5. 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。

1. // 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。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の場合は、 // follow\_link()を使用して、指し示すディレクトリエントリ名のi-nodeを取得します。
2. if (!(bh = find\_entry(&inode,thisname,namelen,&de))) {
3. iput(inode);
4. return NULL;
5. }
6. inr = de->inode; // i-node number of the current directory name part.
7. brelse(bh);
8. dir = inode;
9. if (!(inode = iget(dir->i\_dev,inr))) { // get i-node content.
10. iput(dir);
11. return NULL;
12. }
13. if (!(inode = follow\_link(dir,inode)))
14. return NULL;
15. }
16. }

308

1. /\*
2. \* dir\_namei()
3. \*
4. \* dir\_namei() returns the inode of the directory of the 313 \* specified name, and the name within that directory.
5. \*/
6. //// 指定されたディレクトリ名のi-nodeと最上位のディレクトリ名を取得します。 // Parameters: pathname - ディレクトリのパス名; namelen - パス名の長さ; name - 返される // 最上位のディレクトリ名; 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名の最上位ディレクトリのi-nodeと名前と長さ // です。エラー時にはNULLを返します。
7. static struct m\_inode \* dir\_namei(const char \* pathname,
8. int \* namelen, const char \*\* name, struct m\_inode \* base)
9. {
10. char c;
11. const char \* basename;
12. 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

1. // しかし，返された i-node ポインタは，最後の '/' 文字の前のディレクトリ名の // i-node を指しています．
2. if (!(dir = get\_dir(pathname,base))) // base is the i-node of the starting dir.
3. return NULL;
4. basename = pathname;
5. while (c=get\_fs\_byte(pathname++))
6. if (c=='/')
7. basename=pathname;
8. \*namelen = pathname-basename-1;
9. \*name = basename;
10. return dir;
11. }

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。

1. // 対応するi-nodeを返します。
2. struct m\_inode \* \_namei(const char \* pathname, struct m\_inode \* base,
3. int follow\_links)
4. {
5. const char \* basename;
6. int inr,namelen;
7. struct m\_inode \* inode;
8. struct buffer\_head \* bh;
9. 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を返して終了する。返された一番上の名前の長さが0の場合，そのパス名がディレクトリの最後の項目にちなんで付けられたものであることを意味する。

// 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。

1. // i-nodeを直接検索します。返された名前の長さが0でなければ，再度 dir\_namei() // 関数を呼び出し，新たに指定された開始 // ディレクトリベースでトップレベルのディレクトリ名を検索し，返された情報をもとに同様の判断を行う。311 if (!(dir = dir\_namei(pathname,&namelen,&basename)))
2. return NULL;
3. if (!namelen) /\* special case: '/usr/' etc \*/
4. return dir;
5. if (!(base = dir\_namei(pathname,&namelen,&basename,base)))
6. return NULL;
7. if (!namelen) /\* special case: '/usr/' etc \*/
8. return base;

// そして、返されたトップレベルのディレクトリの中で、指定されたファイル名のディレクトリエントリのi-nodeを探します。もう一度注意してください! 末尾にもディレクトリ名があるが、 // その後ろに「/」が付加されていない場合、最後のディレクトリのi-nodeは返されない! 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。

1. // を使用して、ディレクトリ・エントリのi-node関数find\_entry()を使用して、この状況を個別に処理することができます。この時点で、'de'には見つかったディレクトリ・エントリ・ポインタが入り、'dir'にはそのディレクトリ・エントリを含むディレクトリのi-nodeポインタが入ります。
2. bh = find\_entry(&base,basename,namelen,&de);
3. if (!bh) {
4. iput(base);
5. return NULL;
6. }

// 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

1. // 修正されたフラグです。最後に、i-node ポインタ inode が返されます。さらに、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る場合、 follow\_link() を使用して、それが指し示すディレクトリエントリの // i-node を取得します。
2. inr = de->inode;
3. brelse(bh);
4. if (!(inode = iget(base->i\_dev,inr))) {
5. iput(base);
6. return NULL;
7. }
8. if (follow\_links)
9. inode = follow\_link(base,inode);
10. else
11. iput(base);
12. inode->i\_atime=CURRENT\_TIME;
13. inode->i\_dirt=1;
14. return inode;
15. }

365

//// シンボリックリンクを辿らずに、指定したパス名のi-nodeを取る。

1. // Parameters: pathname - パス名。 // 戻り値：対応するi-nodeです。
2. struct m\_inode \* lnamei(const char \* pathname)
3. {
4. return \_namei(pathname, NULL, 0);
5. }

370

1. /\*
2. \* namei()
3. \*
4. \* 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.
5. \*/

//// 指定されたパス名のi-nodeを取り、シンボリックリンクを辿ることができます。

1. // Parameters: pathname - パス名。 // 戻り値：対応するi-nodeです。
2. struct m\_inode \* namei(const char \* pathname)
3. {
4. return \_namei(pathname,NULL,1);
5. }

382

1. /\*
2. \* open\_namei()
3. \*
4. \* namei for open - this is in fact almost the whole open-routine.
5. \*/

//// 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。

1. // 書き込み権限、実行権限）などの属性があります。新しく作成されたファイルでは，これらの属性は， // 将来のファイルへのアクセスのためにのみ使用されます。 また，読み取り専用のファイルを作成するオープンコールは， // 読み書き可能なファイルハンドルも返します。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：0は正常に返され、それ以外はエラーコードが返されます。res\_inode - 対応するファイルパス名のi-nodeポインタを // 返します。
2. int open\_namei(const char \* pathname, int flag, int mode,
3. struct m\_inode \*\* res\_inode)
4. {
5. const char \* basename;
6. int inr,dev,namelen;
7. struct m\_inode \* dir, \*inode;
8. struct buffer\_head \* bh;
9. 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

1. ファイルオープンフラグに // 追加されます。切り捨てフラグ O\_TRUNC は // ファイルが書き込み可能な場合にのみ有効だからである。次に，現在のプロセスのファイルアクセス許可マスクを使用し， // 与えられたモードの対応するビットをマスクし， // 通常のファイルフラグ I\_REGULAR を追加する。 // このフラグは、開いているファイルが存在せず、 // 作成する必要がある場合に、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。以下の411行目の注釈を参照してください。
2. if ((flag & O\_TRUNC) && !(flag & O\_ACCMODE))
3. flag |= O\_WRONLY;
4. mode &= 0777 & ~current->umask;
5. 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

1. // ディレクトリ名のファイル操作が開かれています。そのため、直接ディレクトリのi-nodeを返し、 // 0を返して終了します。それ以外の場合は、処理が不正なので、i-nodeを戻して、エラーコードを返す。
2. if (!(dir = dir\_namei(pathname,&namelen,&basename,NULL)))
3. return -ENOENT;
4. if (!namelen) { /\* special case: '/usr/' etc \*/
5. if (!(flag & (O\_ACCMODE|O\_CREAT|O\_TRUNC))) {
6. \*res\_inode=dir;
7. return 0;
8. }
9. iput(dir);
10. return -EISDIR; 410 }

// 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，パス名の最後のファイル名に対応するディレクトリエントリ構造を検索し，同時にディレクトリエントリのキャッシュブロックを取得する。キャッシュブロックのポインタが // NULL の場合は，その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

1. // ファイルを作成することしかできません。このとき，ファイルを作成する操作ではない場合や， // ユーザーがディレクトリに書き込む権限を持っていない場合は， // ディレクトリのi-nodeを戻して，対応するエラーコードを返して終了する。
2. bh = find\_entry(&dir,basename,namelen,&de);
3. if (!bh) {
4. if (!(flag & O\_CREAT)) {
5. iput(dir);
6. return -ENOENT;
7. }
8. if (!permission(dir,MAY\_WRITE)) {
9. iput(dir);
10. return -EACCES;
11. }

// 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

1. 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、 // 新しいi-nodeの初期設定を行います：ユーザーID、アクセスモード、モディファイド // フラグを設定します。次に，指定されたディレクトリdirに新しいディレクトリ・エントリを追加します。
2. inode = new\_inode(dir->i\_dev);
3. if (!inode) {
4. iput(dir);
5. return -ENOSPC;
6. }
7. inode->i\_uid = current->euid;
8. inode->i\_mode = mode;
9. inode->i\_dirt = 1;
10. 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， // 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。

1. // 番号を新たに適用した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します。そして， // キャッシュブロックを解放し，ディレクトリの i-node を戻します。そして、新しいディレクトリエントリのi-nodeポインタが // 返され、正常に終了します。
2. if (!bh) {
3. inode->i\_nlinks--;
4. iput(inode);
5. iput(dir);
6. return -ENOSPC;
7. }
8. de->inode = inode->i\_num;
9. bh->b\_dirt = 1;
10. brelse(bh);
11. iput(dir);
12. \*res\_inode = inode;
13. return 0;
14. }

// 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。

1. // 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。
2. inr = de->inode;
3. dev = dir->i\_dev;
4. brelse(bh);
5. if (flag & O\_EXCL) {
6. iput(dir);
7. return -EEXIST;
8. }
9. // 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、アクセスモードがwrite-onlyかread-write、またはアクセス許可がない場合は、 // i-nodeを戻して、アクセス許可のエラーコードを返します。
10. if (!(inode = follow\_link(dir,iget(dev,inr))))
11. return -EACCES;
12. if ((S\_ISDIR(inode->i\_mode) && (flag & O\_ACCMODE)) ||
13. !permission(inode,ACC\_MODE(flag))) {
14. iput(inode);
15. return -EPERM;
16. }
17. // 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。truncateフラグが設定されている場合は、 // ファイルサイズを0に切り詰めます。最後に、ディレクトリエントリのi-nodeへのポインタを // 返し、0（成功）を返します。
18. inode->i\_atime = CURRENT\_TIME;
19. if (flag & O\_TRUNC)
20. truncate(inode);
21. \*res\_inode = inode;
22. return 0;
23. }

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。

// この関数は、モードとdevで指定されたfilenameという名前の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、または名前付きパイプ）を // 作成します。

// パラメータ： filename - ファイル名のパス名; mode - 使用する許可と // 作成するノードのタイプを指定する; dev - デバイス番号。

1. // 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。
2. int sys\_mknod(const char \* filename, int mode, int dev)
3. {
4. const char \* basename;
5. int namelen;
6. struct m\_inode \* dir, \* inode;
7. struct buffer\_head \* bh;
8. 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

1. // 見つかった場合は、エラーコードを返します。トップファイル名の長さが0の場合，与えられたパス名の最後にファイル名が // 指定されていないか，プロセスにディレクトリへの書き込み権限が // ないことを意味し，ディレクトリのi-nodeを戻してエラーコードを返す．
2. if (!suser())
3. return -EPERM;
4. if (!(dir = dir\_namei(filename,&namelen,&basename, NULL)))
5. return -ENOENT;
6. if (!namelen) {
7. iput(dir);
8. return -ENOENT;
9. }
10. if (!permission(dir,MAY\_WRITE)) {
11. iput(dir);
12. return -EPERM;
13. }

// 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

1. パス名に対応する // 最後のファイル名が既に存在している場合は， // 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， // ディレクトリの i ノードが戻され，ファイルが既に存在するというエラーコードが返されます。
2. bh = find\_entry(&dir,basename,namelen,&de);
3. if (bh) {
4. brelse(bh);
5. iput(dir);
6. return -EEXIST;
7. }

// 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、 // 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やキャラクタデバイスファイルを作成している場合は、 // 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。

1. // つまり，デバイスファイルに対して，i-nodeのi\_zone[0]には，デバイスファイルで定義された // デバイスのデバイス番号が格納される。次に，i-nodeの修正時間とアクセス時間を現在の時間に設定し，i-nodeの修正フラグを設定します。
2. inode = new\_inode(dir->i\_dev);
3. if (!inode) { // put back inode and return error code if failed.
4. iput(dir);
5. return -ENOSPC;
6. }
7. inode->i\_mode = mode;
8. if (S\_ISBLK(mode) || S\_ISCHR(mode))
9. inode->i\_zone[0] = dev;
10. inode->i\_mtime = inode->i\_atime = CURRENT\_TIME;
11. 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

1. // 要求されたi-nodeの参照接続カウントをリセットし、i-nodeを戻し、エラーコードを返します。
2. bh = add\_entry(dir,basename,namelen,&de);
3. if (!bh) {
4. iput(dir);
5. inode->i\_nlinks=0;
6. iput(inode);
7. return -ENOSPC;
8. }

// 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

1. // ディレクトリ・アイテムを作成します。ディレクトリエントリのi-nodeフィールドを新しいi-node番号と等しくし、 // キャッシュブロックの修正フラグを設定し、ディレクトリと新しいi-nodeを戻し、 // キャッシュブロックを解放し、最後に0（成功）を返します。
2. de->inode = inode->i\_num;
3. bh->b\_dirt = 1;
4. iput(dir);
5. iput(inode);
6. brelse(bh);
7. return 0;
8. }

514

//// ディレクトリを作る（system-callで使用）。

// パラメータ：pathname - パス名、mode - ディレクトリで使用される許可属性。

1. // 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。
2. int sys\_mkdir(const char \* pathname, int mode)
3. {
4. const char \* basename;
5. int namelen;
6. struct m\_inode \* dir, \* inode;
7. struct buffer\_head \* bh, \*dir\_block;
8. struct dir\_entry \* de;

522

// 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

1. // 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， // プロセスにディレクトリへの書き込み権限がないことを意味し， // ディレクトリのi-nodeを戻して，エラーコードを返す。
2. if (!(dir = dir\_namei(pathname,&namelen,&basename, NULL)))
3. return -ENOENT;
4. if (!namelen) {
5. iput(dir);
6. return -ENOENT;
7. }
8. if (!permission(dir,MAY\_WRITE)) {
9. iput(dir);
10. return -EPERM;
11. }

// 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

1. // バイト（2つのディレクトリエントリのサイズ）、ノードのモディファイフラグ、ノードのモディファイタイムとアクセスタイムを設定しています。2つのディレクトリエントリは，それぞれ「...」と「...」のディレクトリに // 使用されます。
2. bh = find\_entry(&dir,basename,namelen,&de);
3. if (bh) {
4. brelse(bh);
5. iput(dir);
6. return -EEXIST;
7. }
8. inode = new\_inode(dir->i\_dev);
9. if (!inode) { // put back inode and return error code if failed.
10. iput(dir);
11. return -ENOSPC;
12. }
13. inode->i\_size = 32;
14. inode->i\_dirt = 1;
15. 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

1. // 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 の // リンクカウントをリセットし，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 i-node の修正フラグを設定する。
2. if (!(inode->i\_zone[0]=new\_block(inode->i\_dev))) {
3. iput(dir);
4. inode->i\_nlinks--;
5. iput(inode);
6. return -ENOSPC;
7. }
8. 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

1. // 対応するブロックをバッファキャッシュに格納する）。同様に，エラーが発生した場合は，ディレクトリのi-nodeを戻し，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のリンクカウントをリセットし，新しいi-nodeを戻し，スペースエラーコードを返さずに終了する。
2. if (!(dir\_block=bread(inode->i\_dev,inode->i\_zone[0]))) {
3. iput(dir);
4. inode->i\_nlinks--;
5. iput(inode);
6. return -ERROR;
7. }

// 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

1. // i-node番号、nameフィールドに". "を設定します。次に'de'は次のディレクトリ・エントリ // 構造体をポイントし、親ディレクトリのi-node番号と名前「...」を構造体に格納します。 // そして，キャッシュブロックモディファイドフラグを設定し，バッファブロックを解放する。新しいi-nodeのmode // フィールドを再初期化し、i-node modifiedフラグを設定します。
2. de = (struct dir\_entry \*) dir\_block->b\_data;
3. de->inode=inode->i\_num; // set the '.' directory entry.
4. strcpy(de->name,".");
5. de++;
6. de->inode = dir->i\_num; // set the '..' directory entry.
7. strcpy(de->name,"..");
8. inode->i\_nlinks = 2;
9. dir\_block->b\_dirt = 1;
10. brelse(dir\_block);
11. inode->i\_mode = I\_DIRECTORY | (mode & 0777 & ~current->umask);
12. inode->i\_dirt = 1;

// ここで、指定され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。

1. // 新しく作成されたディレクトリの名前。失敗した場合（ディレクトリエントリのバッファブロックポインタがNULLの場合を含む）、ディレクトリのi-nodeを戻します。要求されたi-nodeの参照リンクカウントはリセットされ、i-nodeに戻されます。エラーコードを返して終了する。
2. bh = add\_entry(dir,basename,namelen,&de);
3. if (!bh) {
4. iput(dir);
5. inode->i\_nlinks=0;
6. iput(inode);
7. return -ENOSPC;
8. }

// 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

1. // 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、バッファブロックが解放され、最後に0（成功）が返されます。
2. de->inode = inode->i\_num;
3. bh->b\_dirt = 1;
4. dir->i\_nlinks++;
5. dir->i\_dirt = 1;
6. iput(dir);
7. iput(inode);
8. brelse(bh);
9. return 0;
10. }

587

1. /\*
2. \* routine to check that the specified directory is empty (for rmdir)
3. \*/

//// ディレクトリが空であるかどうかをチェックします。

// Parameters: inode - 指定されたディレクトリのi-nodeポインタです。

1. // 戻り値：1 - ディレクトリが空、0 - 空ではない。
2. static int empty\_dir(struct m\_inode \* inode)
3. {
4. int nr,block;
5. int len;
6. struct buffer\_head \* bh;
7. struct dir\_entry \* de;

597

// まず、指定されたディレクトリに存在するディレクトリ・エントリの数を計算し、チェックします。

// 2つのディレクトリエントリの情報が正しいかどうかを確認します。少なくても

// 2つのディレクトリエントリ：名前が「...」と「...」であるエントリのこと。もし、その数が

1. ディレクトリ・エントリの // 数が 2 より少ないか，ディレクトリの i ノードの最初のダイレクト・ブロックが // どのディスク・ブロック番号も指していないか，ダイレクト・ブロックが読めない場合は， // 警告メッセージが表示され，0 が返される。(失敗)となります。
2. len = inode->i\_size / sizeof (struct dir\_entry); // number of directory entries.
3. if (len<2 || !inode->i\_zone[0] ||
4. !(bh=bread(inode->i\_dev,inode->i\_zone[0]))) {
5. printk("warning - bad directory on dev %04x\n",inode->i\_dev);
6. return 0;
7. }

// この時、'bh'で示されるバッファブロックには、ディレクトリエントリデータが入っています。をさせています。

// 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

1. 1つ目のディレクトリ・エントリの // 番号がカレント・ディレクトリのそれと等しくないか、 // 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、 // 2つのディレクトリ・エントリのnameフィールドが". "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。
2. de = (struct dir\_entry \*) bh->b\_data;
3. if (de[0].inode != inode->i\_num || !de[1].inode ||
4. strcmp(".",de[0].name) || strcmp("..",de[1].name)) {
5. printk("warning - bad directory on dev %04x\n",inode->i\_dev);
6. return 0;
7. }
8. // そして、'nr'をディレクトリ・エントリ番号（0から数えて）と等しくし、'de'は3番目のディレクトリ・エントリを指し、 // ディレクトリ内の残りの（len - 2）ディレクトリ・エントリをすべてループして、i-node numberフィールドが0（使用済み）でないかどうかを確認します。
9. nr = 2;
10. de += 2;
11. 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。

1. // 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(または、ファイルが削除されたなど、 // 使用されていない)場合は、次のブロックの読み込みを続行し、 // 読み込みができない場合は、エラーで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させます。
2. if ((void \*) de >= (void \*) (bh->b\_data+BLOCK\_SIZE)) {
3. brelse(bh);
4. block=bmap(inode,nr/DIR\_ENTRIES\_PER\_BLOCK);
5. if (!block) {
6. nr += DIR\_ENTRIES\_PER\_BLOCK;
7. continue;
8. }
9. if (!(bh=bread(inode->i\_dev,block)))
10. return 0;
11. de = (struct dir\_entry \*) bh->b\_data;
12. }

// 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

1. // バッファブロックは解放され，0が返される。そうでなければ，ディレクトリ内のすべてのディレクトリエントリが // 問い合わせられていなければ，ディレクトリエントリ番号 'nr' がインクリメントされ， // 'de' が次のディレクトリエントリに向けられ，検出が継続される。
2. if (de->inode) {
3. brelse(bh);
4. return 0;
5. }
6. de++;
7. nr++; 630 }
8. // ここでコードが実行されると，使用されたディレクトリエントリが // ディレクトリに見つからないことを意味し（もちろん，最初の2つを除いて），その後，バッファブロックが解放され，1 // が返される。
9. brelse(bh);
10. return 1; // directory is empty!
11. }

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。

1. // 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。
2. int sys\_rmdir(const char \* name)
3. {
4. const char \* basename;
5. int namelen;
6. struct m\_inode \* dir, \* inode;
7. struct buffer\_head \* bh;
8. struct dir\_entry \* de;

642

// 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

1. // 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， // プロセスにディレクトリへの書き込み権限がないことを意味し， // ディレクトリのi-nodeを戻して，エラーコードを返す。
2. if (!(dir = dir\_namei(name,&namelen,&basename, NULL)))
3. return -ENOENT;
4. if (!namelen) {
5. iput(dir);
6. return -ENOENT;
7. }
8. if (!permission(dir,MAY\_WRITE)) {
9. iput(dir);
10. return -EPERM;
11. }

// 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従って、 // find\_entry()関数を使ってディレクトリ・エントリを探し、 // ディレクトリ・エントリを含むバッファ・ブロック'bh'、ディレクトリのi-node 'dir'、およびディレクトリ・エントリ'de'を返す。

// 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

1. // のファイルが既に存在している場合は、エラーコードが返されます。ディレクトリエントリのi-nodeを取得する操作が間違っていた場合、ディレクトリのi-nodeが戻され、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、エラーコードが返されます。
2. bh = find\_entry(&dir,basename,namelen,&de);
3. if (!bh) {
4. iput(dir);
5. return -ENOENT;
6. }
7. if (!(inode = iget(dir->i\_dev, de->inode))) {
8. iput(dir);
9. brelse(bh);
10. return -EPERM;
11. }

// 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、この3つのオブジェクトの情報をチェックすることで、 // 削除の実行可能性を検証します。

//

// 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。

1. // そのため、ディレクトリのi-nodeとエントリのi-nodeを戻し、 // バッファブロックをリリースして、エラーコードを返します。
2. if ((dir->i\_mode & S\_ISVTX) && current->euid &&
3. inode->i\_uid != current->euid) {
4. iput(dir);
5. iput(inode);
6. brelse(bh);
7. return -EPERM;
8. }
9. // ディレクトリエントリのi-nodeのデバイス番号が、このエントリを含むディレクトリのデバイス番号と // 等しくない場合、または削除するディレクトリの参照されるリンクカウントが1よりも // 大きい（シンボルリンクがあることを示すなど）場合、ディレクトリは削除できません。 // そのため，削除す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る。
10. if (inode->i\_dev != dir->i\_dev || inode->i\_count>1) {
11. iput(dir);
12. iput(inode);
13. brelse(bh);
14. return -EPERM;
15. }

// 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。

1. // ". "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。その後、ディレクトリエントリのi-nodeと削除されるディレクトリのi-node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。
2. if (inode == dir) { /\* we may not delete ".", but "../dir" is ok \*/
3. iput(inode);
4. iput(dir);
5. brelse(bh);
6. return -EPERM;
7. }

// 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

1. // ディレクトリの場合は、この削除操作の前提となるものが全く存在しないことになります。そして、削除されるディレクトリ・エントリのi-nodeとそのディレクトリのi-nodeが戻され、バッファ・ブロックが解放され、エラー・コードが返される。
2. if (!S\_ISDIR(inode->i\_mode)) {
3. iput(inode);
4. iput(dir);
5. brelse(bh);
6. return -ENOTDIR;
7. }
8. 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。そして，削除すべきディレクトリ名を含むディレクトリのi-nodeと削除すべきディレクトリのi-nodeを // 戻し，バッファブロックを解放し，エラーコードを返します。
9. if (!empty\_dir(inode)) {
10. iput(inode);
11. iput(dir);
12. brelse(bh);
13. return -ENOTEMPTY;
14. }

// 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、バッファブロック修正フラグが設定されたことを // 確認して、バッファブロックを解放する。そして、削除されたi-nodeのリンク数を設定します。

1. // ディレクトリを0（アイドルを示す）にして、i-node modifiedフラグを設定します。
2. if (inode->i\_nlinks != 2)
3. printk("empty directory has nlink!=2 (%d)",inode->i\_nlinks);
4. de->inode = 0;
5. bh->b\_dirt = 1;
6. brelse(bh);
7. inode->i\_nlinks=0;
8. 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

1. // ノードのフラグです。最後に，削除す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のi-nodeを戻し，0を返します（削除操作は成功）．
2. dir->i\_nlinks--;
3. dir->i\_ctime = dir->i\_mtime = CURRENT\_TIME;
4. dir->i\_dirt=1;
5. iput(dir);
6. iput(inode);
7. return 0;
8. }

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、どのプロセスもそのファイルを開いていない場合、そのファイルも削除され、占有されていたデバイススペースが解放されます。

// パラメータ： name - ファイル名（パス名）です。

1. // 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。
2. int sys\_unlink(const char \* name)
3. {
4. const char \* basename;
5. int namelen;
6. struct m\_inode \* dir, \* inode;
7. struct buffer\_head \* bh;
8. 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

1. // 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， // プロセスにディレクトリへの書き込み権限がないことを意味し， // ディレクトリのi-nodeを戻して，エラーコードを返す。
2. if (!(dir = dir\_namei(name,&namelen,&basename, NULL)))
3. return -ENOENT;
4. if (!namelen) {
5. iput(dir);
6. return -ENOENT;
7. }
8. if (!permission(dir,MAY\_WRITE)) {
9. iput(dir);
10. return -EPERM;
11. }
12. // 次に，指定されたディレクトリのi-nodeとディレクトリ名に従って， // find\_entry()関数を使ってディレクトリ・エントリを探し， // ディレクトリ・エントリを含むバッファ・ブロック'bh'，ディレクトリのi-node 'dir'，ディレクトリ・エントリ'de'を返します。 // そして、ディレクトリ・エントリ'de'のi-node番号に従って、 // iget()関数を使って対応するi-nodeを取得します。
13. bh = find\_entry(&dir,basename,namelen,&de);
14. if (!bh) {
15. iput(dir);
16. return -ENOENT;
17. }
18. if (!(inode = iget(dir->i\_dev, de->inode))) {
19. iput(dir);
20. brelse(bh);
21. return -ENOENT;
22. }

// 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、この3つのオブジェクトの情報をチェックすることで、 // 削除の実行可能性を検証します。

//

// 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を // 戻して

1. // 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。
2. if ((dir->i\_mode & S\_ISVTX) && !suser() &&
3. current->euid != inode->i\_uid &&
4. current->euid != dir->i\_uid) {
5. iput(dir);
6. iput(inode);
7. brelse(bh);
8. return -EPERM;
9. }
10. // 指定されたファイル名がディレクトリの場合，削除することはできません。そのときは、 // 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、 // 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、エラー・コードが返されます。
11. if (S\_ISDIR(inode->i\_mode)) {
12. iput(inode);
13. iput(dir);
14. brelse(bh);
15. return -EPERM;
16. }
17. // 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。
18. if (!inode->i\_nlinks) {
19. printk("Deleting nonexistent file (%04x:%d), %d\n",
20. inode->i\_dev,inode->i\_num,inode->i\_nlinks);
21. inode->i\_nlinks=1;
22. }
23. // 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。そして、ファイル名ディレクトリエントリのi-node number // フィールドを0に設定し、ディレクトリエントリが解放されたことを示し、バッファブロックモディファイドフラグを設定し、バッファブロックを解放します。
24. de->inode = 0;
25. 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

1. // i-nodeのリンク数から1を引いた値が0になり、この時点でどのプロセスもファイルを開いていない場合、 // iput()が呼ばれてi-nodeが戻され、占有していたデバイス空間が解放されると、ファイルも削除される。fs/inode.cの183行目を参照。
2. inode->i\_nlinks--;
3. inode->i\_dirt = 1;
4. inode->i\_ctime = CURRENT\_TIME;
5. iput(inode);
6. iput(dir);
7. return 0;
8. }

766

//// シンボリックリンクを作成します。

// 既存のファイルにシンボリックリンク（ソフトリンクとも呼ばれる）を作成することができます。

1. // 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。
2. int sys\_symlink(const char \* oldname, const char \* newname)
3. {
4. struct dir\_entry \* de;
5. struct m\_inode \* dir, \* inode;
6. struct buffer\_head \* bh, \* name\_block;
7. const char \* basename;
8. int namelen, i;
9. char c; 775

// まず、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、エラーコード

// が返されます。新しいパス名にファイル名が含まれていない場合は， // 新しいパス名のディレクトリのi-nodeを戻して，エラーコードを返す。また，ユーザが新しいディレクトリへの書き込み権限を // 持っていない場合は，リンクを確立することができない。

1. // そして、新しいパス名のディレクトリのi-nodeを戻して、エラーコードを返します。
2. dir = dir\_namei(newname,&namelen,&basename, NULL);
3. if (!dir)
4. return -EACCES;
5. if (!namelen) {
6. iput(dir);
7. return -EPERM;
8. }
9. if (!permission(dir,MAY\_WRITE)) {
10. iput(dir);
11. return -EACCES;
12. }
13. // ここで、ディレクトリで指定されたデバイス上に新しいi-nodeを申請し、i-nodeの // モードをシンボリックリンクタイプとプロセスで指定されたモードマスクコードに設定し、 // i-node modifiedフラグを設定します。
14. if (!(inode = new\_inode(dir->i\_dev))) {
15. iput(dir);
16. return -ENOSPC;
17. }
18. inode->i\_mode = S\_IFLNK | (0777 & ~current->umask);
19. 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

1. // ブロック番号を設定し，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，対応するディレクトリのi-nodeを // 戻し，新しく要求されたi-nodeのリンクカウントをリセットし， // 新しいi-nodeを戻し，スペースエラーコードを返さない。
2. if (!(inode->i\_zone[0]=new\_block(inode->i\_dev))) {
3. iput(dir);
4. inode->i\_nlinks--;
5. iput(inode);
6. return -ENOSPC;
7. }
8. inode->i\_dirt = 1;
9. // 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、バッファキャッシュに // ブロックを配置することです）。エラーがあった場合、ディレクトリのi-nodeを戻し、 // 新しく要求されたi-nodeのリンクカウントをリセットし、 // 新しいi-nodeを戻して、エラーコードを返す。
10. if (!(name\_block=bread(inode->i\_dev,inode->i\_zone[0]))) {
11. iput(dir);
12. inode->i\_nlinks--;
13. iput(inode);
14. return -ERROR;
15. }

// 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

// ユーザー空間にあるシンボリックリンク名の文字列を， // ディスクブロックがあるバッファブロックにコピーし， // バッファブロックモディファイドフラグを設定します。ユーザーが提供した文字列がNULLなしで // 終わらないように、バッファブロックのデータ領域の最後のバイトにNULLを置く必要があります。

1. // そして，バッファブロックを解放し，i-nodeの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等しくなるように設定し，i-nodeの // モディファイドフラグを設定します．
2. i = 0;
3. while (i < 1023 && (c=get\_fs\_byte(oldname++)))
4. name\_block->b\_data[i++] = c;
5. name\_block->b\_data[i] = 0;
6. name\_block->b\_dirt = 1;
7. brelse(name\_block);
8. inode->i\_size = i;
9. 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、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

1. // シンボリックリンクのファイル名が既に存在している場合，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，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 // 戻され，ファイルが既に存在しているというエラーコードが返される。
2. bh = find\_entry(&dir,basename,namelen,&de);
3. if (bh) {
4. inode->i\_nlinks--;
5. iput(inode);
6. brelse(bh);
7. iput(dir);
8. return -EEXIST;
9. }

// 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 (

1. // 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が戻され，要求されたi-nodeの参照リンクカウントがリセットされて， // i-nodeが戻され，エラーコードが返されます。
2. bh = add\_entry(dir,basename,namelen,&de);
3. if (!bh) {
4. inode->i\_nlinks--;
5. iput(inode);
6. iput(dir);
7. return -ENOSPC;
8. }
9. // 最後に，新しいディレクトリエントリのi-nodeフィールドが新しいi-node番号と等しくなり， // バッフ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，ディレクトリと新しい // i-nodeが戻され，最後に0（成功）が返される。
10. de->inode = inode->i\_num;
11. bh->b\_dirt = 1;
12. brelse(bh);
13. iput(dir);
14. iput(inode);
15. return 0;
16. }

836

//// 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。

1. // 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。
2. int sys\_link(const char \* oldname, const char \* newname)
3. {
4. struct dir\_entry \* de;
5. struct m\_inode \* oldinode, \* dir;
6. struct buffer\_head \* bh;
7. const char \* basename;
8. int namelen; 844

// 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

1. // 元のファイルパス名に対応するi-node 'oldinode'を取る。これが0の場合はエラーを意味し、エラーコードを返します。元のパス名がディレクトリ名に対応する場合、i-nodeは戻され、エラーコードも返されます。
2. oldinode=namei(oldname);
3. if (!oldinode)
4. return -ENOENT;
5. if (S\_ISDIR(oldinode->i\_mode)) {
6. iput(oldinode);
7. return -EPERM;
8. }

// 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

1. // 元のパス名が戻され，エラーコードが返される。ファイル名が新しいパス名に含まれていない場合は，元のパス名のi-nodeと新しいパス名のディレクトリが元に戻され，エラーコードが返されます。
2. dir = dir\_namei(newname,&namelen,&basename, NULL);
3. if (!dir) {
4. iput(oldinode);
5. return -EACCES;
6. }
7. if (!namelen) {
8. iput(oldinode);
9. iput(dir);
10. return -EPERM;
11. }

// 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

1. // のエラーコードを返します。また，ユーザが新しいディレクトリに書き込む権利を持っていない場合は， // リンクが確立できないので，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を // 戻して，エラーコードを返します。
2. if (dir->i\_dev != oldinode->i\_dev) {
3. iput(dir);
4. iput(oldinode);
5. return -EXDEV;
6. }
7. if (!permission(dir,MAY\_WRITE)) {
8. iput(dir);
9. iput(oldinode);
10. return -EACCES;
11. }

// 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

1. // リンクを作成します。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， // 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， // エラーコードが返されます。
2. bh = find\_entry(&dir,basename,namelen,&de);
3. if (bh) {
4. brelse(bh);
5. iput(dir);
6. iput(oldinode);
7. return -EEXIST;
8. }

// 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

1. 新たに追加されたディレクトリエントリを含む // バッファブロックがセットされ，バッファブロックが解放され， // ディレクトリの i-node が戻される。
2. bh = add\_entry(dir,basename,namelen,&de);
3. if (!bh) {
4. iput(dir);
5. iput(oldinode);
6. return -ENOSPC;
7. }
8. de->inode = oldinode->i\_num;
9. bh->b\_dirt = 1;
10. brelse(bh); 888 iput(dir);
11. // そして、元のノードのリンクカウントを1増やし、その変更時間を現在の時間に修正し、i-node modifiedフラグを設定する。最後に、元のパス名のi-nodeを戻して、0（成功）を返します。
12. oldinode->i\_nlinks++;
13. oldinode->i\_ctime = CURRENT\_TIME;
14. oldinode->i\_dirt = 1;
15. iput(oldinode);
16. return 0;
17. }

895

## 12.8 file\_table.c

## 12.8 file\_table.c

12.8.1 機能

file\_table.cのプログラムは現在空で、ファイルテーブルの配列のみが定義されています。

### 12.8.2 Code annotation

プログラム 12-7 linux/fs/file\_table.c

1. /\*
2. \* linux/fs/file\_table.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/ 6

// <linux/fs.h> ファイルシステムのヘッダファイルです。ファイルテーブルの構造（file, buffer\_head, // m\_inodeなど）を定義します。

7 #include <linux/fs.h>.

8

9 struct file file\_table[NR\_FILE]; // ファイルテーブルの配列（全64項目）。

10

## 12.9 block\_dev.c

ここからはファイルシステムプログラムの第3部で、block\_dev.c、char\_dev.c、pipe.c、file\_dev.c、read\_write.cの5つのプログラムがあります。最初の4つのプログラムは、主にファイルシステムのデータアクセス操作を実装するread\_write.cのサービスを提供しています。read\_write.cプログラムは、主にシステムコールであるsys\_write()とsys\_read()を実装しています。これら5つのプログラムは、ブロックデバイス、キャラクタデバイス、パイプ "デバイス"、ファイルシステム "デバイス "とのシステムコールのインタフェースドライバと考えることができます。これらの関係は、図12-26で表すことができます。システムコールのsys\_write()やsys\_read()は、パラメータで与えられたファイル記述子の属性に基づいてファイルの種類を判断し、対応するデバイス・インターフェース・プログラムのリード／ライト関数を呼び出し、これらの関数はそれに応じてドライバコードを実行します。

sys\_

write()

sys\_

read()

read\_write.c

Access normal files

file\_write(),file\_read()

Access pipe files

pipe\_

write

()

,pipe\_

read

()

Access block device files

block\_write(),block\_read()

Access char device

files

rw\_char()

。

Operating on

tty\_write(),tty\_read() and

rw\_memory() etc. based on main

devie nr.

Sys

-

call

file\_dev.c

pipe.c

block\_dev.c

char\_dev.c

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

block\_dev.c プログラムは、ブロック・デバイス・ファイルのデータ・アクセス操作プログラムに属します。このファイルには、ブロック・デバイスのオリジナル・データを直接読み書きするための2つのブロック・デバイス・リード／ライト関数、block\_read()およびblock\_write()が含まれています。これらの2つの関数は、システムコール関数read()とwrite()によって呼び出され、他の場所では参照されません。

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。

chars

chars1

chars2

offset

pos

Block

n

Block n+1

Block n

+

2

count

図12-27 ブロックデータ読み書き動作ポインタ位置図

ユーザーバッファは、ユーザープログラムの実行開始時にシステムによって割り当てられるか、実行中に動的に適用されます。この関数を呼び出す前に，システムはユーザバッファが使用する仮想リニアアドレスを，主記憶領域の対応するメモリページにマッピングする。

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. / 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> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。

// <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。

// のディスクリプター/割り込みゲートなどが定義されています。

7 #include <errno.h>

8

1. #include <linux/sched.h>
2. #include <linux/kernel.h>
3. #include <asm/segment.h>
4. #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数の配列であるhd\_sizes[]に 総ブロック数の配列の各項目は、サブデバイス番号で決まるサブデバイスが所有するデータブロックの // 総数に対応する。

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

1. ブロックの // エッジでは，データがあるブロック全体をまず読み出し， // 書き込み位置からデータを埋めていく必要がある。そして、データの完全なブロックをディスクに // 書き込む（つまり、バッファキャッシュに処理する）。
2. int block\_write(int dev, long \* pos, char \* buf, int count)
3. {

// 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

1. // ディスクブロックの読み書きが開始され、ブロック内の最初のバイトが書き込まれるオフセット位置'offset'が取得されます。
2. int block = \*pos >> BLOCK\_SIZE\_BITS; // the block number where pos is located.
3. int offset = \*pos & (BLOCK\_SIZE-1); // offset value in the data block.
4. int chars;
5. int written = 0;
6. int size;
7. struct buffer\_head \* bh;
8. register char \* p; // local register variable.

25

// ブロックデバイスファイルを書き込む場合、書き込まれるデータブロックの総数は、 // もちろん、指定されたデバイスで許可される最大のデータブロック数を超えてはいけません。

1. // 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、 // 関数パラメータで指定された書き込みデータの長さを比較し、制限します。システム内のデバイスで長さが指定されていない場合は、 // デフォルトの長さである 0x7fffff（2GBブロック）が使用されます。
2. if (blk\_size[MAJOR(dev)])
3. size = blk\_size[MAJOR(dev)][MINOR(dev)];
4. else
5. 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。

1. // 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、 // 次の操作に備えます。バッファブロック操作が失敗した場合は，書き込まれたバイト数が // 返され，バイトが書き込まれなかった場合は，エラーコード（負の数）が返される。
2. while (count>0) {
3. if (block >= size)
4. return written?written:-EIO;
5. chars = BLOCK\_SIZE - offset; // bytes that can be written to this block.
6. if (chars > count)
7. chars=count;
8. if (chars == BLOCK\_SIZE)
9. bh = getblk(dev,block); // buffer.c, lines 206 and 322.
10. else
11. bh = breada(dev,block,block+1,block+2,-1);
12. block++;
13. if (!bh)
14. 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'counts'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

1. // ユーザーバッファから 'chars' バイトを、バッファブロック内の書き込み開始位置にコピーする。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， // 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）。
2. p = offset + bh->b\_data;
3. offset = 0;
4. \*pos += chars;
5. written += chars; // The total number of bytes written.
6. count -= chars;
7. while (chars-->0)
8. \*(p++) = get\_fs\_byte(buf++);
9. bh->b\_dirt = 1;
10. brelse(bh);
11. }
12. return written; // Returns the number of bytes written. 54 }

55

//// ブロックリード機能 - 指定し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。

1. // 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。
2. int block\_read(int dev, unsigned long \* pos, char \* buf, int count)
3. {
4. int block = \*pos >> BLOCK\_SIZE\_BITS;
5. int offset = \*pos & (BLOCK\_SIZE-1);
6. int chars;
7. int size;
8. int read = 0;
9. struct buffer\_head \* bh;
10. 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

1. // 関数のパラメータで指定された読み取りデータの長さを比較して制限するために、まずデバイスのブロックの // 長さが取り出されます。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、デフォルトの長さである 0x7fffff (2GB ブロック) が使用されます。
2. if (blk\_size[MAJOR(dev)])
3. size = blk\_size[MAJOR(dev)][MINOR(dev)];
4. else
5. 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

1. // 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。読まれたバイトがない場合は、 // エラーコードが返されます。その後、ブロック番号を1つ増やして、次の操作に備えます。
2. while (count>0) {
3. if (block >= size)
4. return read?read:-EIO;
5. chars = BLOCK\_SIZE-offset;
6. if (chars > count)
7. chars = count; 76 if (!(bh = breada(dev,block,block+1,block+2,-1)))
8. return read?read:-EIO;
9. block++;
10. // 次に，ポインタpはまず，データを読み込むディスクブロックのバッファブロック内の位置を指す。最後のループ読み込み操作のデータが1ブロックに満たない場合は、必要なバイトをブロックの先頭から読み込む必要があるため、あらかじめオフセットを0に設定しておく必要があります。その後、ファイル内のオフセットポインタ'pos'を読み込むバイト数'chars'分だけ進め、読み込むバイト数を統計値'read'に累積します。そして、読み取る必要のあるカウント値'count'から、読み取るべきカウント数'chars'を引く。そして、バッファブロックの'p'が指す開始点から'chars'バイトをユーザバッファにコピーし、ユーザバッファのポインタを前方に移動させる。このコピーが完了すると、バッファブロックは解放されます。
11. p = offset + bh->b\_data;
12. offset = 0;
13. \*pos += chars;
14. read += chars; // The total number of bytes read in.
15. count -= chars;
16. while (chars-->0)
17. put\_fs\_byte(\*(p++),buf++);
18. brelse(bh);
19. }
20. return read; // returns the number of bytes read. 89 }

90

## 12.10 file\_dev.c

### 12.10.1 Function

### file\_dev.cファイルには、file\_read()とfile\_write()という2つの関数があります。これらの関数は、システムコール関数の read() と write() によって、通常のファイルを読み書きする際にも使用されます。前述のファイルblock\_dev.cと同様に、このファイルもファイルデータにアクセスするためのものですが、このプログラムの関数はファイルのパス名を指定して操作します。ファイルのi-nodeとファイル構造は、関数のパラメータで与えられます。iノードの情報により，対応するデバイス番号を得ることができ，ファイル構造により，ファイルの現在の読み書きポインタ位置を得ることができます。前のファイルblock\_dev.cの関数では、パラメータにデバイス番号とファイル内の読み書き位置を直接指定しています。これは特に、/dev/hd0デバイスファイルのようなブロックデバイスファイルを操作するために使われます。

### 12.10.2 Code annotation

プログラム 12-9 linux/fs/file\_dev.c

1. / 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が含まれています。 // <fcntl.h> ファイル制御のヘッダファイルです。ファイルとそのディスクリプターに使用される // 操作制御定数シンボルの定義が含まれています。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

1. // <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。
2. #include <errno.h>
3. #include <fcntl.h>

9

1. #include <linux/sched.h>
2. #include <linux/kernel.h>
3. #include <asm/segment.h>

13

14 #define MIN(a,b) (((a)<(b))?(a):(b)) // 15 #define MAX(a,b) (((a)>(b))?(a):(b)) // a, b の最大値を取得します。

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。Buf」はユーザ空間でのバッファの位置を指定し、「count」は読み取る必要のあるバイト数を指定する。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)である。

1. 17 int file\_read(struct m\_inode \* inode, struct file \* filp, char \* buf, int count) 18 {。
2. int left,chars,nr;
3. struct buffer\_head \* bh;

21

// まず、パラメータの有効性を判断します。読み込むべきバイト数が // 0以下の場合は0を返します。読み込むバイト数が0でない場合は、データを完全に読み取るか、問題が発生するまで、 // 以下のループ操作を行います。

// 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。

1. // 読み込み操作が失敗する。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， // バッファブロックポインタは NULL に設定されます。(filp->f\_pos)/BLOCK\_SIZE を使って， // ファイルのカレントポインタが存在するデータブロック番号が計算されます．
2. if ((left=count)<=0)
3. return 0;
4. while (left) {
5. if (nr = bmap(inode,(filp->f\_pos)/BLOCK\_SIZE)) { // inode.c, line 140.
6. if (!(bh=bread(inode->i\_dev,nr)))
7. break;
8. } else
9. 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。

// ここで、小さい値は、この操作で読み取るべきバイト数「chars」を表します。(BLOCK\_SIZE - nr) > left」の場合、そのブロックは、 // 読み込まれるべき最後のデータであり、そうでなければ、 // 次のブロックのデータを読み込む必要がある。

1. // 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、今回読み込まれるバイト数'chars'を // 進める。残りのバイト数'left'も、'chars'バイト分だけ // 減算する必要があります。
2. nr = filp->f\_pos % BLOCK\_SIZE;
3. chars = MIN( BLOCK\_SIZE-nr , left );
4. filp->f\_pos += chars;
5. left -= chars;
6. // 上記のデータがデバイスから読み込まれた場合、pをバッファブロック内のデータが読み込まれる場所に向け、 // 'chars'バイトをユーザバッファ'buf'にコピーし、そうでない場合は、 // 'chars'バイトを0にしてユーザバッファに書き込む。
7. if (bh) {
8. char \* p = nr + bh->b\_data;
9. while (chars-->0)
10. put\_fs\_byte(\*(p++),buf++);
11. brelse(bh);
12. } else {
13. while (chars-->0)
14. put\_fs\_byte(0,buf++);
15. } 43 }

// 最後に，i-nodeのアクセス時間を現在の時間に修正し， // 読み込んだバイト数を返します。読み込んだバイト数が0の場合は、エラーコードが返されます。

1. // CURRENT\_TIME は、include/linux/sched.h の 142 行目で定義されているマクロで、 // UNIX 時間を計算するために使用されます。つまり、1970年1月1日の0:00から現在の時刻までを計算します。単位は秒です。
2. inode->i\_atime = CURRENT\_TIME;
3. return (count-left)?(count-left):-ERROR;
4. } 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する。'buf' はユーザ空間におけるバッファの位置を // 指定し、'count' は書き込むバイト数である。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未満) である。

1. 48 int file\_write(struct m\_inode \* inode, struct file \* filp, char \* buf, int count) 49 {。
2. off\_t pos;
3. int block,c;
4. struct buffer\_head \* bh;
5. char \* p;
6. int i=0;

55

1. /\*
2. \* ok, append may not work when many processes are writing at the same time 58 \* but so what. That way leads to madness anyway.
3. \*/
4. // 最初にデータがファイルのどこに書き込まれるかを決定します。データをファイルに追加したい場合は， // 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書きポインタに書き込まれます。
5. if (filp->f\_flags & O\_APPEND)
6. pos = inode->i\_size;
7. else
8. 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

1. // 存在しない場合は作成します。結果としてディスクブロック番号が0になった場合、作成に失敗し、 // ループを終了します。それ以外の場合は、ディスクブロック番号に応じてデバイス上のディスクブロックを読み込み、 // エラーが発生した場合はループを終了します。
2. while (i<count) {
3. if (!(block = create\_block(inode,pos/BLOCK\_SIZE)))
4. break;
5. if (!(bh=bread(inode->i\_dev,block)))
6. 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

1. // をブロックの最後まで書き込んでください。cが書き込まれる残りのバイト数(count - i)よりも大きい場合は、 // 今回は単にc = (count - i)バイトを書き込む。
2. c = pos % BLOCK\_SIZE;
3. p = c + bh->b\_data;
4. bh->b\_dirt = 1;
5. c = BLOCK\_SIZE-c;
6. if (c > count-i) c = count-i;

// データを書き込む前に、次のループ操作で書き込むファイルの位置を // 事前に設定します。そこで、書き込みが必要なバイト数だけ、'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

1. // i-nodeのファイル長フィールドを修正し、i-nodeの修正フラグを設定します。そして、今回書き込まれるバイト数を、書き込まれたバイト数'i'に加えて、 // ループ判定を行う。 // 次に，ユーザバッファbufからキャッシュバッファブロックのpが指す開始位置に， // cバイトがコピーされる。コピー後、バッファブロックは解放される。
2. pos += c;
3. if (pos > inode->i\_size) {
4. inode->i\_size = pos;
5. inode->i\_dirt = 1;
6. }
7. i += c;
8. while (c-->0)
9. \*(p++) = get\_fs\_byte(buf++);
10. brelse(bh);
11. }

// 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。

1. // そして，ファイル読み書きポインタを調整する。つまり，操作によってデータがファイルの最後に // 追加されなかった場合は，ファイルの読み書きポインタが現在の読み書き位置 // 「pos」に調整され，ファイルの i-node の修正時刻が現在の時刻に変更される。 // 最後に、書き込まれたバイト数が返されます。書き込まれたバイト数が0の場合は， // エラーコード-1が返される。
2. inode->i\_mtime = CURRENT\_TIME;
3. if (!(filp->f\_flags & O\_APPEND)) {
4. filp->f\_pos = pos;
5. inode->i\_ctime = CURRENT\_TIME;
6. }
7. return (i?i:-1);
8. }

91

## 12.11 pipe.c

### 12.11.1 Function

パイプライン操作は、プロセス間の通信を行う最も基本的な方法です。pipe.cプログラムは、パイプファイルの読み書き操作関数read\_pipe()とwrite\_pipe()を含み、パイプシステムコールsys\_pipe()を実装しています。これら2つの関数は、システムコールread()とwrite()の低レベルな実装関数でもあり、read\_write.cでのみ使用されています。

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。

P

ipe buffer size (PAGE\_SIZE)

Pipe data size (PIPE\_SIZE)

tail

(

i\_zone

[1])

head

i\_zone

[

(

0

])

Buffer pointer

)

(

i\_size

図12-28 パイプバッファ動作図

read\_pipe()は、パイプライン内のデータを読み取るために使用します。パイプラインにデータがない場合は、パイプラインの書き込み処理を覚醒させ、自身はスリープ状態になります。データが読み込まれていれば、それに応じてパイプの先頭ポインタが調整され、データがユーザーバッファに渡されます。パイプラインのデータがすべて取り出されると、パイプラインの書き込みを待っているプロセスも起こされ、読み込まれたデータのバイト数が返されます。パイプラインの書き込みプロセスがパイプラインの操作を終了した場合、この関数は直ちに終了し、読み込んだバイト数を返します。

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、シグナル操作関数のプロトタイプを定義しています。

// <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が含まれています。 // <termios.h> 端末入出力機能のヘッダファイルです。主に非同期通信ポートを制御するターミナル // インターフェースを定義しています。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/mm.h> メモリ管理のヘッダファイルです。ページサイズの定義と、いくつかのページ // リリース関数のプロトタイプが含まれています。

// <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。

1. // <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。
2. #include <signal.h>
3. #include <errno.h>
4. #include <termios.h>

10

1. #include <linux/sched.h>
2. #include <linux/mm.h> /\* for get\_free\_page \*/
3. #include <asm/segment.h>
4. #include <linux/kernel.h>

15

//// パイプリード機能。

1. // パラメータの'inode'はパイプのi-node、'buf'はユーザデータのバッファポインタ、 // 'count'は読み込んだバイト数です。
2. int read\_pipe(struct m\_inode \* inode, char \* buf, int count)
3. {
4. 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。

// 操作を行います。ループ操作の間、現在のパイプにデータがない場合（size=0）、 // ノードを待っているプロセスが起こされますが、これは通常、書き込みパイプラインのプロセスです。

// 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

1. // kernel/signal.c プログラムです。
2. while (count>0) {
3. while (!(size=PIPE\_SIZE(\*inode))) { // get data length in the pipe. 22 wake\_up(& PIPE\_WRITE\_WAIT(\*inode));
4. if (inode->i\_count != 2) /\* are there any writers? \*/
5. return read;
6. if (current->signal & ~current->blocked)
7. return read?read:-ERESTARTSYS;
8. interruptible\_sleep\_on(& PIPE\_READ\_WAIT(\*inode));
9. }

// 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

1. // 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、 // すでに読まれたバイト数 'read' に加算されます。
2. chars = PAGE\_SIZE-PIPE\_TAIL(\*inode);
3. if (chars > count)
4. chars = count;
5. if (chars > size)
6. chars = size;
7. count -= chars;
8. 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

1. // '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。パイプ内のデータは、 // ユーザーバッファにコピーされます。パイプのi-nodeの場合，パイプバッファのブロックポインタは // i\_sizeフィールドにあります。
2. size = PIPE\_TAIL(\*inode);
3. PIPE\_TAIL(\*inode) += chars;
4. PIPE\_TAIL(\*inode) &= (PAGE\_SIZE-1);
5. while (chars-->0)
6. put\_fs\_byte(((char \*)inode->i\_size)[size++],buf++);
7. }
8. // パイプの読み取り操作が終了すると、パイプラインを待っているプロセスが起こされ、 // 読み取ったバイト数が返されます。
9. wake\_up(& PIPE\_WRITE\_WAIT(\*inode));
10. return read;
11. }

45

//// パイプの書き込み機能。

1. // パラメータの'inode'はパイプのi-node、'buf'はデータバッファポインタ、'count'はパイプに書き込まれるバイト数を // 示します。
2. int write\_pipe(struct m\_inode \* inode, char \* buf, int count)
3. {
4. 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

1. // 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。そうでなければ、現在のプロセスをパイプ上でスリープさせて、パイププロセスがデータを読み取るのを待つことで、パイプのスペースを // 解放します。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。
2. while (count>0) {
3. while (!(size=(PAGE\_SIZE-1)-PIPE\_SIZE(\*inode))) {
4. wake\_up(& PIPE\_READ\_WAIT(\*inode));
5. if (inode->i\_count != 2) { /\* no readers \*/
6. current->signal |= (1<<(SIGPIPE-1));
7. return written?written:-1;
8. }
9. sleep\_on(& PIPE\_WRITE\_WAIT(\*inode));
10. }

// 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

1. // 書き込むバイト数を'count'から今回書き込んだバイト数'chars'を引いて、今回書き込んだバイト数を'written'に追加します。
2. chars = PAGE\_SIZE-PIPE\_HEAD(\*inode);
3. if (chars > count)
4. chars = count;
5. if (chars > size)
6. chars = size;
7. count -= chars;
8. 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。

1. //（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。そして、 // ユーザーバッファから'chars'バイトをパイプのヘッドポインタの先頭にコピーします。パイプのi-nodeの場合、パイプバッファのブロックポインタは、 // i\_sizeフィールドにあります。
2. size = PIPE\_HEAD(\*inode);
3. PIPE\_HEAD(\*inode) += chars;
4. PIPE\_HEAD(\*inode) &= (PAGE\_SIZE-1);
5. while (chars-->0)
6. ((char \*)inode->i\_size)[size++]=get\_fs\_byte(buf++);
7. }
8. // パイプの書き込み操作が終了すると、パイプを待っていたプロセスが起こされ、 // 書き込まれたバイト数が返されます。
9. wake\_up(& PIPE\_READ\_WAIT(\*inode));
10. return written;
11. }

75

//// パイプのsystem-callを作成します。

1. // 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： filees - ファイルハンドルの配列。fildes[0]はパイプのデータを読むために， // fildes[1]はパイプにデータを書き込むために使われます。成功した場合は0、エラーの場合は-1を返します。
2. int sys\_pipe(unsigned long \* fildes)
3. {
4. struct m\_inode \* inode;
5. struct file \* f[2]; // array of file structures.
6. int fd[2]; // array of file handles.
7. int i,j; 82
8. // 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が0のアイテム）を取り出し、参照カウントを1に設定します。 フリーアイテムが1つしかない場合は、アイテムを解放します（参照カウントをリセットします）。2つの空きアイテムが見つからない場合は-1を返します。
9. j=0;
10. for(i=0;j<2 && i<NR\_FILE;i++)
11. if (!file\_table[i].f\_count)
12. (f[j++]=i+file\_table)->f\_count++;
13. if (j==1)
14. f[0]->f\_count=0;
15. if (j<2)
16. 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

1. // 1つだけ空いているファイルハンドルがあれば，そのハンドルを解放する。2 つのフリーのハンドルが見つからない場合は、 // 上で取得した 2 つのファイル構造項目が解放され、 -1 が返されます。
2. j=0;
3. for(i=0;j<2 && i<NR\_OPEN;i++)
4. if (!current->filp[i]) {
5. current->filp[ fd[j]=i ] = f[j];
6. j++;
7. }
8. if (j==1)
9. current->filp[fd[0]]=NULL;
10. if (j<2) {
11. f[0]->f\_count=f[1]->f\_count=0;
12. return -1;
13. }

// 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。

1. // 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 // 構造体の項目が適宜解放され，-1 が返される。
2. if (!(inode=get\_pipe\_inode())) { // fs/inode.c, line 231.
3. current->filp[fd[0]] =
4. current->filp[fd[1]] = NULL;
5. f[0]->f\_count = f[1]->f\_count = 0;
6. return -1;
7. }

// パイプi-nodeのアプリケーションが成功した場合、2つのファイル構造は、 // 両方が同じパイプi-nodeを指すように初期化され、readとwriteのポインタを0に設定します。

1. // 最初のファイル構造のファイルモードは読み込みに設定され， // 2番目のファイル構造のファイルモードは書き込みに設定される。最後に，ファイルハンドル配列をユーザ空間の // 配列にコピーする。成功した場合は0を返して終了します。
2. f[0]->f\_inode = f[1]->f\_inode = inode;
3. f[0]->f\_pos = f[1]->f\_pos = 0;
4. f[0]->f\_mode = 1; /\* read \*/
5. f[1]->f\_mode = 2; /\* write \*/
6. put\_fs\_long(fd[0],0+fildes);
7. put\_fs\_long(fd[1],1+fildes);
8. return 0;
9. }

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。

1. // 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。
2. int pipe\_ioctl(struct m\_inode \*pino, int cmd, int arg)
3. {

// コマンドが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ものであれば、 // パイプのデータ長の値をユーザー引数で指定された場所に入れ、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。

1. switch (cmd) {
2. case FIONREAD:
3. verify\_area((void \*) arg,4);
4. put\_fs\_long(PIPE\_SIZE(\*pino),(unsigned long \*) arg); 124 return 0;
5. default:
6. return -EINVAL;
7. }
8. }

129

## 12.12 char\_dev.c

### 12.12.1 Function

char\_dev.cプログラムには、主にrw\_ttyx()、rw\_tty()、rw\_memory()、rw\_char()といったキャラクタデバイスファイルへのアクセス関数が含まれています。また、項目番号がメジャーデバイス番号を表すデバイス読み書き関数ポインタテーブルがあります。

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. / 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> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。

1. // <asm/io.h> io のヘッダーファイルです。ioポートを操作する関数を、 // マクロの組み込みアセンブリ言語の形で定義する。
2. #include <errno.h>
3. #include <sys/types.h>

9

1. #include <linux/sched.h>
2. #include <linux/kernel.h>

12

1. #include <asm/segment.h>
2. #include <asm/io.h>

15

1. extern int tty\_read(unsigned minor,char \* buf,int count); // terminal read.
2. 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 - 読み書きコマンド; minor - ターミナルサブデバイス番号; buf - バッファ; // cout - 読み書きバイト数; pos - 読み書き操作のカレントポインタ, このポインタはターミナル操作では // 役に立たない。

// 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。

1. 21 static int rw\_ttyx(int rw,unsigned minor,char \* buf,int count,off\_t \* pos) 22 {。
2. return ((rw==READ)?tty\_read(minor,buf,count):
3. tty\_write(minor,buf,count));
4. }

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 {。

1. プロセスが対応する制御端末を持っていない場合、 // 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、実際の読み書きバイト数を // 返します。 29 if (current->tty<0)
2. return -EPERM;
3. return rw\_ttyx(rw,current->tty,buf,count,pos);
4. }

33

1. //// メモリデータの読み書き。実装されていません。
2. static int rw\_ram(int rw,char \* buf, int count, off\_t \*pos)
3. {
4. return -EIO;
5. }

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。

1. 39 static int rw\_mem(int rw,char \* buf, int count, off\_t \* pos) 40 {。
2. return -EIO;
3. }

43

1. //// 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。
2. static int rw\_kmem(int rw,char \* buf, int count, off\_t \* pos)
3. {
4. return -EIO;
5. }

48

// ポートのリード/ライト操作機能。

1. // パラメータ: rw - 読み書きコマンド; buf - バッファ; cout - 読み書きバイト数; // pos - ポートアドレス。戻り値: 実際に読み書きされたバイト数。
2. static int rw\_port(int rw,char \* buf, int count, off\_t \* pos)
3. {
4. 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

1. // 64kの場合、1バイトのリードまたはライト操作がサイクリックに実行されます。コマンドが読み込まれた場合、 // 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。コマンドが書き込まれた場合、 // 1バイトのコンテンツがユーザーデータバッファから取り出され、ポートiに出力されます。
2. while (count-->0 && i<65536) {
3. if (rw==READ)
4. put\_fs\_byte(inb(i),buf++);
5. else
6. outb(get\_fs\_byte(buf++),i);
7. i++; // increase port address. [??] 59 }
8. // 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て、 // 読み書きされたバイト数を返します。
9. i -= \*pos;
10. \*pos += i;
11. return i;
12. } 64

//// メモリのリード/ライト操作機能です。メモリのマスターナンバーは1で、ここでは0～5のサブデバイスの処理のみが実装されています。

65 static int rw\_memory(int rw, unsigned minor, char \* buf, int count, off\_t \* pos) 66 {。

1. // 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。
2. switch(minor) {
3. case 0: // device file name is /dev/ram0 or /dev/ramdisk.
4. return rw\_ram(rw,buf,count,pos);
5. case 1: // device file name is /dev/ram1 or /dev/mem or ram.
6. return rw\_mem(rw,buf,count,pos);
7. case 2: // device file name is /dev/ram2 or /dev/kmem.
8. return rw\_kmem(rw,buf,count,pos);
9. case 3: // device file name is /dev/null. 75 return (rw==READ)?0:count; /\* rw\_null \*/
10. 76 ケース 4: // デバイスファイル名は/dev/port 77 return rw\_port(rw,buf,count,pos);
11. default:
12. return -EIO;
13. }
14. }

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

1. // 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。
2. static crw\_ptr crw\_table[]={
3. NULL, /\* nodev \*/ 87 rw\_memory, /\* /dev/mem etc \*/
4. NULL, /\* /dev/fd \*/
5. NULL, /\* /dev/hd \*/
6. rw\_ttyx, /\* /dev/ttyx \*/
7. rw\_tty, /\* /dev/tty \*/ 92 NULL, /\* /dev/lp \*/ 93 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 - 読み書きコマンド、 dev - デバイス番号、 buf - バッファ、 count - 読み書きバイト数、 pos - 読み書きポインタ。

1. // 戻り値：実際に読み書きしたバイト数。
2. int rw\_char(int rw,int dev, char \* buf, int count, off\_t \* pos)
3. {
4. crw\_ptr call\_addr;

98

1. // 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。デバイスが対応する読み書き機能を持っていない場合も、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読み書き操作関数が呼び出され， // 実際に読み書きされたバイト数が返される。
2. if (MAJOR(dev)>=NRDEVS)
3. return -ENODEV;
4. if (!(call\_addr=crw\_table[MAJOR(dev)]))
5. return -ENODEV;
6. return call\_addr(rw,MINOR(dev),buf,count,pos);
7. }

105

## 12.13 read\_write.c

### 12.13.1 Function

read\_write.cプログラムは、ファイル・オペレーティング・システムのコールであるread()、write()、lseek()を実装しています。read()とwrite()は、ファイルの種類に応じて、前の4つのファイルに実装されているread関数やwrite関数を呼び出します。lseek()は、ファイルの読み書きポインタを設定するために使用します。

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・リード操作を実行し、読み取ったバイト数を返します。ディレクトリ・ファイルまたは通常のフォーマル・ファイルであれば、file\_dev.cのファイル・リード関数file\_read()を呼び出します。write()システム・コールの実装はread()と同様です。

### 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。

### 12.13.2 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 stat{} があります。 // と定数が含まれています。

// <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/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

1. // <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。
2. #include <sys/stat.h>
3. #include <errno.h>
4. #include <sys/types.h>

10

1. #include <linux/kernel.h>
2. #include <linux/sched.h>
3. #include <asm/segment.h>

14

1. // fs/char\_dev.c, line 95.
2. extern int rw\_char(int rw,int dev, char \* buf, int count, off\_t \* pos);
3. // パイプ操作関数の読み込み fs/pipe.c, line 16.
4. extern int read\_pipe(struct m\_inode \* inode, char \* buf, int count);
5. // パイプ操作関数の記述 fs/pipe.c, line 46.
6. extern int write\_pipe(struct m\_inode \* inode, char \* buf, int count);
7. fs/block\_dev.c 56行目 // ブロックデバイス読み込み操作機能
8. extern int block\_read(int dev, off\_t \* pos, char \* buf, int count);
9. // ブロックデバイスの書き込み機能 fs/block\_dev.c, line 16.
10. extern int block\_write(int dev, off\_t \* pos, char \* buf, int count);
11. // ファイル操作関数の読み込み fs/file\_dev.c, line 17.
12. extern int file\_read(struct m\_inode \* inode, struct file \* filp,
13. char \* buf, int count);
14. // ファイル操作関数を書く fs/file\_dev.c, line 48.
15. extern int file\_write(struct m\_inode \* inode, struct file \* filp,
16. 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。

1. // 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, 現在の読み書きの位置から), SEEK\_END // (2, ファイルの最後から) です。
2. int sys\_lseek(unsigned int fd,off\_t offset, int origin)
3. {
4. struct file \* file;
5. 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

1. // 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、エラーコードを // 返して終了します。なぜなら、パイプの先頭と末尾のポインタは自由に移動できないからです
2. if (fd >= NR\_OPEN || !(file=current->filp[fd]) || !(file->f\_inode)
3. || !IS\_SEEKABLE(MAJOR(file->f\_inode->i\_dev)))
4. return -EBADF;
5. if (file->f\_inode->i\_pipe)
6. 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。

1. // Origin = SEEK\_SETで、ファイルの先頭にあるファイル読み書きポインタを原点として // 設定することが要求されます。オフセット値がゼロより小さい場合はエラーコードが返され、そうでない場合はファイル読み書きポインタをオフセットと等しく設定します。 35 switch (origin) {
2. case 0:
3. if (offset<0) return -EINVAL;
4. file->f\_pos=offset;
5. 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。

1. // ファイルの現在の読み取り/書き込みポインタの原点です。ファイルの現在のポインタにオフセット値を加えた値が 0 より小さい場合は、エラーコードが返されます。それ以外の場合は、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。
2. case 1:
3. if (file->f\_pos+offset<0) return -EINVAL;
4. file->f\_pos += offset;
5. 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

1. // ファイルの末尾からのオフセット値です。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より小さい場合は，エラーコードを返す。それ以外の場合は，ファイル長にオフセット値を加えた値に // 読み書きポインタを再配置する。
2. case 2:
3. if ((tmp=file->f\_inode->i\_size+offset) < 0)
4. return -EINVAL;
5. file->f\_pos = tmp;
6. break;
7. // オリジンの設定が無効な場合は，エラーコードが返される。最後に，再配置後のファイル読み書きポインタの // 値が返される。
8. default:
9. return -EINVAL;
10. }
11. return file->f\_pos;
12. }

54

//// Read file system-call.

1. // パラメーターの'fd'はファイルハンドル、'buf'はユーザーバッファ、'count'は読むべきバイト数を表す。
2. int sys\_read(unsigned int fd,char \* buf,int count)
3. {
4. struct file \* file;
5. 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

1. // プログラムの最大オープンファイル番号 NR\_OPEN よりも // 大きいか、読み込むバイト数の値が 0 よりも // 小さいか、ハンドルのファイル構造ポインタが NULL の場合は、エラーコードを返して終了します。読み込むバイト数が 0 に等しい場合は、0 を返す。
2. if (fd>=NR\_OPEN || count<0 || !(file=current->filp[fd]))
3. return -EINVAL;
4. if (!count)
5. 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

// を返します。キャラクタファイルであれば，キャラクタデバイスの読み取り操作を行い， // 読み取ったキャラクタ数を返す。ブロックデバイスファイルであれば，ブロックデバイスのリード

1. // 演算を行い、読み込んだバイト数を返します。
2. verify\_area(buf,count);
3. inode = file->f\_inode;
4. if (inode->i\_pipe)
5. return (file->f\_mode&1)?read\_pipe(inode,buf,count):-EIO;
6. if (S\_ISCHR(inode->i\_mode))
7. return rw\_char(READ,inode->i\_zone[0],buf,count,&file->f\_pos);
8. if (S\_ISBLK(inode->i\_mode))
9. 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。

1. // ファイルサイズよりも大きい場合は，ファイルサイズから // 現在の読み書きポインタの値を引いた値を読み取りバイト数とし，読み取りバイト数が 0 の場合は 0 を返す。そして // ファイルの読み込み操作を行い、読み込んだバイト数を返して終了します。
2. if (S\_ISDIR(inode->i\_mode) || S\_ISREG(inode->i\_mode)) {
3. if (count+file->f\_pos > inode->i\_size)
4. count = inode->i\_size - file->f\_pos;
5. if (count<=0)
6. return 0;
7. return file\_read(inode,file,buf,count);
8. }
9. // ここで実行されると、ファイルの属性を判断できないことを意味します。そして、 // ノードファイルモードを表示し、エラーコードを返します。
10. printk("(Read)inode->i\_mode=%06o\n\r",inode->i\_mode);
11. return -EINVAL;
12. }

82

//// Write file system-call.

1. // 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'は書き込まれるバイト数です。
2. int sys\_write(unsigned int fd,char \* buf,int count)
3. {
4. struct file \* file;
5. 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

1. // がプログラムの最大オープンファイル番号NR\_OPENより大きいか、 // 読むべきバイト数の値が0より小さいか、ハンドルのファイル構造ポインタがNULLの場合、 // エラーコードを返して終了します。読み込むバイト数が 0 に等しい場合は 0 を返します。
2. if (fd>=NR\_OPEN || count <0 || !(file=current->filp[fd]))
3. return -EINVAL;
4. if (!count)
5. 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

1. // 通常のファイルであれば、ファイルの書き込み操作を行い、書き込まれたバイト数を // 返して終了します。
2. inode=file->f\_inode;
3. if (inode->i\_pipe)
4. return (file->f\_mode&2)?write\_pipe(inode,buf,count):-EIO;
5. if (S\_ISCHR(inode->i\_mode))
6. return rw\_char(WRITE,inode->i\_zone[0],buf,count,&file->f\_pos);
7. if (S\_ISBLK(inode->i\_mode))
8. return block\_write(inode->i\_zone[0],&file->f\_pos,buf,count);
9. if (S\_ISREG(inode->i\_mode))
10. return file\_write(inode,file,buf,count);
11. // ここで実行されると、ファイルの属性を判断できないことを意味します。そこで、 // i-nodeのファイル属性を表示し、エラーコードを返して終了します。
12. printk("(Write)inode->i\_mode=%06o\n\r",inode->i\_mode);
13. return -EINVAL;
14. }

104

### 12.13.3 User Program Read/Write Operation

上記のプログラムを読めば，ユーザプログラムにおける読み書きの操作がどのように行われるかが理解できるはずです。以下では，read関数を例にとり，ユーザプログラムにおけるファイルの読み取り関数呼び出しがどのように実行され，完了するかを説明します。

1. 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。
2. System call interface functions;
3. Memory allocation management functions;
4. Set of signal processing functions;
5. String processing functions;
6. Standard input/output functions;
7. 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が高くなり、ユーザープログラムのシステム独立性を高めることができます。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);

この2つの関数の第1パラメータはファイルディスクリプター、第2パラメータは読み書きされるデータを格納するための文字バッファ配列、第3パラメータは読み書きされるべきデータのバイト数です。関数の戻り値は、1回の呼び出し時に送信されたバイト数である。ファイルの読み取り操作では、戻り値は読み取りたいデータよりも小さくなることがあります。戻り値が0の場合は、ファイルの終端に達したことを意味し、-1の場合は、読み取り操作でエラーが発生したことを意味する。書き込みの場合、実際に書き込まれたバイト数が返されます。この値が第3引数で指定された値と一致しない場合は、書き込み操作にエラーが発生したことを意味する。読み取り関数の場合、ライブラリでの実装は以下の通りです。

#define \_\_LIBRARY\_\_

#include <unistd.h>

\_syscall3(int,read,int,fd,char \*,buf,off\_t,count)

ここで，\_syscall3()は，include/unistd.hヘッダファイルの189行目で定義されているマクロです。このマクロを上記の特定のパラメータで展開すると，次のようなコードが得られます。

int read(int fd, char \*buf, off\_t count)

{

long \_\_res; \_\_asm\_\_ volatile (

"int $0x80"

: "=a" (\_\_res)

: "" (\_\_NR\_read), "b" ((long)(fd)), "c" ((long)(buf)), "d" ((long)(count))); if (\_\_res>=0) return int \_\_res; errno=-\_\_res;

1を返す。

}

この拡張マクロは、読み取り操作関数の具体的な実装であり、そこからシステムカーネルに入って実行されるプログラムであることがわかります。埋め込みアセンブリステートメントを使用して、関数番号\_\_NR\_read(3)のLinuxシステムコール割り込み0x80を実行します。この割り込みコールは、実際に読み込まれたバイト数をeax(\_\_res)レジスタに返します。返された値が0より小さい場合は、読み取り操作のエラーが発生したことを意味しますので、エラー番号を反転してグローバル変数errnoに格納し、-1の値を呼び出したプログラムに返します。

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)

本機能では、まずパラメータの有効性を判定する。ファイルディスクリプターの値が，システムが同時に開いているファイルの最大数（20個）よりも大きい場合や，読み込むサイズが0よりも小さい場合，あるいはファイルがまだ開かれていない（ファイルディスクリプターをインデックスとするファイル構造体ポインタがnull）場合には，負のエラーコードを返す。その後、カーネルは、読み込んだデータを格納するバッファサイズが適切かどうかを検証する。検証の過程で、カーネルプログラムは、指定された読み取りバイト数に応じて、バッファbufのサイズを検証する。bufが小さすぎる場合、システムはそれを拡張します。そのため、ユーザプログラムが開いたメモリバッファが小さすぎると、後者のデータが破壊されてしまう可能性があります。

* 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，次の種類の読み取り操作関数を呼び出し，実際に読み取ったバイト数を返す。
* 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。

file\_read()

Library

read()

wrapper function

Execute sys

-

call INT 0x80

handler

Execute kernel function

sys\_read() and call the

relevant

C

function

according to

the file type

Call

read()

pipe\_read()

block\_read()

tty\_read()

C

Library

system\_call.s

write\_read.c

file\_dev.c

pipe.c

block\_dev.c

char\_dev.c

User level

Kernel level

User program

図12-29 Read()関数呼び出しの実行手順

## 12.14 open.c

この章の最初から、説明されているすべての手順は、この章の第4部の手順であるファイルシステムの高レベルの操作と管理の部分に含まれています。本節では、open.c、exec.c、stat.c、fcntl.c、ioctl.cの5つのプログラムを掲載しています。

### 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が含まれています。 // <fcntl.h> ファイル制御のヘッダファイルです。ファイルとそのディスクリプターに使用される // 操作制御定数シンボルの定義が含まれています。

// <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。

// <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、修正時間の構造体と、utime() // プロトタイプが定義されています。

// <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 stat{} があります。 // と定数が含まれています。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/tty.h> ttyヘッダーファイルは、tty\_ioというシリアル通信のためのパラメータや定数を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

1. // <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。
2. #include <string.h>
3. #include <errno.h>
4. #include <fcntl.h>
5. #include <sys/types.h>
6. #include <utime.h>
7. #include <sys/stat.h>

13

1. #include <linux/sched.h>
2. #include <linux/tty.h>
3. #include <linux/kernel.h>

17

18 #include <asm/segment.h>.

19

//// ファイルシステムのステータス情報を取得します。

// 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。

// 成功すると，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， // ubuf が指す ustate 構造体に追加される。ustat 構造体は次のように定義されている。

1. // include/sys/types.hです。
2. int sys\_ustat(int dev, struct ustat \* ubuf)
3. {
4. return -ENOSYS; // Return error code, not been implemented yet. 23 }

24

//// 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

1. // 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。timesポインタがNULLの場合，システムの現在時刻が指定されたファイルのアクセス時間と修正時間のドメインを設定するために使用されます。
2. int sys\_utime(char \* filename, struct utimbuf \* times)
3. {
4. struct m\_inode \* inode;
5. long actime,modtime;

29

// 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

1. // ファイル名。与えられたアクセス時間および修正時間構造体のポインタの時間が // NULLでない場合，ユーザが設定した時間値が構造体から読み込まれる。そうでない場合は，システムの現在の時刻を用いて // ファイルのアクセス時刻と修正時刻を設定する。
2. if (!(inode=namei(filename)))
3. return -ENOENT;
4. if (times) {
5. actime = get\_fs\_long((unsigned long \*) &times->actime);
6. modtime = get\_fs\_long((unsigned long \*) &times->modtime);
7. } else
8. actime = modtime = CURRENT\_TIME;
9. // そして、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の修正フラグを設定し、i-nodeを戻して、0を返します。
10. inode->i\_atime = actime;
11. inode->i\_mtime = modtime;
12. inode->i\_dirt = 1;
13. iput(inode);
14. return 0;
15. }

43

1. /\*
2. \* XXX should we use the real or effective uid? BSD uses the real uid, 46 \* so as to make this call useful to setuid programs.
3. \*/

//// 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

1. // (値4)、W\_OK(2)、X\_OK(1)、F\_OK(0)は、チェックしているファイルが // 読み取り可能、書き込み可能、実行可能、ファイルが存在することを示します。アクセスが許可されている場合は0を、そうでない場合は // エラーコードを返します。
2. int sys\_access(const char \* filename,int mode)
3. {
4. struct m\_inode \* inode;
5. 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

1. // 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ビットで構成されているため、上位ビットをすべてクリアするには8進数07の "AND "が必要である。ファイル名に対応するi-node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る。 // i-nodeが存在する場合は、ファイルの属性コードで、i-nodeが戻されます。また，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。
2. mode &= 0007;
3. if (!(inode=namei(filename)))
4. return -EACCES; // error code: No access rights.
5. i\_mode = res = inode->i\_mode & 0777;
6. 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。

1. // そうでなければ，現在のプロセスユーザがファイルの所有者と同じグループに属していれば，ファイルの // グループ属性が取られ，そうでなければ，resの最下位3ビットがファイルのアクセス権 // の「その他」となる。
2. if (current->uid == inode->i\_uid)
3. res >>= 6;
4. else if (current->gid == inode->i\_gid)
5. res >>= 6; // Here should be "res >>= 3;" [??]
6. // この時、resの下位3ビットは、現在のプロセスとファイルの関係に応じて選択されたアクセス属性ビットです。では，この3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、アクセスは許可され、 // 0を返します。
7. if ((res & 0007 & mode) == mode)
8. return 0;
9. /\*
10. \* XXX we are doing this test last because we really should be
11. \* 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.
12. \*/
13. // 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である場合や，ファイルが誰でも実行・検索できる場合は0を返し，そうでない場合はエラーコードを返します。
14. if ((!current->uid) &&
15. (!(mode & 1) || (i\_mode & 0111)))
16. return 0;
17. 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。

1. // 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。
2. int sys\_chdir(const char \* filename)
3. {
4. 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

1. 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。そのため、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は // 戻され、エラーコードが返されます。
2. if (!(inode = namei(filename)))
3. return -ENOENT; // error code: file or directory does not exist.
4. if (!S\_ISDIR(inode->i\_mode)) {
5. iput(inode);
6. return -ENOTDIR; // error code: not a directory name.
7. }
8. // 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、新しく設定された作業ディレクトリのi-nodeに // ポイントして、0を返します。
9. iput(current->pwd);
10. current->pwd = inode;
11. return (0);
12. }

90

//// ルートディレクトリの変更。

// 指定し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'/'に設定する。

1. // 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。
2. int sys\_chroot(const char \* filename)
3. {
4. 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使用されます。

1. // 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が存在しない場合、 // エラーコードが返されます。i-nodeがディレクトリのi-nodeでない場合、i-nodeは // 戻され、エラーコードが返されます。
2. if (!(inode=namei(filename)))
3. return -ENOENT;
4. if (!S\_ISDIR(inode->i\_mode)) {
5. iput(inode);
6. return -ENOTDIR;
7. }
8. // そして、現在のプロセスの元のルートのi-nodeを解放して、指定されたディレクトリ名のi-nodeに // リセットして、0を返します。
9. iput(current->root);
10. current->root = inode;
11. return (0);
12. }

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。

1. // 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。
2. int sys\_chmod(const char \* filename,int mode)
3. {
4. 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。ファイルのモードは、 // ファイルのi-nodeにあるので、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。

1. // 現在のプロセスの実効ユーザーIDがファイルのi-nodeのユーザーIDと異なり、 // 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（アクセス権なし）。
2. if (!(inode=namei(filename)))
3. return -ENOENT;
4. if ((current->euid != inode->i\_uid) && !suser()) {
5. iput(inode);
6. return -EACCES; 115 }
7. 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修正フラグが設定され、i-nodeが戻され、0が返されます。
8. inode->i\_mode = (mode & 07777) | (inode->i\_mode & ~07777);
9. inode->i\_dirt = 1;
10. iput(inode);
11. return 0;
12. }

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。

1. // 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。
2. int sys\_chown(const char \* filename,int uid,int gid)
3. {
4. struct m\_inode \* inode;

125

1. // この呼び出しは、ファイルのi-nodeにユーザとグループのIDを設定するために使用されるので、まず、 // 与えられたファイル名のi-nodeを取得する必要があります。現在のプロセスがスーパーユーザでない場合は、 // i-nodeを戻し、エラーコード（アクセス権なし）を返します。
2. if (!(inode=namei(filename)))
3. return -ENOENT;
4. if (!suser()) {
5. iput(inode);
6. return -EACCES;
7. }
8. // そうでなければ、パラメータで提供された値を使って、ファイルのi-nodeのユーザIDとグループIDを設定し、フラグを修正するためにi-nodeを設定し、i-nodeを戻して、0を返します。
9. inode->i\_uid=uid;
10. inode->i\_gid=gid;
11. inode->i\_dirt=1;
12. iput(inode);
13. return 0;
14. }

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。これは、オープンファイルがtty端末文字デバイスの場合に、 // 現在のプロセス属性とシステムttyテーブルの修正と設定をチェックするために // 使われます。

// 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

1. // 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。
2. static int check\_char\_dev(struct m\_inode \* inode, int dev, int flag)
3. {
4. struct tty\_struct \*tty;
5. 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、あるプロセスが

//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、プロセスが実際に使用している/dev/ttyxx // デバイスの1つに対応する仮想デバイスです。プロセスが制御端末を持っている場合、そのタスク構造の tty // 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、デバイスNo.4のサブデバイスがパラメータに与えられていれば、そのサブデバイス番号が直接取得されます。取得したデバイスNo.4のサブデバイスの番号が、デバイスNo.

1. 4が0より小さい場合は、プロセスに制御端子がないか、デバイス番号が正しくない場合は-1を返す。
2. if (MAJOR(dev) == 4 || MAJOR(dev) == 5) {
3. if (MAJOR(dev) == 5)
4. min = current->tty;
5. else
6. min = MINOR(dev);
7. if (min < 0)
8. 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，そのデバイスが他のプロセスに使用されていることを示しているので， // そのキャラクタデバイスを再び開くことはできず，-1が返される。それ以外の場合は， // ポインタ tty に tty テーブルの対応する構造体アイテムを指させます。

// 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

// がグループリーダーであり、制御端末を持たず、tty構造体のsessionフィールドが // 0（その端末がまだどのプロセスグループの制御端末にもなっていないことを示す）であれば、 // この端末装置をプロセスの制御端末としてminに設定することが許される。

// 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

1. // 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 // 等しく設定します。
2. if ((IS\_A\_PTY\_MASTER(min)) && (inode->i\_count>1))
3. return -1;
4. tty = TTY\_TABLE(min);
5. if (!(flag & O\_NOCTTY) &&
6. current->leader && 156 current->tty<0 &&
7. tty->session==0) {
8. current->tty = min;
9. tty->session= current->session;
10. tty->pgrp = current->pgrp;
11. }

// 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

// 文字端末装置の設定。読み取り動作を満足させるために，最小読み取り文字数を0，タイムアウトのタイミングを0とし，端末装置をノンカノニカルモードに設定する。ノンブロッキングモードは，ノンカノニカルモードでのみ動作する。

1. // 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、AUXキューに何文字入っていても、 // プロセスは何文字を読んですぐに戻ります。
2. if (flag & O\_NONBLOCK) {
3. TTY\_TABLE(min)->termios.c\_cc[VMIN] =0;
4. TTY\_TABLE(min)->termios.c\_cc[VTIME] =0;
5. TTY\_TABLE(min)->termios.c\_lflag &= ~ICANON;
6. } 167 }
7. return 0;
8. }

170

//// ファイルを開く（または作成する）。

// パラメータの'flag'はオープンファイルのフラグで、値を取ることができます。O\_RDONLY, O\_WRONLY または

// O\_RDWR、およびO\_CREAT、O\_EXCL、O\_APPEND、その他のフラグの組み合わせです。この呼び出しで

// 新しいファイルを作成する際には、モードを使ってファイルの許可属性を指定します。これらの属性

// S\_IRWXU（ファイルの所有者が読み取り、書き込み、実行が可能）、S\_IRUSR（ユーザーが読み取り可能）、S\_IRWXG

(グループメンバーが読み取り、書き込み、実行できる）などがあります。新規に作成されたファイルについては、これらの

のプロパティは、将来のファイルへのアクセスにのみ適用されます。読み取り専用のファイルを作成するオープンコールは

ファイルは，読み書き可能なファイルハンドルも返す。呼び出し操作が成功した場合は、 // 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

1. // ファイルsys/stat.h, fcntl.hです。
2. int sys\_open(const char \* filename,int flag,int mode)
3. {
4. struct m\_inode \* inode;
5. struct file \* f;
6. int i,fd; 176

// パラメータを先に処理します。ユーザーが設定したファイルモードと、プロセスの

許可されたファイルモードを生成するための // モードマスクです。開いている

// ファイルの場合、プロセス構造体のファイル構造体ポインタの配列を検索して

1. // 自由項目を見つけます。空き項目のインデックス番号fdがハンドル値となります。空き項目がない場合は、 // エラーコードが返されます（無効なパラメータ）。
2. mode &= 0777 & ~current->umask;
3. for(fd=0 ; fd<NR\_OPEN ; fd++)
4. if (!current->filp[fd]) // free item found.
5. break;
6. if (fd>=NR\_OPEN)
7. 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

// プログラムを実行し、新しいプログラムが子プロセスで実行を開始します。ファイルハンドルの // close\_on\_exec のビットがセットされていれば、 // execve() が実行されたときに対応するファイルハンドルがクローズされ、 そうでなければファイルハンドルは常にオープンされます。

// 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。

1. 開いているファイルの // （参照カウントが0のエントリ）を指定し、fにファイル // テーブルの配列を指定させて、検索を開始しています。なお、184行目のポインタ代入「0+file\_table」は、「file\_table」と「&file\_table[0]」に相当しますが、この方がわかりやすいかもしれません。
2. current->close\_on\_exec &= ~(1<<fd);
3. f=0+file\_table;
4. for (i=0 ; i<NR\_FILE ; i++,f++)
5. if (!f->f\_count) break; // idle item found.
6. if (i>=NR\_FILE)
7. 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

1. // ファイルオープン操作が成功した場合，inode はオープンされたファイルの i-node ポインタとなります // エラーを示しているので，適用されたばかりのファイル構造は解放され，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 はオープンされたファイルの // i-node ポインタとなる。
2. (current->filp[fd]=f)->f\_count++;
3. if ((i=open\_namei(filename,flag,mode,&inode))<0) {
4. current->filp[fd]=NULL;
5. f->f\_count=0;
6. return i; 194 }

// 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。異なるタイプのファイルに対しては、特別な処理を行う必要があります。もし、文字デバイス

ファイルが開かれたら、check\_char\_dev()関数を呼び出して、現在の // プロセスがファイルを開くことができるかどうかをチェックする必要があります。許可されていれば（この関数は0を返します）、check\_char\_dev()の中で

1. // 制御端末はファイルオープンフラグに応じてプロセスに設定されます。 // キャラクタデバイスファイルのオープンが許可されていない場合は、上記で要求されたファイルアイテムとハンドルリソースを // 解放することしかできず、エラーコードを返します。
2. /\* ttys are somewhat special (ttyxx major==4, tty major==5) \*/
3. if (S\_ISCHR(inode->i\_mode))
4. if (check\_char\_dev(inode,inode->i\_zone[0],flag)) {
5. iput(inode);
6. current->filp[fd]=NULL;
7. f->f\_count=0;
8. return -EAGAIN; // resource is temporarily unavailable.
9. }
10. // 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。置き換えられた場合、キャッシュバッファ内のデバイスのすべてのバッファブロックを // 無効にする必要があります。
11. /\* Likewise with block-devices: check for floppy\_change \*/
12. if (S\_ISBLK(inode->i\_mode))
13. check\_disk\_change(inode->i\_zone[0]);
14. // 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フィールドに開いているファイルのi-nodeを設定し、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。
15. f->f\_mode = inode->i\_mode;
16. f->f\_flags = flag;
17. f->f\_count = 1;
18. f->f\_inode = inode;
19. f->f\_pos = 0;
20. return (fd);
21. }

213

//// 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。

1. // 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。
2. int sys\_creat(const char \* pathname, int mode)
3. {
4. return sys\_open(pathname, O\_CREAT | O\_TRUNC, mode);
5. }

218

// 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。

1. // 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。
2. int sys\_close(unsigned int fd)
3. {
4. struct file \* filp;

222

// 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

1. // パラメータ）が返されます。その後，プロセスの close\_on\_exec ビットマップ内のファイルハンドルの // 対応するビットをリセットする。また，ファイルハンドルのファイル構造ポインタが NULL の場合は， // エラーコードが返されます。
2. if (fd >= NR\_OPEN)
3. return -EINVAL;
4. current->close\_on\_exec &= ~(1<<fd);
5. if (!(filp = current->filp[fd]))
6. 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

1. // ファイルがあるので，0（成功）を返します。ここで参照カウントが0になると、ファイルにはプロセスの参照がなく、 // ファイル構造がフリーになったことになります。そして，ファイルのi-nodeを解放して， // 0を返します。
2. current->filp[fd] = NULL;
3. if (filp->f\_count == 0)
4. panic("Close: file count is 0");
5. if (--filp->f\_count) 232 return (0);
6. iput(filp->f\_inode);
7. return (0);
8. }

236

## 12.15 exec.c

### 12.15.1 Function

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、システムコール割り込み(int 0x80)の関数番号\_\_NR\_execve()のCハンドラであり、exec()関数群のカーネル実装関数でもあります。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ており、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データ構造が格納されています。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。

1. / // 実行ファイルのヘッダ構造です。
2. struct exec {
3. unsigned long a\_magic; /\* Use macros N\_MAGIC, etc for access \*/
4. unsigned a\_text; /\* length of text, in bytes \*/
5. unsigned a\_data; /\* length of data, in bytes \*/
6. unsigned a\_bss; /\* length of uninitialized data area for file, in bytes \*/
7. unsigned a\_syms; /\* length of symbol table data in file, in bytes \*/
8. unsigned a\_entry; /\* start address \*/
9. unsigned a\_trsize; /\* length of relocation info for text, in bytes \*/ 14 unsigned a\_drsize; /\* length of relocation info for data, in bytes \*/

15 };

プログラムがfork()を使って子プロセスを生成するとき、通常はexec()クラスタ関数の一つが

* 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 -- set the initial space start pointer; initialize the space page pointer array to (NULL); get the i-node of the file according to the execution file name; calculate the number of parameters and environment variables number; check file type and execute permission;
* According to the header data structure at the beginning of the executable file, the information is processed therein - the file header is read according to the executed file i-node; If it is a shell script (the first line starts with #!), analyze the shell program name and its parameters, and execute the shell program with the executable file as a parameter; check whether it is executable according to the magic number of the file and the length of the segment;
* The initialization operation before running the new file on the current calling process -- point to the i-node of the new executable file; reset the signal processing handle; set the local descriptor base address and segment length according to the header information; set the parameter and environment parameter page pointer; modify the contents of each field of the process.
* Replace the return address of the original call execve() program on the stack with the new executable running address and run the newly loaded executable program.

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、そのコマンドラインから入力されたコマンドライン引数を受け取ります。例えば、次のようなファイル名リストのコマンドを入力した場合。

ls

-

l /home/john/

シェルプロセスは新しいプロセスを作成し、そこで/bin/lsコマンドを実行します。このとき、/bin/ls実行ファイルがロードされたときのコマンドラインにあった3つのパラメータls、-l、/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[])

1つ目はプログラム実行時のコマンドライン上の引数の数で、通常はargc(argument count)です。2つ目は文字列の引数へのポインタの配列(argv -- argument vector)です。各文字列はパラメータを表し、argvの配列の最後は常にnullで終わるようになっています。通常，argv[0]は実行中のプログラムの名前なので，argcの値は少なくとも1である。上記の例では、図12-30に示すように、argc=3、argv[0]、argv[1]、argv[2]はそれぞれ'ls'、'-l'、'/home/john/'、argv[3]=NULLとなります。

NULL

ls

\

0

-

1

\

0

/

home/john/

\

0

argv

図 12-30 コマンドライン引数ポインタ配列 argv[].

また、Main()にはオプションで3つ目のパラメータがあり、実行ファイルの環境設定をカスタマイズしたり、環境設定パラメータの値を提供するための環境変数パラメータが含まれています。これは、文字列引数へのポインタの配列でもあり、環境変数の値であることを除けば、NULLで終わります。プログラムが環境変数を明示的に使用する必要がある場合、main()は次のように宣言することができます。

int main(int argc, char \*argv[], char \*envp[])

環境文字列は、以下のような形式です。

VAR\_NAME=somevalue

VAR\_NAMEは環境変数の名前を表し、等号の後の文字列はこの環境変数に割り当てられた値を表します。コマンドライン・プロンプトで、シェルの内部コマンド「set」を入力すると、現在の環境における環境パラメータのリストが表示されます。コマンドライン・パラメーターと環境文字列は、以下に説明するように、プログラムが実行を開始する前に、プログラム・メモリー領域のユーザー・スタックの一番上に置かれます。

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。

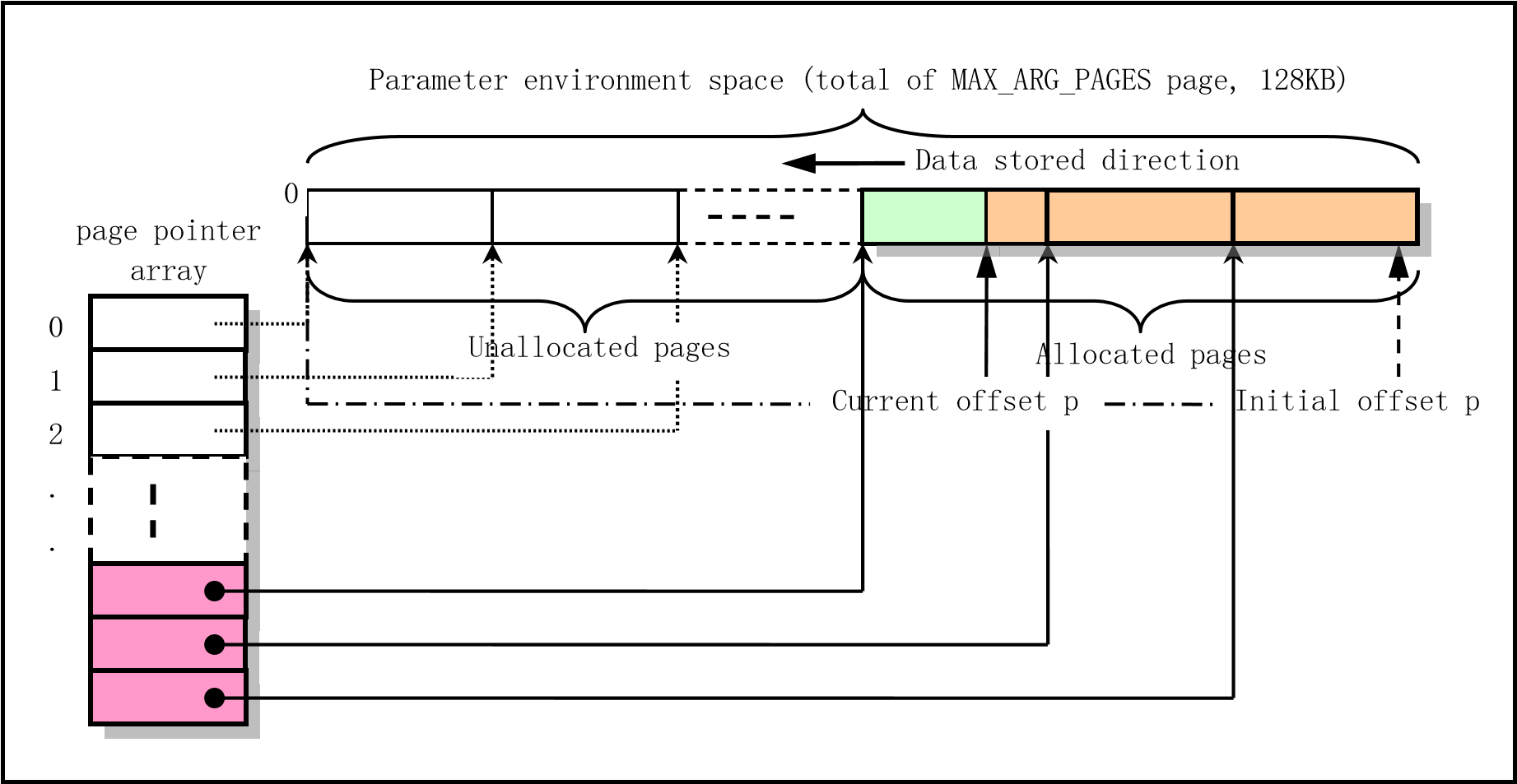


図12-31 パラメーターと環境変数の文字列スペース

copy\_string()が実行された後、カーネルコードは図12-32に示すように、プロセスの論理アドレス空間の先頭からパラメータと環境変数のポインタになるようにpを調整します。これは、プロセスが占有する最大の論理空間サイズである64MBから、パラメータと環境変数が占有するサイズ（128KB-p）を差し引く方法です。また、p1の左部分は、パラメータと環境変数のポインタテーブルを保持するためにcreate\_tables()関数を使用し、p1はポインタテーブルの先頭を指すように再び左に調整されます。そして、得られたポインタはページアラインされ、最後に初期スタックポインタspが得られます。

0

0



128

KB

64

MB

p

p

1

=

p + 64MB

-

128

KB



sp

図12-32 pを初期スタックポインタに変換する方法

create\_tables()関数は、与えられた現在のスタックポインタ値 p とパラメータ変数値 argc と環境変数数 envc に基づいて、新しいプログラムスタック内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し、調整されたスタックポインタ値を返すために使用される。その後、ポインタのページアラインメントを行い、最終的に初期スタックポインタspを得る。作成後のスタックポインタテーブルの形態を図12-33に示す。

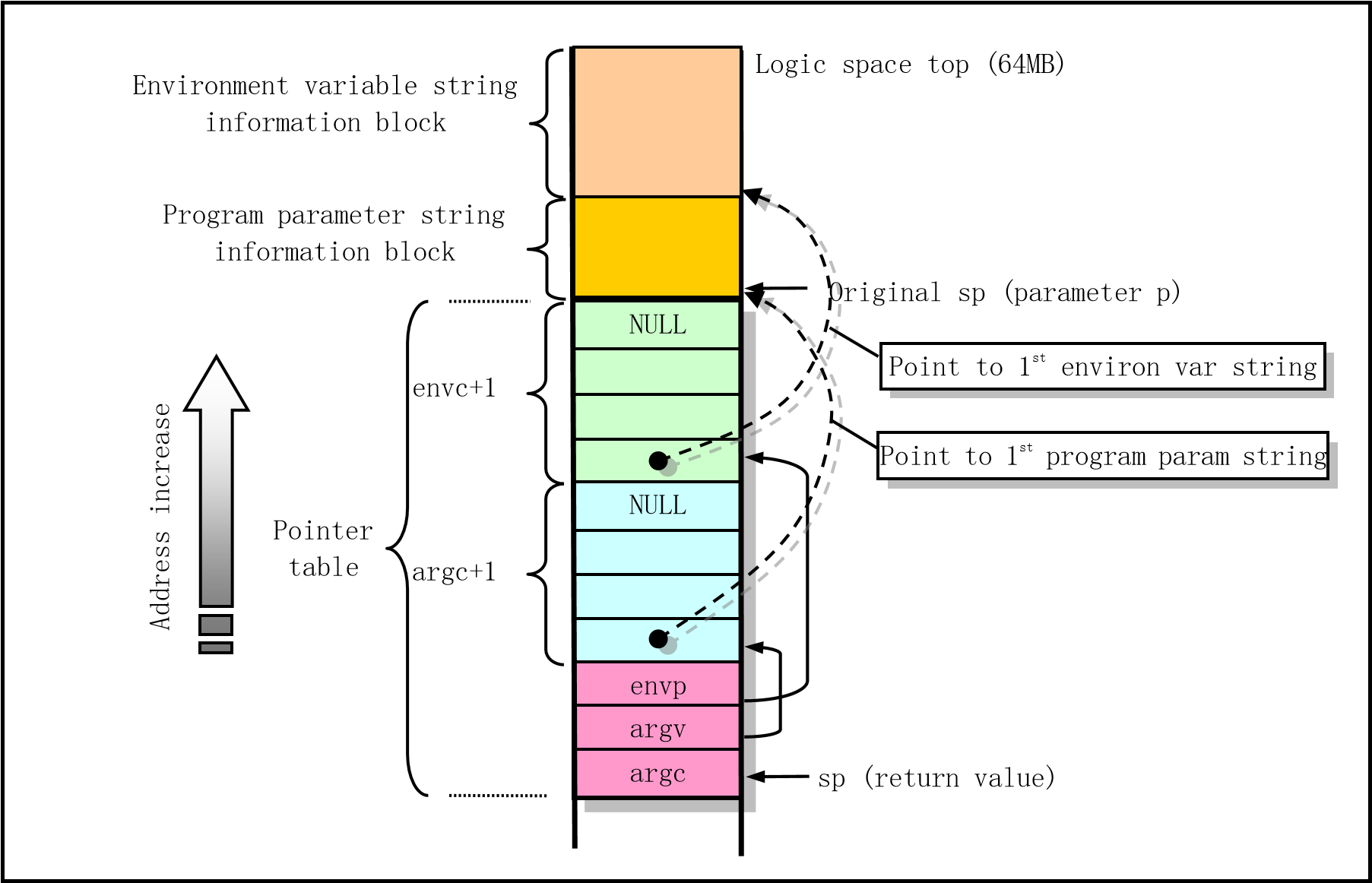


図 12-33 新プログラムスタックのポインタテーブルの模式図

関数do\_execve()が最後に戻るとき、元の呼び出しシステム割り込みプログラムのスタック上のコードポインタeipを、新しい実行プログラムを指すコードエントリポイントに置き換え、スタックポインタを新しい実行ファイルのスタックポインタespに置き換えています。その後、このシステムコールのリターン命令により、最終的にスタック内のデータがポップアップされ、図12-34のようにCPUが新しい実行ファイルを実行することになります。

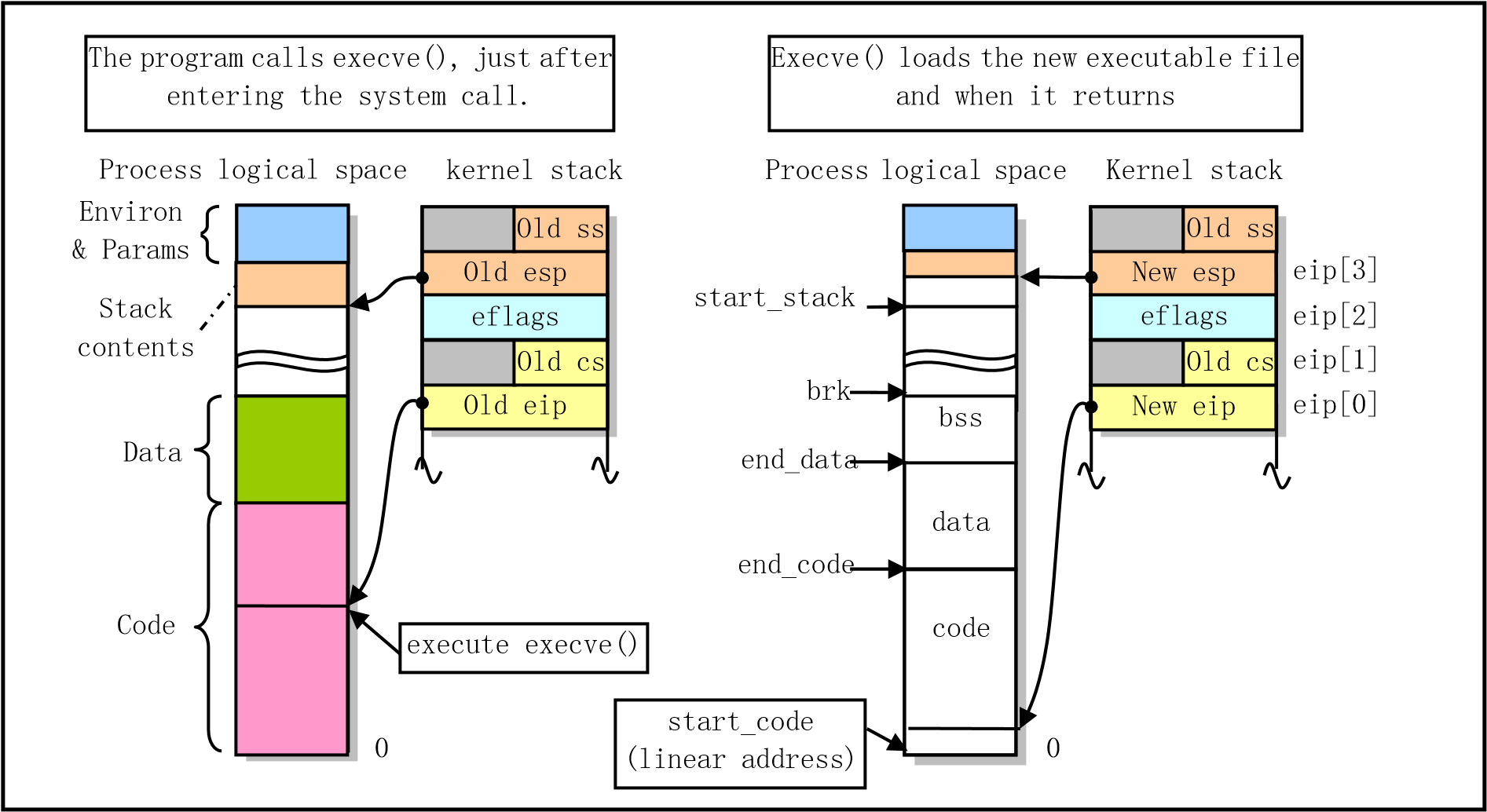


図 12-34 実行ファイルの読み込み時のスタック内のespおよびeipの変化

図の左半分は、プロセスロジックの64MB空間に元の実行プログラムが残っている場合、右半分は元の実行コードやデータが解放され、スタックやコードポインタが更新された場合です。図の斜線（カラー）の部分には、コードやデータの情報が入っています。プロセスタスク構造体のstart\_codeは、CPUの線形空間のアドレスであり、残りの変数値は、プロセス論理空間のアドレスである。

### 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

1. /\*
2. \* #!-checking implemented by tytso.
3. \*/

10

1. /\*
2. \* Demand-loading implemented 01.12.91 - no need to read anything but
3. \* the header into memory. The inode of the executable is put into 14 \* "current->executable", and page faults do the actual loading. Clean.
4. \*
5. \* 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、シグナル操作関数のプロトタイプを定義しています。

// <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。

// <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 stat{} // と定数が含まれています。 // 定数が含まれています。

// <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ファイルです。ファイルテーブルの構造（file,buffer\_head, // m\_inodeなど）を定義します。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <linux/mm.h> メモリ管理のヘッダファイルです。ページサイズの定義と、いくつかのページ // リリース関数のプロトタイプが含まれています。

// <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。

1. 20 #include <signal.h> 21 #include <errno.h>
2. #include <string.h>
3. #include <sys/stat.h>
4. #include <a.out.h>

25

1. #include <linux/fs.h>
2. #include <linux/sched.h>
3. #include <linux/kernel.h>
4. #include <linux/mm.h>
5. #include <asm/segment.h>

31

1. extern int sys\_exit(int exit\_code); // kernel/exit.c, line 365.
2. extern int sys\_close(int fd); // fs/open.c, line 219.

34

1. /\*
2. \* MAX\_ARG\_PAGES defines the number of pages allocated for arguments 37 \* and envelope for the new program. 32 should suffice, this gives 38 \* a maximum env+arg of 128kB !
3. \*/
4. #define MAX\_ARG\_PAGES 32

41

//// 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。

// プロセスのライブラリファイルを選択し、プロセスの現在のライブラリファイルの // i-nodeフィールドを、ここで指定されたライブラリファイルのi-nodeで // 置き換えます。パラメータの指定されたライブラリが空の場合、 // プロセスの現在のライブラリファイルが解放されます。

1. // 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。
2. int sys\_uselib(const char \* library)
3. {
4. struct m\_inode \* inode;
5. 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

1. // TASK\_SIZE (64MB)となります。そのため、プロセスの論理アドレス空間のサイズがTASK\_SIZEと // 等しくない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。
2. if (get\_limit(0x17) != TASK\_SIZE)
3. return -EINVAL;
4. if (library) {
5. if (!(inode=namei(library))) /\* get library inode \*/
6. return -ENOENT;
7. } else
8. inode = NULL;
9. /\* 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

1. // 実行ファイルであるため、実際のライブラリコードは、実際に使用されるときに // メモリにロードされます。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置かれ、そのサイズは // 4MBの倍数となります（linux/sched.hファイルを参照）。
2. iput(current->library);
3. current->library = NULL;
4. base = get\_base(current->ldt[2]);
5. base += LIBRARY\_OFFSET; // linux/sched.h, line 26.
6. free\_page\_tables(base,LIBRARY\_SIZE);
7. current->library = inode;
8. return 0;
9. }

63

1. /\*
2. \* create\_tables() parses the env- and arg-strings in new user 66 \* memory and creates the pointer tables from them, and puts their 67 \* addresses on the "stack", returning the new stack pointer value.
3. \*/

//// 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。

1. // この関数は、スタックポインタを返します。
2. static unsigned long \* create\_tables(char \* p,int argc,int envc)
3. {
4. unsigned long \*argv,\*envp;
5. unsigned long \* sp;

73

// 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．

1. // 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ブロックポインタ、そしてコマンドラインパラメータの数をスタックにプッシュします。 // 以下のポインタプラス1の操作により、spが4バイト増加することに注意してください。
2. sp = (unsigned long \*) (0xfffffffc & (unsigned long) p);
3. sp -= envc+1; // That is sp = sp - (envc+1);
4. envp = sp;
5. sp -= argc+1;
6. argv = sp;
7. put\_fs\_long((unsigned long)envp,--sp); // put in user space.
8. put\_fs\_long((unsigned long)argv,--sp);
9. put\_fs\_long((unsigned long)argc,--sp);
10. // そして、コマンドラインのポインタと環境変数のポインタを // 上記の予約した場所に配置し、最後にNULLポインタを配置します。
11. while (argc-->0) {
12. put\_fs\_long((unsigned long) p,argv++);
13. while (get\_fs\_byte(p++)) /\* nothing \*/ ; // p points to next param string.
14. }
15. put\_fs\_long(0,argv);
16. while (envc-->0) {
17. put\_fs\_long((unsigned long) p,envp++);
18. while (get\_fs\_byte(p++)) /\* nothing \*/ ; // p points to next param string.
19. }
20. put\_fs\_long(0,envp);
21. return sp; // returns the current new stack pointer constructed. 93 }

94

1. /\*
2. \* count() counts the number of arguments/envelopes
3. \*/

//// 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．

1. // 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。
2. static int count(char \*\* argv)
3. {
4. int i=0;
5. char \*\* tmp;

102

1. if (tmp = argv)
2. while (get\_fs\_long((unsigned long \*) (tmp++))) 105 i++;



1. return i;
2. }

109

1. /\*
2. \* 'copy\_string()' copies argument/envelope strings from user
3. \* memory to free pages in kernel mem. These are in a format ready 113 \* to be put directly into the top of new user memory.
4. \*
5. \* Modified by TYT, 11/24/91 to add the from\_kmem argument, which specifies 116 \* whether the string and the string array are from user or kernel segments:
6. \*
7. \* from\_kmem argv \* argv \*\*
8. \* 0 user space user space 120 \* 1 kernel space user space
9. \* 2 kernel space kernel space
10. \*
11. \* We do this by playing games with the fs segment register. Since it
12. \* 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 - 環境空間のページポインタへの引数の配列; p - 引数テーブルの空間内の // オフセットポインタで，常にコピーされた文字列のヘッダを指す; // from\_kmem - 文字列のソースフラグ。

// 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、pポインタはコピー情報の増加に伴って徐々に減少し、 // 常にパラメータ文字列の先頭を指すようになる。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能力を与えるために TYT(Tytso) が追加した新しいパラメータであるべきである。 // execve()がスクリプトファイルを実行する能力を持っていない場合、全てのパラメータ文字列は、 // ユーザーデータ空間にある。

1. // 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。
2. static unsigned long copy\_strings(int argc,char \*\* argv,unsigned long \*page,
3. unsigned long p, int from\_kmem)
4. {
5. char \*tmp, \*pag;
6. int len, offset = 0;
7. unsigned long old\_fs, new\_fs;

133

// まず、現在のセグメントレジスタDS（カーネルデータセグメントを指す）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とold\_fsに格納します。

1. 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。
2. if (!p)
3. return 0; /\* bullet-proofing \*/ // offset pointer verification
4. new\_fs = get\_ds();
5. old\_fs = get\_fs();
6. if (from\_kmem==2) // string and string pointers all in kernel space.
7. 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。

1. // (DS)で，文字列ポインタtmpをカーネルデータ空間に取り込んだ直後に， // FSを復元する。そうでない場合は，FSの値を変更することなく， // ユーザ空間からtmpへの文字列ポインタを直接取り込む。
2. while (argc-- > 0) {
3. if (from\_kmem == 1) // FS points to kernel space if ponters in it.
4. set\_fs(new\_fs);
5. if (!(tmp = (char \*)get\_fs\_long(((unsigned long \*)argv)+argc)))
6. panic("argc is wrong");
7. if (from\_kmem == 1) // if string pointers in kernel.
8. 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が文字列の終わりを指すようになる。文字列の長さが、この時点でパラメータ空間と環境空間に残っている自由長を // 超えた場合、空間が足りなくなる。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が返される。

1. // しかし、パラメータや環境変数が128KBであるため、このようなことは起こらないはずです。147 len=0; /\* ゼロパディングを忘れるな \*/ 148 do {.
2. len++;
3. } while (get\_fs\_byte(tmp++));
4. if (p-len < 0) { /\* this shouldn't happen - 128kB \*/
5. set\_fs(old\_fs);
6. return 0;
7. }

// 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、このような

1. 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'はこの関数の最初で0に初期化されているので、次の'(offset-1 < 0)'のチェックは確実に // 真となり、'offset'はページ範囲内の現在のpポインタに再設定されます。
2. while (len) {
3. --p; --tmp; --len;
4. if (--offset < 0) {
5. offset = p % PAGE\_SIZE;
6. 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ポインタがある空間メモリページがまだ存在しない場合， // 空きメモリページを申請して，ページポインタを配列に埋める必要があります。それと同時に、 // ページポインタの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。

1. 161 if (!(pag = (char \*) page[p/PAGE\_SIZE]) && 162 !(pag = (char \*) page[p/PAGE\_SIZE] = )
2. (unsigned long \*) get\_free\_page()))
3. return 0;
4. if (from\_kmem==2) // If string in kernel space, 166 set\_fs(new\_fs); // FS points to kernel space.

167

1. }

// 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

1. // メモリページの環境変数 'pag'.
2. \*(pag + offset) = get\_fs\_byte(tmp);
3. } 171 }
4. // 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，FS // セグメントレジスタの元の値が復元され， // パラメータ空間と環境空間にコピーされたパラメータのヘッダオフセットが返されます。
5. if (from\_kmem==2)
6. set\_fs(old\_fs);
7. return p;
8. }

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テーブル LDT 内のディスクリプタのセグメントベースアドレスと長さの制限を変更し、パラメータと環境空間ページをデータセグメントの最後に配置する。 // パラメータ： text\_size - ファイルヘッダの a\_text フィールドによって // 与えられるコードセグメントの長さの限界値; page - パラメータと環境空間のページポインタの配列です。

// データセグメントの制限値（64MB）を返します。

1. 177 static unsigned long change\_ldt(unsigned long text\_size,unsigned long \* page) 178 {。
2. unsigned long code\_limit,data\_limit,code\_base,data\_base;
3. 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

1. 読み込み中のプログラム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。つまり、 // 186,188行目のセグメントのベースアドレスを設定する2つのステートメントは、 // 冗長なので省略することができます。
2. code\_limit = TASK\_SIZE;
3. data\_limit = TASK\_SIZE;
4. code\_base = get\_base(current->ldt[1]); // include/linux/sched.h, line 277
5. data\_base = code\_base;
6. set\_base(current->ldt[1],code\_base);
7. set\_limit(current->ldt[1],code\_limit); 188 set\_base(current->ldt[2],data\_base);
8. set\_limit(current->ldt[2],data\_limit);
9. /\* 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

1. // セグメントをページごとに表示します。ライブラリファイルのコードはプロセス空間の最後を占め、 // 4MBの倍数になります。put\_dirty\_page()関数は、物理ページをプロセスロジック空間にマッピングするために使用されます。mm/memory.cファイルを参照してください。
2. \_\_asm\_\_("pushl $0x17\n\tpop %%fs"::);
3. data\_base += data\_limit - LIBRARY\_SIZE;
4. for (i=MAX\_ARG\_PAGES-1 ; i>=0 ; i--) {
5. data\_base -= PAGE\_SIZE;
6. if (page[i]) // put the page if exists.
7. put\_dirty\_page(page[i],data\_base);
8. }
9. return data\_limit; // return the length limit of data segment(64MB). 199 }

200

1. /\*
2. \* 'do\_execve()' executes a new program.
3. \*
4. \* NOTE! We leave 4MB free at the top of the data-area for a loadable 205 \* 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）、関数番号 \_\_NR\_execve によって呼び出される関数です。

// 関数の引数は、システムコールが実行された後、関数が呼び出されるまでの間、 // スタックに徐々にプッシュされる値です（kernel/system\_call.s、217行目）。 // 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 // // // // (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る。 // (3) 216行目で、この関数do\_execve()を呼ぶ前にスタックにプッシュされた // システム割り込みを呼ぶプログラムコードポインタeipを指す。

// 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 - 環境変数ポインタの配列へのポインタ。

1. // 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。
2. int do\_execve(unsigned long \* eip,long tmp,char \* filename,
3. char \*\* argv, char \*\* envp)
4. {

// 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)スペースのページポインターの配列です。 // 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。

1. // 218行目 - 引数＆環境空間の末尾を指すのに使われる整数。
2. struct m\_inode \* inode;
3. struct buffer\_head \* bh;
4. struct exec ex;
5. unsigned long page[MAX\_ARG\_PAGES]; // array of page pointers in A&E space.
6. int i,argc,envc;
7. int e\_uid, e\_gid; // effective user and group ID.
8. int retval;
9. int sh\_bang = 0; // control if we execute script file.
10. 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

実行ファイルのための // 環境文字列変数です。前の行では、pが128KB空間の最後のロングワードに初期配置されるように設定されています。初期化パラメータや環境変数の操作では、128KB空間の現在の位置を示すためにpが使用されます。

// 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。

1. // 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。そして、関数のパラメータに従って、 // コマンドラインパラメータの数 argc と、環境文字列の数 envc を計算します。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。
2. if ((0xffff & eip[1]) != 0x000f)
3. panic("execve called from supervisor mode");
4. for (i=0 ; i<MAX\_ARG\_PAGES ; i++) /\* clear page-table \*/
5. page[i]=0;
6. if (!(inode=namei(filename))) /\* get executables inode \*/
7. return -ENOENT;
8. argc = count(argv); // The number of command line arguments.
9. envc = count(envp); // The number of environment variables.

228

1. restart\_interp:
2. if (!S\_ISREG(inode->i\_mode)) { /\* must be regular file \*/
3. retval = -EACCES;
4. goto exec\_error2; // if not regular file, jump to line 376.
5. }

// 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

モードでは， // "set-group-id "フラグが設定される。この2つのフラグは主に， passwd などの特権ユーザ（スーパーユーザ root など）のプログラムを一般ユーザが // 実行できるようにするために使用される．

// 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

1. 実行ファイルのグループIDには // 実行プロセスのID（egid）が設定され、そうでなければ // カレントプロセスのegidが設定されます。ここで，決定された2つの値は，変数e\_uidとe\_gidに一時的に // 格納される。
2. i = inode->i\_mode;
3. e\_uid = (i & S\_ISUID) ? inode->i\_uid : current->euid; 236 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

// 実行ファイルグループのアクセス許可フラグが最下位3ビットになるように、 // プロセスでは、属性を3ビット右にシフトしています。そうでなければ，このときの // モードの最下位3ビットは，他のユーザの実行ファイルへのアクセス許可フラグとなる。

// 次に、現在のプロセスに実行ファイルを実行する権限があ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

// プロセスユーザーがスーパーユーザーでない場合、現在のプロセスに実行ファイルを実行する権限がないことを示します。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。

1. 237 if (current->euid == inode->i\_uid) 238 i >>= 6;
2. else if (in\_group\_p(inode->i\_gid))
3. i >>= 3;
4. if (!(i & 1) &&
5. !((inode->i\_mode & 0111) && suser())) {
6. retval = -ENOEXEC;
7. goto exec\_error2;
8. }

// 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， // 別のシェルプログラムを実行してこのスクリプトファイルを実行したりする。まず，実行ファイルの最初のブロックを // バッファブロックに読み込んで，バッファブロックのデータを ex 構造体にコピーする。 // 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ある。

// 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（もしあれば）を入力し， // 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， // インタプリタのコマンドライン引数空間に入れる。

// 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

// スクリプトファイルのインタープリタをカーネルで実行します。そして，スクリプトファイル名などのパラメータを設定した後， // インタプリタのi-nodeを取り出し，229行目にジャンプする。

// 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。

1. スクリプトファイルを確認して処理した後、 // 再度このフラグを立てます。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用され、 // そのような設定を繰り返す必要はありません。
2. if (!(bh = bread(inode->i\_dev,inode->i\_zone[0]))) {
3. retval = -EACCES;
4. goto exec\_error2;
5. }
6. ex = \*((struct exec \*) bh->b\_data); /\* read exec-header \*/
7. if ((bh->b\_data[0] == '#') && (bh->b\_data[1] == '!') && (!sh\_bang)) {
8. /\*
9. \* This section does the #! interpretation.
10. \* Sorta complicated, but hopefully it will work. -TYT
11. \*/

256

1. char buf[128], \*cp, \*interp, \*i\_name, \*i\_arg;
2. unsigned long old\_fs;

259

// ここから先は、スクリプトファイルからインタープリター名とそのパラメーターを抽出して

// インタープリタ名、そのパラメータ、スクリプトファイル名を環境パラメータに入れる

// ブロックを作成します。まず、スクリプトファイルの1行目にある文字'#!'の後の文字列をコピーします。

1. // このバッファには、スクリプトのインタープリタ名 (例: /bin/sh) と、 // インタープリタのいくつかのパラメータが含まれている。次に buf の内容を処理します。最初の改行を NULL で置き換え、 // スペースタブを削除します。
2. strncpy(buf, bh->b\_data+2, 127);
3. brelse(bh); // release the buffer block.
4. iput(inode); // put back the i-node of the script file.
5. buf[127] = '\0';
6. if (cp = strchr(buf, '\n')) {
7. \*cp = '\0'; // delete the space, tabs.
8. for (cp = buf; (\*cp == ' ') || (\*cp == '\t'); cp++);
9. }
10. if (!cp || \*cp == '\0') { // if the line contains nothing, error!
11. retval = -ENOEXEC; /\* No interpreter name found \*/
12. goto exec\_error1;
13. }

// 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。

1. // インタープリターです。この行の分析は以下の通りです。まず最初の文字列を取得します。これはインタープリタ名であるべきで、i\_nameはその名前を指しています。インタープリタ名の後に文字があれば、それは引数の文字列であるべきで、i\_argはその文字列を指している。
2. interp = i\_name = cp;
3. i\_arg = 0;
4. for ( ; \*cp && (\*cp != ' ') && (\*cp != '\t'); cp++) {
5. if (\*cp == '/')
6. i\_name = cp+1;
7. }
8. if (\*cp) {
9. \*cp++ = '\0'; // add a NULL to the end of interpreter name.
10. i\_arg = cp; // i\_arg points to the arguments.
11. }
12. /\*
13. \* OK, we've parsed out the interpreter name and 284 \* (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。コンテンツの1行目が「#!/bin/bash -iarg1 -iarg2」の場合、ここで1行目にパラメータを配置した後、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。コピーされないオリジナルの引数の1つは、オリジナルの実行ファイル名で、これはここでのスクリプトファイルの名前であり、以下で処理される // 。

// 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。

1. // とpはプログラムの引数文字列の最初を指します。また，copy\_strings()の最後のパラメータ // (0)は，パラメータ文字列がユーザ空間にあることを示しています。
2. if (sh\_bang++ == 0) {
3. p = copy\_strings(envc, envp, page, p, 0);
4. p = copy\_strings(--argc, argv+1, page, p, 0);
5. }
6. /\*
7. \* Splice in (1) the interpreter's name for argv[0]
8. \* (2) (optional) argument to interpreter
9. \* (3) filename of shell script
10. \*
11. \* This is done in reverse order, because of how the 296 \* user environment and arguments are stored.
12. \*/

// 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。

1. // 文字列がカーネル空間にある場合は、以下の301行目と304行目に示すように、copy\_strings()の最後のパラメータを2に設定する必要があります。
2. p = copy\_strings(1, &filename, page, p, 1);
3. argc++;
4. if (i\_arg) { // copy multiple arguments of the interpreter.
5. p = copy\_strings(1, &i\_arg, page, p, 2);
6. argc++;
7. }
8. p = copy\_strings(1, &i\_name, page, p, 2);
9. argc++;
10. if (!p) {
11. retval = -ENOMEM;
12. goto exec\_error1;
13. }
14. /\*
15. \* OK, now restart the process with the interpreter's inode.
16. \*/

// 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 // インタープリタを実行します。インタープリタのi-nodeを取得するためには，namei()関数を使う必要がありますが，この関数で使われるパラメータ（ファイル名）はユーザデータから取得しています。

// つまり，セグメントレジスタ FS が指す空間からである。そのため，namei()関数を呼び出す前に，一時的にFSがカーネルデータ空間を指すようにする必要があります．

1. // 関数は， namei() が戻った後，カーネル空間からインタープリタの名前を取得し， // FS のデフォルト設定を復元することができます．その後、restart\_interp (229行目)にジャンプして、新しい // 実行ファイル -- スクリプトファイルのインタープリタ -- を再処理します。
2. old\_fs = get\_fs();
3. set\_fs(get\_ds());
4. if (!(inode=namei(interp))) { /\* get executables inode \*/
5. set\_fs(old\_fs);
6. retval = -ENOENT;
7. goto exec\_error1;
8. }
9. set\_fs(old\_fs);
10. goto restart\_interp;
11. }

// 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

1. // (ZMAGIC)であるか、コードとデータの再配置部分が0でないか、(コード部＋データ部＋ヒープ)が50MB以上であるか、実行ファイルサイズが(コード部 //＋データ部＋シンボルテーブル＋実行ヘッダ)以下であるかのいずれかです。
2. brelse(bh);
3. if (N\_MAGIC(ex) != ZMAGIC || ex.a\_trsize || ex.a\_drsize ||
4. ex.a\_text+ex.a\_data+ex.a\_bss>0x3000000 ||
5. inode->i\_size < ex.a\_text+ex.a\_data+ex.a\_syms+N\_TXTOFF(ex)) {
6. retval = -ENOEXEC;
7. goto exec\_error2;
8. }

// 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

1. // ページ(1024バイト)の範囲内では実行できません。デマンドページング技術では、実行ファイルの内容をページ単位で読み込む必要があるため、 // 実行ファイルイメージのコードとデータは、ページ境界から開始する必要があります。
2. if (N\_TXTOFF(ex) != BLOCK\_SIZE) {
3. printk("%s: N\_TXTOFF != BLOCK\_SIZE. See a.out.h.", filename);
4. retval = -ENOEXEC;
5. goto exec\_error2;
6. }

// 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。

1. // プログラムの文字列です。実際には、pは128KBのパラメータ空間と環境空間のオフセット値なので、p=0の場合は、環境変数とパラメータ空間のページがすでに一杯であることを意味します。
2. if (!sh\_bang) {
3. p = copy\_strings(envc,envp,page,p,0);
4. p = copy\_strings(argc,argv,page,p,0);
5. if (!p) {
6. retval = -ENOMEM;
7. goto exec\_error2;
8. }
9. }
10. /\* OK, This is the point of no return \*/
11. /\* 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

// プロセスタスクの中にあります。最後に，このシステムコールを実行するプログラムの // リターンアドレスeip[]を実行ファイルのコードの先頭に指し示す。こうすることで， // システムコールが終了して戻ってきたときに， // 新しい実行ファイルのコードを実行することになります。

// 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、引数や環境ポインタのテーブルをユーザースタックに格納することで、メモリマネージャがユーザースタック空間の物理メモリページをマッピングするため、ページフォルト例外が発生します。

//

// 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

1. // ただし、SIG\_IGNハンドルについてはリセットする必要はありません。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、 // フラグをリセットします。
2. if (current->executable)
3. iput(current->executable);
4. current->executable = inode;
5. current->signal = 0;
6. for (i=0 ; i<32 ; i++) {
7. current->sigaction[i].sa\_mask = 0;
8. current->sigaction[i].sa\_flags = 0;
9. if (current->sigaction[i].sa\_handler != SIG\_IGN)
10. current->sigaction[i].sa\_handler = NULL;
11. }
12. for (i=0 ; i<NR\_OPEN ; i++)
13. if ((current->close\_on\_exec>>i)&1)
14. sys\_close(i);
15. 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

// 元のプログラムのコードとデータのセグメントに対応するページテーブルで指定された物理メモリページとページテーブル自体が解放されます。この時点で、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。

1. // 新しい実行ファイルのテーブルエントリを作成し，関連する実行ファイルページを // メモリに読み込みます。また，"last task used coprocessor "が現在のプロセスを指している場合は， // NULLが設定され，used mathフラグがリセットされます。
2. free\_page\_tables(get\_base(current->ldt[1]),get\_limit(0x0f));
3. free\_page\_tables(get\_base(current->ldt[2]),get\_limit(0x17));
4. if (last\_task\_used\_math == current)
5. last\_task\_used\_math = NULL;
6. 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。

1. // つまり、p はスタックポインタの値に変換されています。その後、内部関数の // create\_tables() を呼び出して、プログラムの main() をパラメータとして、 // スタック空間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し、 // そのスタックポインタを返します。
2. p += change\_ldt(ex.a\_text,page);
3. p -= LIBRARY\_SIZE + MAX\_ARG\_PAGES\*PAGE\_SIZE;
4. 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

1. // ヒープエンドフィールド brk = a\_text + a\_data + a\_bss. brkは、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）の // 終了位置を示すために使用され、カーネルは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、 // プロセスの実効ユーザーIDと実効グループIDを再設定します。
2. current->brk = ex.a\_bss +
3. (current->end\_data = ex.a\_data +
4. (current->end\_code = ex.a\_text));
5. current->start\_stack = p & 0xfffff000;
6. current->suid = current->euid = e\_uid;
7. 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

1. // リターン命令は、スタック上のデータをポップアップして、CPUに // 新しい実行ファイルを実行させるので、最初にシステム割込みを呼び出したプログラムには // 戻らない。
2. eip[0] = ex.a\_entry; /\* eip, magic happens :-) \*/
3. eip[3] = p; /\* stack pointer \*/ 375 return 0;
4. exec\_error2:
5. iput(inode); // put back the i-node. 378 exec\_error1:
6. for (i=0 ; i<MAX\_ARG\_PAGES ; i++)
7. free\_page(page[i]); // release memory pages if error occurs.
8. return(retval); // return error code.
9. }

383

## 12.16 stat.c

### 12.16.1 Function

stat.cプログラムは、ファイル・ステータス情報を取得するためのシステム・コール関数stat()およびfstat()を実装し、取得した情報をユーザー・バッファに格納する。ファイルのステータス情報は、以下のようなstat構造体に格納されており、すべてのフィールド情報は、ファイルのi-nodeから取得することができる。stat()はファイル名を、fstat()はファイルハンドル(ディスクリプタ)を用いて情報を取得する。ファイルinclude/sys/stat.hをご参照ください。

|  |  |  |  |
| --- | --- | --- | --- |
| 1. // 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。 2. struct stat { 3. dev\_t st\_dev; // device number that contains the file. | | | |
| 8 | ino\_t | st\_ino; | // file i-node number. |
| 9 | umode\_t | st\_mode; | // file type and mode. |
| 10 | nlink\_t | st\_nlink; | // number of links to the file. |
| 11 | uid\_t | st\_uid; | // user identification number of the file. |
| 12 | gid\_t | st\_gid; // group number of the file. | |
| 13 | dev\_t | st\_rdev; // device number (if the file is a char or block device file). | |
| 14 | off\_t | st\_size; // file size (bytes) if the file is a regular file. | |
| 15 | time\_t | st\_atime; | // last access time. |
| 16 | time\_t | st\_mtime; | // last modified time. |
| 17  18 }; | time\_t | st\_ctime; | // i-node last changed time. |
| 12.16.2 | **コードアノテーション**  Program 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が書かれています。 // <sys/stat.h> ファイルの状態を表す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 stat{} が含まれています。 // 定数を含みます。

// <linux/fs.h> ファイルシステムのヘッダファイルです。ファイルテーブルの構造（file, buffer\_head, // m\_inodeなど）を定義します。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

1. // <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。
2. #include <errno.h>
3. #include <sys/stat.h> 9
4. #include <linux/fs.h>
5. #include <linux/sched.h>
6. #include <linux/kernel.h>
7. #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。

1. // パラメータinodeはファイルのi-node、statbufはユーザーデータ空間にあるstatファイルの状態構造のポインタであり、取得した状態情報を格納するために使用される。
2. static void cp\_stat(struct m\_inode \* inode, struct stat \* statbuf)
3. {
4. struct stat tmp;
5. int i; 19
6. // 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， // 対応するファイルのi-nodeの情報を一時的にtmpにコピーする。最後に，これらのステータス情報を // ユーザバッファにコピーする。
7. verify\_area(statbuf,sizeof (struct stat));
8. tmp.st\_dev = inode->i\_dev; // device number that contains the file.
9. tmp.st\_ino = inode->i\_num; // file i-node number.
10. tmp.st\_mode = inode->i\_mode; // file type and mode.
11. tmp.st\_nlink = inode->i\_nlinks; // number of links to the file.
12. tmp.st\_uid = inode->i\_uid; // user id.
13. tmp.st\_gid = inode->i\_gid; // group id.
14. tmp.st\_rdev = inode->i\_zone[0]; // device no.(if the file is a char/block file).
15. tmp.st\_size = inode->i\_size; // file size (bytes).
16. tmp.st\_atime = inode->i\_atime; // last access time.
17. tmp.st\_mtime = inode->i\_mtime; // last modified time. 31 tmp.st\_ctime = inode->i\_ctime; // i-node last changed time. 32 for (i=0 ; i<sizeof (tmp) ; i++)

33 put\_fs\_byte(((char \*) &tmp)[i],i + (char \*) statbuf); 34 }.

35

//// 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。

1. // 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。
2. int sys\_stat(char \* filename, struct stat \* statbuf)
3. {
4. struct m\_inode \* inode;

39

1. // まず、ファイル名から対応するi-nodeを見つけ、次にi-nodeのファイルの状態情報をユーザバッファにコピーし、最後にi-nodeを戻します。
2. if (!(inode=namei(filename)))
3. return -ENOENT;
4. cp\_stat(inode,statbuf);
5. iput(inode);
6. return 0;
7. }

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する。ファイルパス名にシンボリックリンクのファイル名がある場合は、 // シンボリックファイル自体のステータス情報が取得され、リンクは踏まれません。

// 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。

1. // 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。
2. int sys\_lstat(char \* filename, struct stat \* statbuf)
3. {
4. struct m\_inode \* inode;

50

1. // 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ファイルであれば、 // リンクを辿りません。そして、i-nodeのファイルステータス情報をユーザバッファにコピーして、 // i-nodeを戻します。
2. if (!(inode = lnamei(filename))) // get the i-node without following the link.
3. return -ENOENT;
4. cp\_stat(inode,statbuf);
5. iput(inode);
6. return 0;
7. }

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 // ユーザーバッファポインタである。

1. // 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。
2. int sys\_fstat(unsigned int fd, struct stat \* statbuf)
3. {
4. struct file \* f;
5. 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り、 // ファイルのi-nodeを取得します。そして，i-nodeのファイルステータス情報をユーザバッファにコピーする。

1. // ファイルハンドルの値が，プログラム NR\_OPEN で開くことのできる最大ファイル数を // 超える場合，ハンドルのファイル構造体ポインタが NULL の場合，または対応するファイル構造体の i-node フィールドが NULL の場合，エラーが発生し，エラーコードが返されます。
2. if (fd >= NR\_OPEN || !(f=current->filp[fd]) || !(inode=f->f\_inode))
3. return -EBADF;
4. cp\_stat(inode,statbuf);
5. return 0;
6. }

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，シンボリックリンクが指すファイルの // パス名文字列）を読み込んで，ユーザバッファに格納します。バッファが小さすぎると， // シンボリックリンクの内容は切り捨てられます。

// 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。

1. int sys\_readlink(const char \* path, char \* buf, int bufsiz)
2. {
3. struct m\_inode \* inode;
4. struct buffer\_head \* bh;
5. int i;
6. char c; 75
7. // 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 // バッファサイズ bufsi は 1 から 1023 の間でなければならない。次に、シンボリックリンクファイルの i-node を取得し、 // ファイルのデータコンテンツの最初のブロックを読み込む。その後、i-nodeを戻す。
8. if (bufsiz <= 0)
9. return -EBADF; 78 if (bufsiz > 1023)
10. bufsiz = 1023;
11. verify\_area(buf,bufsiz);
12. if (!(inode = lnamei(path)))
13. return -ENOENT;
14. if (inode->i\_zone[0])
15. bh = bread(inode->i\_dev, inode->i\_zone[0]);
16. else
17. bh = NULL;
18. iput(inode);
19. // ファイルのデータが正常に読み込まれた場合、最大 bufsiz 文字がファイルからユーザバッファにコピーされ、NULL 文字はコピーされません。最後にバッファブロックを解放し、コピーされたバイト数を返す。
20. if (!bh)
21. return 0;
22. i = 0;
23. while (i<bufsiz && (c = bh->b\_data[i])) {
24. i++;
25. put\_fs\_byte(c,buf++);
26. }
27. brelse(bh);
28. return i;
29. }

98

## 12.17 fcntl.c

### 12.17.1 Function

fcntl.cプログラムは、ファイル制御システムコールfcntl()と、2つのファイルハンドル（ディスクリプター）複製システムコールdup()およびdup2()を実装しています。Dup2()は新しいハンドルの最小値を指定し、dup()は現在の最小値を持つ未使用のハンドルを返します。Fcntl()は、開いているファイルの状態を変更したり、ファイルハンドルをコピーしたりするのに使用します。ハンドルの複製操作は、主にファイルの標準入出力のリダイレクトやパイプ操作に使用されます。本プログラムで使用している定数記号の一部は，include/fcntl.hファイルで定義されています。このプログラムを読む際には、このヘッダーファイルも参照することをお勧めします。 関数 dup() および dup2() が返す新しいファイルハンドルは、コピーまたは複製される元のハンドルと同じファイルテーブルエントリを使用します。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()関数を使用して新しいハンドルを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。

File struct

0

1

2

3

mode flags

f\_count

f\_pos

f\_inode

inode info (file

mode, size, etc)

Memory i

-

node table (32 items)

inode\_table[NR\_INODE]

File list (64 items)

file\_table[NR\_FILE]

I

nfo only in

memory

Array

of

file

pointers in the

process

struct

M

emory i

-

node

structure

図12-35 dup(1)，dup2(1,3)関数実行後のファイル構造

また、本プログラムの内部関数dupfd()を見ると、dup()やdup2()関数で新たに作成されたファイルハンドルについては、フラグclose\_on\_execがクリアされる、つまりexec()クラスの関数が実行されても、dup()で作成されたファイルハンドルはクローズされないことがわかります。

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。パラメータに制御コマンドcmdを指定して、4つの機能を関数内に統合しています。 -cmd = F\_DUPFD、ファイルハンドルを複製します。このとき、fcntl()の戻り値は、第3パラメータで指定された値以上の新しいファイルハンドルです。新しく作成されたファイルハンドルは、元のハンドルと同じファイルエントリを使用しますが、その実行終了フラグビットはリセットされます。このコマンドの場合，関数dup(fd)はfcntl(fd, F\_DUPFD, 0)，関数dup2(fd, newfd)は "close(newfd); fcntl(fd, F\_DUPFD, newfd); " といった記述と同じです。

-cmd = F\_GETFD or F\_SETFD。これら2つのコマンドは、ファイルハンドルのフラグclose\_on\_execの対応するビットを読み込んだり設定したりするために使用されます。フラグを設定する場合、関数の第3引数にはフラグの新しいビット値が渡されます。

-cmd = F\_GETFL または F\_SETFL です。この2つのコマンドは，それぞれファイル操作やアクセスフラグを読み込んだり設定したりするために使用する。これらのフラグは、RDONLY、O\_WRONLY、O\_RDWR、O\_APPEND、O\_NONBLOCKです。

詳細は，include/fcntl.hファイルを参照してください。操作を設定する場合，関数の第3パラメータには，ファイル操作およびアクセスフラグの新しい値を指定し，O\_APPENDおよびO\_NONBLOCKフラグのみを変更することができます。

-cmd = F\_GETLK, F\_SETLK, F\_SETLKWのいずれか。これらのコマンドは、ファイルロックフラグの読み取りまたは設定に使用されますが、Linux 0.12カーネルでは、ファイルレコードロック機能は実装されていません。

### 12.17.2 Code annotation

プログラム 12-16 linux/fs/fcntl.c

1. / 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> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。

// <fcntl.h> ファイル制御のヘッダファイルです。ファイルとそのディスクリプターに使用される操作制御の定数記号の // 定義です。

1. // <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 stat{} があります。 // と定数が含まれています。
2. #include <string.h>
3. #include <errno.h>
4. #include <linux/sched.h>
5. #include <linux/kernel.h>
6. #include <asm/segment.h>

12

1. #include <fcntl.h>
2. #include <sys/stat.h>

15

16 extern int sys\_close(int fd); // ファイルシステムコールを閉じる。(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は新しいファイルハンドルの // 最小番号を指定します。

1. // 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。
2. static int dupfd(unsigned int fd, unsigned int arg)
3. {

// 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

1. // ファイルハンドルは，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に注意してください。ファイルハンドルは，実際にはファイル構造ポインタ配列のアイテムインデックス番号であることに注意してください。
2. if (fd >= NR\_OPEN || !current->filp[fd])
3. return -EBADF;
4. if (arg >= NR\_OPEN)
5. return -EINVAL;
6. // 次に、プロセスのファイル構造ポインタ配列の中から、 // arg と同じかそれ以上のインデックス番号を持ち、 // まだ使用されていないエントリを探します。見つかった新しいハンドルが、 // 開いているファイルの最大数 NR\_OPEN を超える場合 (つまり、空きアイテムがない場合) は、エラーコードが返されます。
7. while (arg < NR\_OPEN)
8. if (current->filp[arg])
9. arg++; 27 else
10. break;
11. if (arg >= NR\_OPEN)
12. 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

1. dup()を用いた場合、 // クローズされません。その後，ファイル構造体ポインタは元のハンドル fd へのポインタと // 等しくなり，ファイル参照カウントが 1 つ増加します。最後に， // 新しいファイルハンドル arg が返される。
2. current->close\_on\_exec &= ~(1<<arg);
3. (current->filp[arg] = current->filp[fd])->f\_count++;
4. return arg;
5. }

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を複製し、新しいファイルハンドルの値はnewfdと同じになります。newfd // がすでに開いている場合は、先に閉じます。

// 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。

1. // 新しいファイルハンドルを返します。
2. int sys\_dup2(unsigned int oldfd, unsigned int newfd)
3. {
4. sys\_close(newfd); // closed the newfd first if already opened.
5. 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の // 未使用のハンドル値になります。

1. // パラメータ: 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。
2. int sys\_dup(unsigned int fildes)
3. {
4. return dupfd(fildes,0);
5. }

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値である。 // ファイルの設定操作やアクセスフラグコマンド F\_SETFL, arg は新しいファイル操作やアクセスモードである。 // ファイルロックコマンド F\_GETLK, F\_SETLK, F\_SETLKW の場合、 arg は flock 構造体へのポインタである。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。

1. // 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は新しいファイルハンドルを返し、F\_GETFDはファイルハンドルのフラグclose\_on\_execを返し、F\_GETFLはファイル操作とアクセスフラグを返します。
2. int sys\_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)
3. {
4. 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。

1. // を異なるコマンド cmd に変更します。ファイルハンドルの値がプロセスのオープンファイルの最大数 NR\_OPEN よりも大きい場合や、ハンドルのファイル構造体ポインタが NULL の場合は、 // エラーコードが返されます。
2. if (fd >= NR\_OPEN || !(filp = current->filp[fd]))
3. return -EBADF;
4. 12.18 ioctl.c
5. switch (cmd) {
6. case F\_DUPFD: // duplicate file handle.
7. return dupfd(fd,arg);
8. case F\_GETFD: // get the close\_on\_exec flag bit for the file.
9. return (current->close\_on\_exec>>fd)&1;
10. case F\_SETFD: // set/reset close\_on\_exec flag. set if arg is 1.
11. if (arg&1)
12. current->close\_on\_exec |= (1<<fd);
13. else
14. current->close\_on\_exec &= ~(1<<fd);
15. return 0;
16. case F\_GETFL: // get the file status flag and access mode.
17. return filp->f\_flags;
18. case F\_SETFL: // set status and access mode (append, non-block).
19. filp->f\_flags &= ~(O\_APPEND | O\_NONBLOCK);
20. filp->f\_flags |= arg & (O\_APPEND | O\_NONBLOCK);
21. return 0;
22. case F\_GETLK: case F\_SETLK: case F\_SETLKW: // not implemented. 71 return -1;
23. default:
24. return -1;
25. }
26. }

76

## 12.18 ioctl.c

### 12.18.1 Function

### ioctl.cプログラムは，入出力制御システム-コールioctl()を実装しています。ioctl()関数は，特定のデバイスドライバごとのインタフェース制御関数と考えることができます。この関数は，ファイルハンドルで指定されたデバイスファイルのドライバ内のIO制御関数を呼び出し，主にttyキャラクタデバイスのtty\_ioctl()関数を呼び出して端末のI/Oを制御する。ttyデバイスのプロパティは、通常、POSIX.1標準で定義されたtermios関連関数を使用する際に、ユーザープログラム内で設定することができます（include/termios.hファイルの最後の部分を参照）。それらの関数（tcflow()など）はコンパイルされたライブラリ libc.a に実装されており、プログラム中の ioctl()関数はシステムコールを通じて実行されたままになっています。

### 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。

12.18 ioctl.c

// <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 stat{} があります。 // と定数が含まれています。

1. // <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。
2. #include <string.h>
3. #include <errno.h>
4. #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

1. // Ioctl操作関数ポインタテーブル。
2. static ioctl\_ptr ioctl\_table[]={
3. NULL, /\* nodev \*/
4. NULL, /\* /dev/mem \*/
5. NULL, /\* /dev/fd \*/
6. NULL, /\* /dev/hd \*/
7. tty\_ioctl, /\* /dev/ttyx \*/
8. tty\_ioctl, /\* /dev/tty \*/
9. NULL, /\* /dev/lp \*/
10. 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記述子が有効かどうかを判断し， // i-nodeのファイルモードに従ってファイルタイプをチェックし， // 特定のファイルタイプに従って関連するio処理関数を呼び出す．

// 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。

1. // 戻り値：成功すれば0、そうでなければエラーコードを返します。 31 int sys\_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
2. {
3. struct file \* filp;
4. 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。そのファイルハンドルが

// 開放可能なファイルの数が少ない場合や，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

1. // プロセスがパイプを操作する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します。実行権限がある場合はPipe\_ioctl()が呼ばれ，そうでない場合は // 無効なファイルのエラーコードが返される。
2. if (fd >= NR\_OPEN || !(filp = current->filp[fd]))
3. return -EBADF;
4. if (filp->f\_inode->i\_pipe)
5. 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

1. // ファイルの場合，エラーコードが返される。キャラクタデバイスやブロックデバイスのファイルの場合は， // ファイルのi-nodeからデバイス番号を取得する。デバイス番号がシステム内のデバイス数よりも大きい場合は， // エラー番号が返される。
2. mode=filp->f\_inode->i\_mode;
3. if (!S\_ISCHR(mode) && !S\_ISBLK(mode))
4. return -EINVAL;
5. dev = filp->f\_inode->i\_zone[0]; // device no. for device type file.
6. if (MAJOR(dev) >= NRDEVS)
7. return -ENODEV;
8. // そして，IO制御テーブルioctl\_tableに従って， // 対応するデバイスのioctl関数ポインタを見つけ，その関数を呼び出す。デバイスがテーブルの中に対応する関数を // 持っていない場合は，エラーコードが返される。
9. if (!ioctl\_table[MAJOR(dev)])
10. return -ENOTTY;
11. return ioctl\_table[MAJOR(dev)](dev,cmd,arg);
12. }

50

## 12.19 select.c

### 12.19.1 Function

Linuxプログラマは、データストリームが断続的に送信されるI/Oデバイスにアクセスするために、複数のファイル記述子を同時に使用する必要があると感じることがよくあります。このような状況に対応するために、複数のread()、write()コールを使用するだけでは、そのうちの1つのコールがブロックされ、プログラムがファイルディスクリプターを待機させてしまう可能性があります。同時に、他のファイルディスクリプターが読み書きできるようになっていても、処理が間に合わないことがあります。

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することができます。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);

この関数は5つのパラメータを使用します。最初のパラメータwidthは、後述する3つのディスクリプターセットの中で最大のディスクリプターの値に1を加えたものです。この値は実際には、カーネルコードがディスクリプターの数をチェックするための値の範囲である。次の3つのパラメータは、ファイルディスクリプターセット型のfd\_setへのポインタで、読み取り操作のディスクリプターセットreadfds、書き込み操作のディスクリプターセットwritefds、例外条件が発生するディスクリプターセットexceptfdsを指している。これらの3つのポインタはいずれもNULLにすることができ，対応するセットに関心がないことを示す．3つのポインタがすべてNULLの場合は，select()関数をより正確なタイマーとして使用することができます（sleep()関数は第2レベルの精度しか得られません）．

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。

...

fd31

fd0

fd1

fd2

fd3

fd30

0

0

0

1

0

0

...

fd\_set

width = 4

1

1

1

1

0

0

...

mask

0

0

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

読み込み操作を行うディスクリプターfd3を監視する必要がある場合は、readfdsセットのfd3（ビット4）を1に設定し、書き込み操作を行うファイルディスクリプターfd1を監視する必要がある場合は、writefdsセットのfd1を1に設定する必要がある。 fd3がすべてのディスクリプターセットの中で最大のディスクリプター値である場合、第1パラメータの幅は4となる。fd\_set型変数の操作を容易にするために，Linuxシステムでは，include/sys/time.hというファイルで定義されている以下のような4つのマクロが用意されている。これらのマクロは，ディスクリプタセットのクリア，ディスクリプタセット内の任意のディスクリプタのビットのセット/リセット/検出に使用される。

#define FD\_ZERO(fdsetp) (\*(fdsetp) = 0) // ディスクリプタセットの全ビットをゼロにします。#define FD\_SET(fd,fdsetp) (\*(fdsetp) |= (1 << (fd))) // ディスクリプタのビットをセットします。

#define FD\_CLR(fd,fdsetp) (\*(fdsetp) &= ~(1 << (fd))) // ディスクリプタのビットをリセットします。

#define FD\_ISSET(fd,fdsetp) ((\*(fdsetp) >> fd) & 1) // ディスクリプタのビットをテストする。

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。これは，以下のような timeval 型（include/sys/time.h ファイルで定義されている）の構造体へのポインタである。

struct timeval { long tv\_sec; /\* seconds \*/ long tv\_usec; /\* microseconds \*/ };

パラメータのタイムアウト・ポインタがNULLの場合は、ディスクリプタ・セットで指定されたディスクリプタのいずれかが動作可能になるまで無期限に待機することを意味します。ただし，プロセスがシグナルを受信した場合には，待機処理は中断され，select()は-1を返し，グローバル変数errnoにはEINTRが設定されることになります。 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、待ちがないことを意味します。この時点で、select()関数を使用して、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ます。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もあり、-1を返す。 一般に、select()が-1を返した場合はエラーを示し、select()が0を返した場合は指定された条件でディスクリプタの準備ができていないことを示し、select()が正の値を返した場合はディスクリプタセット内でアクセス可能なファイルディスクリプタの数を示す。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。

1. #define \_\_LIBRARY\_\_
2. #include <sys/time.h>
3. #include <unistd.h>

04

05 int select(int nd, fd\_set \* in, fd\_set \* out, fd\_set \* ex, struct timeval \* tv) 06 {。

1. // まず、戻り値の変数 \_\_res を定義し、レジスタ変数 \_\_fooebx // を最初のパラメータへのポインタとして定義します。次に、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、 // eax = select システムコール関数番号; ebx は最初のパラメータの nd ポインタです。
2. long \_\_res;
3. register long \_\_fooebx \_\_asm\_\_ ("bx") = (long) &nd;
4. \_\_asm\_\_ volatile ("int $0x80"
5. : "=a" (\_\_res)
6. : "0" (\_\_NR\_select),"r" (\_\_fooebx));
7. // 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 を設定してから -1 を返します。
8. if (\_\_res >= 0)
9. return (int) \_\_res;
10. errno = -\_\_res;
11. return -1;
12. }

実際、select.cのプログラムはもっと複雑です。プログラムの27行目でリーナス氏が言ったように "If you understand what I'm doing here, then you understand how the Linux sleep/wakeup mechanism works." kernel/sched.cプログラムと同様に、このプログラムでもadd\_wait()関数とfree\_wait()関数の理解が最大の難関です。この2つの関数の動作原理を理解するためには、sched.cプログラムのsleep\_on()関数を参考にするとよいでしょう。これらの関数は、いずれもあるリソースのタスク待ち行列の処理に関わっているからです。以下では、まずsys\_select()システムコールの主な動作原理を説明し、次にselect()がどのように待ち行列を処理するかを詳しく説明します。

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。

有效项

nr

entry[0]

entry[1]

entry[2]

entry[3]

entry[59]

wait\_table

Total

NR

\_OPEN

\*3

wait entry

old\_task

wait\_address

図12-37 ウェイトテーブル構造図

waitテーブルは、add\_wait()関数とfree\_wait()関数を使って動作します。記述子の準備ができていないとき、add\_wait()は、記述子に対応するタスク待ち行列に現在のプロセスを追加するために使用されます。待ち行列に項目を追加する前に、まず、追加したい待ち項目と同じ待ち項目を持つ待ち行列ヘッダポインタフィールドをwaitテーブルの中で検索します。すでに存在している場合は、待ち行列に追加されずにそのまま戻ります（つまり、異なる待ち行列に1つの待ち項目だけが挿入されます）。そうでない場合は、待ち行列の項目のwait\_addressフィールドが待ち行列のヘッドポインタを指し、old\_taskフィールドが待ち行列のヘッドポインタがもともと指していたタスクを指します。そして、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が与えられている（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。

proc\_list

wait\_address

old\_task

Old wait

task

proc\_list

W

aiting queue

head pointer

c

urrent

task

Old

wait

task

(

)

a

(

b

)

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

sleep\_on()関数が呼び出されたために、すでに待ち行列に入っている待ちタスクが挿入され、select関数を呼び出している現在のタスクを待ち行列に挿入した後に、別のプロセスがsleep\_on()関数を使って挿入したとします。このとき、待ち行列全体の構造は、図12-39のようになります。

W

aiting queue

head pointer

wait\_address

old\_task

proc\_list

select

task

task3

Old wait

task2

tmp

ptr

Old wait

task1

NULL

New wait

task4

tmp

ptr

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

図からわかるように、waitテーブルのエントリold\_taskのポインタフィールドは、sleep\_on()関数のtmpポインタと全く同じであり、wait\_addressフィールドは、waitテーブルのwait\_tableに同じwait queueポインタを持つエントリが追加されるのを防ぐためにselectにのみ使用されています。したがって、free\_wait()関数を使用してwaitテーブルのアイテムをクリアする場合、free\_wait()で使用されるアルゴリズムは、タスクが起こされたときのsleep\_on()関数とまったく同じです。

待機資源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、待機キューのヘッドポインタが指すタスクが起こされます。そして、そのタスクは、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（state = 0）にして、自分自身を割り込み禁止の待ち状態にします。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覚醒して実行を開始するのを待ってから、自身を覚醒させるのです。そして、スケジューラーを再実行します。

### 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。

### 12.19.2 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ファイルです。ファイルテーブルの構造（file, buffer\_head, // m\_inodeなど）を定義します。

// <linux/kernel.h> カーネルのヘッダーファイルです。カーネルでよく使われる機能のプロトタイプ定義が含まれています。

// <linux/tty.h> ttyヘッダーファイルは、tty\_ioというシリアル通信のためのパラメータや定数を定義しています。

// <linux/sched.h> スケジューラのヘッダファイルは、タスク構造体task\_struct、初期タスク0のデータ、 // ディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロ文を定義しています。

// <asm/segment.h> セグメント操作用のヘッダーファイルです。セグメントレジスタの操作のために、埋め込みアセンブリ関数が定義されています。

// <asm/system.h> システムのヘッダーファイルです。ディスクリプタ／割込みゲートなどを定義／変更する組み込みアセンブリマクロが定義されている。

// <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 stat{} があります。 // と定数が含まれています。

// <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。

// <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。

1. // <signal.h> シグナルのヘッダファイルです。シグナルシンボル定数、シグナル構造体、シグナル操作関数のプロトタイプを定義しています。
2. #include <linux/fs.h>
3. #include <linux/kernel.h>
4. #include <linux/tty.h>
5. #include <linux/sched.h> 12
6. #include <asm/segment.h>
7. #include <asm/system.h>

15

1. #include <sys/stat.h>
2. #include <sys/types.h>
3. #include <string.h>
4. #include <const.h>
5. #include <errno.h>
6. #include <sys/time.h>
7. #include <signal.h>

23

1. /\*
2. \* Ok, Peter made a complicated, but straightforward multiple\_wait() function.
3. \* I have rewritten this, taking some shortcuts: This code may not be easy to
4. \* follow, but it should be free of race-conditions, and it's practical. If you
5. \* understand what I'm doing here, then you understand how the linux sleep/wakeup 29 \* mechanism works.
6. \*
7. \* Two very simple procedures, add\_wait() and free\_wait() make all the work. We
8. \* 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

1. typedef struct {
2. struct task\_struct \* old\_task;
3. struct task\_struct \*\* wait\_address;
4. } wait\_entry;

41

1. typedef struct {
2. int nr;
3. wait\_entry entry[NR\_OPEN\*3];
4. } 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，ディスクリプタに関連付けられた待ち行列のヘッダポインタです。例えば， // tty のリードバッファキューのウェイトキューヘッドポインタは 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ある。

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

1. 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ときには， // すぐに書き込みができるようにしなければならない。
2. if (!wait\_address)
3. return;
4. for (i = 0 ; i < p->nr ; i++)
5. if (p->entry[i].wait\_address == wait\_address)
6. 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

1. wait table entry の // old\_task フィールドは、 wait queue header // ポインタが指すタスクを指しており (無い場合は NULL)、次に wait queue header が現在のタスクを指すようにします。最後に、 // ウェイトテーブルのアイテムカウント値 nr を 1 ずつ増加させます。
2. p->entry[p->nr].wait\_address = wait\_address;
3. p->entry[p->nr].old\_task = \* wait\_address;
4. \*wait\_address = current;
5. p->nr++;
6. }

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

1. // 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、各待ち行列にある他のタスクを起こすために使用されます。これは kernel/sched.c の sleep\_on() 関数の // 後半部分とほぼ同じであり，その説明を参照してほしい．
2. static void free\_wait(select\_table \* p)
3. {
4. int i;
5. 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

1. キューヘッダが指す // をレディ状態（state = 0）にして、自分自身を // 割り込み不可能な待機状態、つまり、これらの後続のキューイングされたタスクが // 起こされるのを待って、このタスクを起こすために再度実行するように設定します。そこで、スケジューラーを再実行します。
2. for (i = 0; i < p->nr ; i++) {
3. tpp = p->entry[i].wait\_address;
4. while (\*tpp && \*tpp != current) {
5. (\*tpp)->state = 0;
6. current->state = TASK\_UNINTERRUPTIBLE;
7. schedule();
8. }

// 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

1. // がこの時点でNULLではなくタスクを指している場合は、 // キューにまだタスクがある（\*tppが空ではない）ので、タスクを準備状態にしてウェイクアップします。最後に、待ち行列のアイテム数フィールド nr を // 0 にして、待ち行列テーブルをクリアします。
2. if (!\*tpp)
3. printk("free\_wait: NULL");
4. if (\*tpp = p->entry[i].old\_task)
5. (\*\*tpp).state = 0;
6. }
7. p->nr = 0;
8. }

81

//// i-node に応じた tty を取得します。

1. // ファイルが，i-nodeに従ったキャラクタターミナルデバイスファイルであるかどうかを調べる．そうであれば， // そのtty構造体ポインタを返し，そうでなければNULLを返します．
2. static struct tty\_struct \* get\_tty(struct m\_inode \* inode)
3. {
4. int major, minor;

85

1. // 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 // 5（制御端末）または4でない場合はNULLを返す。
2. if (!S\_ISCHR(inode->i\_mode))
3. return NULL;
4. if ((major = MAJOR(inode->i\_zone[0])) != 5 && major != 4)
5. return NULL;

// メジャーデバイス番号が5であれば、プロセスのttyフィールドはその端末デバイス番号であり、 // そうでなければ、キャラクターデバイス番号のマイナーデバイス番号に等しい。

// 端末のデバイス番号が0より小さい場合は、プロセスが端末を制御していないか、 // 端末を使用していないことを示しているので、NULLを返し、そうでない場合は

1. // 対応する tty 構造体のポインタ。
2. if (major == 5)
3. minor = current->tty;
4. else
5. minor = MINOR(inode->i\_zone[0]);
6. if (minor < 0)
7. return NULL;
8. return TTY\_TABLE(minor);
9. }

98

1. /\*
2. \* The check\_XX functions check out a file. We know it's either
3. \* a pipe, a character device or a fifo (fifo's not implemented)
4. \*/

//// リードインレディを確認する。

// ファイル読み込み操作の準備ができているかどうか、つまり、端末のリードバッファのキューに読み込むべき文字が // 二次的にあるかどうか、または、パイプファイルが空でないかどうかを確認する。

1. // パラメータwaitはwaitテーブルのポインタ，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)
2. {
3. 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

1. // 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、 // パイプが空であるかどうかを判断します。パイプのi-nodeのi\_zone[0]とi\_zone[1]フィールドには、パイプの現在のヘッドポインタとテールポインタが格納されています。
2. if (tty = get\_tty(inode))
3. if (!EMPTY(tty->secondary))
4. return 1;
5. else
6. add\_wait(&tty->secondary->proc\_list, wait);
7. else if (inode->i\_pipe)
8. if (!PIPE\_EMPTY(\*inode))
9. return 1;
10. else
11. add\_wait(&inode->i\_wait, wait);
12. return 0;
13. }

119

//// チェックライトアウトの準備ができました。

// ファイル書き込み操作の準備ができているかどうか、つまり、端末の書き込みバッファキュー write\_q に // 空き場所があるかどうか、パイプファイルがいっぱいになっていないかどうかをチェックします。

1. // パラメータwaitは待機テーブルポインタ，inodeはファイルのi-nodeポインタです． // ディスクリプタが書き込める場合は1を，そうでない場合は0を返します。
2. static int check\_out(select\_table \* wait, struct m\_inode \* inode)
3. {
4. 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

1. // パイプラインに文字を書き込むための空きスペースがある。ある場合は1を返し、 // ない場合（パイプラインがいっぱい）は、現在のタスクをパイプラインの // i-nodeの待ち行列に追加して0を返します。
2. if (tty = get\_tty(inode))
3. if (!FULL(tty->write\_q))
4. return 1;
5. else
6. add\_wait(&tty->write\_q->proc\_list, wait);
7. else if (inode->i\_pipe)
8. if (!PIPE\_FULL(\*inode))
9. return 1;
10. else
11. add\_wait(&inode->i\_wait, wait);
12. return 0;
13. }

136

//// 異常な状態を確認する。

// ファイルが異常な状態にあるかどうかをチェックします。パイプファイルの場合、2つのパイプ記述子の一方または両方がこの時点でクローズされていれば // 1を返し、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て0を返す。

// 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．

1. // 137 static int check\_ex(select\_table \* wait, struct m\_inode \* inode)
2. {
3. struct tty\_struct \* tty;

140

1. if (tty = get\_tty(inode))
2. if (!FULL(tty->write\_q))
3. return 0;
4. else
5. return 0;
6. else if (inode->i\_pipe)
7. if (inode->i\_count < 2)
8. return 1;
9. else
10. add\_wait(&inode->i\_wait,wait);
11. return 0;
12. }

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

1. // そうでなければプロセスはスリープ状態になり、タイムアウトが切れるか、ディスクリプターが置かれた待ち行列のプロセスが目覚めたために、 // プロセスは継続して実行されることになります。
2. int do\_select(fd\_set in, fd\_set out, fd\_set ex,
3. fd\_set \*inp, fd\_set \*outp, fd\_set \*exp)
4. {
5. int count; // the number of ready descriptors
6. select\_table wait\_table;
7. int i;
8. 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。

1. は、 // ユーザーが与えたディスクリプターセットにある。有効な記述子は、パイプファイル記述子、 // キャラクターデバイスファイル記述子、FIFO記述子のいずれかであり、それ以外のタイプは、 // 無効な記述子としてEBADFエラーを返す。
2. mask = in | out | ex;
3. for (i = 0 ; i < NR\_OPEN ; i++,mask >>= 1) {
4. if (!(mask & 1)) // not in the descriptor set.
5. continue;
6. if (!current->filp[i]) // file not opened.
7. return -EBADF;
8. if (!current->filp[i]->f\_inode) // file i-node is null.
9. return -EBADF;
10. if (current->filp[i]->f\_inode->i\_pipe) // valid: a pipe file.
11. continue;
12. if (S\_ISCHR(current->filp[i]->f\_inode->i\_mode)) // valid: a char dev file.
13. continue;
14. if (S\_ISFIFO(current->filp[i]->f\_inode->i\_mode)) // valid: a FIFO file.
15. continue;
16. return -EBADF; // all the rest are invalid.
17. }

// 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われている

// 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、またはイン異常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。

1. // の条件を満たす。記述子が動作可能な状態であれば、関連する記述子セットに対応するビットが設定され、動作可能な記述子のカウント数が // 1ずつ増加する。183行目の「mask += mask」という記述は、「mask << 1」と同等です。
2. repeat:
3. wait\_table.nr = 0;
4. \*inp = \*outp = \*exp = 0;
5. count = 0;
6. mask = 1;
7. for (i = 0 ; i < NR\_OPEN ; i++, mask += mask) {
8. // 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にあり， // そのディスクリプターが読み取り操作の準備ができている場合，ディスクリプターセットの対応するビットを // 1に設定し，準備ができているディスクリプターのカウントを1つ増加させる。
9. if (mask & in)
10. if (check\_in(&wait\_table,current->filp[i]->f\_inode)) {
11. \*inp |= mask; // set the bit in the set.
12. count++; // ready number.
13. }
14. // 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ディスクリプターセットの中にあり， // そのディスクリプターが書き込み操作の準備ができている場合，ディスクリプターセットの対応するビットが // 1に設定され，準備ができているディスクリプターのカウントが1つ増加する。
15. if (mask & out)
16. if (check\_out(&wait\_table,current->filp[i]->f\_inode)) {
17. \*outp |= mask;
18. count++;
19. }
20. // このときチェックしたディスクリプタが異常ディスクリプタセットにあり，そのディスクリプタが // 異常な状態であれば，異常ディスクリプタセットの対応するビットを1にセットし， // 異常ディスクリプタのカウントを1つ増加させる。
21. if (mask & ex)
22. if (check\_ex(&wait\_table,current->filp[i]->f\_inode)) {
23. \*exp |= mask;
24. count++;
25. }
26. }

// 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、この時点で待ち受けているディスクリプターがある場合や、待ち受け時間が経過していない場合は、現在のプロセスの状態を割込み可能なスリープ状態にして、スケジューラを実行して他のタスクを実行します。

1. // カーネルがこのタスクを再びスケジューリングするときには、free\_wait()を呼び出して、関連する待ち行列のタスクの前後のタスクを起こし、リピートラベル(178行目)にジャンプして、気になるディスクリプターがあるかどうかを再検出します。
2. if (!(current->signal & ~current->blocked) &&
3. (wait\_table.nr || current->timeout) && !count) {
4. current->state = TASK\_INTERRUPTIBLE;
5. schedule();
6. free\_wait(&wait\_table); // the task is awakened and returned to here.
7. goto repeat;
8. }
9. // 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、あるいは待ち時間が経過して待つべきディスクリプターがない場合、free\_wait()を呼び出して待ち行列のタスクを // 起こし、準備の整ったディスクリプターの数を返します。
10. free\_wait(&wait\_table);
11. return count;
12. }

210

1. /\*
2. \* Note that we cannot return -ERESTARTSYS, as we change our input
3. \* parameters. Sad, but there you are. We could do some tweaking in 214 \* the library function ...

215 \*/ // パラメータ \*timeout は処理中に変更されます。

//// 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。

パラメータで指定さ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コピーし， // 待機時間を設定してから do\_select() を呼び出す。返却後，結果はユーザ空間に // コピーバックされる。

// 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．

1. // 返り値が0より小さい場合は，実行中にエラーが発生したことを示し， // 返り値が0に等しい場合は，指定された待ち時間内に操作可能なディスクリプターがないことを示し， // 返り値が0より大きい場合は，操作可能なディスクリプターの数を示している。
2. int sys\_select( unsigned long \*buffer )
3. {
4. /\* 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。

1. // ポインタの引数で渡される引数。
2. int i;
3. fd\_set res\_in, in = 0, \*inp; // read in fd set.
4. fd\_set res\_out, out = 0, \*outp; // write out fd set.
5. fd\_set res\_ex, ex = 0, \*exp; // abnormal fd set.
6. fd\_set mask; // descriptor value range (nd) mask code.
7. struct timeval \*tvp; // wait time structure pointer.
8. 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

// ディスクリプタのセット（すなわち，第1パラメータndの値）は，ユーザプログラムが関心を持つすべての // ディスクリプタのマスクに設定される。例えば，nd = 4 の場合， mask = 0b00001111 // (合計 32 ビット) となる．

1. 227 mask = ~((~0) << get\_fs\_long(buffer++)); 228 inp = (fd\_set \*) get\_fs\_long(buffer++);
2. outp = (fd\_set \*) get\_fs\_long(buffer++);
3. exp = (fd\_set \*) get\_fs\_long(buffer++);
4. tvp = (struct timeval \*) get\_fs\_long(buffer);

232

1. if (inp) // read in set.
2. in = mask & get\_fs\_long(inp);
3. if (outp) // write out set.
4. out = mask & get\_fs\_long(outp);
5. if (exp) // abnormal set.
6. 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， // 時間構造体の時間値がユーザー空間から取得され， // 現在の目盛りの値がjiffies of

// 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。

1. // 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1000000で割ると対応する秒数が得られ、これにシステムのtick // per second HZを掛けると、tv\_usecがtick値に変換されます。
2. timeout = 0xffffffff;
3. if (tvp) {
4. timeout = get\_fs\_long((unsigned long \*)&tvp->tv\_usec)/(1000000/HZ);
5. timeout += get\_fs\_long((unsigned long \*)&tvp->tv\_sec) \* HZ;
6. timeout += jiffies;
7. }
8. 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，処理結果をユーザデータ領域にコピーして， // ユーザに返すために使用されます。競合状態を避けるために、do\_select()を呼び出す前に割り込みを無効にし、 // 関数が戻った後に割り込みを有効にする必要があります。

// 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

1. // であれば、この値をユーザーに返します。プロセスの wait delay field timeout がすでに // 現在のシステムの jiffies 以下であれば、 do\_select() がタイムアウトで返される可能性があることを意味するので、残り時間の値を 0 に設定する。246 cli(); // intを無効にする。
2. i = do\_select(in, out, ex, &res\_in, &res\_out, &res\_ex);
3. if (current->timeout > jiffies)
4. timeout = current->timeout - jiffies;
5. else
6. timeout = 0;
7. 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。do\_select()が返したレディディスクリプターの数が // 0より少ない場合は、実行エラーを示しているので、エラー番号を返します。

// 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

1. // をユーザバッファ空間に書き出す。時刻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， // 秒とマイクロ秒の値に変換することも必要です。
2. current->timeout = 0;
3. if (i < 0)
4. return i;
5. if (inp) {
6. verify\_area(inp, 4);
7. put\_fs\_long(res\_in,inp); // readable descriptor set.
8. }
9. if (outp) {
10. verify\_area(outp,4);
11. put\_fs\_long(res\_out,outp); // writable descriptor set.
12. }
13. if (exp) {
14. verify\_area(exp,4);
15. put\_fs\_long(res\_ex,exp); // abnormal descriptor set.
16. }
17. if (tvp) {
18. verify\_area(tvp, sizeof(\*tvp));
19. 12.20 まとめ
20. put\_fs\_long(timeout/HZ, (unsigned long \*) &tvp->tv\_sec); // seconds.
21. timeout %= HZ;
22. timeout \*= (1000000/HZ);
23. put\_fs\_long(timeout, (unsigned long \*) &tvp->tv\_usec); // microseconds.
24. }
25. // 現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は， // 割り込みエラーの番号が返される。それ以外の場合は，レディディスクリプタの数が // 返される。
26. if (!i && (current->signal & ~current->blocked))
27. return -EINTR;
28. return i;
29. }

279

## 12.20 Summary

本章では、まず、MINIXのファイルシステムの構造と構成をあげて説明し、ファイルシステム内のディレクトリ構造、ディレクトリエントリ、ファイルパス名の構造を説明する。その後、Linuxにおける高速バッファ（バッファ・キャッシュ）の構造と使い方を詳しく説明し、カーネル内の他のプログラムがバッファ・ブロックを利用してブロック・デバイスにアクセスする方法を説明する。続いて、ファイルシステムコードの実装から、そのプログラムコードを、基盤となる一般的なファイルシステム機能、ファイルアクセス操作コード、ファイルアクセス・制御システムコールインタフェースの3つの側面から詳細に説明しています。

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。