

10文字デバイスドライバ

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	サイズ	最終更新日(GMT)	Desc.
 Makefile	3618 バイト	1992-01-12 19:49:17	
 コンソール.c	23327 バイト	1992-01-12 20:28:33	
 キーボード.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	
 シリアル.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 バイト	1992-01-11 04:02:37	

10.1 主な機能

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 ターミナルドライバーの基本原理

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

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

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

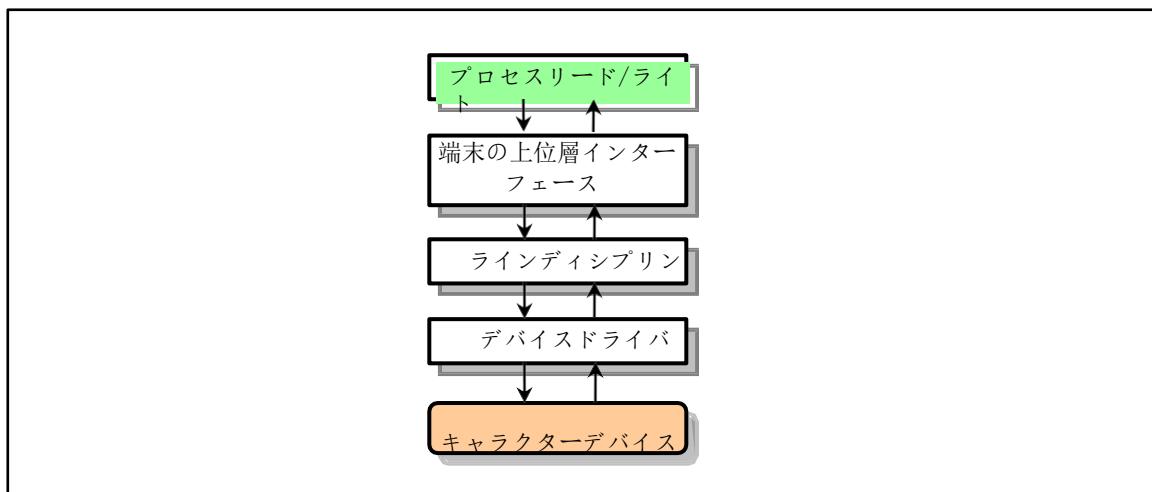


図10-1 端子ドライバ制御フロー

10.1.2 Linuxがサポートする端末機器の種類

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 ルート	tty	5,	07月30日	1992年	tty	// コントロール端子
crw-w-w-w	1 ルート	tty	4,	07月30日	1992年	tty0	// 仮想端末のエイリアスです。
crw-w-w-w	1 ルート	tty	4,	17月30日	1992年のコンソ	ール	// コンソール
crw-w-w-w	1 ルート	その他	4,	17月30日	1992年	TTY1	// バーチャルターミナル1
crw-w-w-w	1 ルート	tty	4,	27月30日	1992年	TTY2	
crw-w-w-w	1 ルート	tty	4,	37月30日	1992年	TTY3	
crw-w-w-w	1 ルート	tty	4,	47月30日	1992年	TTY4	
crw-w-w-w	1 ルート	tty	4,	57月30日	1992年	TTY5	
crw-w-w-w	1 ルート	tty	4,	67月30日	1992年	TTY6	
crw-w-w-w	1 ルート	tty	4,	77月30日	1992年	TTY7	
crw-w-w-w	1 ルート	tty	4,	87月30日	1992年	TTY8	
crw-rw-rw-	1 ルート	tty	4,	647月30日	1992年	ttys1	// シリアルポート端子1
crw-rw-rw-	1 ルート	tty	4,	657月30日	1992年	ttys2	
crw-w-w-w	1 ルート	tty	4,	1287月30日	1992年	ptyp0	// 一次疑似端末。
crw-w-w-w	1 ルート	tty	4,	1297月30日	1992年	ptyp1	
crw-w-w-w	1 ルート	tty	4,	1307月30日	1992年	ptyp2	
crw-w-w-w	1 ルート	tty	4,	1317月30日	1992年	ptyp3	
crw-w-w-w	1 ルート	tty	4,	1927月30日	1992年	ptyp0	// スレーブPTY

crw-rw-rw-	1 ルート	tty	4, 193	7月30日	1992年	ttyp1
crw-rw-rw-	1 ルート	tty	4, 194	7月30日	1992年	ttyp2
crw-rw-rw-	1 ルート	tty	4, 195	7月30日	1992年	ttyp3

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。

1. シリアルポート端子 (`/dev/ttysn`)

シリアルポート端末とは、コンピュータのシリアルポートを利用して接続される端末機器のことである。コンピュータはシリアルポートを1つのキャラクターデバイスとして扱う。当時はターミナルに接続することが主な用途であったため、しばらくの間、これらのシリアルポートデバイスはターミナルデバイスと呼ばれることが多い。これらのシリアルポートに対応するデバイスファイル名は、`/dev/ttys0`、`/dev/ttys1`などで、デバイス番号はそれぞれ(4, 64)、(4, 65)などで、DOSシステム下のCOM1、COM2に対応しています。ポートにデータを送信するには、コマンドラインで標準出力をこれらの特別なファイル名にリダイレクトします。例えば、「echo test>」と入力するとコマンドプロンプトで「`/dev/ttys1`」と入力すると、ttys1ポートに接続されたデバイスに「test」という単語が送信されます。

2. 疑似ターミナル (`/dev/ptyp`, `/dev/ttyp`)

擬似端末（またはPseudo - TTY、PTYと略す）とは、一般的な端末のように機能する装置であるが、その装置は端末のハードウェアとは一切関係がない。擬似端末装置は、他のプログラムに端末風のインターフェイスを提供するために使用され、主にネットワークを介してホストにログインする際に、ネットワークサーバーやログインシェルプログラムに端末風のインターフェイスを提供したり、X Windowウィンドウで動作する端末プログラムに端末風のインターフェイスを提供するために使用されます。もちろん、ターミナル・インターフェースを使用する任意の2つのプログラム間でデータの読み書きチャンネルを確立するために、疑似ターミナルを使用することもできます。2つのアプリケーションやプロセスにターミナル形式のインターフェイスを提供するために、疑似ターミナルはペアになっており、一方をマスター・ターミナルまたは疑似ターミナルマスターと呼び、他方をスレーブ・ターミナルまたは疑似ターミナルスレーブと呼ぶ。`ptyp1`と`ttyp1`のようにペアになった擬似端末の論理デバイスでは、`ptyp1`がマスターまたはコントロール端末、`ttyp1`がスレーブとなる。いずれかの疑似端末に書き込まれたデータは、カーネルを介してペアの疑似端末が直接受信します。例えば、マスター・デバイスの`/dev/ptyp3`とスレーブ・デバイスの`/dev/ttyp3`の場合、プログラムが`ttyp3`をシリアルポートデバイスとして扱うと、そのポートに対する読み書き操作が対応する論理端末デバイスの`ptyp3`に反映され、`ptyp3`はプログラムが読み書き操作に使用する別の論理デバイスとなります。このようにして、2つのプログラムがこの論理デバイスを介して相互に通信することができ、スレーブ・デバイス`ttyp3`を使用している一方のプログラムは、シリアルポートと通信しているとみなすことができます。これは、2つの論理デバイス間のパイプライン操作によく似ています。

擬似端子スレーブ・デバイスの場合は、シリアルポート・デバイスを使用するように設計されたプログラムであれば、ロジック・デバイスを使用することができますが、マスター・デバイスを使用するプログラムの場合は、擬似端子・マスター・デバイスを使用するように特別に設計されています。例えば、インターネット上で誰かがtelnetプログラムを使ってあなたのコンピュータに接続した場合、telnetプログラムは疑似端末・マスター・デバイス`ptyp2`への接続を開始し、gettyプログラムがその上で実行されるはずです。

対応する`ttyp2`ポートに送られます。telnetが相手側から文字を取り出すと、その文字は`ptyp2`と`ttyp2`を介してgettyプログラムに渡され、gettyプログラムは`ttyp2`、`ptyp2`、telnetプログラムを介して「login:」の文字列情報をネットワークに送ります。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

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。

3. コントロルターミナル (`/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。

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、`ioctl()`コールを `TIOCNNTTY`(Terminal IO Control NO TTY)

パラメータを使用して、コントロルターミナルを放棄することができます。

4. コンソール (`/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タイプのターミナル（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。

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、最初に接続される

仮想端末`tty1`。

5. その他の端末の種類

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 端末のデータ構造

```
struct tty_struct {  
    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バッファキューナードの情報を格納するために使用されます。  
    struct termios termios; // Terminal ioのプロパティと制御文字。  
    int pgrp; // 属しているプロセスグループ。  
    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。  
    void (*write)(struct tty_struct* tty); // tty書き込み関数へのポインタ。  
    struct tty_queue read_q; // tty read queue.  
    struct tty_queue write_q; // tty write queue.  
    struct tty_queue secondary; // ttyの補助キュー（または正規のキュー）です。  
};  
extern struct tty_struct tty_table[]; // tty構造体の配列です  
.
```

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`*read()`は、端末装置の入力処理関数である。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 {
    符号なしのロングデータ。                                // 現在のデータの統計。
    struct task_struct * proc_list;                      // バッファ内のデータのヘッダです。
    char buf[1024];                                       // データのテールポインタ。
                                                       // このバッファキューを待っているプロセス。
                                                       // 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、割込み処理中に呼び出されるC関数（`copy_to_cooked()`）が文字変換を行います。例えば、プロセスが端末にデータを書き込む場合、端末のドライバが正規モード関数`copy_to_cooked()`を呼び出し、ユーザバッファ内の全データをバッファキューに書き込み、そのデータを端末に送信して表示する。端末でキーが押されると、トリガされたキーボード割り込みハンドラは、キースキャンコードに対応する文字をリードキュー`read_q`に入れ、カノニカルモードハンドラを呼び出して`read_q`の文字を処理し、補助キュー`secondary`に入れます。同時に、端末装置のエコーフラグ（L_ECHO）が設定されている場合は、その文字も書き込みキュー`write_q`に入れ、端末の書き込み関数を呼び出して画面に表示します。通常、パスワードの入力など特殊な条件を除き、エコーフラグは設定されています。これらのフラグの値を変更するには、端末の`termios`構造体の情報を変更すればよい。

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

その内容は以下の通りです。

```
struct termios {
    符号化されていません。           入力モードフラグ /* */
    符号化されていない   char      入力モードフラグ
    c_line;  符号化されていな  /* 出力モードフラグ */
    い char c_cc[NCCS];          制御モードフラグ /* */
};                                制御モードフラグ           //またはス
                                    /* ローカルモードフラグ   ピード
                                    */
                                    /* ラインの規律 */
                                    制御文字 /* */ 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は、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す。また、ISIGフラグが設定されている場合は、端末から発行される制御文字INTR、QUIT、SUSPを受信する際に、システムが対応する信号を生成する必要があることを意味する。termios.hファイルの169~183行目を参照。

上記の4種類のフラグセットはすべて符号なしの長さで、各ビットがフラグを表すことができる所以、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)を修正することで

他のキーで生成された端末の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ます。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、端末がロードモードで動作するように設定できます。このとき、カノニカル・モード・ハンドラは、ユーザが入力したデータをそのままユーザに転送し、キャリッジ・リターンは通常の文字として扱われる。したがって、ユーザーが

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

の情報を提供します。

```
[/root]# stty
-----キャラクタ-----
INTR: '^C' QUIT: '^`' ERASE: '^H' KILL: '^U' EOF: '^D'
TIME: 0 : MIN 1 SWTC: '^@' START: '^Q' STOP: '^S'
SUSP: '^Z' EOL: '^D' EOL2: '^@' LNEXT: '^V' ^W
DISCARD: (^ディスクアンド) reprint: '^R' rwerase: '^W'
-----コントロールフラグ-----
-cstopb cread -parenb -parodd hupcl -clocal -crtscs
ボーレート9600 Bits:CS8
-----Input Flags-----
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr
 icnl -iuclc ixon -ixany ixoff -imaxbel
-----出力フラグ-----
OPOST -OLCUC ONLCR -ONOCR -ONLRET -OFILL -OFDEL
遅延モード。Cr0 n10 tab0 bs0 ff0 vt0
-----ローカルフラグ-----
isig icanon -xcase echo -echoe -echok -echonl -noflsh
-停止する echoctl echoprt echoke -flusho -pendin -iexten
行数 0 コル数 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。

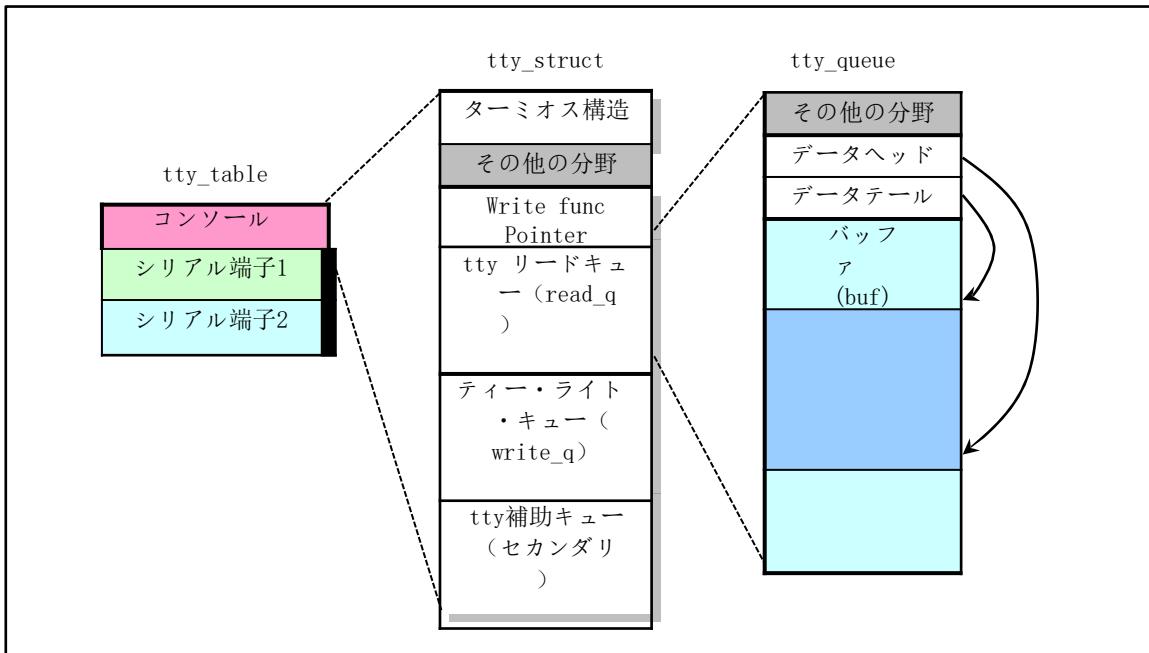


図10-2 ターミナルプログラムのデータ構造

4. 正統派モードと非正統派モード

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セッサ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。

1. キャノニカルモード

`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。`copy_to_cooked()`関数では、この文字と行末文字EOL、EOL2はキャリッジリターンとして扱われます。読み込み操作機能で遭遇した文字は

はすぐに戻ります。EOF文字はキューに入らず無視されます。

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。

2. 非正統派モード

`c_iflag`セットの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。

MINは、読み取り操作を行う際の最小文字数を、**TIME**は、文字の読み取りを待つタイムアウト値を指定します（単位は1/10秒）。それらによるとの値は、4つのケースで説明できます。

1. MIN>0, TIME>0

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）。

2. MIN>0, TIME=0

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）。

3. MIN=0, TIME=0

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。

4. 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 コンソールターミナルおよびシリアルターミナルデバイス

Linux 0.12では、ホスト上のコンソール端末と、シリアルハードウェア端末の2種類の端末が使用できるようになっている。コンソール端末は、カーネル内のキーボード割り込みハンドラプログラム`keyboard.s`と表示制御プログラム`console.c`によって操作・管理される。上位の`tty_io.c`プログラムから渡される表示文字や制御メッセージを受け取り、ホストの画面に文字を表示する制御を行う。同時に、

コンソール（ホスト）は、キーボードで生成されたコードを`keyboard.s`を介して`tty_io.c`プログラムに転送します。シリアル端末装置は、コンピュータのシリアルポートに回線を介して接続されており、

カーネル内のシリアルプログラム`rs_io.s`を介して`tty_io.c`と直接情報を交換しています。

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

この2つのプログラムが実装しているのは、シリアル端末機のROMに入っている端末処理プログラムの役割（通信部分を除く）に相当し、また、通常のPCの端末エミュレーションソフトのようなものでもあります。つまり、この2つのプログラムはカーネルの中にあるとはいえ、独立して見ることができます。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に通点については、図10-3を参照のこと。したがって、一般的なハードウェアの端末装置や端末エミュレーションプログラムの動作原理をある程度理解していれば、この2つのプログラムを読むことに何の困難も生じないだろう。

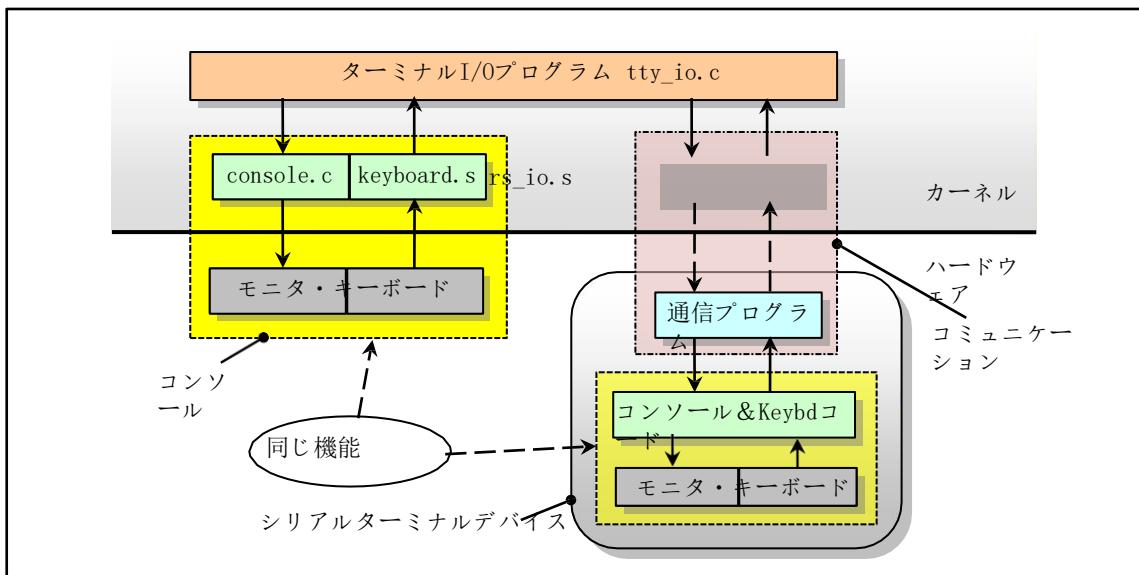


図10-3 コンソール端末とシリアル端末のデバイス図

10.1.5.1 コンソールドライバ

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、ユーザーがキーボードで文字を入力すると、キーボードの割り込み応答が発生します（割り込み要求信号IRQ1、割り込み番号INT33に対応）を使用して、ハンドラを実行します。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、使用している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`のリードキュー`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。

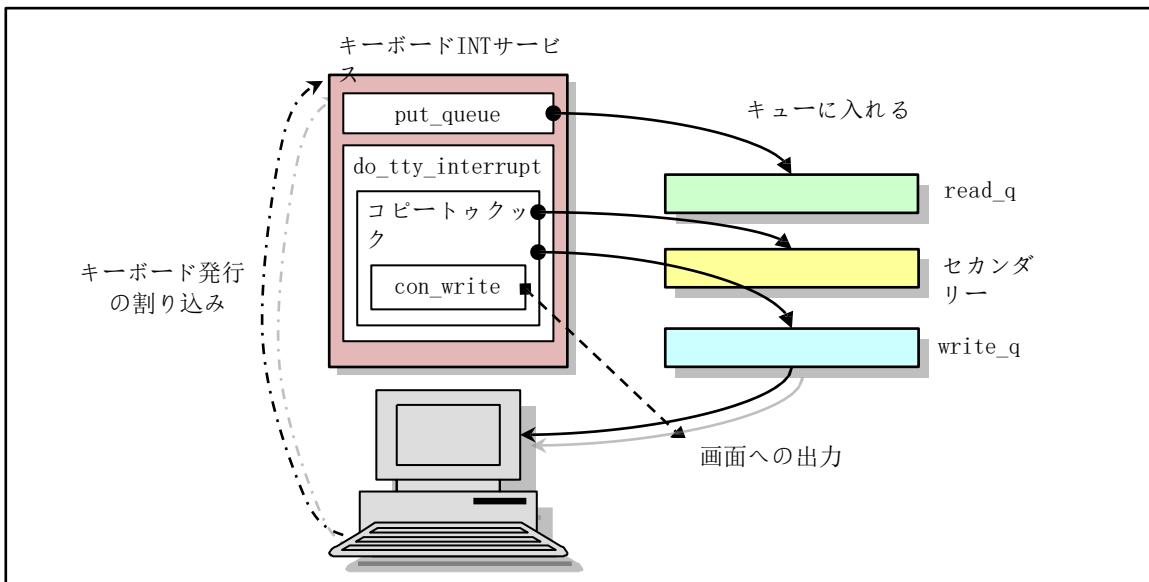


図10-4 コンソールキーボード割り込みサービス

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、この時点ですでに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にはつきりと示されています。

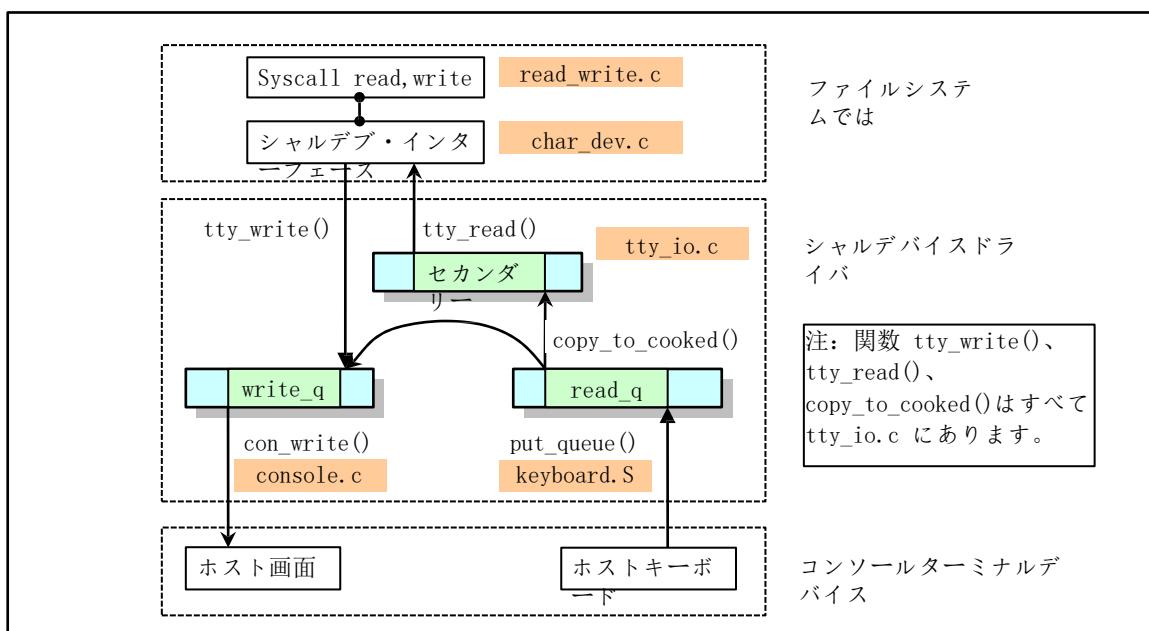


図10-5 コンソールキャラクターキューと機能の関係

10.1.5.2シリアルターミナルドライバ

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ブ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り込みは(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。文字を受信した場合、プログラムはまず、その文字をリード・バッファ・キューread_qに入れ、copy_to_cooked()関数を呼び出して、正統的なモードの文字を文字行単位で取り出して、補助キュー「secondary」に入れます。文字の送信が必要な場合には、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、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確認し、まだ文字がある場合には、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が割り込み要求を発行します。このとき、シリアルポートの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、C割り込みハンドラのdo_tty_interrupt()関数です。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several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。シリアル割り込み処理では、割込みの原因に応じて、書き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、送信ホールドレジスタエンブティイネーブル割り込みビットがマスクされます

これにより、このような割り込みが再び発生するのを防ぐことができます。

シリアル端子の書き込み関数rs_write()はserial.cプログラムに、シリアル割り込みプログラムはrs_io.sに実装されています。シリアル端子の3つのバッファキューと関数の関係は図10-6を参照してください。

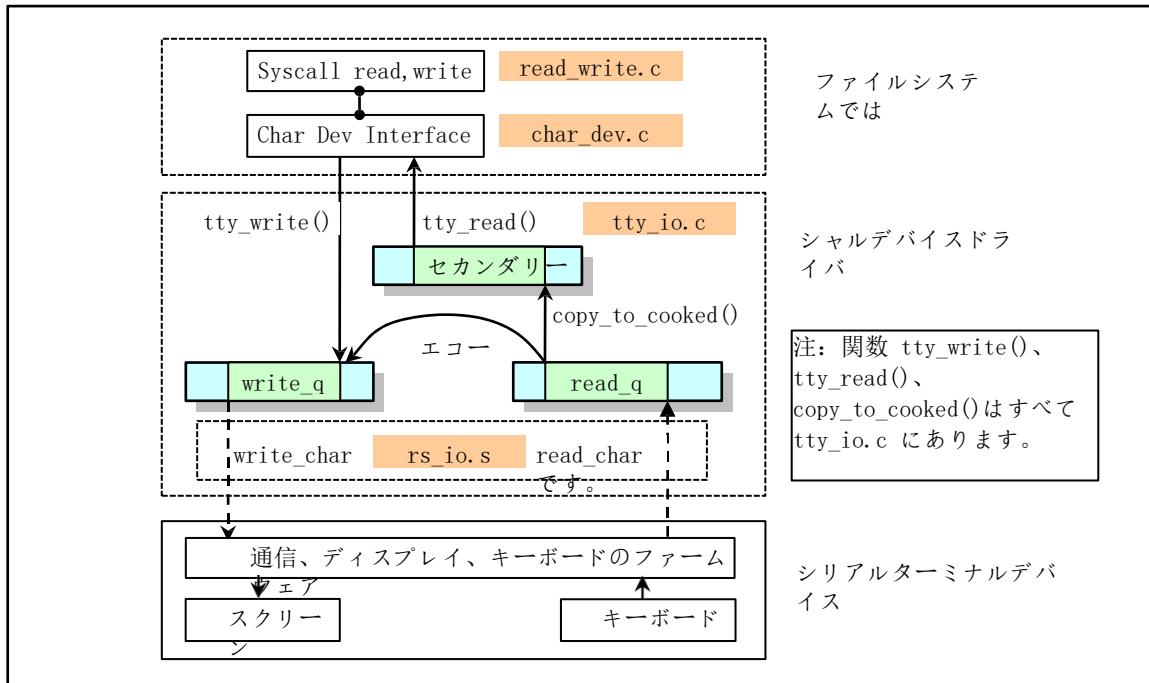


図10-6 シリアルターミナル機器のキャラクターキューと機能の関係

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、との処理方法は全く同じである。

10.1.6 ターミナル・ドライバー・インターフェース

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ーフェースは、`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。

2. キーボード.S

1. 機能説明

`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バイト目は時と同じスキャンコードになります。

が押されたキーである。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、アセンブラーが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 コードアナノテーション

プログラム 10-1 `linux/kernel/chr_drv/keyboard.S`

```

1  /*
2  * linux/kernel/keyboard.S 3
3  *
4  * (C) 1991 Linus Torvalds 5
5  */ (C) 1991 Linus Torvalds 6
6  * アメリカのキーボードパッチを提供してくれたAlfred
7  * Leung氏 ドイツのキーボードパッチを提供して
8  * くれたWolfgang Thiel氏 フランスのキーボー
9  * ドを提供してくれたMarc Corsini氏
10 */
11 */
12
13 /* フィンランド語キーボード用のKBD_FINNISH
14 * USタイプはKBD_US
15 * ドイツ語キーボード用のKBD_GR
16* KBD_FR (Frenchのキーボード用
17 */
18 // キャラクタマッピングのコードテーブルを後から選択する際に使用するキーボードの種類を定
19 // 義めぬKBD_FINNISH // リーナスはフィンランド人
20 .テキスト
21 // グローバル変数として宣言され、init時にキーボード割り込みディスクリプターを設定するた
22 めに使用されます。
23 .globl _keyboard_interrupt // keyboard_interrupt()と同じです。

```

```

22
23 /* 
24 * これらはキーボードの読み取り機能のためのも
のです。
25 // Sizeは、キーボードバッファのキューリの長さをバイト単
26 // 位で表します。/* must be a power of two !そして、tty_io.c !!!!*/
27 // 同じでなければなりません。*/
28 // キーボードバッファキューリデータ構造 tty_queue の各フィールドのオフセットを以下に示しま
す。
29 head<include/linux/tty.h, />int y24queue構造体のheadフィールドのオフセット。
30 tail = 8 // テールフィールドのオフセット
31 proc_list = 12 // 待機処理フィールドのオフセット（このバッファを待つ）。
32 buf = 16 // tty_queue構造体のバッファフィールドのオフセット。
33
34 // このプログラムでは、3つのフラグバイト（mode, leds, e0）を使用しています。各ビットの意
味は以下の通りです。
35 // 各バイトフラグは以下の通りです。
36 // (1) 'mode' は、特殊キーの押下状態フラグである。の状態を示すのに使われます。
37 // 大文字と小文字変換をcaps（）、代替キー←→（al右のCtrlキーが押された状態）、シフトキー。
38 // 、ビット6-capsキーの状態 態、bit 2 - 左のCtrlキーが押され
39 // 。た状態、bit 1 - 右のシフトキーが
40 // bit 5 - 右altキーを押したとき 押された状態、bit 0 - 左のシフト
41 // (2) 'leds' 4は、左altキーを押したとき表示が変更される状態。つまり
42 // num-lock, caps-lock, scroll-lockを示すLEDチップ。
43 // ビット7-3はすべてゼロ、使用し bit1 num-lock（初期設定1）。
44 // ません。ビット2はキャップスロ ビット0のスクロールロック。
45 // (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。
46 // を1文字または2文字のスキャンコードで受信します。通常、スキャンコード0xe0を受信した場合、
それは
47 // 8ビットを接続された端子を意味します。
モード続いて2文字であることと表示するため、番組表の説明をご覧ください。
48 // 34個の 0 /* num-lock, caps, scroll-lock mode (nom-lock on) */
49 LED. .バイト
50 e0: 2
51 .バイト
52 /* con_init は実在する割り込みルーチンで、以下のように読み込みます。
53 * キーボードのスキャンコードを適切な形式に変換します。
54 * ascii character(s).
55 */
56 // Note: 上記のオリジナルコメントは廃止されました。
57 ///// キーボード割り込みハンドラのエントリポイント。
58 // 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、インターフェース
59 // 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
60 // CPUが要求に応答したときに、//。割り込みハンドラは、キースキャンコードを
61 // キーボードコントローラポート(0x60)に対応するスキャンコードサブルーチンを呼び出して
62 // 処理を行います。
63 // 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。
64 // スキャンコードが0xe0または0xe1の場合。そうであれば、直ちにキーボードコントローラに応答す
る
65 // 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
66 // 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。
67 // 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
68 // 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、// 处理サブルーチンを行いま
す。

```

```

// スキャンコードに対応する文字を読み込み文字バッファのキューに入れる read_q.
// そして、キーボードコントローラに応答し、EOI信号を送信した後、関数
// do_tty_interrupt()を呼び出して（実際にはcopy_to_cooked()を呼び出して）文字を処理する
// read_q の中の // を、セカンダリの補助キューに入れます。
42_keyboard_interrupt:
43    プッシュル
44    %eax プッ
45    シュル
46    %ebx プッ
47    シュル
48    %ecx プッ
49    シュル          // dsとesをカーネルデータセグメントに設
50    %edx プッ      定します。
51    シュル %ds
52    プッシュル
53    %es            // 黒画面の時間カウント (blankinterval) を設定しま
54    movl $0x10,%eax // %eax はスキャンコード */
55    mov %ax,%ds      // スキャンコードをalに読み込ませる。
56    mov %ax,%es      // スキャンコードが0xe0の場合、set_e0
57    movl _blankinterval,%eaxにジャンプします。
58    movl %eax,_blankcount // スキャンコードが0xe1の場合、set_e1
59    xorl %eax,%eax // ヘジャンプ
60    .table$0xe60table(%eax,4) // キーハンドラ: key_table + eax*4 (513行目参照).
61    mmpb $0xe0,%alリターン後にe0フラグをリセット。
62    je set_e0 cmpb
63
// 次の $0xe1,%e1-72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を使つたキー一ポート回路です。ポート0x61は、8255Aの出力ポートBのアドレスです。ビット
このポートの//7 (PB7) は、キーボードデータの処理を無効にしたり有効にしたりするために使用さ
れます。このプログラム
//は、受信したスキャンコードに対応して、まずキーボードを無効にし、次に
64_60_既ぐ場合は$0x61,%alが使えるようにするポートBの状態を取得します。PB7はキーボードの有効化/無効
65_61_化に使用されます。
66_62_   jmp 1f          // 暫く待つ。
67_63_   1: jmp 1f        // alのbit7を設定
68_64_   1: orb $0x80,%al //します。
69_65_   jmp 1f          // ポートBのbit7を設定してキーボードを
70_66_   1: jmp 1f        // 無効にする。
71_67_   1: andb $0x7F,%al // alのbit7をリセットします。
72_68_   アウトプ %al,$0x61 // ポートBのbit7をリセットしてキーボード
73_69_   movb $0x20,%al // を有効にする。
74_70_   outb %al,$0x20 // EOI信号を8259Aチップに送る。
75_71_   pushl $0          // コンソール tty = 0 で、パラメータとしてプッシュ
76_72_   ロール          // されます。
77_73_   _do_tty_interrupt // カノニカルモードに変換
78_74_   addl $4,%esp     // スタックのパラメータを破棄します。
79_75_   ポップ %es
80_76_   ポップ %ds
81_77_   popl %edx
82_78_   popl %ecx
83_79_   popl %ebx
84_80_   popl %eax
85_81_   アイレット

```

```

85 set_e0: movb $1, e0          // 0xe0を受信すると、e0フラグのbit0がセットされます。
86      jmp e0_e1
87 set_e1: movb $2, e0          // 0xe1を受信すると、e0フラグのbit1がセットされます。
88      jmp e0_e1
89
90 /*
91 * このルーチンは、バッファに最大8バイトを充填します。
92 * %ebx:%eax. (%ebxはHigh)となります。バイトが書き込まれるのは
93 * %eaxが0になるまで、%al,%ah,%eal,%eah,%bl,%bh ... を順に並
べる。 94 */
94
// まず、コンソールの読み込みバッファキューリード_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リード_q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。バッファが満杯でなければ、ebx:eaxのデータを右の
// を8ビット分移動させて（つまり、ahの値をalに、blをahに、bhをblに）、上記の処理を繰り返し
// ます。
95 pushl %ecx // すべての文字が処理されるまで、現在のヘッドポインタの値が保存されます。
96 // そして pushl %edx キューを待っているプロセスがあるかキャラクタを取得します。クリップ
97 // します。 movl _table_list, %edx # read-queue for console
98 put\_queue movl head(%edx), %ecx // キューの先頭ポインタを取る -> ecx.
99
100 1:   movb %al, buf(%edx, %ecx) // alの中のcharを頭のポインタの位置に入れる。
101     incl %ecx // 1バイト分前に移動しました。
102     ANDL $SIZE-1, %ECX // ヘッドポインタを調整する。
103     cmpl tail(%edx), %ecx // バッファフル - すべてを破棄する
104     Je 3f
105     shrld $8, %ebx, %eax // ebxの右8ビットをeaxに移動、ebxは変更なし。
106     Je 2f // まだキャラクターがいるか? if noならジャンプ。
107     shr $8, %ebx // ebxの値を8ビット右に移動
108     JMP 1B
109 2:   movl %ecx, head(%edx) // すべての文字がキューに入っていれば、ヘッドポインタ
110     // が保存されます。
111     movl proc_list(%edx), %ecx // このキューのプロセスポインタを待っている?
112     // テストル %ecx, %ecx // このキューを待っているプロセスがあるかどうかをチエ
113     // ックします。
114 3:   Je 3f // No, then jump (to line 114).
115     movl $0, (%ecx) // はい、プロセスは目覚めています。
116 // ここから先は、ジャンプテーブルのポインタに対応する各キーのサブルーチンとなる
117 // key_table で、上記60行目のステートメントから呼び出されます。ジャンプテーブルkey_tableの
118 // 始まりは
119 // 513行目で
120 // 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。
121 // ctrlやaltの//。スキャンコードの前に0xe0のスキャンコードを受信した場合 (e0フラグがセットされ
122 // ている)。
123 // ctrキーも$0x10であるctrlキーとaltキーが複数ない差ある意味のモードでセット
124 // です// モードがctrlまたはaltに対応して設定されます。
125     movb $0x10, %al // 0x10(bit4)は、モードの左altキーです。
126 alt:

```

```

121 1:    CMPB $0,E0          // e0 set?(押されたキーは右のctrl/altキー)?
122      Je 2f              // いいえ、ジャンプします(124行目へ)。
123      addb %al,%al       // はい、右クリックフラグに変更します(bit3またはbit5)
124 2:    orb %al,mode      // モードフラグの対応するビットを設定します。
125 // このコードは、ctrlキーやaltキーが離されたときのスキャンコードを処理し、リセットして
// モードフラグの該当ビット。演算では、それを判断して
// は、e0フラグの有無に応じて、キーボードの右側にあるctrlキーまたはaltキーである。
// を設定します。
126 unctrl: movb $0x04,%al // bit2はモードの左ctrlキー用。
127      jmp 1f
128 unalt: movb $0x10,%al   // 0x10は、モード時の左ALTキーのビット4です。
129 1:    cmpb $0,e0          // e0 set?(右のctrl/altキーが解放されているか?)?
130      Je 2f              // いいえ、ジャンプします(132行目へ)
131      addb %al,%al       // yesの場合は、リセットフラグビット(bit3またはbit5)
132 2:    notb %al          // モードフラグの対応するビットをリセットします。
133      とb %al,mode
134      レット
135 // このコードは、左右のシフトキーを押したり離したりしたときのスキャンコードを処理します
// モードの対応するビットを設定、リセットするための //。
136のlshiftです。
137      orb $0x01,mode     // 左シフトキーが押されると、モードビット0を設定
138      レット
139のunshift。
140      とB $0xfe,mode     // 左シフトキーが離されると、モードビット0がリセ
141      レット
142 rshift
143      orb $0x02,mode     // 右シフトキーが押されると、モードbit1を設定し
144      レット
145 unrshiftです。
146 // 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。イエスなられ必ずしも、そうでなければモードフラグのビット6(capsキー
147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
148ビット(test bit $0x80, mode)します。 // bit7セッタ?(キャップスが押された?)?
149      jne 1f              // はい、169行目にジャンプします。
150      xorb $4, leds        // leds flagでcaps-lock(bit2)を反転させ
151      xorb $0x40, mode      // モードフラグのキャップスステート
152      orb $0x80, mode
153 // このコードは、ledsフラグに従って、LED(16)を反転させます。またはオフにしま
す。 // モードフラグのビット7(キャップスが押
154 set_ledsコール kb_wait   // れれてない口)を設定します。ファが空になるのを待ち
155      movb $0xed,%al        // set leds コマンド */
156      outb %al,$0x60        // ポート0x60にkbdコマンド0xedを送信。
157      コール kb_wait
158      movb leds,%al         // leds flagをパラメータとして取得します。
159      outb %al,$0x60
160      レット
161      とB $0x7f, mode       // キャップスが解放されると、モードフラグのbit7を
162個のキャップなし。        // リセットします。
163      レット

```

164巻。

```

165      testb $0x03, mode           // ctrlキーも押された?
166      je 1f                      // いいえ、ジャンプします(169行目へ)。
167      コール                   // はい、メモリ情報を表示します(mm/memory.c, line
168      _show_mem jmp             457)
169 1:    2f                      // いいえ、プロセスの状態情報を表示します
170 2:    コール                   (kernel/sched.c, 45)
171      _show_state xorb        // スクロールキーが押されたら、LEDのビット0を反転さ
172 num:   $1, leds                せる。
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
174      xorb $2, leds            オフにします。
175 /*     jmp set_leds            // numキーが押されたら、LEDフラグのbit1を反転させる

176 * curosr-key/numeric keypadのカーソルキーはここで処理されます。
177* テンキーのチェックなど 178*/          // leds フラグに基づいて LED インジケータをオンまたは
// 次のコードは、まず右のテンキーでスキャナを発行されているかどうかを判断します。
// キーボードの//ボタンを押します。そうでない場合は、このサブロティンを終了します。キーボード
の最後のキーDel(0x53)が押されたら
// テンキーが押されたときに、「Ctrl-Alt-Del」キーが押されたかどうかを判断します。
// コンビネーションが押されました。コンビネーションキーが押された場合、システム再起動コード
にジャンプします。
// 10.2.3項に記載されているXTキーボードスキャンコード表を参照してください。
// のリスト、または付録のキーボードスキャンコード表の最初のセットを参照してください。見ての
通り
// 表から、テンキーのキーのスキャンコード範囲は[0x47~0x53]となります。
180 // また、サブキー$0x47も使用する場合はキヌリ召ノ不が0x47-e0の範囲にない場合は、(198)を
179のカーソルです。                                返します。
181      jb 1f                      // つまり、テンキーからはスキャンコードが生成されません。
182      cmpb $12, %al              // (0x53 - 0x47 = 12)
183      ja 1f
184 // 12となった場合は、「del」キーが押されたことを示すので、続けて
// '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。
185      testb $0x0c, mode          // ctrlキーが押されているか、押されていなければジャンプする。
186      je cur2
187      testb $0x30, mode          // Altキーが押された?
188      jne reboot                // はい、そしてシステムの再起動(594行目)にジ
                                         ャンプします。
// 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,
// 插入、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、カーソル移動や插入/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
189 curの動作処理を呼び出す。ただし、numlockが移動を強制する場合(未e0ゲタがゼット?)?
す。// テンキーとして使われる)で、シフトキーも押されている場合は、キーパッドキーを
190 カーソル移動の操作として、この時/e0が設定されていれば、curにジ
                                         ャンプします。
191      Testb $0x02, Leds          /* num-lockではない場合、カーソルは強制的に使用されます。*/
                                         // ledsのテスト
192      jne cur // シフトが押されると、numのLEDが点灯して操作を行います。
                                         // カーソルを動かす
193 // 最後にtestb $0x03,数字キーとし、使用しない場合は、強制的に移ナンバーテーブルの
// num_tableは、スキャンコードに翻訳せられ、対応する数値文字が

```

キーへの // が取り出され、バッファキューに置かれます。文字数が多いので
// 一度にキャラクターキューに入れるには<=8>が必要なので、その前にebxをクリアする必要があ
ります。

```

195 // put_queueにジャンプして実行する。95 行目のコメントを参照して
196     movb %ah,%eax          // eaxをインデックスとするデジタル文字を取得
197     jmp put_queue          // します。
198 1:    レット              // パラメータ: 1~8chars in ebx:eax.
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 // 文字が最初に取得されます。もし
// 文字が<='9' (5, 6, 2, 3) の場合、Page Up、Page Dn、Insertのいずれかであることを意味しま
す。
// またはDeleteキーを押すと、関数文字列に'~'という文字が追加されます。
// しかし、このカーネルはそれらを認識して処理することはありません。コードは次に、コンテンツ
を
axの//をeaxのハイワードに入れて、'esc['をaxに入れて、ハイワードの文字で
200 #/ eaxの内容を移動し形成します最後にcharを取得します。
ルル // キャンセル用$先%ah+。
201     ja ok_cur            // <='9' (5, 6, 2, 3)の場合、ページアップ/ダウンまたは
202     movb $'~, %ah         // Ins/Delキーです。
203 ok_cur: shll $16, %eax // 文字'~'を追加する必要があります。
204     movw $0x5b1b, %ax      // axの内容をeaxの上位ワードに移動。
205     xorl %ebx, %ebx       // axに「esc []」を入れる
206     jmp put_queue
207
208
209 #if defined(KBD_FR)
210 num_tableです。
211 .ascii "789 456 1230." // テンキーのキーに関連付けられたASCIIコード。
212 #else
213 num_table:
214 .ascii "789 456 1230,"
215 #endif
216 cur_table:
217     .ascii "HA5 DGC YB623" // キーパッドの矢印、Ins、Delキーに関連付けられてい
218                         // ます。
219 /*
220 * このルーチンはファンクション・キー
221 * を処理する 221 */ 。

// ファンクションキースキヤンコードをエスケープ文字列に変換するサブルーチンであり
// が読み取りキューに格納されます。コードがチェックするファンクションキーの範囲はF1~F12で
す。
// ファンクションキーF1~F10のスキヤンコードは0x3B~0x44、F11と
// F12は0x57, 0x58です。まず、F1--F12を、対応するシーケンス番号に変換します
// 0--11となり、関数キー一テーブルfunc_tableを照会して、対応する
// エスケープ文字のシーケンスで、文字キューに入れられます。の間に Alt キーが押されると
// ファンクションキーを押すと、制御端子の切り替え動作のみが行われます。
223 ファンクでサブ $0x3B, %al // F1のスキヤンコードは0x3B
224     jb end_func          // ファンクションキーではない場合は
                           // Ret
225     cmpb $9, %al          // F1--F10 ?
226     jbe ok_func           // Yesの場合はジャンプ
227     サブ $18, %al          // キー F11, F12 ?
228     cmpb $10, %al          // キー F11 ?
229     jb end_func           // F11以上のオーダーがあった場合は
                           // 撤退。

```

```

230      cmpb $11,%al          // キー F12 ?
231      ja end_func          // 注文がF12のものよりも大きくない場合は退出。
232 ok_func:
233      テストb              // 左Altが押された?
234      $0x10, mode jne      // 変更コンソールにジャンプする if
235      alt_func cmpl        yes.
236      $4,%ecx              /* 十分なスペースがあるかどうかを確認
237      p�vend_func_table(%eax,4),%edx ます。// キーのエスケープ文字列を取得します。
238      xorl %ebx,%ebx        // 十分なスペース(4バイト)がない場合
239      jmp put_queue         合は、retします。

// 以下のコードは、alt + Fnキーの組み合わせでバーチャルコントロールの変更を処理します。
// 端末です。この時、eaxはファンクションキーのインデックスまたはオーダー番号(F1 -- 0)であり、対応する
バーチャルコントロールの端末番号に //。
240 alt_func pushl %eax          // インデックス番号をバーチャルコンソール番号として
241      コール               プッシュします。
242      _change_console popl  // chr_dev/tty_io.c, line 87.
243 end_func %eax               // パラメータを破棄します。
244
245      レ
246      ツ
247 /* ト
248 * ファンクションキー送信 F1:' esc [ [ A ] F2:' esc [ [ B ]
など。 249 */
250 func_table:
251      . long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
252      . long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
253      . long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b
254

// スキャンコード-ASCII文字対応表。
// 対応するキーのスキャンコードは、キーボードに応じてASCII文字にマッピングされます。
// 前に定義されたタイプ(FINNISH, US, GERMAN, FRANCH)を使用しています。
255 #if define(KBD_FINNISH)
256 key_map:
257      . バイト 0,27           // スキャンコード0x00, 0x01の場合。
258      . ascii "1234567890+"  // 0x02,...0x0c,0x0dのスキャンコードに対応しています。下
の図も同様です。
259      . バイト 127,9
260      . ascii "qwertyuiop}"
261      . バイト 0,13,0
262      . ascii "asdfghjkl|{"
263      . バイト 0,0
264      . アスキイ "", zxcvbnm, .-
265      . バイト 0,*0,0,32     /* 36-39 */
266      . フィル 16,1,0       /* 3A-49 */
267      . バイト '-0,0,0,0,+  /* 4A-4E */
268      . バイト 0,0,0,0,0,0   /* 4F-55 */
269      . バイト '<
270      . フィル 10,1,0
271

272 shift_map:                 // シフトを同時に押したときのマッピングテーブル。
273      . バイト 0,27
274      . ascii !"#$%&/()=?`"
275      . バイト 127,9
276      . ascii "QWERTYUIOP]`"

```

```

277      .バイト 13, 0
278      .ascii "ASDFGHJKL¥¥["
279      .バイト 0, 0
280      .ascii "*ZXCVBNM;:_"
281      .バイト 0, ',*, 0, 32      /* 36-39 */
282      .フィル 16, 1, 0          /* 3A-49 */
283      .バイト ',-, 0, 0, 0, '+' /* 4A-4E */
284      .バイト 0, 0, 0, 0, 0, 0  /* 4F-55 */
285      .バイト ',>
286      .フィル 10, 1, 0
287
288 alt_map:                      // altキーを同時に押したときのマッピングテーブル。
289      .バイト 0, 0
290      .ascii "¥0@0$¥0{[]}¥¥0"
291      .バイト 0, 0
292      .バイト 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
293      .バイト ',~, 13, 0
294      .バイト 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
295      .バイト 0, 0
296      .バイト 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
297      .バイト 0, 0, 0, 0      /* 36-39 */
298      .フィル 16, 1, 0          /* 3A-49 */
299      .バイト 0, 0, 0, 0, 0      /* 4A-4E */
300      .バイト 0, 0, 0, 0, 0, 0  /* 4F-55 */
301      .バイト ',|
302      .フィル 10, 1, 0
303
304 #elif defined(KBD_US)        // キーボードのマッピングテーブル
305
306 key_map:
307      .バイト 0, 27
308      .ascii "1234567890-="
309      .バイト 127, 9
310      .ascii "qwertyuiop[]"
311      .バイト 13, 0
312      .アスキー
313      "asdfghjkl;''"
314      .バイト ',~, 0
315      .ascii "¥zxcvbnm,.//      /* 36-39 */
316      .バイト 0, ',*, 0, 32    /* 3A-49 */
317      .フィル 16, 1, 0          /* 4A-4E */
318      .バイト ',-, 0, 0, 0, '+' /* 4F-55 */
319      .バイト 0, 0, 0, 0, 0, 0
320      .バイト ',<
321      .フィル 10, 1, 0
322
323 shift_map:
324 .バイト 0, 27
325      .ascii "!!@#$%^&*()_+"
326      .バイト 127, 9
327      .ascii "QWERTYUIOP{}"
328      .バイト 13, 0
329      .ascii "ASDFGHJKL:¥""
```

```

330      .バイト ',~,0
331      .ascii "|ZXCVBNM<>?"
332      .バイト 0,'*,0,32      /* 36-39 */
333      .フィル 16,1,0        /* 3A-49 */
334      .バイト '-,0,0,0,'+    /* 4A-4E */
335      .バイト 0,0,0,0,0,0    /* 4F-55 */
336      .バイト '>
337      .フィル 10,1,0
338
339 alt_map:
340      .バイト 0,0
341      .ascii "$0@0$0{[]}$0"
342      .バイト 0,0
343      .バイト 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
344      .バイト ',~,13,0
345      .バイト 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
346      .バイト 0,0
347      .バイト 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
348      .バイト 0,0,0/*.          36-39 */
349      .fill 16,1,0/*.          3A-49 */
350      .バイト 0,0,0,0/*.          4A-4E */
351      .バイト 0,0,0,0,0/*.          4F-55 */
352      .バイト '|'
353      .フィル 10,1,0
354
355 #elif defined(KBD_GR)           // ドイツ語キーボードのマッピングテー
356               ブル
357 key_map:
358      .バイト 0,27
359      .ascii "1234567890"
360      .バイト 127,9
361      .ascii "qwertzuiop@"
362      .バイト 13,0
363      .ascii "asdfghjkl[]`"
364      .バイト 0,'#'
365      .アスキイ "yxcvbnm,-"
366      .バイト 0,'*,0,32      /* 36-39 */
367      .フィル 16,1,0        /* 3A-49 */
368      .バイト '-,0,0,0,'+    /* 4A-4E */
369      .バイト 0,0,0,0,0,0    /* 4F-55 */
370      .バイト '<
371      .フィル 10,1,0
372
373
374 shift_map:
375 .バイト 0,27
376      .ascii !"#$%&/()=?`"
377      .バイト 127,9
378      .ascii "QWERTZUIOP$*"
379      .バイト 13,0
380      .ascii "ASDFGHJKL{}~"
381      .バイト 0,''
382      .ascii "YXCVBNM;:_"

```

```

383     .バイ 0,* ,0,32      /* 36-39 */
384     ト
385     .fill 16,1,0          /* 3A-49 */
386     .バイ ',-,0,0,0,'+   /* 4A-4E */
387     ト
388     .バイ 0,0,0,0,0,0,0,0 /* 4F-55 */
389     ト
390 alt_map:
391     .バイ 0,0
392     ト
393     .ascii "¥0@0$¥0{[]}¥¥0"
394     .バイト 0,0
395     .バイト '@,0,0,0,0,0,0,0,0,0
396     .バイト '^,13,0
397     .バイト 0,0,0,0,0,0,0,0,0,0
398     .バイト 0,0,0,0,0,0,0,0,0,0
399     .バイ 0,0,0,0          /* 36-39 */
400     ト
401     .fill 16,1,0          /* 3A-49 */
402     .バイ 0,0,0,0,0        /* 4A-4E */
403 #elif defined(KBD_FR)
404     .バイ 0,0,0,0,0,0,0,0 // フランスのキーボードのマッピングテ
405     ト
406 key_map:
407     :バイト '|,0,27
408     ト
409     :¶¶¶i10;&{¥'0'(-)_)_@)="
410     .バイト 127,9
411     .アスキー "アザーティテ
412     ュイオップ^$"
413     .バイト 13,0
414     .ascii "qsdfghjklm|"
415     .バイト '^,0,42       /* coin sup gauche, don't know,      */
416     [*/mu].                  */
417     .ascii "wxcvbn,;:!"
418     .バイト 0,* ,0,32      /* 36-39 */
419     .フィル 16,1,0          /* 3A-49 */
420     .バイト ',-,0,0,0,'+   /* 4A-4E */
421     .バイト 0,0,0,0,0,0     /* 4F-55 */
422     .バイト '<
423 shift_map:フィル 10,1,0
424 .バイト 0,27
425     .ascii "1234567890]+"
426     .バイト 127,9
427     .ascii "AZERTYUIOP<>"
428     .バイト 13,0
429     .ascii "QSDFGHJKLM%"
430     .バイト '^,0,'#
431     .ascii "WXCVBN?./¥"
432     .バイト 0,* ,0,32      /* 36-39 */
433     .フィル 16,1,0          /* 3A-49 */

```

```

436      .バイト'-'          /* 4A-4E */
437      ,0,0,0,'+'         /* 4F-55 */
438      .バイト0,0,0,0,0,0
439      .バイト'>
440      .フィル10,1,0
441 alt_map:
442      .バイト
443      0a@ccii "¥0~#[[|`^¥^@]]"
444      .バイト0,0
445      .バイト'@,0,0,0,0,0,0,0,0
446      .バイト',~,13,0
447      .バイト0,0,0,0,0,0,0,0,0
448      .バイト0,0
449      .バイト0,0,0,0,0,0,0,0,0
450      .バイ0,0,0,0          /* 36-39 */
        ト
451      .フィ16,1,0          /* 3A-49 */
        ル
452      .バイ0,0,0,0,0          /* 4A-4E */
        ト
453      .バイ0,0,0,0,0,0,0          /* 4F-55 */
        ト
454 #error "KBD_type not defined"
455 #endif .フィ10,1,0
456 /* ル
466 * do_selfは「通常の」キー、つまり意味を変えないキーを扱います。
467 #else で、1文字だけのリターンを持つもの。
463 */
// コードはまず、対応するキャラクターマッピングテーブル (alt_map、shift_map、key_map) を選択します。
// モードフラグに応じてマッピングテーブルを検索し、スキャancodeに応じて
ボタンの//を押すと、対応する文字 (ASCIIコード) が表示されます。そして、ボタンを押したときに
// 現在の文字がctrlボタンやaltボタンで同時に押され、その文字が
// ASCIIコードの値で、一定の変換を行います。最後に、変換された結果の文字
// は、読み取りバッファのキューに格納されます。
// 次のコードでは、まず、alt_map、shift_map、key_mapのいずれかのマッピングテーブルを選択します。
465 // モードフラグは基底であります。          // alt_map table address -> ebx (use alt_map table).
466 do_self: testb $0x20, mode             /* alt-gr */ // 右のaltキーが押された?
467     jne 1f                           // Yesならラベル1にジャンプ。
468     lea shift_map,%ebx            // そうでなければ、shift_mapテーブルを使用します。
469     testb $0x03, mode             // シフトキーが押されたか?
470     jne 1f                           // Yesならラベル1にジャンプ。
471     lea key_map,%ebx            // そうでない場合は、key_mapテーブルを使用します。

// これで、使用するマッピングテーブルが決まりました。次は、マッピングの対応する文字
スキャancodeの値に応じて // テーブルを取得します。対応する文字がない場合は
// マッピングテーブルでは、(turns to none、495行目)と返されます。
472 1:    movb (%ebx,%eax),%al           // マッピングテーブルから対応するcharを取得します
473     orb %al,%al                   .
474     Je none                      // 0以外のASCIIコードの文字ですか?
                                // noneにジャンプし、コードがゼロの場合はリターン
// ctrlキーが押されているかcapsキーがぬまぢされていて、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。
// 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。
475    testb $0x4c, mode          /* ctrl or caps */ // ctrlが押されているか、capsロックがかかっているか?
476    je 2f                   // そうでなければラベル2にジャンプします。
477    cmpb $'a,%al           // 文字が'a'の範囲内にあるかどうかをチェックする --
478    jb 2f                   // 文字が'a'の範囲内にあるかどうかをチェックする --
479    cmpb $'}',%al           '}''
480    ja 2f                   // そうでない場合は、ラベル2にジャンプします。
481    subb $32,%al           // charのASCIIコード値が0x20だけデクリメントされます
。

// 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
// は、[0x00--0x1F]の範囲で対応する制御文字を生成することができます。例えば
// ctrl + 'M' で対応するキャリッジリターン制御文字が生成されます。
482 2: testb $0x0c, mode /* ctrl */ // ctrlキーが押されたか?
483    Je 3f                  // ジャンプフォワード そうでない場合はラベル3。
484    CMPB $64,%AL           // を確認してください。charは、[0x40--0x5F]の範囲です
。
485    jb 3f                  // ラベル3にジャンプ でなければ
486    CMPB $64+32,%AL
487    subb $64,%al           // 制御用char [0x00--0x1f]に変換します。

// 左altキーが同時に押された場合、文字のビット7が設定される。つまり
// 0x7f以上の拡張文字セットの文字を生成することができます。
この時に///。
489 3: testb $0x10, mode      /* Left ALT */ // 左ALTが押された?
490    je 4f                  // ラベル4にジャンプしない場合
491    orb $0x80,%al          // charのビット7を設定します。

// 最後に、alの文字を読み込みバッファのキュに入れます。
492 4: andl $0xff,%eax       // 1文字のみ
493    xorl %ebx,%ebx
494    call put_queue
495 なし    ret
。
496
497 *マイナスは、独自のルーチンを持っていて、「E0h」の前に
498 *マイナスのスキャンコードは、テンキーの
499 *スラッシュが押され
500 ました。 501 */
501
// なお、フィンランド語やドイツ語のキーボードでは、スキャンコード0x35が'-'キーに対応しています。
502 の /> 26行目以降をご覧ください。 // e0フラグが0x01になっている?(e0受信 ?)
スです。
503    jne do_self            // の場合は、通常の処理に移ります。
504    movl $'/',%eax          // それ以外の場合は、'-'を'/'に置き換えます
。
505    xorl %ebx,%ebx
506    jmp put_queue
507
508 *このテーブルは、スキャンコードが実行されたときに、どのルーチンを呼び出すかを決定します。
509 *ゲットしました。ほとんどのルーチンは do_self を呼び出すだけですが、
次のような場合には何もしません。
510 *彼らはメイク・オア・ブレイクです。

```

[512 */ // 「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に注意。](#) [513 key_table:](#)

```

514     . long none,do_self,do_self,do_self      /* 00-03 s0 esc 1 2 */
515     . long do_self,do_self,do_self,do_self    /* 04-07 3 4 5 6 */
516     . long do_self,do_self,do_self,do_self    /* 08-0B 7 8 9 0 */
517     . long do_self,do_self,do_self,do_self    0C-0F + ' bs tab */。
518     . long do_self,do_self,do_self,do_self    /* 10-13 Q W E R */
519     . long do_self,do_self,do_self,do_self    /* 14-17 t y u i */
520     . long do_self,do_self,do_self,do_self    /* 18-1B o p } 。^ */
521     . long do_self,ctrl,do_self,do_self      /* 1C-1F enter ctrl a s */ .
522     . long do_self,do_self,do_self,do_self    /* 20-23 d f g h */
523     . long do_self,do_self,do_self,do_self    /* 24-27 j k l | */
524     . long do_self,do_self,lshift,do_self     /* 28-2B { para lshift , */
525     . long do_self,do_self,do_self,do_self    /* 2C-2F z x c v */
526     . long do_self,do_self,do_self,do_self    /* 30-33 b n m , */
527     . long do_self,minus,rshift,do_self      /* 34-37 .- rshift */
528     . long alt,do_self,caps,func            /* 38-3B Alt Sp Caps F1 */
529     . long ファンク、ファンク、ファンク、フ /* 3C-3F f2 f3 f4 f5 */ (注)
530     . long ファンク、ファンク、ファンク、フ /* 40-43 F6 F7 F8 F9 */ (注)
531     . long func,num,scroll,cursor          /* 44-47 f10 num scr home */
532     . long cursor,cursor,do_self,cursor    /* 48-4B up pgup - left */
533     . long cursor,cursor,do_self,cursor    4C-4F n5 right + end */。
534     . long カーソル、カーソル、カーソル、カ /* 50-53 dn pgdn ins del */ の
535     . long none,none,do_self,func           ようになります。
536     . long func,none,none,none             /* 54-57 sysreq ? < f11 */
537     . long なし、なし、なし、なし、なし    /* 58-5B f12 ??? */
538     . long なし、なし、なし、なし、なし    /* 5C-5F ??? */
539     . long なし、なし、なし、なし、なし    /* 60-63 ? ??? */
540     . long なし、なし、なし、なし、なし    /* 64-67 ??? */
541     . long なし、なし、なし、なし、なし    /* 68-6B ??? */
542     . long なし、なし、なし、なし、なし    /* 6C-6F ??? */
543     . long なし、なし、なし、なし、なし    /* 70-73 ? ??? */
544     . long なし、なし、なし、なし、なし    /* 74-77 ??? */
545     . long なし、なし、なし、なし、なし    /* 78-7B ??? */
546     . long なし、なし、なし、なし、なし    /* 7C-7F ??? */
547     . long なし、なし、なし、なし、なし    /* 80-83 ? br br br */
548     . long なし、なし、なし、なし、なし    /* 84-87 br br br */
549     . long なし、なし、なし、なし、なし    /* 88-8B br br br */
550     . long なし、なし、なし、なし、なし    /* 8C-8F br br br */
551     . long なし、なし、なし、なし、なし    /* 90-93年 br br br */
552     . long なし、なし、なし、なし、なし    /* 94-97 br br br */
553     . long なし、なし、なし、なし、なし    /* 98-9B br br br */
554     . long none,unctrl,none,none          /* 9C-9F br unctrl br br */
555     . long なし、なし、なし、なし、なし    /* A0-A3 br br br */
556     . long none,none,unshift,none        /* A4-A7 br br br */
557     . long なし、なし、なし、なし、なし    /* A8-AB br br unshift br */
558     . long なし、なし、なし、なし、なし    .
559     . long none,none,unrshift,none       /* AC-AF br br br */
560     . long unalt,none,uncaps,none       /* B0-B3 br br br */
561     . long なし、なし、なし、なし、なし    /* B4-B7 br br unrshift br */
562     . long なし、なし、なし、なし、なし    .
563     . long なし、なし、なし、なし、なし    /* B8-BB unalt br uncaps br */
564     . long なし、なし、なし、なし、なし    /* BC-BF br br */
565     . long なし、なし、なし、なし、なし    /* C0-C3 br br br */
566     . long なし、なし、なし、なし、なし    /* C4-C7 br br br */
567     . long なし、なし、なし、なし、なし    /* C8-CB br br br */

```

```

565      . long なし、なし、なし、なし、な      /* CC-CF br br br */
566      . long なし、なし、なし、なし、な      /* D0-D3 br br br */。
567      . long なし、なし、なし、なし、な      /* D4-D7 br br br */。
568      . long なし、なし、なし、なし、な      /* D8-DB br ???*/
569      . long なし、なし、なし、なし、な      /* DC-DF ???*/
570      . long なし、なし、なし、なし、な      /* E0-E3 e0 e1 ?*/
571      . long なし、なし、なし、なし、な      /* E4-E7 ???*/
572      . long なし、なし、なし、なし、な      /* E8-EB ???*/
573      . long なし、なし、なし、なし、な      /* EC-EF ???*/
581 * タイムアウトはありません。バッファが空にならなければハングア
574 * ップします。. long なし、なし、なし、なし、な      /* F0-F3 ???*/
582 * し
583 kb_wait: long なし、なし、なし、なし、な      /* F4-F7 ???*/
584     pushl %eax
585 1:   . inb $0x64,%al なし、なし、なし、なし、な // kbdコンキ中8-FBのアタスを読み
586     テストル 取る。
587     . $0x02,%al なし、なし、なし、なし、な // 入力バックPCが空であるかどうかを
588     1b    テストする(0)
589     popl %eax          // 空でなければラベル1にジャンプ
590 // また、ret
591 * ものののの引てはシま、まニボコドに問ひで答えせマが起動く動くのを待まちます。
592 * コントローラがリセットラインをローにパル
スする。 593 */
// 本サブルーチンでは、値0x1234を物理メモリアドレス0x472に書き込みます。この位置は
// リブートモードフラグを設定します。ブートプロセス中、ROM BIOSはリブートモードフラグを読
み込んで
// の値に基づいて次の実行を指示します。値が0x1234の場合、BIOSは次の実行をスキップします。
// メモリ検出処理を行い、ウォームブート処理を行います。この値が0の場合は、コールドブート
594再起動処理を行います。
ます。    コール kb_wait           // コントローラのバッファが空になる
595    movw $0x1234,0x472        の待ちます。
596    movb $0xfc,%al            メモリチェックを行わない */
597    outb %al,$0x64            don't do memory check
598    jmp die                  パルスリセットとA20のLow */
599
```

3. インフォメーション

1. PC/ATキーボードインターフェースプログラミング

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ピン) は、以下のリセット動作の実装に使用されます。

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

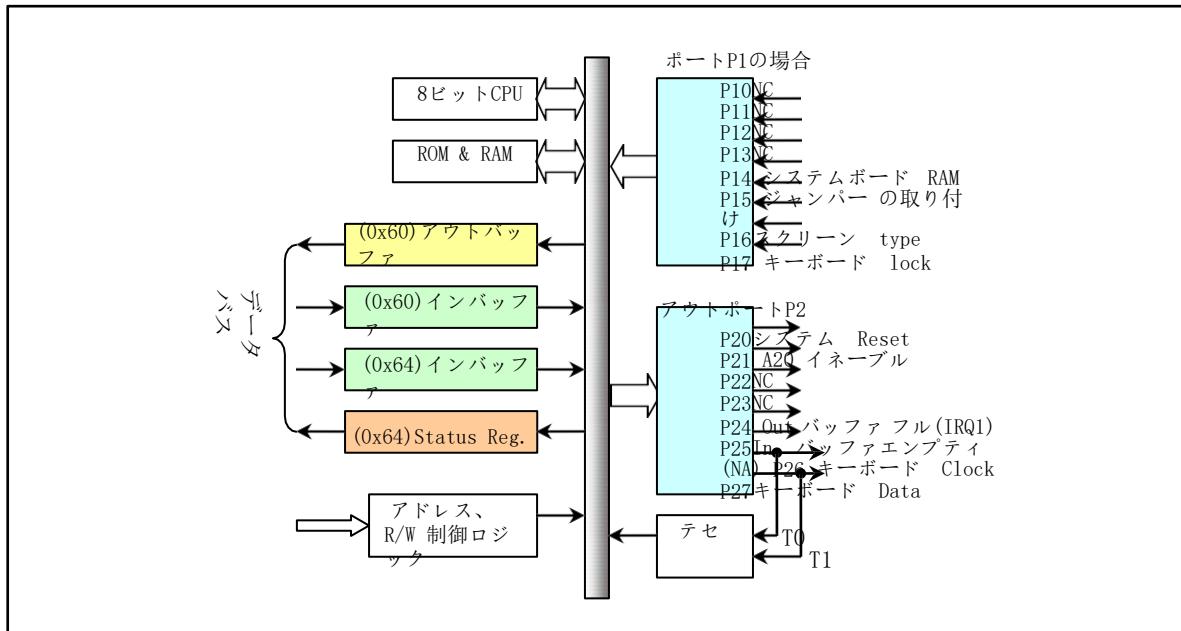


図10-7 キーボードコントローラ804Xの回路図

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

ポート	R/W	名前	目的
0x60	リード	データポートまたは出力バッファ	8ビットのリードオンリーレジスタです。キーボードコントローラがスキャンコードを受信すると またはキーボードからのコマンド応答があった場合、一方ではステータスレジスタビット0=1をセットし、他方では割り込みIRQ1を発生させます。 通常は、ステータスポートのビット0=1の時にのみ読み出すべきです。
0x60	書く	入力バッファ	コマンドや後続のパラメーターをキーボードに送信する際や キーボードコントローラにパラメータを書き込む。10種類以上のキーボードがある コマンドを使用する場合は、表の後の説明を参照してください。通常は 、ステータスポートのビット1=0のときにのみ書き込みます。
0x61	リード/ライト		このポートは、8255Aの出力ポートB（P2）のアドレスであり、ハードウェア・リセットされます。 8255Aを使用したPC標準キーボード回路の処理を行います。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有効にします 。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。 ビット6=0 キーボードクロックを強制的に低くし、キーボードからの

			のデータを提供しています。 ビット5-0 これらのビットは、キーボードとは独立しており、PPI（Programmable Parallel Interface）に使用されます。
0x64	リード	ステータスレジスタ	<p>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。</p> <p>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)。</p> <p>ビット6=受信タイムアウト (キーボードの転送でIRQ1が生成されない)、ビット5=送信タイムアウト (キーボードが応答しない)。</p> <p>Bit 4=0 インヒビットスイッチ。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) である。</p> <p>=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) ; Bit 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 = セルフテストに合格。</p> <p>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)。</p> <p>Bit 0 = 出力バッファがいっぱいになった (データポート0x60にシステム用のデータがある)。</p>

10.2.3.2 キーボードコマンド

0x64 デバイスはポート0x60に接続されています。このパラメータは、キーを書き込みます。このパラメータは、キーをボードコマンドを送信するためです。キーを受信してから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 (これも下に書き込まれます) 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。

コマンド	持っています。 パラメータ	説明	表10-2 キーボードコマンド一覧
0xed	はい。	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キーです。 ビット1=num-lockキー、ビット0=scroll-lockキー。	
0xee	いいえ	診断応答。キーボードは0xeeを返すはずです。	
0xef		Reserved.	
0xf0	はい。	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 - スキャンコードセット2を選択 (AT、PS/2の場合、これがデフォルト)、0x03 - スキャンコードセット3を選択。	552
0xf1		Reserved.	
0xf2	いいえ	キーボードの識別番号を読み取ります (2バイト読み取り)。ATキーボードが返すレスポンスコード0xfa。	

		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：Rate = 1 / ((8 + A) * 2 ^ B * 0.00417)となります。このパラメータのデフォルト値は0x2cです。
0xf4	いいえ	キーボードを有効にする。
0xf5	いいえ	キーボードを無効にします。
0xf6	いいえ	キーボードのデフォルトパラメーターを設定します。
0xf7-0xfd		Reserved.
0xfe	いいえ	スキャンコードを再送します。このコマンドは、システムが検出してキーボードからのデータ送信が正しく行われていない。
0xff	いいえ	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。 演算処理は 1. キーボードは、コマンド受信後すぐにコマンド0xfaを送信して応答します。 2. キーボードコントローラは、キーボードロックとデータラインをハイレベルに設定します。 3. キーボードがBATの操作を開始します。 4. 正常に終了した場合、キーボードは0xaaを、そうでない場合は0xfdを送信し、スキャンを停止します。

10.2.3.3 キーボード・コントローラ・コマンド

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、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。

コマンド	持っています。 パラメータ	表10-3 キーボードコントローラコマンド一覧 説明
0x20	いいえ	キーボードコントローラへの最後のコマンドバイトは、システムリード用のポート0x60に置かれます。
0x21-0x3f	いいえ	下位5ビットで指定されたコントローラの内蔵RAMのコマンドを読み出すコマンドの
0x60-0x7f	はい。	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（parity、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）。 ビット5 PCモード（スキャンコードのparity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 キーボード操作を禁止（キーボードロックをLowにする）。 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します。 ビット0は、出力レジスタがフルになったときに割り込みを発生させることができます。
0xaa	いいえ	キーボードコントローラのフルテストを初期化します。成功すると0x55、失敗すると0xfcを返します。
0xab	いいえ	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はエラーフリー。

		0x01 キーボードクロックラインがLow（常にLow、Lowスタック）。 0x02 キーボードクロックライン はハイ、0x03 キーボードデータラ インはロー。 0x04 キーボードのデータラインがハイになっています。
0xac	いいえ	診断ダンプ。804Xの16バイトRAM、出力ポート、入力ポートの状態を順次、システムに出力されます。
0xad	いいえ	キーボード操作を禁止する（コマンドバイトのビット4を1にする）。
0xae	いいえ	キーボード操作を有効にします（リセットコマンドバイトのビット4=0）。
0xc0	いいえ	804Xの入力ポートP1を読み込み、0x60にして読み込みます。
0xd0	いいえ	804Xの出力ポートP2を読み込み、0x60にして読み込みます。
0xd1	はい。	ライト804Xの出力ポートP2は、オリジナルのIBM PCでは出力ポートのビット2を使って制御しています。 A20のゲートです。なお、ビット0（システムリセット）は常にセットされている必要があります。
0xe0	いいえ	テスト端子T0とT1の入力は、出力バッファに送られ、読み取ることでシステムを使用しています。ビット1：キーボードデータ、ビット0：キーボードクロック。
0xed	はい。	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はCaps-lockキーです。 ビット1=Num-lockキー、ビット0=Scroll-lockキー。
10.2.3.4 キーボードスキャノード		出力ポートにパルスを送信します。このコマンド・シーケンスは、出力ポートP20-23を制御します。 PCはすべてノンエンコードのキーは、キーボードです。キーボードの固有のキーには左から右へ、左から下へとポジション番号が付いており、XTとAT機のキーボードの位置が大きく異なります。キーが押されたときにキーが離されたときには「ブレイク」スキャノードと呼ばれます。XTキーボードの各キーのスキャノードを表10-4に示します。

F1 F2	~ 1 2 3 4 5 6 7 8 9 0 - = „ ^ „)	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 [] 0f 10 11 12 13 14 15 16 17	ホー	↑	PgUp	PrtSc
3D 3E	18 19 1a 1b	ム	48	49	37
		47			
F5 F6	cntl a s d f g h j k l ; ' enter 1d 1e 1f 20 21 22 23 24 25	←	5	→	-
3F 40	26 27 28 1c	4B	4C	4D	4A
F7 F8	lshft z x c v b n m , . / rshft 2a 2c 2d 2e 2f 30 31 32	エ	↓	PgDn	+
41 42	33 34 35 36	ン	50	51	4E
		ド	4F		
F9 F10	ALT スペース CAPLOCK	Ins	デ		
4ボタン	38の各ボタンには、それぞれ対応するスキャノードがバイ	52の下位7ビット（ビット6			
~0)	ト7=0はスキャ	53）を示します。ビット			

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キーのキーボードの場合、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、キー位置番号は30で、メイクコードとスキャンコードも30（0x1e）、ブレークコードはメイクコードに0x80を加えた0x9eとなります。

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

さらに、PrtScnキーとPause/Breakキーという非常に特殊なキーがあります。キーを押すと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の列が発生する。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。

3. コンソール.c

1. 機能説明

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⁵⁵⁵を中心構成されています。通常のモードでは、表示文字の書き込み

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

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

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

10.3.2 コードアノテーション

プログラム 10-2 linux/kernel/chr_drv/console.c

```

1  /*
2  * linux/kernel/console.c
3  *
4  *      (C) 1991 Linus Torvalds
5  */
6
7 /*
8 * console.c
9 *
10 * このモジュールは、コンソールの io 関数を実装しています。
11 *      'void con_init(void)'
12 *      'void con_write(struct tty queue * queue)'
13 * 願わくば、これがかなり完成度の高いVT102の実装となるように。
14 *
15 * John T Kohl氏のおかげでビープ音が発生しました。
16 *
17 * バーチャルコンソール、スクリーンブランкиング、スクリーンダンピング、カラー、グラフィックス
18 * Peter MacDonaldによるCharsとVT100の機能強化。
19 */
20 /*
21 * 注意!!!割り込みを一時的に無効にしたり有効にしたりすることができます。
22 * (ビデオIOに単語を入れる)が、これはキーボードでも動作します。
23 * 割り込み。キーボードの入力時には、割り込みが有効になっていないことがわかっています。
24 *
25 * トランプゲートを使用しているため、割り込みが発生します。すべてがうまくいくことを願っています。
26 */
27
28 /*
29 * 主にGalen Hunt氏による、様々なビデオカードをチェックするコードです。
30 * <g-hunt@ee.utah.edu>
31 */
32
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_structや
// 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リの形式です。
// <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> 文字列のヘッダーファイルです。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ーファイルです。システムの様々なエラー番号を含みます。
33 #include <linux/sched.h>
34 #include <linux/tty.h>
35 #include <linux/config.h>
36 #include <linux/kernel.h> (日本語)
37
38 #include <asm/io.h>
39 #include <asm/system.h>
40 #include <asm/segment.h>
41
42 #include <string.h>
43 #include <errno.h>
44
45 // 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
46 // す。
47 #define ICRNL '^w^'
48 // 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。
49 #define DEF_TERMIOS_OPOST 0
50 #define DEF_TERMIOS_ISIG 0
51 #define DEF_TERMIOS_I_SETSIG 0
52 #define DEF_TERMIOS_I_SETSIG 0
53 } // struct termios
54
55
56 /*
57 * これらは起動時のsetup-routineによって設定されます。
58 */
59 // boot/setup.sとシステムパラメーターテーブルの読み込みと保存に関するコメントを参照してください。
60 // セットアッププログラムによる
61 #define ORIG_X (*((unsigned char *)0x90000)) // 元のカーソルのカラムNo.
62 #define ORIG_Y (*((unsigned char *)0x90001)) // 元のカーソル行番号
63 #define ORIG_VIDEO_PAGE (*((unsigned short *)0x90004)) // 現在のビデオページ。
64 #define ORIG_VIDEO_MODE (((*(unsigned short *)0x90006) & 0xff)) // 表示モード。
65 #define ORIG_VIDEO_COLS (((*(unsigned short *)0x90006) & 0xff00) >> 8) // 画面のcolumn。
66 #define ORIG_VIDEO_LINES (((*(unsigned short *)0x9000e) & 0xff)) // 画面の行数。
67 #define ORIG_VIDEO_EGA_AX (*((unsigned short *)0x90008)) // 現在の関数。

```

```

67 #define ORIG_VIDEO_EGA_BX (*(unsigned short *)0x9000a)      // ディスペンサーのサイズ、カ
68 #define ORIG_VIDEO_EGA_CX (*(unsigned short *)0x9000c)      // ラーモード。
69
70 // 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
71 #define VIDEO_TYPE_MDA 0x10 /* モノクロの文字表示 */
72 #define VIDEO_TYPE_CGA 0x11 CGAディスプレイ /* CGAディスプ
73 #define VIDEO_TYPE_EGAM 0x20 レイ
74 #define VIDEO_TYPE_EGAC 0x21 EGA/VGAのモノクロモード */。
75 #define NPAR_16           /* EGA/VGAのカラーモード
76
77 #define NR_CONSOLES = 0; // システムがサポートしているバーチャルコンソールの数で
78
79 extern void keyboard_interrupt(void); // キーボード割り込みハンドラ (keyboard.S内)。 80
// 以下のスタティック変数は、このファイルで使用されているグローバル変数の一部です。

```

```

81 static unsigned char video_type;          /* 使用しているディスプレイの種類 */
82 static unsigned long video_num_columns;    /* テキストの列数 */
83 static unsigned long video_mem_base;        /* ビデオメモリのベース */
84 static unsigned long video_mem_term;        /* ビデオメモリの終了 */
85 static unsigned long video_size_row;        /* 1行あたりのバイト数 */
86 static unsigned long video_num_lines;       /* テストラインの数 */
87 static unsigned char video_page;           /* 動画の初期ページ */
88 static unsigned short video_port_reg;       /* ビデオレジスタセレクトポート */
89 static int can_do_colour = 0; // フラグ、カラー機能を使える。 91
// ハーネシ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。現在の情報をすべて含んでいます。
// vc_origin と vc_scr_end は、対応するディスプレイメモリの位置です。
// 現在処理されているバーチャルコンソールで使用されている開始行と最終行に
// vc_video_mem_start と vc_video_mem_end は、ディスプレイの
// 現在の仮想コンソールで使用されているメモリ領域です。
92 static struct {
93     符号なし ショート vc_video_erase_char; // char属性を消去 & char (0x0720)
94     符号なし チャー vc_attr;             // char属性。
95     符号なし チャー vc_def_attr;         // デフォルトの属性です。
96     int vc_bold_attr;                  // 太字のchar属性。
97     符号なし ロング vc_ques;           // クエスチョンマークを表示します。
98     符号なし ロング vc_state;           // エスケープシーケンスやコントロールシーケンスの状態。
99     符号なし ロング vc_restate;         // エスケープや制御シーケンスの次の状態です。
100    符号なし ロング vc_checkin;         // vc_checkinです。
101    符号なし ロング vc_origin;          /* EGA/VGA の高速スクロールに使用されます
102    符号なし ロング vc_scr_end;         /* EGA/VGA の高速スクロールに使用されます
103    符号なし ロング vc_pos;            // 現在のカーソルのmem位置です。
104    符号なし ロング vc_x, vc_y。        // 現在のカーソルの列、行の値。
105    符号なし ロング vc_top, vc_bottom; // スクロール時の上下のラインnr。
106    符号なし ロング vc_npar, vc_par[NPAR]; // エスケープシーケンス param nr と配列。
107    符号なし ロング vc_video_mem_start; /* ビデオRAMの開始位置 */

```

```

113 } vc_cons[MAX_CONSOLES];
114
// 参考までに、現在使用しているコンソールのシンボルを以下のように定義します。
// 処理されます。上記と同じ意味です。ここで、currconsは現在の仮想端末
// vc_cons[]構造体を使用して、関数の引数に番号を入力します。
115 #define origin          (vc_cons[currcons].vc_origin)
116 #define scr_end         (vc_cons[currcons].vc_scr_end)
117 #define pos             (vc_cons[currcons].vc_pos)
118 #define top             (vc_cons[currcons].vc_top)
119 #define bottom          (vc_cons[currcons].vc_cons[currcons].
120 #define x               ons].vc_bottom) (
121 #define y               vc_cons[currcons].vc_x) (
122 #define state           vc_cons[currcons].vc_y) (
123 #define restate          vc_cons[currcons].vc_state) (
124 #define checkin         vc_cons[currcons].vc_restate)
125 #define npar            (vc_cons[currcons].vc_checkin)
126 #define par             (vc_cons[currcons].vc_npar)
127 #define ques            (vc_cons[currcons].vc_par)
128 #define attr             (vc_cons[currcons].vc_ques)
129 #define saved_x          (vc_cons[currcons].vc_attr)
130 #define saved_y          (vc_cons[currcons].vc_saved_x)
131 #define translate        (vc_cons[currcons].vc_saved_y)
132 #define video_mem_start (vc_cons[currcons].vc_video_mem_start)
133 #define video_mem_end   (vc_cons[currcons].vc_video_mem_end)
134 #define def_attr          (vc_cons[currcons].vc_def_attr)
135 #define video_erase_char (vc_cons[currcons].vc_video_erase_char)
136 #define iscolor           (vc_cons[currcons].vc_iscolor)
137
138 int blankinterval = 0;                      // ブラックスクリーンの間隔
139 int blankcount = 0;                         .
140                                         // ブラックスクリーンのタイ
141 static void sys beep(void);                 // メタモードビープ機能。
142
143 /*
144 * ESC-Zやcsi0cに対して端末が答える内容です。
145 * クエリ (=vt100レスポンス)。
146 */
147 // csi - Control Sequence Introducer.
// ホストは、端末に対して、デバイス属性制御シーケンスの応答を
// デバイスアトリビュート(DA)制御シーケンスで、パラメータなし、またはパラメータ0('ESC
// [c' or
// 'ESC [0c') (ESC-Zも同じ働きをします)。端末は、以下のシーケンスを送信して応答します。
// ホストです。このシーケンス(すなわち「ESC [?1; 2c」)は、端末がVT100互換であることを示し
// ます。
高度なビデオ機能を備えた//ターミナル。
148 #define RESPONSE "#033[?1;2c"
// 使用する文字セットを定義します。上の部分は通常の7ビットのASCIIコードで、これは
// US文字セットに対応しています。下の部分はVT100端末のライン文字に対応しています。
// デバイス、つまり、チャートライインを表示するキャラクタセットです。
149 static char *translations[] = "0123456789:;<=>?""
150 /* 通常の ASCII */
151 /* vt100 */
152 /* vt100 */
153 /* vt100 */
154 /* vt100 */

```

```

155     " !¥#$%&' ()*+, -./0123456789:;=>"  

156     "@abcdefghijklmnopqrstuvwxyz[=]= " "004" "007" "007" "370"  

157     "007" "275" "267" "326" "323" "327" "304" "304" "307" "266" "322"  

158     "272" "363" "362" "343" "007" "234" "007" ""  

159 };  

160  

161 #define NORM_TRANS_(translations)[0]  

162 #define GRAF_TRANS_(translations)[1]  

163  

164 ////////////////////////////////////////////////////////////////////  

164 // カーソルの現在の位置をトラッキングします。  

164 // パラメータ: currcons - 現在の仮想端末; new_x - カーソルの列番号。  

164 // new_y - カーソルラインの番号です。  

164 // 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。  

164 // 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  

164 // パラメータの有効性を確認します。与えられたカーソルの列数が最大値を超えた場合は終了します  

164 // 。  

164 // 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  

164 // 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  

164 // が更新され、ディスプレイメモリの位置posに対応します。なお、すべての変数  

164 // 関数内 // は、実際には vc_cons[currcons] 構造体の対応するフィールドです。  

164 // と以下の機能があります。  

164 /* 注意! Gotoxyはx=video_num_columnsがOKだと思っています。*/  

165 static inline void gotoxy(int currcons, int new_x, unsigned int new_y)  

166 {  

167     if (new_x > video_num_columns || new_y >= video_num_lines)  

168         return;  

169     x = new_x;  

170     y = new_y;  

171     pos = origin + y*video_size_row + (x<<1); // 1つのcolは2バイト必要なので、x<<1。  

172 }  

173  

173 ////////////////////////////////////////////////////////////////////  

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。  

173 // まず、ディスプレイカードの種類がカラーカードであるかどうかを確認し、コンソールの  

173 // パラメータで指定された // がフロントコンソールで、2つの条件が満たされない場合は終了します。  

173 // それ以外の場合は、スクロール開始アドレスをディスプレイカード上のレジスタに出力します。  

174 static inline void set_origin(int currcons)  

175 {  

175 // まず、ディスプレイカードの種類を決定します。EGA/VGAカードの場合は、画面上の  

175 // 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  

175 // 範囲（エリア）は  

175 // 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  

175 // があります。  

175 // 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  

175 // の場合は直接終了します。さらに、フォアグラウンドのコンソールでのみ操作を行うため、以下  

175 // の場合のみ(video_type_ != VIDEO_TYPE_EGAC && video_type_ != VIDEO_TYPE_EGAM)  

176 // 現在のコンソールcurrconsはフォアグラウンドのコンソールなので、メモリスタートを設定する  

177 // 必要がある(currcons != fg_console)  

178 // スクロール開始線に対応する位置です。  

179  

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

179 // よって  

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

179 // 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。  

179 // 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。  

179 // 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  

179 // ます。  

179 // 右は2で割るということで、画面上の1文字を表現することにもなります。

```

```

// 2バイトのメモリで出力値は、デフォルトのディスプレイメモリの開始位置を基準にして動作します。
// video_mem_base の位置、例えば EGA/VGA カラーモードの場合は video_mem_base = physical
180 // メモリアドレス0xb8000。
181     cli(); // データレジスタr12を選択してハイバイトを出力する。
182     outb_p(12, video_port_reg); // データレジスタr12を選択してハイバイトを出力する。
183     outb_p(0xff&((origin-video_mem_base)>>9), video_port_val);
184     outb_p(0xff&((origin-video_mem_base)>>9), video_port_val); // データレジスタr12を選択しvideo_port_val出力する。
185     sti();
186 }
187

188 static void scrup(int currcons)
189 {
190     // 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
191     // が最下段の行数以上であれば、ローリング動作の条件となる
192     // が満たされない。また、EGA/VGAカードの場合は、画面上の範囲（エリア）を指定して
193     // スクロールは、MDAモノクロディスプレイカードでは全画面スクロールしかできません。
194     // この関数は、ディスプレイタイプがEGAとMDAの場合、別々に処理します。ディスプレイタイプが
195     // EGA の場合は
196     // まぬココルをほまずンデオヌヅドカニドがEGA/VGAをタイプの場合は対応していません。ムーブメントに分けられるbottomとtop
197     // を返すことができます。
198     if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
199     {
200         // 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
201         // 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
202         // 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
203         // 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
204         // カーソルと文字ポインタの位置 scr_end で画面の終わり end は
205         // を追跡します。最後に、新しいスクリーンウィンドウのメモリ開始位置の原点が
206         // ディスプレイコントローラー。
207         if (! top && bottom == video_num_lines)
208         {
209             origin += video_size_row;
210             pos += video_size_row;
211             scr_end += video_size_row;
212
213             // 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
214             // 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
215             // 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。
216             // 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。
217             // 画面のウィンドウが下に移動しました。続いて、画面メモリの状況に応じて
218             // データが移動すると、現在の画面に対応するスタートポインタ、カーソル位置の
219             // ポインタと、それに対応する画面端のメモリポインタscr_endが再調整されます。
220
221             // この組み込みアセンブリコードは、まず、（画面の文字）に対応するメモリデータを移動させます
222             // 行番号-1) 行をディスプレイメモリの開始位置video_mem_startに追加してから
223             // 後続のメモリ位置に1行のスペース（消去）文字データを入れる。
224             // %0 - eax (消去文字 + 属性); %1 - ecx ((画面の行数 -1))
225             // 文字数に対応する / 2、長い言葉で動く); %2 - edi (メモリを表示する
226             // 開始位置 video_mem_start); %3 - esi (スクリーン・ウィンドウ・メモリの開始位置の原点)。

```

```

// 移動の方向。[edi] -> [esi], move ecx long words.
198     if (scr_end > video_mem_end) {...  

199         asm ("cld##"  

200             "rep##"           // 明確な方向性  

201             "movsl##"          // 移動を繰り返すデ  

202             "movl _video_num_columns, %1##t" "rep##t"  

203             // 一行分のスペースを埋める。 "stosw"  

204             :: "a"(video_erase_char)です。  

205             "c" ((video_num_lines-1)*video_num_columns>>1),  

206             "D"(video_mem_start)です。  

207             "S" (原点)  

208             cx", "di", "si") 。  

209         scr_end -= origin-video_mem_start;  

210         pos -= origin-video_mem_start;  

211         origin = video_mem_start;  

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

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

// 新しいライン  

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

// (最終行の先頭にある対応するメモリ位置)を参照してください。
213     } else {  

214         asm ("cld##"  

215             "rep##" // 一行分のスペースを埋める。  

216             "stosw"  

217             :: "a"(video_erase_char)です。  

218             "c"(video_num_columns)です。  

219             "D" (scr_end-video_size_row)  

220             : "cx", "di") 。  

221     }  

222     // set_origin(currcons);  

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

// は  

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

// が削除されます。このとき、画面の全行に対応する表示メモリデータ  

// 指定された行頭から行末までの // がそのまま1行分上に移動して、消去  

// 新たに表示される行には、 // の文字が入ります。  

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

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

223 // - esi (最上位1列目が配置されているメモリの位置)です。  

224     asm ("cld##"  

225         "rep##"           // トップ+1からボトムまで繰り返  

226            します。  

227             "movsl_n##"  

228             "movl _video_num_columns, %%ecx##t"  

229             "rep##" // 空白文字を改行で埋める。  

230             "stosw"  

231             :: "a"(video_erase_char)です。  

232             "c" ((bottom-top-1)*video_num_columns>>1),  

233             "D" (原点+video_size_row*top) です。  

234             "S" (原点+ビデオサイズ_row*(top+1))  

             cx", "di", "si") 。

```

```

235         }
236     }
// ディスプレイの種類がEGAではなくMDAの場合、以下のような移動操作を行います。なぜなら
// MDAディスプレイカードは、画面全体をスクロールするだけで、自動的に
// 表示メモリの範囲を超えた（つまり、ポインタが自動的にスクロールする）ので
// 画面の内容に対応したメモリを、表示とは別に処理しない。
// メモリの範囲です。処理方法は、EGAの非フルスクリーン動作と全く同じです。
237    その他      /* EGA/VGAではありません。
238     {
239         asm (
240             "cld\t"
241             "rep\t"
242             "movsl_n\t"
243             "movl_video_num_columns, %%ecx\t"
244             "rep\t"
245             "stosw\t"
246             :: "a"(video_erase_char)です。
247             "c" ((bottom-top-1)*video_num_columns>>1),
248             "D" (原点+video_size_row*top) です。
249             "S" (原点+ビデオサイズ_row*(top+1))
250             cx", "di", "si" );
251     }
252 }

//// 表示内容が1行分下にスクロールします。
// 画面スクロールウィンドウを1行上に移動し、対応する画面スクロール領域の内容を
// が1行下に移動します。新しい行は、移動の開始行の上に表示されます。処理は
// メソッドは scrup() と似ていますが、データのオーバーレイを行うのは
// メモリデータで、コピー動作が逆になります。つまり、最後の文字からのコピーは
// カウントダウンの2行目を最終行にして、3行目の文字をコピーします。
// カウントダウンの//行を、最後の2行目にする、といった具合です。
253 static void scrdown(int currcons)
254 {...}
// 同様に、スクロールエリアには少なくとも2行が必要です。の先頭行番号が2行以上であれば
// スクロールエリアの最下段の行番号以上の条件である
ローリング動作のために // を満たしていません。また、EGA/VGAカードに対しては、指定した
// MDAモノクロディスプレイカードでは、画面上のスクロール可能な範囲（エリア）が
// フルスクリーンスクロールを行います。ディスプレイの開始位置までウィンドウが移動するので
// 最大でエリアメモリ、画面の終わりに対応する表示メモリポインタscr_end
// ウィンドウが実際のディスプレイメモリの端を超えないように、通常のメモリデータの移動のみ
// をここで処理する必要があります。
255     if (bottom <= top)
256         を返すことができます。
257     if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
258     {
// %0 - eax (消去文字 + 属性); %1 - ecx (対応する長い単語の数)
// 上から下までの行数-1); %2 - edi (の最後の長い単語の位置)
// ウィンドウの右下隅); %3 - Esi (2行目の最後のロングワードの位置
// ウィンドウのカウントダウンの//)。
// 移動の方向。[esi] -> [edi], move ecx long words.
259         asm ( `std\t` ).           // 演出フラグを立てる!
260             "rep\t"
261             "movsl_n\t"
262             "addl $2, %%edi\t" /* %edi が 4 減らされました。

```

```

263             "movl _video_num_columns, %%ecx\n"
264             "rep\n" // 上の行の空欄を埋めてください。
265             "stosw"
266             :: "a"(video_erase_char)です。
267             "c" ((bottom-top-1)*video_num_columns>>1),
268             "D" (原点+video_size_row*bottom-4) です。
269             "S" (origin+video_size_row*(bottom-1)-4)
270             :"ax", "cx", "di", "si";
271         }
272     // 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）。
273    その他 /* EGA/VGAではありません。
274     {
275         asm ( /*std\n*/
276             "rep\n"
277             "movsl_n\n"
278             "addl $2, %%edi\n" /* edi がデクリメントされました by 4 */
279             .
280             "movl _video_num_columns, %%ecx\n"
281             "rep\n"
282             "stosw"
283             :: "a"(video_erase_char)です。
284             "c" ((bottom-top-1)*video_num_columns>>1),
285             "D" (原点+video_size_row*bottom-4) です。
286             "S" (origin+video_size_row*(bottom-1)-4)
287             :"ax", "cx", "di", "si");
288     }
289
290     //カーソルが最終行に同じ列の位置で1行下の既存行変数を直接変更する y++,
291     // 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
292     // 1行分の文字数に対応）。それ以外の場合は、その内容を移動する必要があります。
293     // 画面のウィンドウを1行分上げる。関数名のlf（ラインフィード）は、処理制御の
294     // 文字 LF。
295     static void lf(int currcons)
296     {
297         も (y+1< bottom) {
298             も
299             y++;
300             pos += video_size_row;           // 1行で占有するバイト数を加算する。
301             を返すことができます。
302         }
303         scrup(currcons)                // コンテンツを1行上に移動します
304     }
305
306     // カーソルは同じ列の1行上に移動します。
307     // カーソルが画面の最初の行にない場合は、カーソルのカレントを直接変更する
308     // 行変数y--を調整し、カーソルに対応する表示メモリの位置pos、minusを調整する
309     // 画面上の文字列1行に対応するメモリの長さのバイト数です。
310     // そうでなければ、スクリーンウィンドウの内容を1行下に移動させる必要があります。このとき、
311     // 関数
312     // 名前のri（リバースインデックス）は、制御文字RIまたはエスケープシーケンス "ESC M" を指します。
313     static void lf ri(int top, currcons)
314     {
315         y--;

```

```

303         pos -= video_size_row;           // 1行が占めるバイト数をマイナスします。
304         を返すことができます。
305     }
306     scrdown(currcons) です。          // コンテンツを1行下に移動します。
307 }
308

// カーソルは0列目に戻ります。
// メモリロケーションposに対応するカーソルを調整する。カーソルの列番号
// *2は、0列が占めるメモリバイトの長さで、その列までのラインでは
// カーソルの位置を示します。関数名のcr（キャリッジリターン）は、制御文字である
// 処理中です。
309 static void cr(int currcons)
310 {
311     pos -= x<<1; // col 0からカーソルのcolまでで占められるバイト x=0;
312
313 }
314

// カーソルの前の文字を消去し、カーソルを1列前に移動させる。
// カーソルが0列目以外の場合、カーソルは2バイトでバックされる。
// メモリポジションpos（画面上の1文字に対応）、次にカレントカーソル
// の列が1だけデクリメントされ、カーソル位置の文字が消去されます。
315 static void del(int currcons)
316 {
317     if (x) {
318         pos -= 2;
319         x--;
320         *(unsigned short *)pos = video_erase_char;
321     }
322 }

323
//// 画面上のコンテンツのうち、カーソル位置に関連する部分を削除する。
// この関数は、ANSI制御シーケンスを処理するために使用されます。文字の削除操作
指定された制御シーケンスに従って、カーソル位置に関連付けられた // を実行します。
// 表示されている文字の一部または全部が消去され、カーソル位置は変更されません。
文字や行が消去されたときに、//。
// この関数で処理されるANSI制御シーケンスは：'ESC [ Ps J'. その中で、Ps = 0
// - カーソルを画面の一番下まで削除することを意味し、1 - カーソルまで画面を削除することを意
味します。
// 2 - 画面全体を消す。
// 機能名csi_J (CSI - Control Sequence Introducer) は、制御の
// シーケンス「CSI Ps J」の処理を行います。引数'vpar'は、「Ps」の値に対応します。
// 上記の制御シーケンスを行います。端末制御コマンドの紹介については、セクション
プログラミリストの後に // 10.3.3.3. よく使われるエスケープシーケンスと制御シーケンス
// の付録3に記載されています。 324 static
void csi_J(int currcons, int vpar) 325 .

326     long count asm ("cx"); // レジスタ変数を使用するように設定します。 long
327     start asm ("di");
328

// まず、削除する文字数と、削除するディスプレイメモリの位置を設定します。
// 上記の3つのケースに応じて削除を開始する switch (vpar) {
329     case 0: /* カーソルからディスプレイの端までを消去する */ (注)
330         count = (scr_end-pos)>>1;
331

```

```

332         start = pos;
333         ブレークします。
334     case 1: /* スタートからカーソルまでを消去する。
335             count = (pos_origin)>>1;
336             start = origin;
337             ブレークします。
338     case 2: /* ディスプレイ全体を消去する。
339             count = video_num_columns * video_num_lines;
340             start = origin;
341             ブレークします。
342
343 // その後、消去文字を使つて返すことができた場所を埋める。
344 // %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)。
345     asm ("cld#\n"
346           "rep#\n"
347           "stosw#\n"
348           :: "c"(カウント)です。
349           "D"(スタート),
350           "a"(video_erase_char)
351           : "cx", "di" );
352
353 static void csi_K(int currcons, int vpar)
354 {。    ロングカウントアズム ("cx")。
355     ロングスタートASM (以下、 "/DI/" )。
356
357 // まず、削除する文字数と、その文字が表示されるディスプレイメモリの位置を指定します。
制御シーケンスの3つの条件に応じて、 // 削除開始が設定されます。
358     スイッチ (vpar) {
359         case 0: /* カーソルから行末までの消去 */ (0)
360             if (x>=video_num_columns)
361                 を返すことができます。
362             count = video_num_columns-x;
363             start = pos;
364             ブレークします。
365         case 1: /* 行頭からカーソルまでを消去する */ (1)
366             start = pos - (x<<1);
367             count = (x< video_num_columns)? x:video_num_columns;
368             ブレークします。
369         ケース 2: /* 行全体を消去 */
370             start = pos - (x<<1);
371             count = video_num_columns;
372             ブレークします。
373             のデフォルトです。

```

```

374          を返す
375      }
376      // その後、消去文字を使ってき 文字が消えた場所を埋める。
377      // %0 - ecx (delete count); %1 -edi (delete address start); %2 -eax (fill erase character).
378      asm ("cld##"
379           "rep##"
380           "stosw##"
381           :: "c"(カウント)です。
382           "D"(スタート)、
383           "a"(video_erase_char)
384           : "cx", "di" );
385
386      //// 表示キャラクターの属性を設定します。
387      // 制御シーケンスは、パラメータに応じて文字表示属性を設定します。すべての
388      // 今後、端末に送られる文字には、ここで指定された属性が使用されます。
389      // 文字表示の属性をリセットするために、再び制御シーケンスが使用されます。
390      // 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
391      // て明るい。
392      // 4 - 下線、5 - 閃光、7 - 反転、22 - 非太字、24 - 下線なし、25 - 閃光なし。
393      // 27 - 通常 ;30--38 - 前景色の設定 ;39 - デフォルトの前景色(白)。
394      // 40--48 - 背景色の設定; 49 - デフォルトの背景色(黒)。
395      void csi_m(int currcons)
396  {
397      int i;
398
399      // 1つの制御シーケンスには、複数の異なるパラメータを設定することができます。そのパラメータ
400      // は
401      // 配列par[]. 次のコードは、各パラメータPsを数に応じて循環的に処理します。
402      // 受信したパラメータnparの
403      // Ps = 0 の場合は、現在のバーチャルコンソールの横に表示される文字属性
404      // には、デフォルトの属性 def_attr が設定されています。初期化時、def_attr は 0x07 に設定さ
405      // れています。
406      // (黒地に白)。
407      // Ps = 1の場合、後に表示される文字属性が太字やハイライトに設定されます。
408      // カラーディスプレイの場合は、文字属性または上位0x08が強調表示されます。
409      // 文字。モノクロディスプレイの場合は、文字に下線を引く。
410      // 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
411      // ルイの場合は、前景色が1つ増え、もう一方の色が使われます。
412      // 現時点では // 0:d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
413      // 元の vc_bold_attr が (par) ない場合は背景色がリセットされ、そうでない場合は
414      // 前景色が反転します。前景色が背景色と同じ場合 break; /* default */.
415      case 1: attr=(iscolor? attr|0x08:attr|0x0f);break; /* bold */.
416      /*case 4: attr=attr|0x01;break; */ /* 下線 */
417      case 4: 太字にする */;
418      if (! iscolor)
419          attr |= 0x01; // モノは下線を引く。
420     その他
421      /*foreground = background */ をチェックします。
422      if (vc_cons[currcons].vc_bold_attr != -1)
423          attr = (vc_cons[currcons].vc_bold_attr&0x0f) | (0xf0&(attr));
424     その他
425      { short newattr = (attr&0xf0) | (0xf&(~attr)) です。
426          attr = ((newattr&0xf)==((attr)>>4)&0xf)?

```

```

403             ((attr)&0xf0) | (((attr)&0xf)+1)%0xf)
404            になります。
405         }
406     }
407 // Ps = 5の場合、現在のパーゴナルコンソールに表示される文字は
// ブリンクに設定されている、つまり、属性バイトのビット7が1に設定されている。
// Ps = 7の場合は、その後に表示される文字が逆に設定され、つまり前景
// と背景色が入れ替わります。
// Ps = 22の場合、それ以降の文字の強調表示を中止する（太字表示を中止する）。
// Ps = 24の場合、モノクロ表示時に後続の文字の下線がキャンセルされる
// と緑はカラー表示のためにキャンセルされます。
// Ps = 25の場合、それ以降の文字の点滅はキャンセルされます。
// 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。
// 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。
408     ケース 5: attr=attr|0x80;break; /* 点滅 */
409     ケース 7: attr=(attr<<4)|(attr>4);break; /* ネガティブ
410             */
411     ケース 22: attr=attr&0xf7;break; /* not bold */.
412     ケース 24: attr=attr&0xfe;break; /* アンダーラインではな
413             */
414     ケース 25: attr=attr&0x7f;break; /* 点滅しない */。
415     ケース 27: attr=def_attr;break; /* ポジティブなイメージ
416             */
417 // 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。
418 // 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ます。参照
419 // 色の値については、番組表の後の説明を参照してください。
420     のデフォルトです。
421     if (!
422         can_do_colour)
423         break;
424     if((par[i]>=30)&&(par[i]<=38))           // フォアグラウンド
425         attr = (attr & 0xf0) | (par[i]-30)      • カラー。
426        となります。
427     else /* 背景色 */                      // 背景色。
428     if((par[(addr+40)&0xff]+(par[(addr+48)]-40)<<4)
429        となります。
430    その他
431         ブレークします。
432     }
433 // メモリの位置を設定します。
434 static inline void set_cursor(int currcons)
435 {
436 // ディスプレイカーソルを設定する必要があるということは、キーボード操作があるということな
437 // ので、次のようにします。
438 // で黒画面操作の遅延カウント値を元に戻すことができます。また、コンソールの
439 // カーソル表示するのためintervalウンドのコンソル黒画面の開始位置を初期化します。
440     if (currcons != fg_console)
441         を返すことができま
442     // 次に、インデックススレジスタポートを使用して、表示制御データレジスタr14（現在の

```

```

// カーソルの表示位置（上位バイト）を表示し、カーソルの現在位置を書き込む
// 上位バイト（9ビット右に移動すると上位バイトが下位バイトに移動して分割される
// を2で割ったもの）。これは、デフォルトのディスプレイメモリの操作に相対するものです。次に、
// インデックスレジスタ
435 // r15を選択して、カーソルの現在位置の下位バイトを書き込む。 cli();
436     outb_p(14, video_port_reg)です。
437     outb_p(0xff&((pos-video_mem_base)>>9), video_port_val);
438     outb_p(15, video_port_reg);
439     outb_p(0xff&((pos-video_mem_base)>>1), video_port_val)となります。
440     sti()です。
441 }
442
// カーソルを隠す。
// カーソルを隠すために、現在のバーチャルコンソールウィンドウの端に設定します。
443 static inline void hide_cursor(int currcons)
444 {
    // 最初に表示制御データレジスタr14（カーソルの現在の表示位置）を選択する
    // 上位バイト）を書き込んだ後、カーソル位置の上位バイトを（9ビット右に移動して
    // は上位バイトを下位バイトに移動して2で割ることを意味します）。その後、r15を選択し、次のように書き込みます。
445 // カーソルの現在位置の下位バイト)です。
446     outb_p(0xff&((scr_end-video_mem_base)>>9), video_port_val);
447     outb_p(15, video_port_reg)となります。
448     outb_p(0xff&((scr_end-video_mem_base)>>1), video_port_val)
449 }    となります。
450
//// VT100 に応答シーケンスを送信します。
// つまり、ホストが端末を要求することに応答して、デバイス属性 (DA) を送信する
// をホストに送信します。ホストは、端末にデバイスアトリビュート (DA) コントロールを送り返す
// よう要求します。
ESC[0c]」または「ESC Z」などのDA制御シーケンスをパラメータなしで送信することで、 // シー
ケンスを作成することができます。
// そして、端末は147行目で定義された応答シーケンス（例：「ESC [?1; 2c」）を送り返します。
ホストのシーケンスに対応して // を表示します。このシーケンスは、ホストに対して、端末がVT100
高度なビデオ機能を備えた//互換性のある端末です。処理は、応答を
// シーケンスをリードバッファのキューに入れ、copy_to_cooked()関数で処理して
// 補助（二次）キューを入れる。
451 static void respond(int currcons) // 147行目 (tty_struct *tty)を定義されています。
452 {
453     cli()です。
454     while (*p) {                                // レスポンスシーケンスをリードキューに入れます
455         PUTCH(*p, tty->read_q);           .
456         p++;                                     // include/linux/tty.h, line 46.
457     }
458     sti()です。
459     copy_to_cooked(tty)です。          // 補助キューを入れる。tty_io.c, 120.
460 }
461
462 }
463
//// カーソル位置にスペース文字を挿入します。
// カーソルの先頭のすべての文字を1スペース右に移動し、挿入する
// カーソルでの消去文字。 464 static
void insert_char(int currcons) 465 {

```

```

466     int i=x;
467     unsigned short tmp, old = video_erase_char;           // erase char (with attribute).
468     unsigned short * p = (unsigned short *) pos;           // カーソルの記憶位置
469
470     while (i++ < video_num_columns)
471         { tmp=*p;
472             *p=old;
473             old=tmp;
474             p++;
475         }
476     }
477
478     //// カーソル位置に行を挿入します。
479     // 画面のウィンドウをカーソルのある行から下にスクロールする
480     // ウィンドウに表示されます。カーソルは新しい、空の行になります。
481     static void insert_line(int currcons)
482     {
483         int oldtop, oldbottom; 481
484
485         // 最初に画面のスクロール開始行の上端と最終行の下端の値を保存してからスクロールする
486         // 画面の内容をカーソルのある行から1行下にする。最後に
487         // スクロールの開始行' top' と最終行' bottom' の元の値 oldtop=top;
488         oldbottom=bottomです。
489
490         // top=yです。                                // スクロール画面の開始行と終了行を設定します。
491         bottom = video_num_lines;
492         scrdown(currcons);                         // 画面の内容が1行下にスクロールする
493         top=oldtop;
494         bottom=oldbottomとなりま
495         す。
496
497         //// 文字を削除します。
498         // カーソル位置の1文字を削除し、カーソルの右にあるすべての文字をシフトさせる
499         // を1スペースだけ左に移動させます。
500         static void delete_char(int currcons)
501         {
502             int i;
503             unsigned short * p = (unsigned short *) pos;
504
505             // カーソルの現在の列の位置xが、画面の右端の列を超えているかどうかを返します。
506             // それ以外の場合は、カーソルの右文字から行末までのすべての文字が
507             // を1スペース分左にシフトします。そして、最後の文字で消去文字を埋める。
508             if (x>=video_num_columns)
509                 を返すことができます。
510             i = xです。
511             while (++i < video_num_columns) {...          // を1スペースだけ左にシフトしたものです
512
513                 *p = *(p+1)です。
514                 p++;
515             }
516             *p = video_erase_char;                      // 最後に消しゴムの文字を記入します。
517         }
518
519         //// カーソルがある位置の1行を削除します。

```

```

// カーソルがある位置の1行を削除し、画面を1行上にスクロールします。
506 static void delete_line(int currcons)
507 {
全
-----
***タ
// まず、画面のスクロール開始行「上」と最終行「下」を保存してから、スクロールの
// 画面スルのある開始時時の内容を表示する「最後」最終終の「下」。
510     oldtop=top;
511     oldbottom=bottomです。
512     top=yです。          // スクロール画面の開始行と終了行を設定します。
513     bottom = video num lines;
514     scrup(currcons)です。    // 画面の内容が1行上にスクロールします。
515     top=oldtop。
516     bottom=oldbottomとなります。
517 }
518

//// カーソル位置に nr 文字を挿入します。
// ANSIのエスケープ文字列を扱う。'ESC [ Pn @'.1つ以上のスペース文字の挿入
// 現在のカーソルの位置にPnは挿入する文字数で、デフォルトは1です。
// カーソルは最初に挿入されたスペース文字の位置のままで。カーソル位置の文字と
// 右のボーダーが右にずれて、右のボーダーを超えた文字が消えてしまいます。
// エスケープシーケンスでのパラメータnr = Pn。
519 static void csi_at(int currcons, unsigned int nr)
520 {
// 挿入された文字数が1行分の文字数を超える場合は、切り捨てられる
// 挿入文字数nrが0の場合、1文字が
// を挿入します。その後、周期的にnr個のスペース文字を挿入します。
521     if (nr > video num columns)
522         nr = video num columns;
523     else if (!nr)
524         nr = 1;
525     while (nr--)
526         insert_char(currcons)で
527     す。
528

//// カーソル位置に nr 行を挿入します。
// ANSIのエスケープシーケンスを扱う：'ESC [ Pn L'.制御シーケンスは、1つまたは複数の空白を挿
入します。
// カーソルの位置にラインが表示されます。操作が完了しても、カーソル位置は変わりません。
// 空白行を挿入すると、カーソル下のスクロールエリアの行が下に移動します。る。
// 表示ページをスクロールして出てきた行がなくなる。エスケープシーケンスのパラメータ nr = Pn
。
529 static void csi_L(int currcons, unsigned int nr)
530 {
// 挿入された行数が画面の最大行数よりも多い場合、その行数は
行の//が画面の行に切り取られます。挿入された行数nrが0の場合、1行の
531 // が挿入されねばます video num lines)
532 // の後、Nrの空白行を周期的には挿入まます;
533     else if (!nr)
534         nr = 1;
535     while (nr--)
536         insert_line(currcons)
    です。

```

537 }

538

```
//// カーソル位置の nr 文字を削除します。
// ANSIのエスケープシーケンスを扱う。'ESC [Pn P'. この制御シーケンスは Pn 個の文字を削除する
// をカーソルから削除します。文字が削除されると、カーソルの右側にあるすべての文字が
// を左にシフトすると、右のボーダーにスペース文字ができます。そのプロパティ
// は最後の左シフト文字と同じになるはずですが、ここでは簡略化されています。
// キャラクターのデフォルトのプロパティ（黒背景白前景空間0x0720）を
// スペース文字を設定します。パラメータ nr = エスケープシーケンスの Pn。
```

539 static void csi_P(int currcons, unsigned int nr)

540 {。

```
// 削除された文字数が1行分の文字数を超える場合は、切り捨てられます
// 削除された文字数nrが0の場合、1文字が
// 削除されます。そして、カーソルで指定された数の文字 nr を繰り返し削除します。
```

```
541     if (nr > video_num_columns)
542         nr = video_num_columns;
543     else if (!nr)
544         nr = 1;
545     while (nr--)
546         delete_char(currcons) で
547     } す。
```

548

```
//// カーソルの先頭にある nr 行を削除します。
// ANSIのエスケープシーケンスを処理します。'ESC [ Pn M' です。制御シーケンスは、1つまたは複数の行を削除する
スクロールエリアでカーソルが置かれている行から // を削除します。行が削除されると
スクロールエリアの削除された行の下にある//行が上に移動し、1つの空白行が
// 一番下の行です。Pnが表示ページに残っている行数よりも大きい場合には
// このシーケンスは、これらの残りの行を削除するだけで、スクロールエリア外では機能しません。
// エスケープシーケンスでのパラメータnr = Pn。
```

549 static void csi_M(int currcons, unsigned int nr)

550 {。

```
// 削除した行数が画面の最大行数よりも多い場合。
// 表示されている行数が切り捨てられます。削除する行数nrが0の場合、1
551 // 行が削除されますが video_num_lines で指定された nr 行を繰り返し削除します。
552     nr = video_num_lines;
553     else if (!nr)
554         nr=1;
555     while (nr--)
556         delete_line(currcons)
557     } です。
```

558

```
//// 現在のカーソル位置を保存します。
```

559 static void save_cur(int currcons)

560 {。

```
561     saved_x=x;
562     saved_y=y;
563 }
```

564

```
//// 保存されたカーソル位置を復元します
```

```
。 565 static void restore_cur(int
currcons) 566 {。
```

```
567 gotoxy(currcons, saved_x, saved_y);
```

568}569570

```

// 以下の列挙定義は、以下のcon_write()関数で
// エスケープシーケンスやコントロールシーケンスを解決します。ESnormalは初期のエントリー状態
// であり
// また、エスケープシーケンスやコントロールシーケンスが処理されるときの状態。
// ESn表記文字制御文字常場換算状態である文字を画面に直接表示しも制御文字ルカリッジリターンな
// ど)を受信した場合は、カーソル位置を設定します。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場合、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のガイド
// コードなので、ESsquareにジャンプして処理を行い、そうでない場合は受信した文字をエスケ
// ープシーケンスとして扱います。キャラクタセットの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で処理しています。デバイス制御文字列シーケン
// ス「ESC P」については、別状態のESsettermで処理します。
// ESsquare - 制御シーケンスブリアンブル('ESC [')を受信したことを示し、次のことを示します
// この時、「[]」文字を受信した場合は、キーボードのファンクションキーから送られてくるシ
// ーケンスである「ESC [[]」シーケンスを受信したことを意味しますので、Esfunckeにジャン
// プして処理を行います。そうでなければ、制御シーケンスのパラメータを受信する準備をする
// 必要があるので、状態ESgetparsを設定して、シーケンスのパラメータ文字を受信して保存す
// る状態に直接入ります。
//
// ESgetpars - この状態は、パラメータの値を受け取ることを示しています。
// この時の制御シーケンスを紹介します。引数は10進数で表現されており、受信した数字文字を
// 数値に変換して、配列par[]に保存します。セミコロン ';' を受信した場合は、この状態の
// まで、受信したパラメータ値を次のデータpar[]に保存します。すべてのパラメータを取得し
// たことを示す数値文字でもセミコロンでもない場合は、状態ESgotparsに移行して処理を行
// 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
// は、この状態で受信した終了文字に応じて、対応する制御シーケンスを処理することができる
// 。ただし、処理の前に、ESsquare状態で「?」を受信した場合、このシーケンスは端末機器の
// プライベートシーケンスである。本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を実行し、シーケ
// ンスの処理後に状態ESnormalに戻す。
//
// ESfuncke - キーボードのファンクションキーからシーケンスを受け取ったことを示す。
// 表示する必要がないもの。そこで、通常の状態ESnormalに戻します。
// ESsetterm - DCS(Device Control String Sequence)状態であることを示す。このとき
// の間に、文字'S'を受信した場合は、初期の表示文字属性に戻ります。また、受信した文字
// が'L'または'1'の場合は、折り返し線表示モードのON/OFFを行います。
//
// 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、それぞれG0とG1で使用する文字セットを指定するためのものです。このとき、文字'0'を
// 受信した場合は、グラフィックの文字セットがG0、G1として選択され、受信した文字が'B'の
// 場合は、通常のASCIIの文字セットがG0、G1の文字セットとして選択されます。
//
571 enum { ESnormal, ESesc, ESsquare, ESgetpars, ESGotpars, ESfuncke,
572 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フrom取り出され、それぞれの
// の文字です。制御文字やエスケープシーケンス、制御シーケンスの場合は、制御処理
カーソルの位置合わせや文字の削除などの//が行われ、通常の文字
カーソルの位置に // が直接表示されます。
// パラメータ tty は、現在のコンソールで使用されている tty 構造体ポインタです。
574 void con_write(struct tty_struct* tty)
575 {
576     int nr;
577     char c;
578     int curcons;
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
// キーボードやプログラム (Ctrl-Sボタンなど) から発行された場合は、機能が停止します。
// 書き込みキューの文字を処理して、関数を終了します。さらに、もし
// 制御文字 CAN (キャンセル、ASCIIコード24、Ctrl-Xで生成) またはSUB (Substitute、26。
// Ctrl-Z) が取られた場合は、エスケープシーケンスまたはコントロールシーケンス中にその文字を
受信した場合。
// その場合、シーケンスは実行されず、直ちに終了し、後続の
// 表示された文字。なお、con_write()関数は、表示された文字を処理するだけです。
// 現在、書き込みキューに入っている文字数をフェッチする際に使用します。それは
// シーケンスが書き込みキューに入っている間の文字数を読み取ることができます。
580 // なので、「状態」は、以下のようなときに、エスケープシーケンスやコントロールシーケンスを処
581 理する別の状態になっている可能性があります。
582 // 関数が最後に終了する。 currcons
583     = tty - tty_table;
584     if (!CHARS(tty->write_q) || (currcons<0))書き込みキューの中の文字数を取得し
585     while (npanic("con_write: illegal tty") とます。
586         if (tty->stopped)
587             break;
588         GETCH(tty->write_q, c);           // 1文字を取得します。
589         if (c == 24 || c == 26)           // 制御用の文字。キャンセル」または「サ
590             state = ENormal;            ブティテュート」。
591             switch(state) {...}
592
// 抽出された文字が通常の表示文字の場合、対応する表示文字
// は、現在マッピングされている文字セットから直接取り出され、ディスプレイメモリに配置されま
す。
// 現在のカーソルがある位置、つまり文字が直接表示されている位置。
// その後、カーソル位置を1文字分だけ右に移動させます。具体的には、もし
// の文字が制御文字でも拡張文字でもない場合、つまり (31<c<127) の場合は
// 現在のカーソルが行末にある場合、または行末以外の位置にある場合、カーソルが
// は次の行の最初の列に移動します。そして、メモリポインタのposを対応する
// をカーソル位置に設定します。その後、その文字はディスプレイメモリposに書き込まれ
592 // カーソルが1列分右に移動し、Eをねじてposも2バイト移動しています。
593         if (c>31 && c<127) { // 通常の表示のチャ。
594             if (x=video_num_columns) { // ラインを変更しますか?
595                 x -= video_num_columns;
596                 pos -= video_size_row;
597                 If (currcons) です。
598             }

```

```

599             Asm ("movb %2,%ah\t")           // 書き込んで
600             "movw %%ax,%1n\t"           ください。
601             :: "a" (translate[c-
602             32]), "m" (*(short
603             *)pos), "m" (attr)
604             : "ax" ) .
605             pos += 2;
606             x++;
607
608             // '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）。
609             // 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。
610             // cがキャリッジリターンCR(13)の場合、カーソルを先頭列(0列目)に移動します。
611             // cがDEL(127)の場合、カーソルの左側の文字が消去され、カーソルが移動します。
612             // を消去した文字位置に戻す。
613             // cがBS（バックスペース、8）の場合、カーソルを1列分左に移動させ、カーソルを
614             // メモリポインタposに対応する
615             } else if (c==27) // ESC - エスケープ・チャ一。
616                 state=ESesc;
617             else if (c==10 || c==11 || c==12)
618                 lf(curcons) です。
619             else if (c==13) // CR - キャリッジリターン。
620                 CR (カーコン)。
621             else if (c==ERASE\_CHAR(tty))
622                 del(curcons) です。
623             else if (c==8) { // BS - バックスペース。
624                 if (x) {
625                     x--;
626                     pos -= 2;
627                 }
628
629             // cが水平タブHT(9)の場合、カーソルは8の倍数の位置に移動し、カーソルの数が
630             // の列数が画面上の最大列数を超える場合は、カーソルを次の行に移動させてください。
631             // cがベルBEL(7)の場合、システムのビープ機能が呼び出され、スピーカーが鳴ります。
632             // cが制御文字SO(14)またはSI(15)の場合は、文字セットG1またはG0を
633             // ディスプレイの文字セットを変更します。
634             } else if (c==9) {                   // HT - Horizontal Tab.
635                 c=8-(x&
636                 7); x +=
637                 c;
638                 if(x+video num columns) {...}
639                     x = video num columns;
640                     pos = video size row;
641                     lf(curcons) です。
642                 }
643                 c=9;
644             } else if (c==7) // BEL - ベル。
645                 sysbeep();
646             else if (c == 14) // SO - Shift out, use G1.
647                 translate = GRAF TRANS;
648             else if (c == 15) // SI - Shift in, use G0.
649                 translate = NORM TRANS;
650
651             ブレークします。
652
653             // ESnormal状態でエスケープ文字ESC(0x1b=27)を受信すると、次のようになります。
654             // 状態の処理を行います。この状態では、cの制御文字やエスケープ文字の処理を行います。
655             // 処理後、デフォルトの状態はESnormalになります。

```

```

637         case ESesc:
638             state = ESnormal;
639             switch (c)
640             {
641                 case '[':           // ESC [ - CSIシーケンス
642                     state=ESsquare;
643                     break;
644                 case 'E':           // ESC E - カーソルが次の行と col 0 に移動します。
645                     gotoxy(currcons, 0, y+1);
646                     break;
647                 case 'M':           // ESC M - 1行上に移動します
648                     ri(currcons);   .
649                     break;
650                 case '['D'.        // ESC D - 次の行に移動します
651                     lf(currcons);   .
652                     break;
653                 case '['Z'.        // ESC Z - デバイスのプロパティを
654                     respond(currcon照会)ます。
655                     break;
656                 case '['7':          // ESC 7 - カーソル位置の保存.
657                     save_cur(currcons);
658                     ブレークします。
659                 case '['8':          // ESC 8 - カーソル位置の復元.
660                     restore_cur(currcons);
661                     ブレークします。
662                 case '['(') :       // ESC (、ESC ) - select
663                     char set. state = ESsetgraph;
664                     ブレークします。
665                 case '['P':          // ESC P - ターミナルのパラメータを設定します
666                     .
667                     state = ESsetterm;
668                     break;
669                 case '['#':          // ESC # - 行属性を変更します。
670                     state = -1;
671                     break;
672                 case '['c':          // ESC c - デフォルトの設定にリセット
673                     tty-> termios =
674                         DEF_TERMIO;   state =
675                         restate = ESnormal; checkin
676                         = 0;
677                         top = 0です。
678                         bottom = video num lines;
679                         /* case break; テンキー */ アブ
680                         /* case '=': リアイコン */ テ
681                         キー
682                         ブレークしま
683
684             // 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
685             // の配列なので、ESsquare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
686             // 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
687             // 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
688             // ャラクターが
689             // 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
690             // が受信されたので、次の状態はESfunckeyに設定されます。受信した文字が「?」でなければ
691             // 処理する状態ESgetparsに直接行く。受信した文字が'?'の場合、それは
692             // 端末機のプライベートシーケンスであること、その後に機能文字が続くこと。

```

```

// だから次の文字に行って、コードを処理するために状態ESgetparsに行く。 case
682     ESsquare:
683         for (npar=0; npar<NPAR;           // パラの配列を初期化しま
684             npar++) par[npar]=0;       す。
685         npar=0;
686         state=ESgetpars;
687         if (c=='[')      ファンクションキ          // 'ESC [['
688         { state=ESfunckey; — /*/*.
689             break;
690         }
691         if (ques=(c=='?'))
692             ブレークします。
693
// この状態は、制御シーケンスのパラメータ値を受信することを示しています。
// この時のことです。パラメータは10進数で表現されており、受け取った
// 数値文字を値に変換して、par[]配列に保存します。セミコロンを受け取った場合
// ';' の場合は、この状態のままで、受信したパラメータ値を次の項目である
// 配列par[]. 数字やセミコロンでない場合は、すべてのパラメータを示す
// が得られたならば、処理するために状態ESgotparsに移動します。
694     case ESgetpars:
695         if (c==';' && npar< NPAR-1) {
696             npar++します。
697             ブレークします。
698         } else if (c>='0' && c<='9') {
699             par[npar]=10*par[npar]+c-'0';
700             ブレークします。
701         } else state=ESgotpars;
702
// 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
// は、端末機器のプライベートシーケンスです。の処理をサポートしていません。
// このシーケンスでは、直接ESnormal状態に復元します。それ以外の場合は、対応する
703 // 制御シーケンスです。シーケンスが処理されると、状態はESnormalに戻ります。 case
704     ESGotpars:
705         if(ques)ESnormal;           // 受信した「?」
706         { ques=0;
707             break;
708         }
709         スイッチ (c
710
// cが'G' または'`'の場合、par[]の最初の値は列番号を表し、それ以外の場合は
// ゼロの場合、カーソルは1列分左に移動します。
// cが'A'の場合、最初の値は、カーソルが上に移動した行数を表します。もし
// パラメータが0の場合、1行上に移動します。
// cが'B' または'e'の場合、最初のパラメータは、次のようにして下に移動した行数を表します。
// カーソルを表示します。パラメータが 0 の場合は、1 行下に移動します。
// cが'C' または'a'の場合、最初のパラメータは、その右にある列数を表します。
// カーソルを表示します。パラメータが0の場合、1列分だけ右にシフトします。
// cが'D' の場合、最初のパラメータは、カーソルの左にある列数を表します。
// パラメータが0の場合は、1列分左にシフトされます。
711         case 'G': case '`':// CSI Pn G -move horizontally
712             if (par[0]) par[0]--;
713             gotoxy(currcons, par[0], y);
714             ブレークします。

```

712 ケース 'A': // CSI Pn Aの移動 をアップして
 います。
 。

713 if (! par[0]) par[0]++;
714 gotoxy(currcons, x, y-par[0]) です
 。
715 ブレークします。

716 ケース 'B': case 'e': // CSI Pn B - move を下げ
 ています。
 。

717 if (! par[0]) par[0]++;
718 gotoxy(currcons, x, y+par[0]) です
 。
719 ブレークします。

720 ケース 'C': case 'a': // CSI Pn C - move のよう
 になります。
 // cが'E'の場合、最初の値はカーソルが下に移動した行数を表し、次の値を返します。
721 // を0列目に移動します。値が0の場合は、1行下に移動します。
722 // cが'F'の場合、最初の値はカーソルが上に移動して、その行数を表し、次のように返します。
723 // を0列目に移動します。パラメータが0の場合は、1行上に移動します。
724 // cが'd'の場合、最初の値はカーソルのある行番号(0から数えて)をmove の左で
 //が必要となります。
725 // cが'H'または'f'の場合、最初の値はカーソルの移動先の行番号を表します。
726 // で、2番目のパラメータは、カーソルを移動させる列番号を表します、x-par[0], y です
727 case 'E': // CSI Pn E - 下に移動、col 0. if (!
728 par[0]) par[0]++;
729 gotoxy(currcons, 0, y+par[0]);
730 ブレークします。
731 case 'F': // CSI Pn F - 上に移動、col 0. if
732 (! par[0]) par[0]++;
733 gotoxy(currcons, 0, y-par[0]);
734 ブレークします。
735 case 'd': // CSI Pn d - カーソル行番号の設定 if
736 (par[0]) par[0]--;
737 gotoxy(currcons, x, par[0]);
738 ブレークします。
739 case 'H': case 'f': // CSI Pn H - set cusor postion.
740 if (par[0]) par[0]--;
741 if (par[1]) par[1]--.
742 gotoxy(currcons, par[1], par[0]);
743 break;
744 // 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、方法を表します。
 // カーソルの位置に関連して、画面がクリアされます。
 // cが'K'('ESC [Ps K')の場合、最初のパラメータは、文字がどのように
 行内の // は、カーソル位置に関連して削除されます。
 // cが'L'('ESC [Pn L')の場合は、カーソル位置にn行が挿入されることを意味する。
 // cが'M'('ESC [Pn M')の場合は、カーソル位置でn行が削除されることを意味する。
 // cが'P'('ESC [Pn P')の場合、カーソル位置で n 文字が削除されることを意味します。
 // 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。
745 case 'J': // CSI Pn J - 画面上の文字を消す。
746 csi_J(currcons, par[0]);
747 ブレークします。
748 case 'K': // CSI Pn K - 一行の文字を消去する。


```

791 }
792     if (c=='b')
793         vc_cons[currcons].vc_bold_attr
794             = par[0];
795     }
796     ブレークし
// ステータスESfunckeyは、以下のマップシーケンスキーからのシーケンスを受信したことを示します。
//
797 // キーボードを表示しません。表示前の必要がないので、通常の状態であるESnormalに戻ります。
// ヨンキーです。
798     state = ESnormal;
799 // 状態ESsettermは、DCS (Device Control String) シーケンス状態であることを示しています。
// この時、文字'S'を受信した場合、初期表示文字属性は
// 復元しました。文字が'L'または'l'の場合、折り畳み表示モードのON/OFFを行う。 case
800     ESsetterm:/* Setterm関数です。*/
801     state = ESnormal;
802     if (c == 'S') {
803         def_attr = attr;
804         video_erase_char = (video_erase_char&0x0ff) | .
805             (def_attr<<8)となっています。
806     } else if (c == 'L')
807         ; /*linewrap on*/
808     else if (c == 'l')
809         ; /*linewrap off*/
     ブレークします。

// 状態ESsetgraphは、設定された文字セットのエスケープシーケンス'ESC ('または'ESC )'が
// を受信しました。これらは、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されます
//
// このとき、文字'0'を受信すると、グラフィック・キャラクタ・セットを
// G0とG1で、受信した文字が'B'の場合、通常のASCII文字セットが選択される
810 G0とG1のキャラクターセットとしての//。
811     case ESsetgraph: // 'CSI (0)' or 'CSI (B)' - select char set.
812         state = ESnormal;
813         if (c == '0')
814             translate = GRAF_TRANS;
815         else if (c == 'B')
816             translate = NORM_TRANS;
817             のデフォブレークします。
818             ルトですstate = ESnormal;
819     }
820 }
// 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。
821     set_cursor(currcons);
822 }
823
824 /*
825 * void con_init(void);
826 *
827 * このルーチンは、コンソールを活 割り込みが入っても何もしない
828 * 性化します。
829 * その他もし、画面に           クリアの場合は、tty_writeを
830 * 適切なエスケープ・シークエース
831 * 保存されている情報を読み取る by setup.sで現在のディスプレイを決定

```

```

832 * タイプに応じてすべてを設定します。
833 */
834 ポイ con_init(void)
835 {
836     登録されたunsigned char a;
837     char *display_desc = "????";
838     char *display_ptr;
839     int currcons = 0; // 現在のコンソール番号
840     ロングベース、ターム
841 // まず、long video memory; // 位置と表示レジスタのインデックスポート番号と表示レジスタのデータポート番号
842 video num columns = ORIG VIDEO COLS... // この関数に固有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)。
843     video size row = video num columns * 2; // ディスプレイの列で使用されるバイト。
844     video num lines = ORIG VIDEO LINES; // ディスプレイの//ライン。
845     video page = ORIG VIDEO PAGE; // 表示ページ。
846     video erase char = 0x0720; // char: 0x20, attr:0x07.
847     blankcount = blankinterval; // 単位: ティク
848
849 // そして、表示モードがモノクロかカラーかに応じて、表示メモリのスタート
// 位置と表示レジスタのインデックスポート番号と表示レジスタのデータポート番号
// がそれぞれ設定されます。取得したBIOS表示モードが7に等しい場合、それは
// モノクロのディスプレイカードです。
850     if (ORIG VIDEO MODE == 7) /* これはモノクロのディスプレイですか? */
851     {
852         video mem base = 0xb0000; // モノクロディスプレイの開始アドレス。
853         video port reg = 0x3b4; // インデックスレジスタポート。
854         video port val = 0x3b5で // データレジスタポート。
855
856 // 次に、ディスプレイカードがモノクロカードかカラーカードかを判断します。
// BIOS割り込みint 0x10関数0x12で取得した表示モード情報に if
// 割り込みで得られたBXレジスタの戻り値が0x10になっていないことを意味します。
// EGAカードなので、初期表示タイプはEGAモノクロです。がありますが
// EGAカードの表示メモリを増やしても、32KBまでしか使用できないので、その分を
// モノクロモードでは0xb0000--0xb8000のアドレス範囲になります。このコードでは、ディスプレイに
// 説明文字列を「EGAm」にすると、画面右上に表示されます。
// システムの初期化時に
// なお、BXレジスタは、カードの種類が変更される前や
// 割り込み後にint 0x10が呼び出されます。割り込みの後にBLの値が変更された場合は
// 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
// 以降のVGAの//。割り込みコールの戻り値が変化していない場合、示す
857 // ディスプレイカードがこの機能をサポートしていないということは、一般的な
858 // モノクロのディスプレイカード。
859     if ((ORIG VIDEO EGA BX & 0xff) != 0x10)
860     {
861         video mem term = 0xb8000; // ビデオメモリの終端アドレス
862         displaydesc = VIDEOTYPE EGAM; // ビデオタイプ (EGAのモノラル)。
863     }
864
865 // BXレジスタの値が0x10に等しい場合、モノクロディスプレイカードMDAを意味する
866 // そして、8KBのディスプレイメモリ
867 // しかない。 else

```

```

862         {
863             video_type = VIDEO_TYPE_MDA; // MDAモノ。
864             video_mem_term = 0xb2000; // メモリエンドアドレス。
865             display_desc = "*MDA "です。
866         }
867 // 表示モードが7でない場合は、カラーディスプレイカードであることを示しています。このとき、
868 // ディスプレイの
869 // テキストモードで使用するメモリの開始アドレスは0xb2000です。*/
870 // それ以外のモードで使用するメモリの終了アドレスは0xc0000です。
871 // その他のモードで使用するメモリの終了アドレスは0xb8000です。
872 // これが外の場合は、ビデオカードのvideo_port_regが0x3d4とならない限り、ビデオカードのvideo_baseが0xb8000でなければなりません。
873 // これが内蔵の場合は、ビデオカードのvideo_port_regが0x3d4とならない限り、ビデオカードのvideo_baseが0xb8000でなければなりません。
874 if((ORIG_VIDEO_EGA_BY & 0xff) != 0x10) // データレジスタポート
875     {
876         video_type = VIDEO_TYPE_EGAC; // EGAタイプ。
877         video_mem_term = 0xc0000; // memの終了アドレス。
878         display_desc = "EGAc";
879     }
880 // その他
881 {
882     video_type = VIDEO_TYPE_CGA; // CGAタイプ。
883     video_mem_term = 0xba000; // メムの終了アドレス。
884     display_desc = "*CGA "となります。
885 }
886 }

// では、現在のディスプレイで開くことのできるバーチャルコンソールの数を計算してみましょう
// カードです。ハードウェアで許可されているバーチャルコンソールの数は、合計で
// メモリの video_memory を、各仮想コンソールが占有するバイト数で割ったもの。のは
// 各バーチャルコンソールが占有するディスプレイメモリの数は、ライン数と同じです。
// 文字の1行あたりのバイト数を掛けたもの: video_num_lines * video_size_row。
// 許可されたバーチャルコンソールの数が、システムで定義された最大数よりも多い場合は
// MAX_CONSOLESに、バーチャルコンソールの数を設定します。もし、仮想コンソールの数が
// このようにして算出された//コンソールは、0であれば1に設定されます。最後に、ビデオメモリの合
// 計数を分割した
// バーチャルコンソールの数によって、各バーチャルコンソールが占有するメモリのバイト数が決ま
887 ります。
888 // コンソール (VC) です。
889 if(NR_CONSOLES > MAX_CONSOLES) video_mem_base = max_consoles = 8
890 else NR_CONSOLES = MAX_CONSOLES;
891 if (! NR_CONSOLES)
892     nr_consoles = 1;
893 video_memory /= NR_CONSOLES; // 各vcのメモリ。
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
896 る */
897 // そして、説明文字列を画面の右上に表示します。メソッドは
898 // 文字列をディスプレイメモリの対応する場所に直接書き込むことができます。

```

```

// まず、ディスプレイポインタ display_ptr が右端の最後の 4 文字を指します。
// の最初の行をコピーし（各文字は2バイト必要なので、マイナス8）、その後、循環的に
// display_ptr = ((char *)video_mem_base) + video_size_row - 8;
897     while (*display_desc)
898     {
899         *display_ptr++ = *display_desc++;
900         display_ptr++;
901     }
902
903
904     /* スクロールに使われる変数の初期化(主にEGA/VGA) **.
905
906 // この時、現在の仮想コンソール番号 currcons は 0 に初期化されていることに注意してください
907 .
908 // つまり、以下は実際に構造体vc_cons[0]のすべてのフィールドを初期化しています。
909 // 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。
910 // 上記の115。以下ではまず、デフォルトのスクロール開始メモリ位置 video_mem_start を設定しま
911 //す。
912 // とデフォルトのスクロールする最終行のメモリ位置（実際には、これらはディスプレイの一部
913 //です
914 // base = origin = video_mem_start + video_num_lines * video_size_row; // デフォルトのスクロール開始位置
915 // term = video_mem_end = base + video_memory; // #0の終了位置
916 // bottom = video_num_lines; // スクロールのボトムライン数。
917 // attr = 0x07; // デフォルトの属性（黒地に白）。
918 // def_attr = 0x07; // デフォルトのchar属性を設定します。
919 // restate = state = ENormal; // エスケープシーケンスの現在と次の状態。
920 // checkin = 0;
921 // ques = 0です。 // クエスチョンマークのフラグが立った。
922 // iscolor = 0です。 // カラーフラッグ。
923 // translate = NORM_TRANS; // 使用されるcharセット（通常のASCIIテーブ
924 //ル）。
925 // vc_cons[0].vc_bold_attr = -1; // 太字のchar属性フラグ(-1は使用しない)。
926
927 // コンソール0の現在のカーソル位置とメモリ位置posを設定した後に
928 // カーソルに対応する // 残りのバーチャルコンソールの構造をループします。
929 // それぞれが占有しているディスプレイメモリの開始位置と終了位置を除いて、残りの
930 // gotoxy(currcons, ORIG_X, ORIG_Y);
931
932     for (currcons = 1; currcons < NR_CONSOLES; currcons++) {
933
934         vc_cons[currcons] = vc_cons[0]; // #0の構造体データをコピー
935         origin = video_mem_start = (base += video_memory);
936         scr_end = origin + video_num_lines * video_size_row;
937         video_mem_end = (term += video_memory);
938         gotoxy(currcons, 0, 0); // カーソルは左上にあります。
939     }
940
941 // 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
942 // る
943 // 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。
944 // 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、
945 IRQ1の
946 // 応答すべきキーボードからの要求信号。最後にキーボードコントローラをリセットする
947 // キーボードが正常に動作を開始できるようにするために、 // を使用します。
948 // update(screen(0x21)&0x20, 0x21); // ウンドの原点と本二塗山を復元を解除します
949 // set_trap_gate(0x21, &keyboard_interrupt); // system.hの36行目を参照してください。
950 // a= inb_p(0x61)となります。 // キーボードポート0x61 (8255A PB) を
951 // 読みます。

```

```

932     outb_p(a|0x80, 0x61)となります          // キーボードを無効にする（ビット7を
933     。                                         設定）。
933     outb_p(a, 0x61)となります。           // キーボードをリセットするには、も
934 }// 現在のフォアグラウンドのコンソールを更新します。         う一度enableしてください。
935 // フォアグラウンドコンソールをfg_consoleで指定したバーチャルコンソールに切り替える
fg_console
// はフォアグラウンドのバーチャルコンソールの番号セットです。
936 void update_screen(void)
938 {      set_origin(fg_console);           // スクロール開始メモリのアドレスを設定し
939     set_cursor(fg_console)            ます。
940 }      。                                         // コントローラ内のカーソル位置を設定しま
941
942 /* from bsd-net-2: */?
943
    //// ピープ音を止める。
    // 8255AのPBポートのビット1とビット0をリセットします。の後のタイマープログラミングの説明を
    参照してください。
    // kernel/sched.cプログラムです。
944 void sysbeepstop(void)
945 {      /* カウンタ2を無効にする */
946     outb(inb_p(0x61)&0xFC, 0x61)
947     となります。
948 }
949
950 int beepcount = 0;                         // ピープ音のカウント（テ
951                                         ィック）。
952
    // ピープ音を鳴らす
    // 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ブ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
    // ピープ音を鳴らすためには、2つのステップが必要です。まず、PBポート(0x61)のビット1とビット0をオン（セット）にし、次に
    // タイマー2チャンネルに特定のタイミング周波数を送るように設定します。8259Aの割り込みコントローラを参照
    boot/setup.sプログラムの後に、//チップのプログラミング方法を参照してください。
    // kernel/sched.cプログラムの後にタイマーの命令を出す。
953 static void set_beep(int arg)
954 {      /* カウンタ2を有効にする */
955     outb_p(inb_p(0x61)|3, 0x61)と
956     なります。
957     /* カウンタ2のセットコマンド 2 byte write */
958     outb_p(0xB6, 0x43)となります        // タイマーコントロールワードレジスタポート
959     。
960     750 HZの場合は0x637を送信 */
961     .
962     outb_p(0x37, 0x42)となります        // チャンネル2にハイ&ローカウントのバイトを
963     。
964     outb(0x06, 0x42)です。
965     /* 1/8秒 */
966     // 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。
967     // 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
968     // 画面をダンプする。ポインタとして機能します。
969     int do_screendump(int arg)
970 {

```

```

967     char *sptr, *buf = (char *)arg;
968     int curcons, l;
969
// この関数は、まずユーザが提供したバッファ容量を確認し、それが十分でない場合は
// が適切に展開されます。そして、その先頭からコンソール番号のカーコンを取ります。その後
// コンソール番号が有効であると判断された場合、コンソール画面のすべてのメモリコンテンツが表
示されます。
// がユーザー バッファにコピーされます。
970     verify_area(buf, video_num_columns*video_num_lines);
971     currcons = get_fs_byte(buf);
972     if ((currcons <= EIO) || (currcons > NR_CONSOLES))
973         currcons--;
974
975     sptr = (char *) origin;
976     for (l=video_num_lines*video_num_columns; l>0 ; l--)
977         put_fs_byte(*sptr++, buf++);
978
979 } // がユーザー バッファにコピーされます。
980
// ブラックスクリーン処理。
// blankInterval の間にユーザーが何もボタンを押さなかった場合、画面は
// 画面を保護するために黒く表示しています。
981 void blank_screen()
982 {
983     if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
984         return;
985 /* ここは空白です。やり方がわからないのですが、*1
986 */
987
// 黒い画面を元に戻す。
// ユーザーが任意のボタンを押すと、ブラックスクリーン状態の画面が復元される。
988 void unblank_screen()
989 {
990     if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
991         return;
992 /* ここでアンブランク */
993 }
994
//// コンソールプリント機能です。
// この関数は、カーネルの表示関数である printk() (kernel/printk.c) によってのみ使用され、印刷
を行います。
// 現在のフォアグラウンド・コンソールのカーネル情報です。処理方法は、ループアウトして
// バッファ内の文字をカーソルの動きに合わせて制御したり、画面に直接表示する
文字の特性に応じて、//。引数 b は、マル終端の文字列
// バッファポインタ。
995 void console_print(const char * b)
996 { int currcons = fg_console;
997     char c;
998
999
// バッファ内の文字を循環させます。現在の文字が改行文字の場合。
// カーソルでキャリッジリターン・ラインフィード操作が行われ、次に次の文字
// が処理されます。それがキャリッジリターンであれば、直接キャリッジリターンを実行します。そ
の後
1000 // 次の文字に進みます。
        while (c = *(b++)) {...
```

```

1001         if (c == 10) {
1002             CR (カーコン)
1003             .
1004             If (curcons) です
1005             .
1006         }
1007         if (c == 13) {
1008             // 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
1009             // 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
1010             // 次の行の先頭に戻ります。そして、その文字を
1011             // 画面に表示されている、カーソルの位置です。カーソルを右に1列移動
1012             // 次の文字の準備をします。
1013             if (x==video_num_columns) {...}
1014                 x -= video_num_columns;
1015                 pos -= video_size_row;
1016                 If (currcons);
1017             }
1018             // レジスタALには、表示する文字が入ります。ここではAHに属性バイトが入ります。
1019             // となり、AXの内容がカーソルメモリの位置posに格納される、つまり、文字
1020             // カーソルの位置に // が表示されます。
1021             Asm ("movb %2,%ah\%t")           // 属性バイト→あ。
1022             "movw %%ax,%1n\%t"           // posにaxの値を入れます。
1023             :: "a"(c)のようにな
1024             // ります。
1025             "m" (*(short *)pos)
1026             // です。
1027             "m" (attr)
1028             : "ax" )。
1029             pos += 2;
1030         }
1031         // 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。
1032         set_cursor(currcons);
1033     }
1034 }
```

3. インフォメーション

1. VGAアダプタのプログラミング

ここでは、IBMの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ディスプレイコントロールカードの汎用プログラミングポートです。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

ポート	R/W	名称と使用方法				
0x3b4/0x3d4	W	6845のインデックスレジスタ。ポート0x3d5からどのレジスタ (r0～r17) にアクセスするかを選択するために使用します。				
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。 各データレジスタの機能説明を表10-6に示す。				
0x3b8/0x3d8	R/W	6845モードコントロールレジスタ ビット7-6未使用 bit5=1が点滅しています。 bit 4=640*200 B/Wグラフィックモード、bit 3=ビデオ信号有効。 bit 2=白黒、0=カラー。 ビット1=320*200のグラフィック、=0のテキスト、ビット 0=80*25のテキスト、=0の 40*25のテキスト。				
0x3b9/0x3d9	R/W	CGAのパレットレジスタ。使用する色を選びます。 ビット7-6未使用 bit 5=1 シアン、マゼンタ、ホワイトのカラーセットを有効にする。 =0 カラーセット（赤、緑、青）を有効にします。 bit 4=1 グラフィックス、テキスト表示の背景色を強調する。 bit 3=1 40*25のボーダー、320*200の背景、640*200の前景色の表示を強化 bit 2=1 赤を表示。40*25のボーダー、320*200の背景、640*200の前景。 bit 1=1 緑を表示します。40*25のボーダー、320*200の背景、640*200の前景、ビット0=1 青を表示。40*25のボーダー、320*200の背景、640*200の前景。				
0x3ba/0x3da	R	CGAステータスレジスタ。 ビット7-4未使用 bit 3=1 virtual retrace, RAMアクセスOK for next 1.25ms; bit 2=1 light pen off, =0 light pen on; ビット1=1 ライトペントリガセット bit 0=1 表示を妨げずにメモリにアクセスする; =0 今回はメモリを使用しない。				
0x3bb/0x3db	W	ライトペントリガセットをリセットする（強制的にライトペンのストロボは有効）。 表10-6 MC6845内部データレジスターと初期値				
0x3dc	R/W	ライトペンのラッチを事前に設定する（強制的なライトペンのストロボは有効）。 グラフィックモード				
Reg.	登録名	ユニット	R/W	40*25モード	80*25モード	グラフィックモード
r0	水平方向の合計	シャル	W	0x38	0x71	0x38
r1	水平方向の表示 水平方向の同期位置	シャ	W	0x28	0x50	0x28
r2	水平方向の同期ポールス幅	ル	W	0x2d	0x5a	0x2d
r3		シャ	W	0x0a	0x0a	0x0a
		ル				
r4	垂直方向の合計	チャーシュー	W	0x1f	0x1f	0x7f
R	垂直方向のトータル走査線	走査線	W	0x06	0x06	0x06
5	アジャスト 垂直方向のディスプレイ表示	シャル	W	0x19	0x19	0x64
R	垂直方向のシン	ロー シ	W	0x1c	0x1c	0x70
6	クポジション	ヤルロ				
R		一				
7						
r8	インターレースモード	587	W	0x02	0x02	0x02
r9	最大スキャンラインアドレス	走査線	W	0x07	0x07	0x01

r10	カーソルスタート	走査線	W	0x06	0x06	0x06
r11	カーソル終了	走査線	W	0x07	0x07	0x07
r12	Start Mem Address (High)		W	0x00	0x00	0x00
r13	Start Mem Address (Low)		W	0x00	0x00	0x00
r14	カーソル位置 (高)		R/W	Vary		
r15	カーソル位置 (低)		R/W	Vary		
r16	ライトペン (高)		R	Vary		
r17	ライトペン (低)		R	Vary		

10.3.3.2 スクロール操作の原理

スクロールとは、画面上の指定された開始行と終了行上の文字列を上に（スクロールアップ）、または下に（スクロールダウン）移動させることをいう。画面を、対応する画面内容を上に表示するウィンドウと考えると

画面の内容を上に移動させると、ウィンドウが表示メモリの下に移動し、画面の内容を下に移動させると、ウィンドウが上に移動することになります。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、プログラムはメモリのデータを移動させます。

画面に対応する実際の表示メモリの先頭まで（video_mem_start）。

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

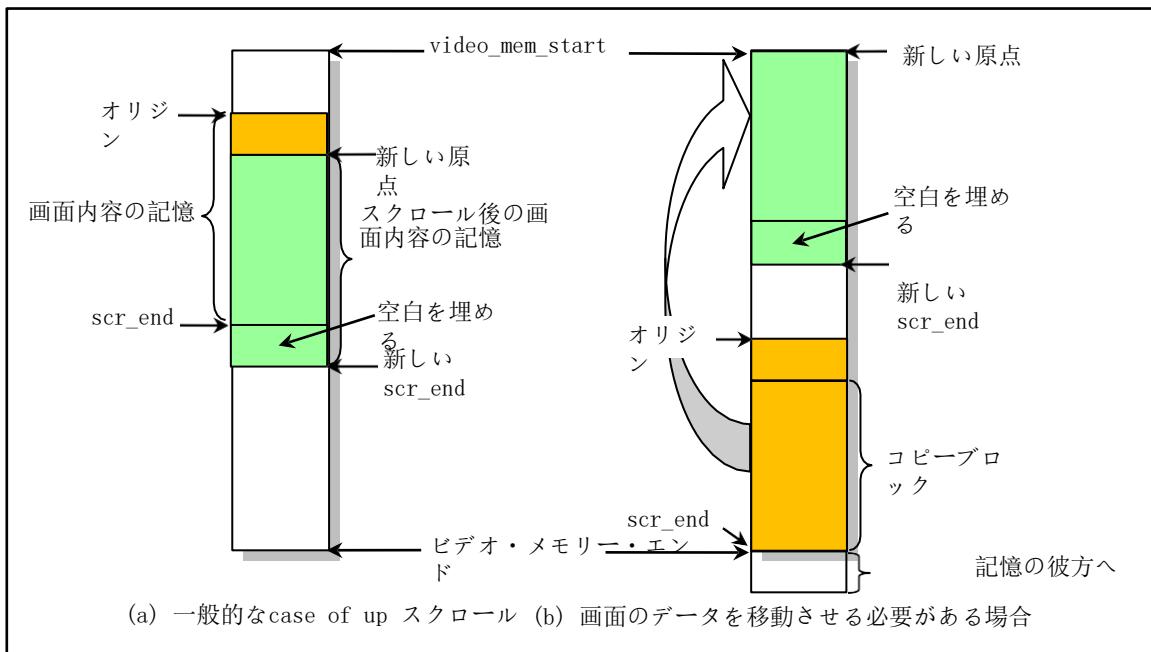


図10-8 スクロールアップ操作の模式図

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 端末制御コマンド

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。

1. 文字符串化方式

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串化規格を定めている。ANSI X3.4-1977とISO 646-1977では、American Standard Code for Information Interchange、つまりASCIIコードセットを定義している。ANSI X3.41-1974とISO 2022.2は、7ビットと8ビットのコードセットのコード拡張技術を規定しています。ANSI X3.32、ANSI X3.64-1979では、ASCIIコードのテキスト文字を使って端末制御文字を表現する方法が開発されています。Linux 0.1xのカーネルでは、Digital Equipment Corporation DEC（現在はCompaqとHPに組み込まれている）のVT100とVT102のターミナルデバイスとの互換性しか実装されていませんが。

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

2.ASCIIコード表

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)も制御文字です。文字、そしてスペース文字 (0x20) は、通常のテキスト文字か制御文字のどちらかになります。制御文字とその機能はANSIによって標準化されており、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コード				GLコード				C1コード		GRコード					
	7ビットコードテーブル								8ビットコード表の右半分							

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（8列目～15列目）の各コードの8ビット目はすべて1なので、これらの文字は8ビット環境でしか使用できません。8ビットコード

テーブルには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種類があります。

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。

3. コントロール機能

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、表示の制御にのみ使用される。

制御機能とは、テキスト文字の処理や送信を行う機能であり、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、テキスト行の削除、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。

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。

4. エスケープシーケンス

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つまたは複数のASCII表示文字を入力します。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。 0x20–0x2f 0x30–0x7e

イントロデューサー	中級	文字	ファイナルチャヤー
		(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）の範囲です。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
イントロデューサー	パラメータの文字 (0または複数のchars)	中級者向けの文字 (1つのchars)	ファイナルチャー

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の中のCSI (0x9b) です。しかし、CSIは7ビットのコード拡張「ESC []」でも拡張できるため、すべての制御シーケンスは、以下の方法で表現できます。

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

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

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

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

```
CSI 5 ; 9 H
0x9b 0x35 0x3b 0x41 0x48
や
。 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

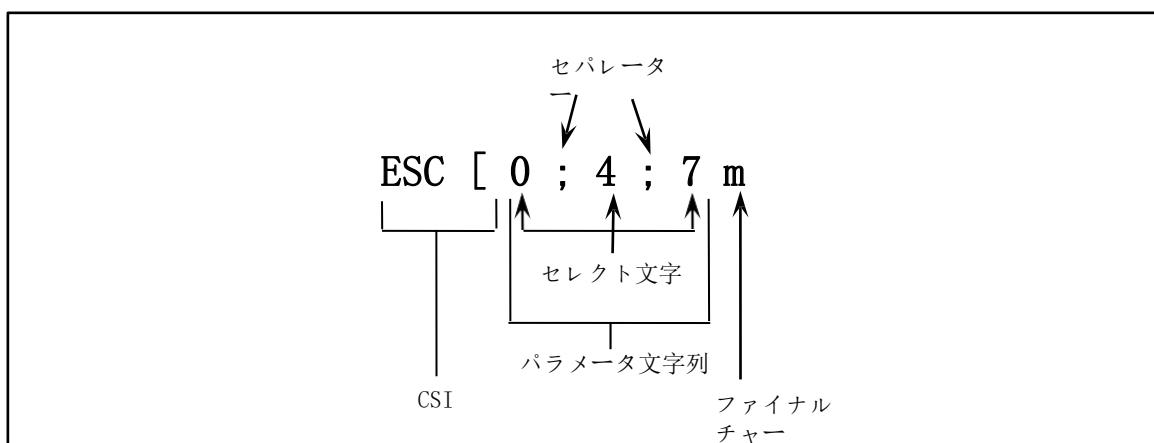


図10-9 制御シーケンスの例

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ます。各制御関数には固有の名前と略記法があります。これらの名前とニーモニックは標準的なものです。デフォルトでは、制御文字や表示文字に対するターミナルの解釈は、使用されているASCII文字セットに依存する。サポートされていない制御コードの場合、ターミナルが取る通常のアクションはそのコードを無視することであることに注意してください。ここに記載されていない文字がターミナルに送信されると、予想外の結果になる可能性があります。

本書の付録では、C0表とC1表の中でよく使われる制御文字の説明と、端末がそれを受け取ったときの動作の概要を示しています。特定の端末では、通常、C0と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。

4. シリアル.c

1. 機能

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

関数rs_write()は、ファイルシステムでキャラクタデバイスファイルを使用する際に呼び出される。プログラムがシリアルデバイスの /dev/tty64 ファイルに書き込む際には、システムコールのsys_write()が実行される(fs/read_write.c内)。システムコールで、読み込まれているファイルがキャラクタデバイスファイルであると判断されると、rw_char()関数(fs/char_dev.c)が呼び出される。この関数は、読み込んでいるデバイスのサブデバイス番号などの情報に応じて、キャラクタデバイスの読み書き機能テーブル(デバイススイッチテーブル)を用いて、rw_tty()を呼び出す。これは最終的に、ここでシリアル端末の書き込み関数rs_write()を呼び出すことになります。

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

2. コードアノテーション

プログラム 10-3 linux/kernel/chr_drv/serial.c

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

```

4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * serial.c
9 *
10 * RS232 io機能を実装しています。
11 *     void rs_write(struct tty_struct * queue);
12 *     void rs_init(void);
13 *とシリアルIOに関わる全ての割り込みを行います。
14 */
15
16 // <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。
17 // 通信です。
18 // <linux/init.h> のデバッガと、起動時のシリアルポート設定を取得する構造体の組み込みアセンブリ関数マクロの記述があります。
19 // <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。
20 // 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。
21 // <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。
22 // マクロの組み込みアセンブリの形式です。
23 #include <linux/tty.h>
24 #include <linux/sched.h>
25 #include <asm/system.h>
26 #include <asm/io.h>
27
28 // 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。
29 #define WAKEUP_CHARS (TTY_BUF_SIZE/4)
30
31 extern void rs1_interrupt(void);           // シリアル1割り込みハンドラ(rs_io.s, 34)
32 extern void rs2_interrupt(void);           // シリアル2割り込みハンドラ(rs_io.s, 38)
33
34 ///////////////////////////////////////////////////////////////////
35 // シリアルポートの初期化
36 // 指定したシリアルポートの送信ボーレート (2400bps) を設定し、すべての割り込みを許可する
37 // のソースは、ライトホールドレジスタ以外は空です。また、2バイトポートの出力時には
38 // また、ラインコントロールレジスタのDLABビット (ビット7) を設定する必要があります。
39 // 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。
40 static void init(int port)
41 {
42     .    outb_p(0x80, port+3) /* ラインコントロールREGのDLABを設定 */
43     .    outb_p(0x30, port)   /* 除算器のLS(48→2400bps)*/
44     .
45     outb_p(0x00, port+1) /* 除算器のMS*/
46     .
47     outb_p(0x03, port+3) /* DLABをリセットします。*/
48     .
49     outb_p(0x0b, port+4) /* DTR, RTS, OUT_2を設定 */
50     .
51 ///////////////////////////////////////////////////////////////////
52 // シリアルポートを初期化でチップ書き込み以外のすべての侵入を可能にする。
53 // 割り込みディスクリプターテーブルのゲートディスクリプター設定マクロset_intr_gate()は
54 // include<asm/io.h>で実装され、データポートを読み込んでリセットする
55 void rs_init(void)                         (?)
56 {

```

```

// 次の2つの文は、2つのシリアルの割込みゲート記述子を設定するために使用されます。
// rs1_interruptは、シリアルポート1の割込みハンドラポインタです。使用される割込みは
// シリアルポート1による//はint 0x24、ポート2はint 0x23です。表5-2およびsystem.hファイルを参照
39 してくださり intr_gate(0x24, rs1_interrupt) ; // ポート1(IRQ4)のintゲートディスクリプタを設
40  定する。
41  set_intr_gate(0x23, rs2_interrupt) ; // ポート2(IRQ3)のintゲートディスクリプタを設
42  定します。
43  init(tty_table[64].read_q->data) ; // ポート1を初期化する (.dataはベースアドレス
44 }  ます。 )。
45  // ポート2の初期化
46 /*
47 * このルーチンは、tty_write が何かを入力したときに呼び出されます。
48 * write_queueのことです。キューが空であるかどうかをチェックして
49 * 割り込みレジスタの設定
50 */
51 * void _rs_write(struct tty_struct * tty);
52 */
53 void rs_write(struct tty_struct * tty)
54 {
    // 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。シリアルインター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、システムはこの処理を繰り返します。
    // と文字を1つずつ送信していきます。書き込みキューの中のすべての文字が送信されると
    // アウト、書き込みキューが空になり、割り込みハンドラが送信ホールドレジスタをリセット
    // 割り込みイネーブルレジスタの // 割り込みイネーブルフラグで、再び送信
    // ホールドレジスタを空にすると、割り込み要求が発生します。この時点で、"ループ"送信動作の
    // 終了します。
55 // 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。
    // をレジストに房。dataができます。アスレポートのよみにレジストアマリを下を取得したいときに割り込みを
56 // 開始することができます。
57 // 送信保持レジスタが空のときに送信する文字。ただ、次のことに注意してください。
58     outb(inb_p(tty->write_q->data+1) | 0x02, tty->write_q->data+1)。
59     sti() です。
60 }

```

3. インフォメーション

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

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

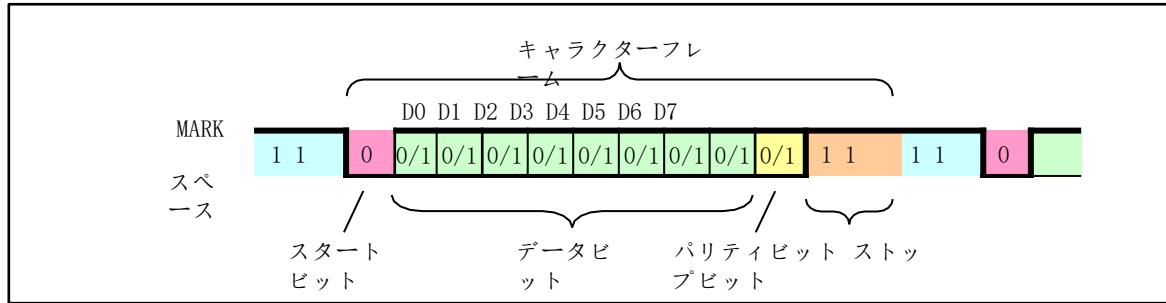


図10-10 非同期シリアル伝送のフレームフォーマット

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できます。(1)パリティエラー。(2)パリティエラー この時点では、プログラムは相手に文字の再送を依頼しなければなりません。このエラーは、プログラムが受信速度よりも遅い速度で文字を取り込んだために発生します。ここでは、プログラムを修正して、文字の周波数を速くする必要があります。このエラーは、受信要求されたフォーマット情報が正しくない場合に発生します。例えば、ストップビットを受信すべきところを、SPACEビットを受信してしまった場合などです。一般的には、回線干渉の他に、通信相手のフレームフォーマットの設定が異なることが考えられます。

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ています。そのため、RS-232C

規格では、多くのMODEM固有の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、他の資料を参照してください。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ーピン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。

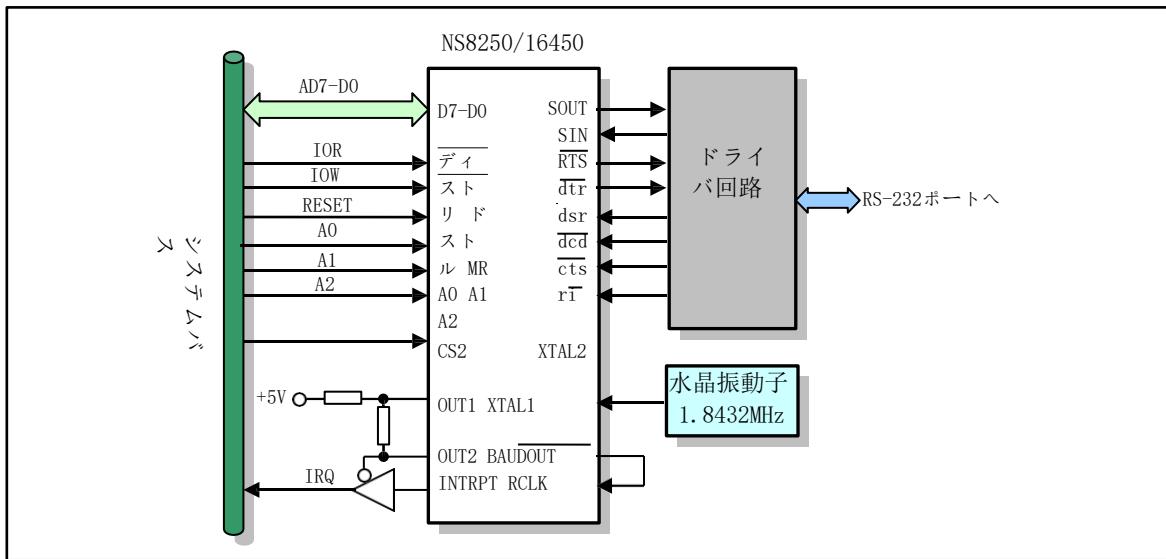


図10-11 NS8250/16450の基本的なハードウェア構成図

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(CTS)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、通信回線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入力する方法があります。PCでは後者の方法で、XTAL1端子に1.8432MHzのクロック信号を直接入力しています。端子BAUDOUTからはUART送信ボーレートの16倍信号が出力され、端子RCLKからはボーレートの受信したデータを両者が接続されているので、PCでデータを送受信する際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。

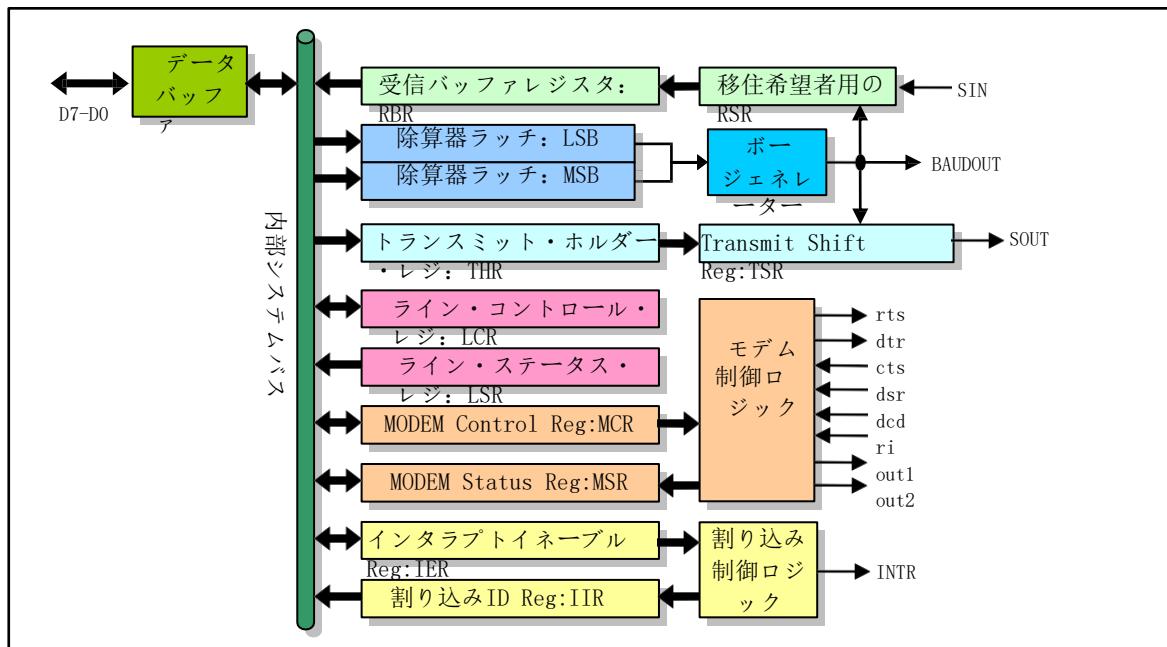


図 10-12 NS8250 UART 内部コンポーネントの基本ブロック図

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。

ポートアドレス	R/W	条件	表10-8 UART内部レジスタのポートと目的
0x3f8 (0x2f8)	W	DLAB=0	送信ホールディングレジスタ (THR) を書き込み、送信される文字を格納します。
	R	DLAB=0	受信バッファレジスタ RBR の読み出し。受信した文字が含まれています。
	R/W	DLAB=1	ボーレートファクターの下位バイト (LSB) を読み書きする。
0x3f9 (0x2f9)	R/W	DLAB=1	ボーレートファクターの上位バイト (MSB) を読み書きする。
	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 送信保留レジスタエンブティ割り込み許可、 Bit 0=1 受信データ割り込み許可。
0x3fa (0x2fa)	R		割り込み識別レジスタ IIR の読み出し。割り込みハンドラを使用して4つのタイプのうち、どのタイプが割り込まれたかを決定します。ビット7~3はすべて0（未使用）。 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の状態が変化、優先度4。リセットするにはMODEMの状態を読み取ってください。

			ビット0=0 保留中の割り込み; =1 割り込みなし。
0x3fb (0x2fb)	W		<p>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ピット (DLAB)。 =0 レシーバ、送信ホールド、または割り込みイネーブルレジスタのアクセス；ビット6=1でブレーク可能。</p> <p>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 パリティあり、=0 パリティなし。</p> <p>ここで、ビット2=1は、データビット長に依存します。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ビットとなり、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ビットとなります。 =0 ストップビットは1ビット。</p> <p>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個の7ビットデータビット = 11 8ビットのデータビット</p>
0x3fc (0x2fc)	W		<p>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。</p> <p>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。</p> <p>ビット3=1、補助ユーザが出力2を指定し、システムへのINTRIPTを有効にする。ビット2=1、補助ユーザが出力1を指定し、PCを使用しない。</p> <p>ビット1=1でRTS送信要求が有効になります。</p> <p>ビット0=1は、データ端子レディーDTRをアクティブにします。</p>
0x3fd (0x2fd)	R		<p>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。 ビット5=1 送信保留レジスタが空で、送信する文字を得ることができる；ビット4=1 ブレーク条件を満たすビット列を受信する。 ビット3=フレームフォー マットエラー、ビット 2=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。 ビット0=1 受信データはレディであり、システムはリード可能である。</p>
0x3fe (0x2fe)	R		<p>MODEMステータスレジスタMSRを読む。δは、信号の変化を示すかの状態になります。</p> <p>ビット7=キャリアディテクト (CD)が有効、ビット6=リングインディケーション(RI)が有効。</p> <p>ビット5=データ・デバイス・レディ (DSR) が有効、ビット4=クリア・トランシミット (CTS) が有効。</p> <p>ビット3=1でδモーリアを検出。 ビット2=1はリング信号のエッジを検出し、ビット1=1はDSR (Data Device Ready) を示す。</p>
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RESET信号がNS8250チップのMRピンを介して、UARTの内部レジスタと制御ロジックをリセットします。その後、UARTを使用したい場合は、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) 送信ボーレートの設定

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

$$\text{ボーレート・ファクター} = \frac{\text{UART Freq.}}{\text{Baudrate} \times 16} = \frac{1.8432 \text{ MHz}}{2400 \times 16} = \frac{1843200}{2400 \times 16} = 48$$

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

b) 転送フォーマットを設定します。

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

はデータのビット長を示しており、0b11の場合はデータ長が8ビットであることを示しています。

c) MODEMコントロールレジスタを設定します。

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

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

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

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

をリセットすることができます。MCRビット3=0の場合でも、MODEMの状態が変化した条件下でUARTはINTRPT端子に割り込み要求信号を発生させることができ、割り込み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) 割り込みイネーブルレジスタ IER の初期化

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条件、すなわち割り込みソースの種類表10-8に示すように、4種類の割り込みソースを選択することができます。対応するビットが1の場合は、その条件で割り込みの発生が許可されていることを意味し、そうでない場合は無効となります。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のイネーブル割り込みフラグを設定することで、割り込みソースが開始するシリアル割り込み処理中に、カーネルプログラムは書き込みキュー内のキャラクタのフェッチを開始し、その出力を送信ホールドレジスタに送ってUARTが送出することができます。UARTが文字を送出すると、送信保持レジスタは空になり、再び割り込み要求が発生します。このように、書き込みキューに文字がある限り、システムはこのプロセスを繰り返し、文字を1つずつ送信していきます。書き込みキューの文字がすべて送出されると、書き込みキューは空になり、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プログラムが読めるようにリードキューのヘッダーに置かれ、そのデータを

シリアル端末に送信する必要のある文字は、書き込みキューの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。⁶⁰²そのため、シリアル割り込みハンドラの最初のタスクは、割り込みが発生する特定の条件を決定することです。これには

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

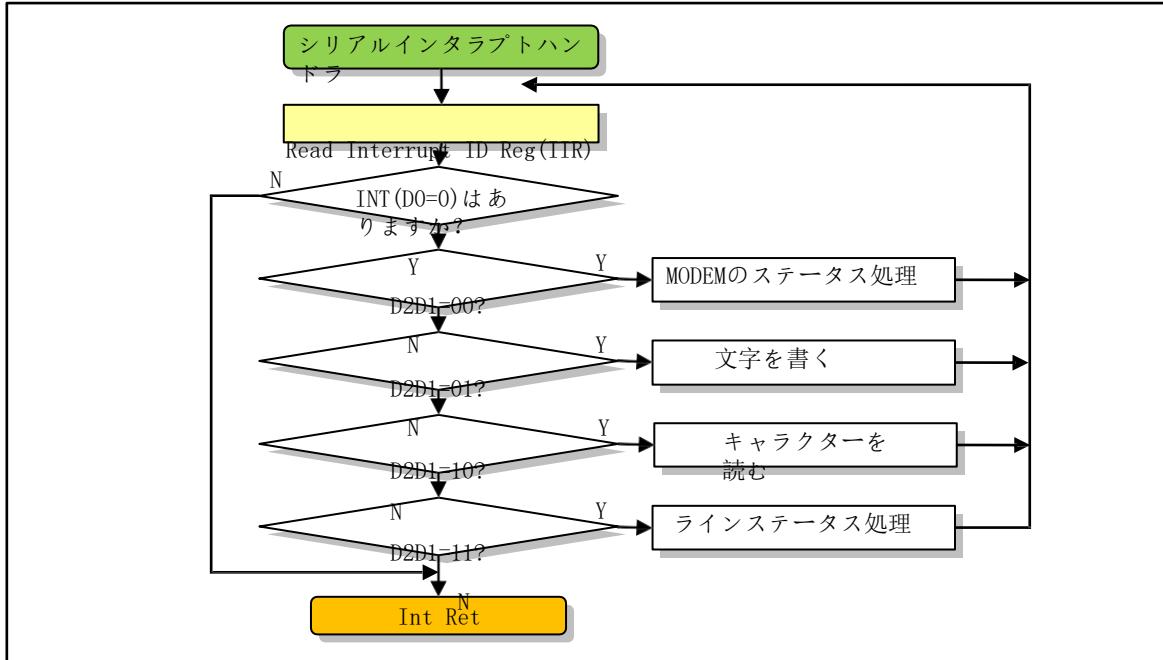


図10-13 シリアル通信割り込み処理ブロック図

IIRの内容を取り出した後は、まずビット0によって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、処理すべき割り込みがある。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。

5. rs_io.s

1. 機能

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

システムがシリアル割り込みを発生させる状況には、次の4種類があります。

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

そのため、このプログラムを読む前に、`include/linux/tty.h`というヘッダーファイルを見ておくといいでしょう。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 コードアノテーション

プログラム 10-4 `linux/kernel/chr_drv/rs_io.s`

```

1  /*
2  * linux/kernel/rs_io.s 3
3  *
4  * (C) 1991 Linus Torvalds 5
5  */ (C) 1991 Linus Torvalds
6
7 /*
8 * rs_io.s 9 *
10 * RS232のIO割り込みを実装したモジュールです。
11 */
12
13 .テキスト
14 .globl _rs1_interrupt,_rs2_interrupt
15
16 // size は読み書きのキューの長さをバイト単位で表したものです。
17 // サイズ = 1024 /* 2の累乗でなければならぬ !!! そして、tty_io.c の値と一致しなければならない !!!*/
18
19 /* these are the offsets into read/write buffer structure */
20 // include/linux/tty.hのtty_queue構造体の各フィールドのオフセットに対応します。
21 // ファイルで、rs_addrはtty_queue構造体のデータフィールドに対応しています。しかし、シリアルの場合
22 // 端末のバッファキューは、このファイルで記述されるポアートのバードオブアセッタス(0x3f88または0x2f8
23 // head保持します。 0x2f8)。
24 tail = 8 // バッファ内のヘッドポインタフィールドのオフセット。
25 proc_list = 12 // バッファ内のテールポインタフィールドのオフセット。
26 buf = 16 // バッファリングされたプロセスフィールドのオフセット
// 待機状態になります。書き込みバッファのキューに256文字以上の文字が残っていないとき。
// 割り込みハンドラは、待機中のプロセスを起こして、文字の入力を続けることができます。
// 書込みキューです。

```

```

27_スタートアッ          /* 再起動時に書き込みキューに残っていた文字数 */
プ = 256                \(`o`)
28_
29_/* これらは、実際の割り込みルーチンです。これらは
31_* 割り込みの発生源を特定し、適切な処置を行います。 32 */
     //// シリアルポート1の割り込みハンドラのエントリポイントです。
     // 初期化時、rs1_interruptのアドレスは割り込みディスクリプター0x24に配置されます。
     8259Aの割り込み要求IRQ4端子に対応する//。まず、シリアルのアドレスである
     // ttyテーブルのターミナル1（シリアルポート1）のリード / ライトバッファキューのポインタが最
     初にpushされる
     // をスタックに格納し（tty_io.c, 81）、rs_int にジャンプして処理を続けます。これにより
     // シリアルポート1とシリアルポート2の処理コードを共有することができます。文字の
     // バッファキュー構造 tty_queue は、include/linux/tty.h の 22 行目 있습니다。
33 .align 2
34 _rs1_interrupt:
35     pushl $_table_list+8    // シリアルポート1のr/wキューのポインタがスタックにpushさ
36     jmp rs_int             ュされます。
37 .align 2
     //// 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。
38 _rs2_interrupt:
39     pushl $_table_list+16 // シリアルポート2のr/wキューのポインタがスタックにpushさ
     れます。

// 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、に従つて
ビット2とビット1に // を入力すると、対応する割込みソースタイプ処理サブルーチンが呼び出され
ます。
// ポインタジャンプテーブルを使用します。各サブルーチンは、対応する割り込みをリセットします
。
// 処理後のUARTのソースです。
// サブルーチンが戻った後、このコードは他の割り込みソースがあるかどうかを判断するためにルー
41 プします。
42 // (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
43 // 割り込み全%edx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。
44 // 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
45 // のないことを確認する IIR ビット 0 = 1 が自動的に設定されます。
46 // 保留中の割り込み、つまり割り込み%edxが終了できるように。
47 rs_int%edx%eax        /* 割り込みなので、できません。
48     pushl $0x10           /* bsがOKであることを知る。ロードしてください。
49     pop %ds               // ds, esがカーネルデータセグメントを指すように
50     pushl $0x10           します。
51     pop %es
52     movl 24(%esp),%edx   // シリアルバッファのキューのアドレスを
53     movl (%edx),%edx      取得します。
54     movl rs_addr(%edx),%edx // シリアルポートの取得またはbaseaddrのベースアドレスを
                           // edx割り込み ID 登録 */
                           // IIRポートアドレスは0x3fa(0x2fa)です。
// IIR（割り込み識別バイト）の内容を取得して、割り込みの原因を特定します。
// 割り込みです。ビット2とビット1で決まる割り込み条件は4種類あります。モデム

```

```

// ステータスの変化、書き込まれる文字（送信）、読み込まれる文字（受信）、ラインの状態
// を変更しています。このコードでは、まず処理すべき割り込みがあるかどうかを判断し、次に
// それに応じて、対応するインタラプトを
55 rep_int:
56     xorl %eax,%eax
57     inb %dx,%al          // 割り込み識別バイトを取得します。
58     testb $1,%al          // bit0 = 0 ?(インタラプトは？)
59     jne end                // いいえ、最後までジャンプします。
60     CMPB $6,%AL           /* これは起こってはならないことですが、 ...*/
61     JA END                // al > 6ならば、最後までジャンプ。
62     movl 24(%esp),%ecx    // シリアルバッファのキューのアドレスを取得 ->
63     pushl %edx             ecx.
64     subl $2,%edx           // IIRポートのアドレスを保存する temperoly.
65     call jmp_table(%eax,2) /* NOTE: 本4ではなべて、名前ははずす0x308と0x48に戻す
// 上記の記述は、処理すべき割り込みがあるときに、ビット0=0の
// AL、ビット2、ビット1が割込みタイプなので、割込みを掛けるのと同じです。
// 2で入力します。そして、アドレスは4バイトなので、ここでは2を掛けて、ジャンプを得ます。
各割込みタイプのアドレスに対応する // テーブル（79行目）を作成し、そこにジャンプして
// 対応する処理
// キャラクター割り込みの送信を許可するには、送信
// ホールディングレジスターフラグ。serial.cプログラムでは、書き込みキューにデータがあるときは
// rs_write()関数は、割り込みイネーブルレジスタの内容を変更して
// 送信ホールドレジスタ割り込みイネーブルフラグ。
66 // システムが切断を送信する必要がある場合は。トアドレス0x3fa（または0x2fa）を復元します
67     jmp rep_int            // 保留中の割り込みを処理するループにジャンプします。
68 終了。      movb $0x20,%al      // 割り込みコントローラにEOI命令を送信します。
69     outb %al,$0x20         EOI */ /* EOI
70     ポップ %ds
71     ポップ %es
72     popl %eax
73     popl %ebx
74     popl %ecx
75     popl %edx
76     addl $4,%esp           # _table_listのエントリーを飛び越えて # キューのアドレスを
                                // 破棄する。
77     アイレット
78 // 各割込み種別処理サブルーチンのアドレスジャンプテーブルです。の4種類があります。
// 割り込み条件。MODEMの状態変化、書き込み（送信）される文字、送信される文字
// 読まれる（受信される）；ラインの状態が変化する。
79 jmp_tableです。
80 long modem_status,write_char,read_char,line_status 81
// この割り込みはMODEMの状態が変化したときに発生します。にリセット操作が行われます。
// MODEMステータスレジスタMSRを読み出して
82 .align 2
83 modem_status:
84     addl $6,%edx           モデムステータスレジスタを読み込んでイントロ
                                // ダクションをクリアする */。
85     inb %dx,%al            // MODEMのステータスレジスタポート。0x3fe
86     レット
87 // このシリアルインタラプトは、ラインの状態が変化したときに発生します。リセット操作が行われる

```

```

ライン・ステータス・レジスタLSRを読み込むことで、
//。
88 .align 2
89 line_stataddl $5,%edx           /* ライン・ステータス・レジスターを読み込んで
                                     イントロダクションをクリアする。*/
90     inb %dx,%al                // LSRポート。0x3fd
91     レット
92
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リード_qの先頭に入れ、ポインタを
// 1文字分の位置。もしヘッドポインタがバッファの終端に達していたら、フォールドさせます。
// バッファの先頭に戻ります。最後に、C関数のdo_tty_interrupt()（つまり。
// copy_to_cooked()）が呼び出され、読み込まれた文字がカノニカルモードに処理されます。
// バッファキュー（セカンダリバッファ）。
// 現在のシリアルポートの番号を知るには、現在のシリアルポートのキューアドレスから、その番号
を差し引いてください。
// キュー・タルベ・アドレス・テーブル・リストを8で割った結果、1がシリアル1、2が
// シリアル2。
94 .align 2inb %dx,%al           // 受信バッファレジスタRBRにある文字をALに読み込みます
95 read_chamdd $%ecx,%edx
96     subl $_table_list,%edx // シリアルバッファのキューアドレス -> edx
97     shr $3,%edx          // 現在のシリアル番号 = (キューアドレス - table_list)
98     movl (%ecx),%ecx    /#8read=queue
99     movl head(%ecx),%ebx // リードキューアヘッドポインタを取得する -> ebx.
100    movb %al,buf(%ecx,%ebx) // charを頭の位置に置く。
101    incl %ebx            // で、頭のポインタを1つ前に移動させます。
102    andl $size-1,%ebx    // ヘッドポインタをバッファの長さで変調します。
103    cmpl tail(%ecx),%ebx // 頭と尻尾のポインタを比較します。
104    je 1f                // queue is full if equal, then jump forward.
105    movl %ebx,head(%ecx) // そうでなければ、新しいヘッドポインタを保存します。
106    addl $63,%edx        // シリアル番号をtty番号（63または64）に変換し、スタックに
107    pushl %edx          プッシュする。
108 1:   _do_tty_interrupt addl 397.
109     $4,%esp             // パラを破棄して、Ret.
110     レット
111
112 // 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
// レジスタイル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
書き込みキューに現在含まれている//を計算し、それが256未満であれば、プロセスは
書き込み操作を待っている // が起こされます。その後、書き込みの最後から1文字を取り出して
// バッファのキューを調整し、テールポインタを調整して保存します。書き込みバッファのキューが
空の場合は、ジャンプ
// に write_buffer_empty のラベルを付けて対応します。
113
114 .align 2
115 write_chamdd $%ecx,%ecx      # write=queue // address -> ecx
116     movl head(%ecx),%ebx    // 書き込みキューアヘッドポインタをebxに取得
117     subl tail(%ecx),%ebx   // charsの数 = head - tail.  #
118     andl $size-1,%ebx     nr chars in queue
119     je write_buffer_empty // head == tail の場合、キューが空であれば、ジャン
120     cmpl $startup,%ebx    プします。
121     ja 1f                 // 書き込みキューの中の文字数を確認します。
122

```

```

123      movl proc_list(%ecx),%ebx          # wake up sleeping process // get wait proc list.
124      テストル %ebx,%ebx             # is there? # any process waiting to write ?
125      Je 1f                         // none, then jump forward to label 1.
126      movl $0, (%ebx)                // そうでなければ、プロセスをウェイクする（実行可能な状態に変更する）。
127 1:   movl tail(%ecx),%ebx          // テール位置からALまでのcharを取得します。
128      movb buf(%ecx,%ebx),%al        // THRレジスタ（ポート0x3f8または0x2f8）に出力します。
129      アウトバウンド %al,%dx       // テールを1つ前に移動させます。
130      incl %ebx                   // テールポインタを変調して保存します。
131      andl $size-1,%ebx           // tail == head ?
132      movl %ebx,tail(%ecx)         //もし真ならば、キューが空であることを意味し、ジャンプ
133      cmpl head(%ecx),%ebx        // 次のコードは、書き込みバッファのキューますwrite_q が空である場合を処理します。もし
134      je write_buffer_empty      //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
135      // 次のコードは、書き込みバッファのキューwrite_q が空である場合を処理します。もし
136      //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
137      // 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
138      // 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
139      // 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
140      // 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
141      // 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
142      // トランスマッショントラブルレジスタ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
143      // 力されると
144      // が再び書き込みキューに入ると、serial.cのrs_write()関数は、再び
145      // 送信保持レジスタが空になったときに割り込みが発生するようにして、UARTが
146      // "自動的に書き込みを行なう" のかの文字を読み込むとsleepingprocess
147      .align 2テストル %ebx,%ebx      # is there any?
148      write_buffer_empty            // 待機しているプロセスがないので、ラベル1にジャンプ
149      // フォワードします。
150      movl $0, (%ebx)              // そうでなければ、プロセスをウェイクアップします。
151      incl %edx                  // Read int enable register IER (0x3f9 or 0x2f9).
152      inb %dx,%al                // 暫く待つ。
153      jmp 1f                      // 送信ホールドレジスタを空にしてマスクする int (ビット1)。
154      jmp 1f                      /* 送信インターラプトを無効にする */
155     アウトバウンド %al,%dx
156      レット

```

6. tty_io.c

1. 機能

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。

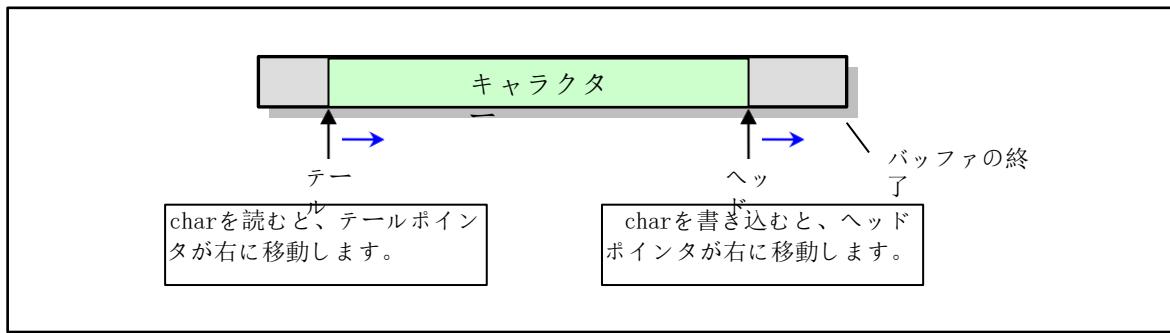


図10-14 ttyキャラクターバッファキューの動作

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ー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ーデバイスファイルであると判断した場合には、キャラクター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。

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。

1. 読み込みキューが空の場合、またはAUXキューが満杯の場合は、最後のステップ (ステップ 10) に進み、そうでない場合は以下の操作を行います。
2. 読み込みキューread_qのテールポインタから1文字を取得し、テールポインタを1文字分前に移動させる。
3. キャリッジリターン(CR)やラインフィード(NL)の文字であれば、端末のターミオス構造にある入力フラグ(ICRNL, INLCR, INOCR)の状態に応じて文字を変換する。たとえば、次のような場合
キャリッジリターン文字を読んでいて、ICRNLフラグが設定されている場合、その文字を改行文字に置き換えます。
4. 大文字→小文字フラグIUCLCが設定されている場合、その文字を対応する小文字に置き換えます。
5. カノニカルモードフラグICANONが設定されている場合、その文字はカノニカルモードで処理されます。
 - a. 削除文字(^U)であれば、補助キューセカンダリの1行分の文字を削除する(キューへッドポインタは、キャリッジリターンやラインフィードが発生するか、キューが空になるまでバックアップされる)。
 - b. 消去文字 (^H) であれば、セカンダリのヘッドポインタで1文字を消去し、ヘッドポインタは1文字分後退します。

- c. ストップ文字(^S)であれば、端末のストップフラグstopped=1を設定します。
- d. 開始文字(^Q)であれば、端末の停止フラグをリセットします。
- 6. 受信キーボード信号フラグISIGが設定されている場合、プロセスのために入力された制御文字に対応する信号を生成します。
- 7. 行末文字 (NLや^Dなど) の場合は、セカンダリキューフォームで行数統計データを1増やします。
- 8. ローカルエコーフラグが設定されている場合は、その文字も書き込みキューフォーム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文字が表示されることになります。
- 9. そのキャラクターを補助キューフォームのセカンダリリーに入れ、上記のステップ1に戻り、リードキューフォームの他のキャラクターをループし続けます。
- 10. 最後に、セカンダリキューフォームにスリープしているプロセスを起こします。

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フォームのデータ構造や、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 コードアノテーション

プログラム 10-5 linux/kernel/chr_drv/tty_io.c

```

1 /*
2 * linux/kernel/tty_io.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * 'tty_io.c' は、コンソールであろうと tty に直交する感覚を与えます。
9 * または rs-channels。また、エコーやクックドモードなども実装されています。
10 */
11 * Kill-line thanks to John T Kohl, and corrected VMIN = VTIME = 0.
12 */
13
// <ctype.h> 文字型ファイルです。文字型変換のためのマクロを定義しています。
// <errno.h> エラー番号のヘッダーファイルです。システムの様々なエラー番号を含みます。
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、および
// 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。LIBRARYが定義されている場合は、システムコール番号と
//      インラインアセンブリ_syscall10()も含まれています。
14 #include <ctype.h>
15 #include <errno.h>
16 #include <signal.h>
17 #include <unistd.h>
18
// アラーム信号の対応するビットマスクを信号ビットマップに与える。
19 #define ALRMASK (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
// 21 #include <linux/sched.h> セグ

メントレジスタ操作用
22 #include <linux/tty.h>
23 #include <asm/segment.h>
24 #include <asm/system.h>
25
    // プロセスグループ関数をキルする（プロセスグループにシグナルを送る）。パラメータ pggrp
    // にはプロセスグループ番号、sigはシグナル、privは優先度を指定します。
    // この関数の主な目的は、指定されたシグナル・シグを各プロセスに
    // 指定されたプロセスグループ pggrp. プロセスへの送信が成功している間は
    // 0を返します。それ以外の場合は、指定されたプロセスグループ番号pggrpに対応するプロセスが見
    // つかなかった場合。
    // エラー番号 -ESRCH が返されます。プロセスグループ番号を持つプロセスが pggrp
    // が見つかっても、シグナルの送信操作に失敗した場合は、送信失敗のエラーコードが返されます。
26 int kill_pg(int pggrp, int sig, int priv); // kernel/exit.c, line 171.
    // 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。
27 int is_orphaned_pg(int pggrp); // kernel/exit.c, line 232. 28
    // termios構造体の3つのモードフラグセットのうちの1つを取得するか、またはフラグが
29 #define L_FLAG(tty,f) ((tty)->_termios.c_lflag & f) // ローカルモードのフ
30 #define I_FLAG(tty,f) ((tty)->_termios.c_iflag & f) // ラグです。
31 #define O_FLAG(tty,f) ((tty)->_termios.c_oflag & f) // 入力モードのフラグ
32
    // termios構造の特別な（ローカルな）モードフラグセットにガチャを取るのフラグ
33 #define L_CANNON(tty) _L_FLAG((tty), _ICANON) // カノニカルモードフラグ。
34 #define L_ISIG(tty) _L_FLAG((tty), _ISIG) // シグナル（INTR、QUITなど）のフラグです
35 #define L_ECHO(tty) _L_FLAG((tty), _ECHO) .
36 #define L_ECHOE(tty) _L_FLAG((tty), _ECHOE) // echo char flag.
37 #define L_ECHOK(tty) _L_FLAG((tty), _ECHOK) // キャノンモードのエコー消去フラグ。
38 #define L_ECHOCTL(tty) _L_FLAG((tty), _ECHOCTL) // キャノンモードのKILL消去ラインフラグ。
39 #define L_ECHOKE(tty) _L_FLAG((tty), _ECHOKE) // エコー制御 char flag.
40 #define L_TOSTOP(tty) _L_FLAG((tty), _TOSTOP) // キャノンモードで消去行とエコーフラグを
41
    // 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ダランド出力にSIGTTOU信号を送る。
42 #define I_UCLC(tty) _I_FLAG((tty), _IUCLC) // 大文字→小文字のフラグを取得します。
43 #define I_NLCR(tty) _I_FLAG((tty), _INLCR) // ラインフィードNLを入力のCRフラグにマ
44 #define I_CRNL(tty) _I_FLAG((tty), _ICRNL) // ッピングする。
45 #define I_NOCR(tty) _I_FLAG((tty), _IGNCR) // NLフラグにキャリッジリターンCR。
46 #define I_RXON(tty) _I_FLAG((tty), _IXON) // CRフラグを無視する。
47
    // 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。
48 #define O_POST(tty) _O_FLAG((tty), _OPOST) // 後処理出力。
49 #define O_NLCR(tty) _O_FLAG((tty), _ONLCR) // NLをCR-NLフラグにマッピングす
        // る。
50 #define O_CRNL(tty) _O_FLAG((tty), _OCRNL) // CRをNLフラグにマッピングする
51 #define O_NLRET(tty) _O_FLAG((tty), _ONLRET) // NLはCR機能のフラグを実行しま
        // す。
52 #define O_LCUC(tty) _O_FLAG((tty), _OLCUC) // 小文字→大文字のフラグ。
53

```

```

// termios構造の制御フラグでボーレートを取得します。CBAUDはボーレートマスク(0000017)です。
54 #define C_SPEED(tty) ((tty)->_termios.c_flag & CBAUD)
// tty端末がハングアップしたかどうかを判断する、つまり送信
// ボーレートはB0(0)です。
55 #define C_HUP(tty)      (C_SPEED((tty)) == B0)
56
57 #ifndef MIN
58 #define MIN(a,b) ((a) < (b) ?(a) : (b))
59#endif
60
// tty端末が使用するバッファキューモードの配列tty_queuesを以下のように定義します。
// そして、ttyターミナルテーブル構造の配列tty_table.QUEUESは、バッファの最大数である
// tty端末で使用されるキューモード。疑似端末は2種類（マスターとスレーブ）に分かれています。
// 各tty端末は、3つのttyバッファキューモードを使用しており、これはバッファリングされた読み取りキューモードである
// キーボードまたはシリアル入力 read_queue、バッファリングされた画面またはシリアル出力の書
61 #define QUEUES (3*(MAX_CONSOLES+NR_SERIALS+2*NR_PTYS)) // 合計54個のキューモード
62 static struct queue *rs_secondary; // フォアグラウンド用のsecondary queue。
63 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モードの開始位置を設定します。
// tty_queues[]配列のtty端末。8つの仮想コンソール端末は、最初の
// tty_queues[]配列の24項目(3 X MAX_CONSOLES)(0--23)です。
// 2つのシリアル端子は次の6項目(3 X NR_SERIALS)を占めています(24--29)。
// 4つの主要な疑似端末は次の12項目(3 X NR_PTYS)(30--41)を占めています。
// 4つのスレーブ疑似端末は、次の12項目(3 X NR_PTYS)(42--53)を占めています。
65 #define con_queues tty_queues
66 #define rs_queues ((3*MAX_CONSOLES) + tty_queues)
67 #define mpty_queues ((3*(MAX_CONSOLES+NR_SERIALS))+ tty_queues)
68#define spty_queues ((3*(MAX_CONSOLES+NR_SERIALS+NR_PTYS))+ のようになります。
// 以下は、さまざまなタイプのttyで使用されるtty構造の開始位置を設定します。
// tty_table[]配列の中にある端末。8つの仮想コンソール端末では、64項目(0~63)を使用できます。
// tty_table[]配列の先頭にある。
// 2つのシリアル端末は、次の2つの項目(64--65)を使用します。
// 4つのメイン擬似端末は、128から始まるアイテムを64アイテム(128--191)まで使用します。
70 #define rs_table (64+tty_table) // 128から始まるアイテムを最大端末の番号まで確保します。
71 #define mpty_table (128+tty_table) // シリアルターミナルのttyテーブルです。
72 #define spty_table (192+tty_table) // メインの疑似端末のttyテーブルです。
73
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
77 /*
78 * これらは、マシンコードハンドラーが使用するテーブルです。
79 * バーチャルコンソールを導入することができます。
80 */
// tty read and write buffer queue structure address table.rs_io.sプログラムで使用され、以下を取得します。
// 読み書き可能なバッファキューモードのアドレス。
81 struct tty_queues *table_chqueues[1], // フォアグラウンド・コンソール・リード/ライト
82     rs_queues + 0, rs_queues + 1, // ・キューモードのアドレス。
83     // シリアルターミナル1のリード/ライトキューモードのアドレスです。
612

```

```

84     _rs_queues + 3, _rs_queues + 4      //シリアルターミナル2のリード/ライトキューの
85     };                                アドレスです。
86
87     //// フォアグラウンドのコンソール機能を変更します。
88     // 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。
89     // Parameters: new_console - 指定された新しいコンソール番号です。
90     void change_console(unsigned int new_console)
91     {...}
92     // 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
93     // は無効です。それ以外の場合は、現在のフォアグラウンドのコンソール番号が設定され、フォアグ
94     // ラウンドの
95     // console read and write queue 構造体の table_list[] のアドレスが更新されます。最後に更新
96     // 現在のタブnew_console-ルの画面で。|| new_console >= NR_CONSOLES)
97     return;
98     fg_console = new_console;
99     table_list[0] = con_queues + 0 + fg_console*3;
100    table_list[1] = con_queues + 1 + fg_console*3;
101    update_screen(); // kernel/chr_drv/console.c, line 936.
102
103 }
104
105    //// 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。
106    // 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。
107    // 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
108    // プロセスへのシグナルがなく、指定されたキュー・バッファが空であれば、プロセスに
109    // 割り込み可能なスリープ状態にして、キューのプロセス待ちポインタがプロセスを指すようにする
110    .
111    static void sleep_if_empty(struct tty_queue * queue)
112    {。    while (!(current->signal & ~current->blocked) && EMPTY(queue))
113        interruptible_sleep_on(&queue->proc_list);
114        sti()です。
115    }
116
117    //// 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。
118    // Parameters: queue - 指定したキューへのポインタ。
119    // プロセスがキューバッファに文字を書き込む前に、キューの状態を決定します。
120    // キューのバッファが一杯になっていない場合に返します。そうでない場合は、プロセスに処理する
121    // シグナルがない場合。
122    // で、キューバッファに残っている空き領域が<128の場合、プロセスを
123    // 割り込み可能なスリープ状態で、キューのプロセス待ちポインタがプロセスを指します。
124    static void sleep_if_full(struct tty_queue * queue)
125    {。    if (! FULL(queue))
126        を返すことができます。
127        cli()です。
128        while (!(current->signal & ~current->blocked) && LEFT(queue)<128)
129            interruptible_sleep_on(&queue->proc_list);
130        sti()です。
131    }
132
133    //// キーが押されるのを待ちます。
134    // 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
135    // 状態に入ります。
136    void wait_for_keypress(void)
137    {

```

```

117     sleep_if_empty(tty_table[fg_console].secondary);
118 }
119
120 // // カノニカルモードの文字列をコピーして変換します。
121 // 指定された端末の読み取りキューにある文字をコピーして、正規の
122 // モード（クックドモード）のキャラクターと、様々なフラグに応じてAUXキューに格納される
123 // 末端のtermios構造に設定されています。
124 void copy_to_cooked(struct tty_struct * tty)
125 {
126     signed char c; 123
127
128     // 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
129     // 。
130     // 3つのキューのポインタがすべてNULLの場合、ttyの初期化関数に問題がある。
131     // も (! (tty->read_q || tty->write_q || tty->secondary)) {
132     // し。
133     //     printf("copy_to_cooked: missing queues\n");
134     // を返すことができます。
135
136     // そうでなければ、ttyのリードキューバッファから取得した各文字を適切に処理します
137     // 端末のtermios構造の入力フラグとローカルフラグに応じて、それを
138     // auxiliary queueに入る secondary.
139     // 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
140     // 文字の//であれば、ループ本体を終了します。そうでなければ、プログラムは1文字を
141     // 読み込みキューの最後にあるポインターを、テールポインターを1つ前に移動させます。
142     // その後、文字コードに応じた処理が行われます。また、_POSIX_VDISABLE(＼＼＼＼＼)の場合は
143     // が定義されている場合、文字コードの値が_POSIX_VDISABLEの値と同じであれば、関数
144     // 対応する特殊制御文字の//が禁止されています。
145     // 一方 (1) {
146     //     if (_EMPTY(tty->read_q))
147     //         ブレークします。
148     //     if (_FULL(tty->secondary))
149     //         ブレークします。
150     //     GETCH(tty->read_q, c) です。 // charを取得して、尻尾が前に移動する。
151
152     // 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
153     // 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
154     // 合は
155     // 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
156     // する
157     // 旗の他の//が改行する場合の文字は小文字に変換されからキャリッジ
158     // リターンフラグNLCRがセットされている場合はCR(13)に変換されます。大文字から小文字への入力
159     // の場合 if (c==13) {
160     //     if (_CRNL(tty))
161     //         c=10;
162     //     else if (_NOCR(tty))
163     //         を続けています。
164     } else if (c==10 && _NLCR(tty))
165     //         c=13;
166     if (_UCLC(tty))
167     //         c=_tolower(c)となります。
168
169     // ローカルモードフラグセットのカノニカルモードフラグCANONが設定されている場合、読み込まれ
170     // る文字は
171     // 以下のように処理されます。
172     // 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
173     // 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。
174     // 一行分の文字がttyの補助キューが空でなく、最後の文字が

```

```

補助キューでフェッチされた//は改行文字ではない NL(10)であり
// ファイル終端文字(^D)またはファイル終端文字が_POSIX_VDISABLEになっていない。
// そして、その他の文字をループして削除処理を行います。この間
// 消去処理では、ローカルエコーフラグECHOが設定されている場合、消去制御文字の
// ERASE (^H)を書き込みキューに入れる必要があります。その文字が制御文字の場合
// (値<32)で、2バイト表現(例: ^V)の場合は、追加のイレーズ
// 文字ERASEを配置する必要があります。その後、tty write関数が呼び出されて、すべての
// の文字を書き込みキューに入れて、端末装置に送信します。最後に、ttyの補助キューの先頭の
// ポイン では1バイトでバックします。
タ      す
。
143     if (L_CANON(tty)) {
144        もし ((KILL_CHAR(tty) != _POSIX_VDISABLE) &&
145             (c==KILL_CHAR(tty))) {
146             /* 入力ラインの削除処理 */
147             while(!(_EMPTY(tty->secondary) ||
148                 (c==LAST(tty->secondary))==10 ||
149                 ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
150                     (c==EOF_CHAR(tty)))))) {
151                 if (L_ECHO(tty)) {
152                     if (c<32) // 制御文字 ?
153                         PUTCH(127, tty->write_q) です。
154                     PUTCH(127, tty->write_q) です。
155                     tty-> write(tty) です。
156                 }
157                 DEC(tty->secondary->head)。
158             }
159             // 文字が削除制御文字ERASE(^H)であり、かつ削除文字が
160             continue; // 行内の他の文字を処理します。
161             // _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
162             // 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
163             // 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
164             // 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
165             // もし
166             // その文字が制御文字(値<32)であり、2バイトで表現された
167             // 表現(例: ^V)では、追加の消去文字ERASEを配置する必要があります。ttyの書き込み
168             // 関数が呼び出され、書き込みキューに入っているすべての文字が端末機器に出力されま
す。
169             // 最後に、tty補助キュー④(ポインタをもどす)後退させて、頭へ戻ります。
170             // 他のキャラチャラしたもの(ERASE_CHAR(tty)) {
171                 if (_EMPTY(tty->secondary) ||
172                     (c==LAST(tty->secondary))==10 ||
173                     ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
174                         (c==EOF_CHAR(tty)))) {
175                     を続けています。
176                     if (L_ECHO(tty)) {
177                         if (c<32)
178                             PUTCH(127, tty-> write_q) です。
179                         PUTCH(127, tty->write_q) です。
180                         tty-> write(tty) です。
181                     }
182                     DEC(tty->secondary->head)。
183                     を続けています。

```

```

176         }
177     }
// IXONフラグが設定されている場合は、端子の停止/開始出力制御文字が有効になります。もし
// このフラグが設定されていない場合、ストップとスタートの文字は、通常の文字として読み込まれ
// 処理を行います。このコードでは、読み込まれた文字が停止文字STOP(`S)の場合は、その文字を
// tty停止フラグ、ttyに出力を一時停止させ、特別な制御文字を破棄する（配置されていない
// を補助キューに入れる）、他の文字の処理を続けます。もし、その文字が
// 開始文字START(`Q)を入力すると、ttyの停止フラグがリセットされ、ttyの出力が回復して
// 制御文字は破棄され、他の文字は引き続き処理されます。
// コンソールでは、tty->write()は、console.cのcon_write()関数と同じです。
// 新しい文字を画面に表示するのを直ちに中止する (chr_drv/console.c, line 586) due
// ストップド=1の発見に//。疑似端末の場合、書き込み操作が中断される
// (chr_drv/pty.c, line 24)のように、ターミナルストップフラグが設定されているからです。シリ
// アルターミナルの場合
// 送信中の端末の停止フラグに応じて、送信を中断する必要がある
178 // 端末ですが、このバージョンは実装されていません
179     。 if (L_ISIXON(TTYP_CHAR(tty) != POSIX_VDISABLE)
&&...
180             (c==STOP_CHAR(tty)) {
181                 tty->stopped=1;
182                 tty-> write(tty) です。
183                 を続けています。
184             }
185             If ((START_CHAR(tty) != POSIX_VDISABLE)
&&...
186             (c==START_CHAR(tty)) {
187                 tty->stopped=0;
188                 tty-> write(tty) です。
189                 を続けています。
190         } // 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
191 // には、信号を生成することができ、制御文字INTR、QUIT、SUSP、DSUSPを受信したときに
// プロセスに対応する信号を生成する必要があります。その文字がキーボードの
// 割り込み(`C)、キーボード割り込み信号SIGINTをプロセス内の全てのプロセスに
// 現在の処理の//グループで、次の文字の処理を続けます。もし、そのような
// 文字が終了文字(``の場合、キーボードの終了信号SIGQUITが全プロセスに送信されます。
// を現在のプロセスのプロセスグループに追加し、次の文字の処理を続けます。
// 文字がサスペンド文字(`Z)の場合、ストップ信号SIGTSTPを現在の
// 処理を行います。同様に、_POSIX_VDISABLE(0)が定義されている場合は、文字コードの値が
// 文字処理中に _POSIX_VDISABLE の値に // 変化した場合は、対応する
// スペシ 制御文字の使用は禁止されています。
ヤル
192         if (L_ISIG(tty)) {
193            もし ((INTR_CHAR(tty) != POSIX_VDISABLE) &&
194             。 (c==INTR_CHAR(tty)) {
195                 kill_pg(tty->pgrp, SIGINT, 1)。
196                 を続けています。
197             }
198            もし ((QUIT_CHAR(tty) != POSIX_VDISABLE)
&&...
199             (c==QUIT_CHAR(tty)) {
200                 kill_pg(tty->pgrp, SIGQUIT, 1)。
201                 を続けています。
202             }
203            もし ((SUSPEND_CHAR(tty) != POSIX_VDISABLE)
&&...616 (c==SUSPEND_CHAR(tty)) {
204

```

```

205                     if (! is_orphaned_pgrp(tty-
206                         >pgrp)) kill_pg(tty->pgrp, SIGTSTP, 1)
207                         .
208                     } // 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。
209                     // 一行分の文字が処理されたことを示す、行番号「secondary.data」現在
210                     // 補助キューに格納されている // を1つずつ増やしていきます。文字の行を取ると
211                     // 関数 tty_read() で補助キューから // 受け取った場合、行番号は 1 つずつ減算されます。
212                     // 315行目をご覧ください。
213                     if (c==10 || (EOF_CHAR(tty) != POSIX_VDISABLE
214                         &&...))
215                         c==EOF_CHAR(tty)))
216                     // 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
217                     // 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ぬし
218                     // 文字が制御文字(値<32)でエコー制御文字フラグECHOCTLの場合
219                     // が設定されると、文字'~'と文字c+64がttyの書き込みキューに入ります(~C, ~H,
220                     // などが表示されます)、そうでない場合は、文字が直接、tty write
221                     // のキューに入ります。最後に、ttyの書き込み操作関数が呼び出されます。
222                     if (L_ECHO(tty)) { .
223                         if (c==10) {
224                             PUTCH(10, tty->write_q) です。
225                             PUTCH(13, tty->write_q) です。
226                         } else if (c<32) {
227                             if (L_ECHOCTL(tty)) { .
228                             PUTCH('~', tty->write_q) です。
229                             PUTCH(c+64, tty-> write_q) です。
230                         }
231                     } else
232                         PUTCH(c, tty->write_q) です。
233                     tty-> write(tty) です。
234                 }
235             // 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。
236             // 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ウェイクアップします。
237             PUTCH(c, tty->secondary) です
238             .
239         }
240     }
241     wake_up(&tty->secondary->proc_list)
242     .
243 /*
244 * プロセスにSIGTTINまたはSIGTTOUを送信する必要があるときに呼び出される
245 * グループ
246 *
247 * システムコールの再起動を要求するのは、システムコールの再起動があった場合
248 * のみです。
249 * デフォルトのシグナルハンドラが使用されています。その理由は、もし
250 * ジョブがSIGTTINやSIGTTOUをキャッチしている場合、シグナルハンドラーは次の
251 * ことを望まないかもしれません。
252 * システムコールをやみくもに再起動することはできません。
253 * 端末のpgrpを現在のpgrpに戻す(おそらく、コントロールの
254 * ttyはログアウト時に解放されています)、無限ループに陥らないためには
255 * システムコールを再起動しながら、常にSIGTTINを生成するようにしています。
256 * またはSIGTTOU。デフォルトのシグナルハンドラは、プロセスを停止させる
257 * 無限ループの問題を回避することができます。おそらく、ジョブコントロール
258 * の
259 * 認識力のある親は、子のプロセスを継続する前に物事を修正します。

```

```

245 */
     //// 端末を使用するプロセスグループのすべてのプロセスに信号を送ります。
     // この関数は、バックグラウンドのすべてのプロセスにSIGTTINまたはSIGTTOUシグナルを送信するために使用されます。
     // グループ内のプロセスが制御端末にアクセスすると、//プロセスグループかどうかにかかわらず
     // バックグラウンドプロセスグループのプロセスがこの2つの信号をブロックしたり、無視したりする
     // 現在のプロセスは、直ちに読み取りと書き込みの操作を終了して戻ります。
246 int tty_signal(int sig, struct tty_struct *tty)
247 {
     // 孤児のプロセスグループのプロセスを停止したくない（行の説明を参照
     // ファイルkernel/exit.cの//232）を使用します。そのため、現在のプロセスグループがオーファンプロセスグ群(ア)の場合
248 // エラーが返されません。return -EIO; // 例外の場合は指定されたpgroupを停止しない(ア)
249 // 現在のプロセスグループ(curren->pgrp, sig, 1); // シグナルを送る sig.
250
     // このシグナルが現在のプロセスによってブロック（マスク）されているか、無視されている場合は
     // 、エラーが返されます。
     // そうでなければ、現在のプロセスがシグナルsigに新しいハンドラーを設定している場合は、次のように返します。
251 // 割り込まれる可能性のある情報を取り戻す)それ以外の場合は
     // システムコールの再起動後に実行されます。
252     ((int) current-> sigaction[sig-1]. sa_handler == 1))
253         return -EIO; /* 私たちのシグナルは無視されます */。
254     else if (current->           g-1]. sa_handler)
255         sigaction[si return - /* We _will_ be 割り込み :-*/ EINTR;
256
     その他
257         return -ERESTARTSYS; /* 私たちは 割り込み :-*/
258             /* (ただし、再 続けることができます
     起動 。) */
259
     //// tty read機能。
260 // 端末の補助キューから指定された数の文字を読み込んで配置する
     // ユーザが指定したバッファに格納されます。パラメータ: channel - サブデバイスの番号, buf -
     // ユーザーが指定した
     // バッファポインタ; nr - 読み込むバイト数。読み込んだバイト数を返します。
261 int tty_read(unsigned channel, char * buf, int nr)
262 {
     struct tty_struct * tty;
     struct tty_struct * other_tty = NULL;
263     char c, * b=buf;
264     int 最小、時間。
265
     // まず、関数のパラメータの有効性を判断し、tty構造体のポインタを取る
     // 端末の //。tty端末の3つのキューpointerがともにNULLの場合、EIOエラー
     // のメッセージが返されます。tty端末が擬似端末の場合、別のtty構造体other_tty
     // 疑似端末に対応したもの取得する必要がある if (channel >
266         255)
267
     return -EIO;
268     tty = TTY_TABLE(channel) です。
269     if (!(tty->write_q || tty->read_q || tty->secondary))
270         return -EIO;
271
     // 現在のプロセスが、ここで処理されているtty端末を使用しているが、プロセスグループが
     // 端末の//番号が現在のプロセスグループ番号と異なっていることを示しています。
     // 現在のプロセスがバックグラウンドプロセスグループのプロセスである場合、つまり
     // フォアグラウンドではありません。そのため、現在のプロセスグループのすべてのプロセスを停止
     // する必要があります。そのため

```

```

// 現在のプロセスグループにSIGTTINシグナルを送信してリターンする必要があり、待機している
// でフォアグラウンドのプロセスグループになってから、読み取り操作を行います。
// さらに、現在の端末が疑似端末の場合、対応する他の疑似
// 端末はother_ttyです。ここでのttyが主要な擬似端末である場合、other_ttyは
// 対応するスレーブの疑似端末、またはその逆です。
273     if ((current->tty == channel) && (tty->pggrp != current->pggrp))
274         return(tty_signal(SIGTTIN, tty));
275     if (channel & 0x80)
276         other_tty = tty_table + (channel ^ 0x40)です。

// 続いて、コードは読み取り文字操作のタイムアウトのタイミング値と最小数を設定します。
VTIMEに対応する制御文字配列の値に基づいて読み込まれる文字の//。
// とVMINです。非正規モードでは、この2つはタイムアウトのタイミング値です。VMINは
// 読み取り操作を満足させるために必要な最小文字数。
// VTIMEは、1/10秒カウントのタイミング値です。
277     time = 10L*tty->                                // リードタイムアウトのタイミング値
278     termios.c_cc[VTIME]; minimum = tty->          を設定します。
// tty端末がrawモード[VMIN]との場合は、最小文字数を設定し読むべき文字数の最小値です。
// 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
// 以下のコードでは、プロセスの読み取りタイムアウト値を
// を指定されたタイムアウト値時間に設定し、残りの文字が読み込まれるのを待ちます。
// 328行目をご覧ください。
// この時点で読み込み文字数の最小値が設定されていない場合は
// 読み込む文字数nrで、タイムアウトのタイミング値が設定されている場合は、プロセスが読み込ん
// だ
// タイムアウトのタイミング値 タイムアウトは、現在のシステム時刻+指定したタイムアウト時間
// に設定されます。
// となり、時間がリセットされます。また、上記で設定された最小読み取り文字数が
// がプロセスで読み取るべき文字数nrよりも大きい場合、minimum=nrとする。その
// は、カノニカルモードでの読み出し操作に対して、制約を受けずに
279 // VTIMEとVMINの対応する制御文字の値を制御し、それらが唯一機能する
280 非正規モード(Rawモード)の操作で // if
281     (L_CANON(tty)) {
282         最小値=nr;
283         current->timeout = 0xffffffff;
284         time = 0;
285     else if (minimum)
286         最小値=timeout = 0xffffffff。
287         if (time)
288     current->timeout = time + jiffies;
289         time = 0となります
290     }
291     if (minimum>nr)
292         最小値=nr;                                // 必要な文字数まで読み込みます。

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

```

```

// 擬似端末で、現在の擬似端末の補助キューに文字を書き込む
// バッファにコピーします。つまり、他の端末は、書き込みキュー・バッファの文字を
// 現在の疑似端末のリード・キュー・バッファを変換して
// カノニカルモード関数で変換された後の補助キュー while (nr>0) {
293     if (other_tty)
294         other_tty-> write(other_tty);
295

// ttyの補助キューが空の場合、または、canonical modeフラグが設定されていて、ttyのリードキューが
// が満杯ではなく、補助キューの文字列数が0であれば、プロセスの
// 読み取り文字のタイムアウト値(0)が設定されていない、または現在のプロセスが信号を受信している。
// であれば、まずループ本体を終了します。そうでない場合は、端末がスレーブ疑似端末であり
// 対応するマスター疑似端末がハングアップしたならば、ループ体も終了します。
// 上記2つの状況のいずれでもない場合は、現在のプロセスを
// 割り込み可能なスリープ状態にし、復帰後に処理を継続する。カーネルは
// カノニカルモードでは、ユーザーにデータを文字列単位で提供するために、少なくとも1つの
296 // このモードでの補助キューの文字のライン、つまりセカンダリ.データの最小値である
297 // は1です。    if (EMPTY(tty->secondary) || (L_CANON(tty) &&
298           !FULL(tty->read_q) && !tty->secondary->data)) {
299     if (! current->timeout ||)
300         (current->signal & ~current->blocked)) {
301     sti()です。
302     ブレークします。
303 }
304     if (IS_A_PTY_SLAVE(channel) && C_HUP(other_tty))
305         ブレークします。
306     interruptible_sleep_on(&tty->secondary->proc_list)
307     .
308     sti()です。
309 }
310 // 以下は、文字のsti()です操作の正式な実行を開始します。数字の
// nr=0になるか、補助バッファが終了するまで、連続してデクリメントされます。
// キューは空です。
// このループでは、補助キューの文字cが最初に取り込まれ、キューのテールポインタの
// テールは1文字分だけ右にシフトします。取得した文字がファイル
// ターミネーター(^\D)または改行文字 NL(10)は、以下に含まれる文字列の行数です。
// 補助キューが1つデクリメントされます。その文字がファイル終了文字 (^\D) の場合は
// で、canonical mode フラグが設定されていれば、ループは中断され、そうでなければ、ファイル
終了文字
// に遭遇していないか、または生（非正規）モードである。このモードでは、ユーザーは
// 文字ストリームを読み取り対象とし、制御文字を認識しません（例えば
// をファイルターミネーターとして使用します）。その後、コードはその文字をユーザーデータバッ
ファに直接入れます。
// bufを読み込んで、読むべき文字数を1つデクリメントします。このとき、もし
読むべき文字の//がすでに0であれば、ループは中断されます。また、端末が
// 空きバッファモードで、読み込んだ文字が改行を連続して構成しまくると、ループも終了します。
// さらに、nr文字が取られていない限り、AUXキューが
311     do {
312         GETCH(tty->secondary, c)です。
313        もし ((EOF_CHAR(tty) != POSIX_VDISABLE &&
314             c==EOF_CHAR(tty)) || c==10)

```

```

315                     tty->secondary->data--。
316            もし ((EOF_CHAR(tty) != POSIX_VDISABLE &&
317             c==EOF_CHAR(tty)) && L_CANON(tty))
318                ブレークします。
319             else {
320                 put_fs_byte(c, b++) です。
321                 if (!--nr)
322                    ブレークします。
323             }
324             if (c==10 && L_CANON(tty))
325                ブレークします。
326         } while (nr>0 && !EMPTY(tty->secondary))。

// この時点で、ttyターミナルがcanonicalモードであれば、改行や
// end of file 文字に遭遇しました。非正規モードであれば、読み込んだnr
// の文字が入っていたり、補助キューが取り出されていました。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
327 現在の          wake_up(&tty->read_q->proc_list)。
328 // 時間のジフティmstime読み取りタイムアウト時間。もちろん、端末がカノニカルモードの場合や
329 // nr文字が読み込まれたのでこの大盛なループを直接終了することができます。
330         if (L_CANON(tty) || b->buf >= minimum)
331             ブレークします。
332         }

// この時点でtty文字のループ操作が終了するので、プロセスの読み取りタイムアウトをリセットす
// る
// タイミング値のタイムアウト。現在のプロセスがシグナルを受信し、かつ、どのキャラクターも
// がまだ読まれていない場合は、システムコール番号を再起動して返され、そうでない場合は
333 // 読んだ文字数(b->buf)を表示します。_
334     if ((current->signal & ~current->blocked) && !(b->buf))
335         return -ERESTARTSYS;
336     return (b->buf)となります。
337 }

338
//// ttyの書き込み機能。
// ユーザーバッファの文字をttyの書き込みキューbufに入れます。
// 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 - サブデバイスの
// 数。
// 書き込まれたバイト数書き込まれたバイト数を返します。
339 int tty_write(unsigned channel, char * buf, int nr)
340 {。    static cr_flag=0;
341     struct tty_struct * tty;
342     char c, *b=buf;
343
344     // まずパラメータの有効性を判断し、tty構造のポインタを
// 端末になります。tty端末の3つのキューbufがともにNULLの場合、EIOエラーメッセージ
// を返します。
345     if (channel > 255)
346         return -EIO;
347     tty = TTY_TABLE(channel) です。

```

```

348     if (!(tty->write_q || tty->read_q || tty->secondary))
349         return -EIO;
// 端末のローカルモードフラグセットでTOSTOPが設定されている場合、バックグラウンドプロセスの
// 出力はSIGTTOUという信号を送る必要があります。このとき、現在のプロセスがttyを使用している
// 場合は
// ここで処理されている端末ですが、端末のプロセスグループ番号が異なります
// 現在のプロセスグループ番号から // を選択すると、現在のプロセスが
// バックグラウンドのプロセスグループ、つまり、そのプロセスはフォアグラウンドではないとい
// うことです。そのため、私たちは
// 現在のプロセスグループのすべてのプロセスを停止します。そのため、SIGTTOUを送る必要があり
350 ます。
351 // 現在のプロセスグループにシグナルを送り、フォアグラウンドプロセスになるのを待って戻る
352 // グループ化してから、書き込み操作を行う。 if
    (L_TOSTOP(tty) &&
     (current->buf->channel) && (tty->pgp != current->pgp))
// ここで、ユーザーバッファから文字を形づけて、それを
// 書き込みキューリターン。書き込むバイト数が0より大きいときは、次のように
// ループ動作を行います。ループ中、この時点でttyの書き込みキューが満杯になっていたら
// カレントプロセスが割込み可能なスリープ状態になる。もし現在のプロセスがシグナル
// 処理するために、ループ本体を終
353     了する while (nr>0) {
354         sleep_if_full(tty->write_q);
355         if (current->signal & ~current->blocked)
356             break;
// 書き込まれる文字数nrがまだ0より大きく、tty write
// キューバッファが満杯にならない場合、以下の操作がサイクリックに実行されます。最初に1を取
// る
// バイトをユーザーバッファから取得します。端末の実行出力処理フラグがOPOSTの場合
357 // 出力モードのフラグセットが設定されていると、その文字に対する後処理動作を行う while
358     (nr>0 && !FULL(tty->write_q)) {
359         c=get_fs_byte(b);
            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。
360 // それ以外の場合は、文字が改行文字 'c' & & O_CRLF の場合にルフィードがリターン関数
361 // フラグ ONLRETがセットされていると、その文字はキャリッジリターン文字 'r' に置き換えられます
362 .
            else if (c=='\r' && O_NLRET(tty))
                c='r' です。
363
// 文字が改行文字 '¥n' であり、キャリッジリターンフラグ cr_flag が
// セットされていますが、カーニューラインフラグ OLCRへの改行がセットされ、cr_flagフラグがセ
// ットされ
// CRが書き込みキューに入ります。その後、次の文字の処理を続けます。
// 小文字→大文字フラグ OLCUC が設定されている場合、その文字は大文字に変換されます。
364 // 文字を表示します。
365     if (c=='\r' && !cr_flag && O_NLCR(tty)) {
366         cr_flag = 1;
367         PUTCH(13, tty->write_q) です。
368         を続けています。
369     }
370     if (O_LCUC(tty))
371         c=toupper(c) となります。
}
// 次に、ユーザデータバッファのポインタbを1バイト進め、書き込むバイト数を指定します。
// は1バイト減ります。cr_flagフラグがリセットされ、そのバイトがtty write

```

```

// queue.
372         b++; nr--;
373         cr_flag = 0;
374         PUTCH(c, tty->write_q)
375     }    です。
// 必要な文字がすべて書き込まれるか、書き込みキューがいっぱいになると、プログラムは終了します。
// のループに入れります。この時点で、対応する tty write 関数が呼び出され、表示されます。
// 書き込みキューの文字をコンソール画面に表示したり、シリアルポートから送信したりすることができます。
// 現在処理されている tty がコンソール端末の場合、tty->write() は con_write(); if
// ttyがシリアル端末の場合、tty->write()はrs_write()関数です。まだバイトがある場合
376 // を書き込むためには、w書き込み戻す。中の文字が取り込まれるのを待つ必要があります。そこで
377 // コール      if (nr>0)
378 // ここでスケジューラーを使つて(他のタスクを先に進めてください。
379 }
380     return (b-buf); // 最終的なリターン          は、書き込まれたバイト数です
381 }
382 /*
383 */
384 * Jeh, 時々、386が本当に好きになります。
385 * このルーチンは割込みから呼び出されます。
386 * このようにしておけば、何の問題もありません。
387 * 割り込みでもスリープすることができるようになるといいで
388 * もちろん、誰かが私の間違いを証明してくれれば、私は
389 * hate intel for all time :-(。私たちには
390 * 注意して、割り込みを復活させてください。
391 * しかし、これを呼び出す前に、チップス。
392 *
393 * 通常の状況では、ここで寝ることはないと思います
394 * いずれにしても、寝ているタスクがあるかもしれないのに、
395 /* それ付良いこと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。
396 /* 全くの無事: tty - 指定されたttyターミナル番号。
397 // 指定されたtty端末のキューの文字をカノニカル（調理済み）にコピーまたは変換する
// モードの文字を、AUXキューに格納します。この関数は、シリアル
// ポート読み取り文字割り込み(rs_io.s, 110)とキーボード割り込み(kerboard.S, 76)です。
398 void do_tty_interrupt(int tty)
399 {
400     copy_to_cooked(TTY_TABLE(tty));
401 }
402 // 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。
403 void chr_dev_init(void)
404 {
405 }
406 // tty端末の初期化機能。
407 // 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。
408 void tty_init(void)
409 {
410     int i;

```

409

```
// まず、全端末のバッファキュー構造を初期化し、初期値を設定する。のために
//シリアルターミナルのリード/ライトバッファのキューは、そのデータフィールドをシリアルポートのベースアドレスに設定します。
```

```
//シリアルポート1は0x3f8、シリアルポート2は0x2f8です。次に、初期にtty構造を設定します。
```

```
すべての端末の //。特殊文字の配列c_cc[]の初期値を定義するのは
```

410 // include/linux/tty.hファイルです。

411 for (i=0 ; i < QUEUES ; i++)412 rs_queues[0].quei(構造体 = t_struct) tty(0x3f8, 0, 0, 0, "",)」をなしますを読む413 rs_queues[1] = (構造体 tty queue) {0x3f8, 0, 0, 0, ""}。 // 書き込みキューです。414 rs_queues[3].quei(構造体 tty queue) {0x2f8, 0, 0, 0, ""}。415 rs_queues[4].quei(構造体 tty queue) {0x2f8, 0, 0, 0, ""}。416 rs_queues[4].tty_table[1] = (struct tty_struct) {... {0, 0, 0, 0, 0, init_c_cc} となります。417 0, 0, 0, ヌル, ヌル, ヌル, ヌル

418 };

419 // 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が設定されています。読み込みバッファ、書き込み
コンソール端末のtty構造にある // バッファ、補助バッファキュー構造
```

```
// は435行目で初期化され、それぞれ対応する構造体アイテムを指しています。
```

```
con_init().....// ttyのキュー構造配列tty_table[]の中にある。
```

422 for (i = 0 ; i < NR_CONSOLES ; i++) {...423 con_table[i] = (struct tty_struct) {...424 ICRNL, /* 受信したCRをNLに変更する */ (注)425 OPOST|ONLCR, /* 発信するNLをCRNLに変更 */。

426 0, // 制御モードフラグの設定

427 IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, // ローカルフラグの設定

428 0, /* コンソールtermio */ // 0 -- TTY.

429 INIT_C_CC} で // 制御文字配列 c_cc[] す。

430 0, /* イニシヤル pggrp */

431 0, /* イニシヤル セッション */

432 0, /* イニシヤル 止まっている */

433 con_writeで // コンソール書き込み機能。
す。434 con_queues+0+i*3, con_queues+1+i*3, con_queues+2+i*3

435 };

436 }

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

```
//シリアルターミナルのtty構造の中にある、リード/ライトと補助バッファのキュー構造です。
```

```
// tty バッファキュー構造配列の対応する構造項目を指している
```

```
// tty_table[].61--73行目の説明を参照してください。
```

438 for (i = 0 ; i < NR SERIALS ; i++) {439 rs_table[i] = (struct tty_struct)

440 {...

441 0, /* 翻訳なし */ // 入力モードフラグの設定

```

441          0, /* 翻訳なし */      */ // 出力モードフラグの設定
442          B2400 | CS8,           // コントロールモードフラグ
443          0,                   set. 2400bps, 8bits.
444          0,                   // ローカルモードフラグが設定されています。
445          INIT_C_CC} です。   // 回線手続き、0 -- TTY。
446          0,                   // 制御文字の配列です。
447          0,                   // 初期のプロセスグループ。
448          0,                   // セッションを開始します。
449          rs_queues+0+i*3, rs_queues+1+i*3, rs_queues+2+i*3 // 初期停止フラグ
450          rs_write です。     // シリアルポートの書き込み機能。
451      };
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
// それらを初期化する必要があります。以下のループでは、まず、tty構造体を初期化します。
// 各マスター疑似端末の対応する tty 構造を初期化します。
453 // スレーブ疑似端子。
454     for (i = 0 ; i< NR_PTYS ; i++) {....
455         mpty_table[i]/* 翻訳なし struct */ // 入力モードフラグの設
456         0, /* 翻訳なし */      定
457         B9600 | CS8 です。    // 制御モードフラグのセットを使用していま
458         0,                   定
459         0,                   // ローカルモード を設定しました。
460         0,                   // フラグ
461         0,                   // ラインの手順。 0--TTY。
462         INIT_C_CC} です。   // コントロール 文字配列です。
463         0,                   // イニシャル プロセスグループ。
464         0,                   // イニシャル セッションを行います。
465         0,                   // イニシャル が停止したフラグ。
466         mpty_write です。    // マスターの疑似書き込み機能。
467         mpty_queues+0+i*3, mpty_queues+1+i*3, mpty_queues+2+i*3
468     };
469     spty_table[i] = (struct tty_struct) {....
470         {0, /* 翻訳なし *//
471         0, /* 翻訳なし *//
472         B9600 | CS8,
473         IXON | ISIG | ICANON, // ローカルモードフラグが設定されている。
474         0,
475         INIT_C_CC} です。
476         0,
477         0,
478         spty_write, // スレーブの疑似書き込み機能。
479         spty_queues+0+i*3, spty_queues+1+i*3, spty_queues+2+i*3
480     };
481 // 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
482 // 初期化され、仮想コンソールの数 NR_CONSOLES と疑似端末の数
483 // システムに含まれるNR_PTYSが表示される。
484     rs_init()
485     printk("%d virtual consoles\n", NR_CONSOLES).
486     printk("%d pty's\n", NR_PTYS).
487 }

```

3. インフォメーション

1. 制御文字 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、この時点では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。

7. tty_ioctl.c

1. 機能

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ーフェースを実現する。

一般ユーザプログラムでは、システムコール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、以下を取得する端末IO制御コマンドTIOCGPGRPでは

ターミナルプロセスのグループ番号（フォアグラウンドプロセスのグループ番号）を指定するために、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()を実装しています。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。

2. コードアナリシス

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> の初期名と、記述子のファイルデ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
// <termios.h> の名が異なるため、主に端末の入出力機能を定義しています。
// 雜同期通信用の制御するヘッダーファイルです。一般的に使用されているいくつかの製品の
// Linux/SELinuxが含まれています。Linux/SELinuxのヘッダーファイルでは、タスク構造体task_struct、データ
// カーネルの使用する機能

```

```

// <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/io.h> Ioのヘッダーファイルです。の形で、ioポートを操作する関数を定義します。
// マクロの組み込みアセンブリです。
// <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
// <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。
// ディスクリプタ/割り込みゲートなどが定義されています。 7 #include <errno.h>
8 #include <termios.h>
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h> (日本語)
12 #include <linux/tty.h>
13
14 #include <asm/io.h>
15 #include <asm/segment.h>
16 #include <asm/system.h>
17
// 次の行では、2つの外部関数プロトタイプとボーレート係数の配列を与えています。
シリアルポートのための //。最初の関数は、セッション番号を取得するために使用されます。
// プロセスグループは、プロセスグループ番号pgrp (kernel/exit.cで定義されています) に従って
所属します。
// 161行目)。2つ目の関数tty_signal()は、プロセス内のすべてのプロセスにシグナルを送るために
使用されます。
chr_drv/tty_io.c の 246 行目で定義されている、指定された tty ターミナルを使用する // グループです。
// ボーレート係数の配列（または除数の配列）は、ボーレートとの対応を示す。
// レートとボーレート係数を示しています。例えば、ボーレートが2400bpsの場合、対応する
// 係数は48(0x30)、9600bpsの係数は12(0x1c)です。後の命令をご覧ください。
// プログラムリスト。
18 extern int session_of_pgrp(int pgrp);
19 extern int tty_signal(int sig, struct tty_struct *tty);
20
21     64, 48, 24, 12, 6, 3
22 static unsigned short quotient[] = {。
23
24
25
26
//// 送信ボーレートの変更。
// Parameters: tty - 端末に対応する tty データ構造体。割り算が
// ラッチフラグDLABがセットされると、ボーレートファクターの下位バイトと上位バイトがそれぞれ
// シリアルポート1のポート0x3f8および0x3f9を介してUARTに書き込まれた、DLABの
// ビットは書き込み後にリセットされます。シリアルポート2の場合、0x2f8と0x2f9の2つのポートが
// あります。
27 static void change_speed(struct tty_struct * tty)
28 {
29     unsigned short port, quot; 30
// この関数はまず、パラメータ tty で指定された端末がシリアル端末であるかどうかをチェックし
// ます。
// し、そうでない場合は終了します。シリアルターミナルのtty構造の場合、readのデータフィールドは
// キューは、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
//が端末のtermios構造の制御モードフラグセットから設定されていることを確認し、取得する。
// ボーレートファクター配列quotient[]から対応するボーレートファクターの値を指定します。
CBAUD          627
//は、コントロールモードフラグが設定されている場合のボーレートビットマスクです。

```

```

31     if (!(port = tty->read_q->data))
32         return;
33     quot = quotient[tty-> termios.c_flag & CBAUD] となります。
// その後、ボーレート係数がUARTチップのボーレート係数ラッチに書き込まれます。
// シリアルポートに対応しています。書き込みの前に、まず、除算器のラッチアクセスを
// ラインコントロールレジスタLCRのビットDLAB（ビット7）を書き込み、16ビットボーレートファクタの
// ローバイトとハイバイトをそれぞれポート0x3f8, 0x3f9に送る。のDLABフラグビットをリセットします。
34 // LCRでoutb_p(0x80, port+3);      /* DLABの設定 */
35     outb_p(quot & 0xff, port);    /* 割り算のLS */ /* 割り算のLS
36     outb_p(quot >> 8, port+1);   /* 除算器のMS */
37     outb_p(0x03, port+3) となり /* DLABのリセット
38     ます。                         */
39 }
40 } sti() です。
41
42 static void flush(struct tty_queue* queue)
43 {
44     cli() です。
45     queue-> head = queue-
46     >tail;
47 } sti() です。
48 static void wait_until_sent(struct tty_struct* tty)
49 {
50     /* 何もない - 実装されていない
51 } */
52
53 static void send_break(struct tty_struct* tty)
54 {
55     /* 何もない - 実装されていない
56 } */
57
58 /**
59 * 端末のtermios構造情報を取得します。
// パラメータ: tty - 端末のtty構造体ポインタを指定する; termios - ユーザーバッファ
// このようにして、Termiosの構造を保持していた。
60 static int get_termios(struct tty_struct* tty, struct termios* termios)
61 {
62     int i;
63     // まず、ユーザバッファポインタが示すメモリ領域が十分であることを確認する。もし
64     // ない場合は、メモリを確保します。次に、指定された端末のtermios構造情報をコピーする
65     // をユーザバッファに送信します。最後に0を返します。
66     verify_area(termios, sizeof (*termios)); // kernel/fork.c, line 24. for (i=0
67     ; i< (sizeof (*termios) ; i++)
68         put_fs_byte( ((char *)&tty-> termios) [i] , i+(char *)termios
69     ); return 0;
70 }
71
72 /**
73 * 端末のtermios構造に情報を設定します。

```

```

// パラメータ: tty - 端末のtty構造体ポインタを指定; termios - ユーザーデータ領域
// termios構造のポインタ。
69 static int set_termios(struct tty_struct* tty, struct termios* termios),
70                     int channel)
71 {
72     int i, rtsig;
73
74     /* 端末の状態を設定しようとしたとき、自分がフォアグラウンドに
75      いなからしたら、SIGTTOUを送信する。シグナルがブロックされた
76      り、無視されたりしたら、操作を実行する。POSIX 7.2) */
77
78     // 現在のプロセスのtty端末のプロセスグループ番号が
79     // 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
80     // 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
81     // 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
82     // 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するプロセスが一時的に
83     // 実行を停止して、先にtermios構造を修正できるようにしています。しかし、もしsend
84     // 関数tty_signal()がERESTARTSYSまたはEINTRの値を返した場合、操作は
85     // 後日、再度実行されます。
86     if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
87         rtsig = tty_signal(SIGTTOU, tty); // chr_drv/tty_io.c, 246行目. if
88         (rtsig == -ERESTARTSYS || rtsig == -EINTR)
89         return rtsig;
90     }
91
92     // 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。
93     // 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
94     //シリアルポート送信ボーレート、シリアルUARTチップ内のボーレートを再度修正
95     // コントロールモードフラグc_cflagのボーレート情報に基づく
96     // 構造体を作成し、最後に0を返します。
97     for (i=0 ; i< (sizeof (*termios)) ; i++)
98         ((char *)&tty-> termios)[i]=get_fs_byte(i+(char
99         *)termios); change_speed(tty);
100    0を返す。
101
102    //// Termio構造の情報を取得します。
103    // パラメータ: tty - 端末のtty構造体ポインタを指定する; termio - ユーザーバッファ
104    // termioの構造情報を保持する//。
105    static int get_termio(struct tty_struct* tty, struct termio* termio)
106    {
107        int i;
108        struct termio tmp_termio;
109
110        // 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
111        // 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。
112        // 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
113        // す。
114        // 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
115        // 後者は短い。したがって、一時的なtermio構造へのコピーの目的である
116        // はデータ型変換のためのものです。最後に、一時的なtermio構造の情報は
117        // verify_area(termio, sizeof (*termio));
118        tmp_termio.c_iflag = tty-> termios.c_iflag;
119        tmp_termio.c_oflag = tty-> termios.c_oflag;
120        tmp_termio.c_flag = tty-> termios.c_flag.

```

```

97     tmp_termio.c_lflag = tty->
98     termios.c_lflag; tmp_termio.c_line = tty->
99     termios.c_line; for(i=0 ; i < NCC ; i++)
100        tmp_termio.c_cc[i] = tty->
101        termios.c_cc[i]; for (i=0 ; i< (sizeof (*termio))
102        ; i++)
103            put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio);
104    }
105
106 /*
107 * これは、386がローバイトファーストの場合にのみ動作します。
108 */
109
110 ///////////////////////////////////////////////////
111 // 端末のtermio構造情報を設定します。
112 // Parameters: tty - 端末のtty構造のポインタを指定する; termio - termioの
113 // 構造体をユーザーデータ領域に配置します。この関数は、ユーザーの情報をコピーするために使用
114 // されます。
115 // 端末のtermios構造体にtermioをバッファリングして0を返す。
116 static int set_termio(struct channel * tty, struct termio * termio,
117 {
118     int i, retsig;
119     struct termio tmp_termio;
120
121     // 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
122     // 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
123     // がフォアグラウンドでない場合は、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
124     // ます。
125     // 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
126     // 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
127     // する必要があります。
128     // を使って実行を一時的に停止し、termiosの構造を先に修正できるようにしました。ただし。
129     // 送信関数 tty_signal() が ERESTARTSYS または EINTR の値を返す場合は、操作
130     // は後で再度実行されます。
131     if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
132         retsig = tty_signal(SIGTTOU, tty);
133         if (retsig == -ERESTARTSYS || retsig == -EINTR)
134             return retsig;
135     }
136
137     // 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。
138     // その構造体を作成し、その情報をttyのtermios構造体にコピーします。の目的は
139     // これは、モードフラグが設定されているタイプを変換するためのもので、つまり、短整数型の
140     // termioを長整数型にしたもので。しかし、2つのフィールドのc_lineとc_cc[]は
141     // の構造は同じです。
142     for (i=0 ; i< (sizeof (*termio)) ; i++)
143         ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
144         *(unsigned short *)&tty-> termios.c_iflag = tmp_termio.c_iflag;
145         *(unsigned short *)&tty-> termios.c_oflag = tmp_termio.c_oflag;
146         *(unsigned short *)&tty-> termios.c_flag = tmp_termio.c_flag;
147         *(unsigned short *)&tty-> termios.c_lflag =
148             tmp_termio.c_lflag; tty-> termios.c_line = tmp_termio.c_line;
149         for(i=0 ; i < NCC ; i++)
150             tty-> termios.c_cc[i] = tmp_termio.c_cc[i] となります。
151
152     // 最後に、ユーザーが端末のシリアルポートの速度を変更した可能性があるため、ボーレートの
153     // のボーレート情報に応じて変更され、シリアルUARTチップの
154     // termios構造体のモードフラグc_flagを設定し、0を返す。
155     change_speed(tty);

```

```

130          0を返す。
131  }
132
133 ///////////////////////////////////////////////////////////////////
134 // tty端末装置の入出力制御機能。
135 // 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。
136 // この関数は、まず、対応する端末の tty 構造体を以下のようにして見つけます。
137 // のデバイス番号を入力し、制御コマンドcmdに従って処理します。
138 int tty_ioctl(int dev, int cmd, int arg)
139 {
140     struct tty_struct * tty;
141     int pgrp;
142
143     // 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
144     // 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
145     // 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
146     // 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
147     // 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
148     // マシンを停止します。メジャーデバイスの番号が5ではなく4の場合は、サブデバイスの
149     // デバイス番号からの // 番号です。サブデバイスの番号は、0(コンソール端末)、1(シリアル
150     // if (MAJOR(dev) == 5) {
151         dev=current->
152             tty; if (dev<0)
153                 panic ("tty_ioctl: dev<0");
154     } else
155         dev=MINOR(dev) となります。
156
157     // 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
158     // 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。
159     // 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
160     // 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
161     // 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
162     // 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
163     fg_console
164     // 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
165     // 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
166     // 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
167     // 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
168     // の tty
169     // 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
170     // の
171
172     // tty構造はtty_table[dev]です。dev = 1 の場合、対応する端末の tty 構造は
173     // はtty_table[0]です。tty_io.cプログラムの70~73行目をご覧ください。
174     // TCGETS: 対応する端末のtermios構造情報を取得します(devの時点でのconsole);
175     // パラメータargはユーザバッファのポインタです。
176     // TCSETSF: ターミオスを設定する前に、出力キュー内のすべてのデータを待って
177     // を処理し、入力キューをフラッシュします。その後、termiosを設定する操作を行います。
178     // TCSETSW: termiosを設定する前に、出力キューのすべてのデータを待って
179     // を処理します。この形式は、パラメータを変更すると出力に影響する場合に必要です。
180     // TCSETS: 対応する端末のtermios構造情報を設定します。この時点では
181     // パラメータargは、termios構造体を保持するユーザバッファポインタです。
182     // TCGETA: 対応する端末のtermio構造の情報を取得します。このとき
183     // パラメータ arg がユーザバッファポインタであることを示す。
184     // TCSETAF: termioを設定する前に、出力キューのすべてのデータが処理されるのを待つ必要がある
185     // となり、入力キューがフラッシュされます。その後、設定端末のtermio操作を行います。

```

```

// TCSETAW: ターミオスを設定する前に、出力キューのすべてのデータを待って
// を処理します。この形式は、パラメータを変更すると出力に影響するような場合に必要です。
// TCSETA: 対応する端末のtermio構造情報を設定します。この時点では
// パラメータargは、termio構造体を保持するユーザバッファポインタです。
// TCSBRK: パラメータargの値が0の場合、出力キューの処理を待つ
// して、ブレークを送ります。
145     スイッチ (cmd) {
146         case TCGETS:
147             return get_termios(tty, (struct termios *) arg);
148         ケース TCSETSF:
149             flush(tty->read_q) です /* フォールスルー
150                 * */
151             ケース TCSETSW:
152                 wait_until_sent(tty) で /* フォールスルー
153                     return set_termios(tty, (struct termios *) arg, dev);
154                 ケース TCSETSです。
155                 return get_termio(tty, (struct termio *) arg);
156             ケース TCSETAF:
157                 flush(tty->read_q); /* フォールスルー*/
158             ケース TCSETAW:
159                 wait_until_sent(tty); /* フォールスルー*/
160             ケース TCSETA:
161                 return set_termio(tty, (struct termio *) arg, dev);
162             ケース TCSBRK:
163                 if (!arg) {
164                     wait_until_sent(tty);
165                     send_break(tty);
166                 }
167             0を返す。
168 // フロー制御を開始 / 停止する。パラメータargがTCOFF (端子制御出力OFF) の場合は
169 // の出力が保留され、TCONであれば、保留されていた出力が復元されます。同じタイミングで
170 // 出力を中断または再開する際には、書き込みキューにある文字を
171 // ユーザーインタラクションのレスポンスを高速化する。argがTCIOFF(Terminal Control Input ON)
172 // の場合は
173 // 入力は中断され、TCIONであれば保留中の入力が再開される。 case
174     TCXONC:
175         switch (arg) {
176             case TCOFF: >stopped = 1; // 端末の出力を停止します。
177                 tty-> write(tty); // キューの出力を書き込みます。
178                 return 0;
179             case TCOON:
180                 tty->stopped = 0; // 端末の出力を元に戻す。
181                 tty->
182                     write(tty);
183 // 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するので
184 // STOP文字を端末の書き込みキューに入れて、入力を中断する
185 // 端末がその文字を受信したとき。パラメータがTCIONの場合は、START
186 // 端末に送信を再開させるために // 文字を送信します。
187 // STOP_CHAR(tty)は、((tty)->termios.c_cc[VSTOP])と定義されており、これを使って
188 // 端末のtermios構造体の制御文字配列の対応する値。もし
189 // 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
190 // _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
191 // 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

```

```

// にし を端末の書き込みキューに入れます。も同様である。
// ていま
す。
178     case TCIOFF:
179         if (STOP_CHAR(tty))
180             PUTCH(STOP_CHAR(tty), tty->write_q);
181         0を返す。
182     case TCION:
183         if (START_CHAR(tty))
184             PUTCH(START_CHAR(tty), tty->write_q)です。
185         0を返す。
186 // 書き込まれたがまだ送信されていないデータをフラッシュします。もし、そのような
187 // データをまだ読み取っていない場合は、それを返す。もし、そのようなデータをまだ読み取っていない場合は、それを返す。
188 // パラメータargが0の場合、入力キューがフラッシュされ、1の場合、出力キューがフラッシュされ
189 // ます。 ケー TCFLSH。
190 // 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。
191     if (arg==0)
192         flush(tty-> read_q)です。
193     else if (arg==1)
194         flush(tty-> write_q)です。
195     else if (arg==2) {
196         flush(tty-> read_q)です。
197         flush(tty-> write_q)です。
198     } else
199         0を返す。 return -EINVAL;
200 // TIOCEXCL: 端末のシリアルラインの専用モードを設定します
201 // TIOCNXCL: 端末のシリアルライン専用モードをリセットする。
202 // TIOCSCTTY: tty を制御端末として設定します。(TIOCNOTTY - 制御端末なし)。
203 // 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
204 // ープを読み取る)。
205 // 数)を確認します。最初にユーザバッファの長さを確認し、次に端末のttyのpgrpフィールドをコ
206 // ピーする
207 // をユーザバッファに移す。この時点では、パラメータargはユーザバッファのポインタで
208 // す。 case TIOCEXCL:
209     return -EINVAL; /* not implemented */
210 case TIOCNXCL:
211 case TIOCGPGRP: -EINVAL; /* not implemented */
212 case TIOCSCTTY:
213     return -EINVAL; /* 制御用語のNIを設定 */ arg);
214     return 0;

// 端末のプロセスグループ番号pgrpを設定する(つまり、フォアグラウンドのプロセスグループ番号
// を設定する)。
// パラメータargは、ユーザバッファ内のプロセスグループ番号pgrpへのポインタになりました。こ
// のようにして
// このコマンドを実行するには、プロセスに制御端末があることが前提となります。
// 現在のプロセスが制御端末を持っていないか、devがその制御端末でない場合。
// または制御端末が処理中の端末devになりましたが、セッション番号の
// プロセスがターミナルデバイスのセッション番号と異なる場合は、ターミナレスの
// エラーメッセージが返されます。
// そして、ユーザ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、パーティション
// エラーメッセージが返されます。それ以外の場合は、端末のプロセスグループ番号を設定します
// 。

```

```

//をpgrpに変更します。この時点で、pgrpはフォアグラウンドのプロセスグループになります。
209 ループになります case TIOCSPGRP: // 関数tcsetpgrp()を実装します。
210     if ((current->tty < 0) ||
211         (current->tty != dev) ||
212         (tty->session != current->session))
213         return -ENOTTY;
214     pgrp=get_fs_long((unsigned long *) arg); // ユーザバッファからの
215     取得 if (pgrp < 0)
216         return -EINVAL;
217     if (session_of_pgrp(pgrp) != current->session)
218         return -EPERM;
219     tty->pgrp = pgrp;
220     return 0;

// TIOCOUTQ: 出力キューに残っている文字数を返します。最初の検証
// ユーザーのバッファの長さに合わせて、キューの中の文字数をユーザーにコピーします。このとき
// パラメータ arg がユーザバッファポインタであることを示す。
// TIOCINQ: 入力補助キューでまだ読み込まれていない文字の数を返す。この関数は
// ユーザバッファの長さが最初に検証され、AUXの文字数が
// キューがユーザーにコピーされます。この時点でのパラメータargは、ユーザバッファのポインタ
221 です。    ケース TIOCOUTQです。
222         verify_area((void *) arg, 4);
223         put_fs_long(CHARS(tty->write_q), (unsigned long *) arg);
224         0を返す。
225     ケース TIOCINQ:
226         verify_area((void *) arg, 4);
227         put_fs_long(CHARS(tty->secondary),
228                     (unsigned long *) arg)を使用しています。
229         0を返す。

// TIOCSTI: 端末の入力操作をシミュレートします。このコマンドは、文字へのポインタを
// をパラメータとし、その文字が端末で入力されたものと仮定します。ユーザーには
// スーパーユーザー権限、または制御端末の読み取り権限を持っていること。
// TIOCGWINSZ: ターミナルデバイスのウィンドウサイズを読み取る (termios.hのwinsize構造体を参
照)。
230 // 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)
231     case TIOCSTI:
232         return -EINVAL; /* not implemented */
233     case TIOCGWINSZ:
234         return -EINVAL; /* not implemented */
235     case TIOCSWINSZ:
236         return -EINVAL; /* not implemented */

// TIOCMBIT: 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ピットフラ
グを返します。
// (termios.hの185~196行目を参照)。
// TIOCMSET: 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。
// TIOCMGET: 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。
236 // TIOCMSETです。MORENO TIOCMGET: タスピンの状態を設定します。ビットがセットされた場合、対応す
237 るステータスが          return -EINVAL; /* で インプリメ */
238           は          ント
239           な          く
240           ケース TIOCMBIS。
241           return -EINVAL; /* で インプリメ */
242           は          ント
243           な          く
244           ケース TIOCMGET。
245           return -EINVAL; /* で インプリメ */
246           は          ント
247           な          く

```

```

// tiocgsoftcar: ソフトウェアキャリア検出フラグを読み取ります (1 - オン、0 - オフ)。
244 // tiocssoftcar: base TIOCGSOFTCAR: ア検出フラグを設定します (1 - オン、0 - オフ)。
245                                     return -EINVAL; /* not implemented */
246     case TIOCSSOFTCAR:
247         return -EINVAL; /* not implemented */
248     のデフォルトです。
249         return -EINVAL;
250     }
251 }
252

```

3. インフォメーション

1. ポーレートとポーレートファクター

ポーレートは、 $\text{ポーレート} = 1.8432\text{MHz} / (16 \times \text{ポーレート係数})$ で計算されます。
共通ポーレートとポーレートファクターの対応関係を表10-9に示す。

表10-9 ポーレートとポーレート係数の表

ポーレート	ポーレート・ファクター		ポーレート	ポーレート・ファクター	
	MSB,LSB	値		MSB,LSB	値
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	0x00,0x0	192			

10.8 まとめ

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を詳しく説明し、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トするための実装コードを指定します。

11 Math Coprocessor (数学)

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

名前	サ	最終更新時刻 (GMT)	デス
 Makefile	3377 バイト	1991-12-31 12:26:48	コ
 add.c	1999 バイト	1992-01-01 16:42:02	
 compare.c	904 バイト	1992-01-01 17:15:34	
 convert.c	4348 バイト	1992-01-01 19:07:43	
 div.c	2099 バイト	1992-01-01 01:41:43	
 ea.c	1807 バイト	1991-12-31 11:57:05	
 error.c	234 バイト	1991-12-28 12:42:09	
 get_put.c	5145 バイト	1992-01-01 01:38:13	
 math_emulate.c	11540 バイト	1992-01-07 21:12:05	
 mul.c	1517 バイト	1992-01-01 01:42:33	

11.1 機能説明

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、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はもちろん、Linux 0.9xのカーネル開発の初期には、数学コプロセッサチップ80387（またはその互換チップ）が高価だったため、一般のPCでは常に贅沢品でした。そのため、以下のような場合を除いては

が大量に科学的な計算をする場合や、特に必要とされる場所では、80387チップは一般的なPCには搭載されない。現在のインテル・プロセッサーには数学コプロセッサ機能が組み込まれているが、現在のOSではコプロセッサのエミュレータ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おり、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、アセンブリやディスアセンブル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カーネルに搭載されている数学コプロセッサエミュレータmath_emulate.cは、80387の動作を完全にエミュレートします。

チップがコプロセッサ命令を実行します。このプログラムは、コプロセッサ命令を処理する前に、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内部のすべてのスタック・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従っ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 浮動小数点データタイプ

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。

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

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。

10進法	2の補数	表11-1 整数の3つの表現方法	符号の大きさ
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（ダブルワード）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、符号の表現に特化したビット（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す。また

浮動小数点数の有効数字（仮数）部分は表現方法で、符号ビットは浮動小数点数全体の符号を表します。

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

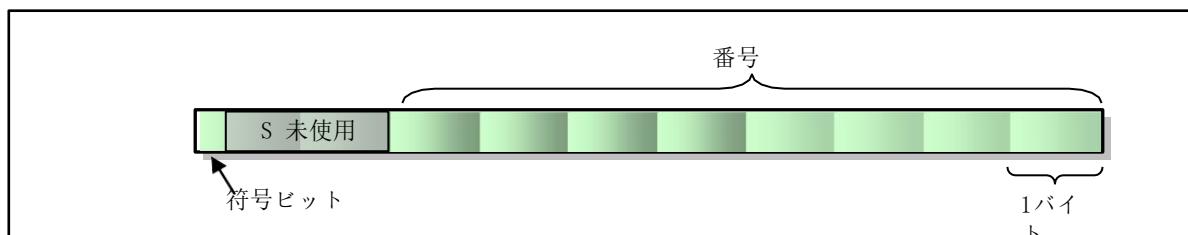


図11-1 エミュレータがサポートする一時的な整数フォーマット

2.2.BCDデータタイプ

BCD（Binary Coded Decimal）コード値は、2進数でコード化された10進数の値です。圧縮BCDコードでは、各バイトが2桁の10進数を表すことができ、4ビットごとに0～9の桁を表します。例えば、10進数59を圧縮BCDコードで表現すると、0x01011001となります。圧縮されていないBCDの場合

符号は、各バイトが下位4ビットのみで1桁の10進数を表現します。

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

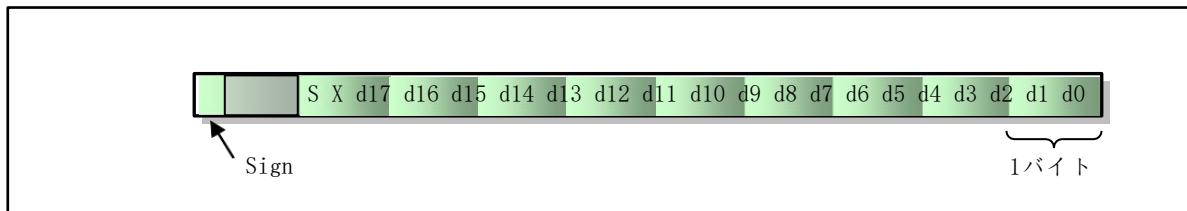


図11-2 80387がサポートするBCDコードデータタイプ

3. 浮動小数点データタイプ

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。



図11-3 浮動小数点の一般的なフォーマット

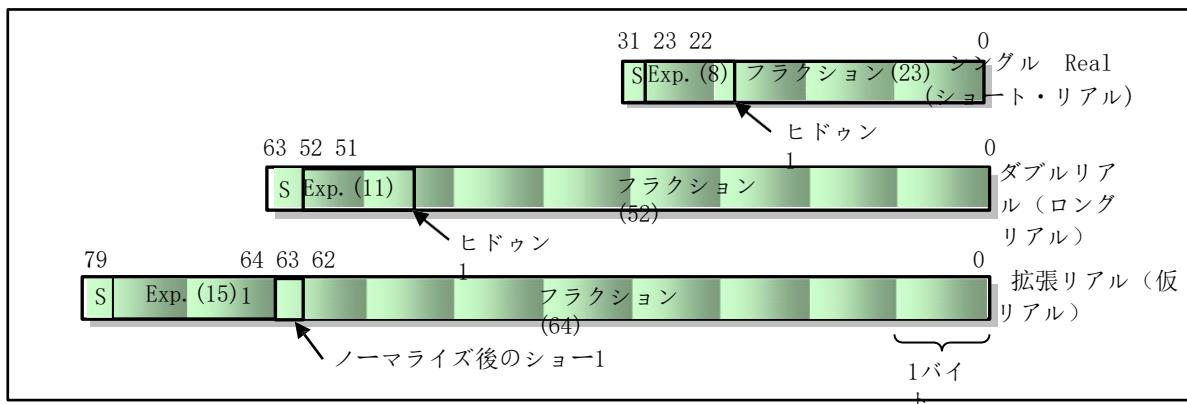


図11-4 80387コプロセッサの実数フォーマット

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數値を調整します。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。

フォーマットの指數フィールドには、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は指數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）の偏った基數は、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、短い実数の指數値である0b10000000は、実際には 2^1 (0b0111111 + 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。

```
62 typedef struct {
```

```
63     長いa, b。          // 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）。
64     short exponent
65 } 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。

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

サイン	バイアスされた指 数	シグニフィカント		タイプ
0/1	11...11	1	11...11	NaNs - QNaNs 静かなNaN
0/1	11...11	1	...	
0/1	11...11	1	10...00	不定形
0/1	11...11	1	01...11	Signalling NaNs - SNaNs
0/1	11...11	1	...	
0/1	11...11	1	00...01	インフィニット
0/1	11...11	1	00...00	
0/1	11...10	1	11...11	規範
0/1	...	1	...	
0/1	00...01	1	00...00	
0/1	00...00	1	11...11	疑似デノルマル
0/1	00...00	1	...	
0/1	00...00	1	00...00	
0/1	00...00	0	11...11	デノルマル
0/1	00...00	0	...	
0/1	00...00	0	00...01	
0/1	00...00	0	00...00	ゼロ

ゼロとは、指数の値と符号の値がともに0で、残りの指数の値が0のものは予約されている、つまり指数の値が0のものは通常の実数値を表すことができない値です。無限とは、指数の値がすべて1で、符号の値がすべて0、残りの値がすべて

指数値の0x11...11も予約済みです。

デノルマル数は、非常に小さな値を表すために使用される特別なクラスの値です。プログレッシブアンダーフローやプログレッシブ精度の低下を示すことがあります。この値は通常、正規化された数値（有効数字の最上位ビットがビット1になるまで左シフト）として表現することが求められますが、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。

擬似デノルマルクラス値は、シニフィカントの最上位ビットが1、デノルマルクラス値のビットが0という値です。擬似デノルマル数は稀なもので、正規化されたクラス数で表すことができますが、そうではありません。上で説明したように、特殊なバイアスのかかった指数0x00...00には正規化された数値の指数0x00...01と同じ値をとることで、擬似的に非正規化されたクラス番号を正規化されたクラス値として表現することができる。

もう一つの特殊なケースはNaN (Not a Number) です。NaNには、シグナルを出すもの（シグナリングNaN）と、シグナルを出さないもの（クワイエットNaN）の2種類があります。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。各データタイプには、不定値を表す数値があります。整数の場合、最大の負の数を用いてその不定値を表します。

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 数学コプロセッサの機能と構造

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き、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。

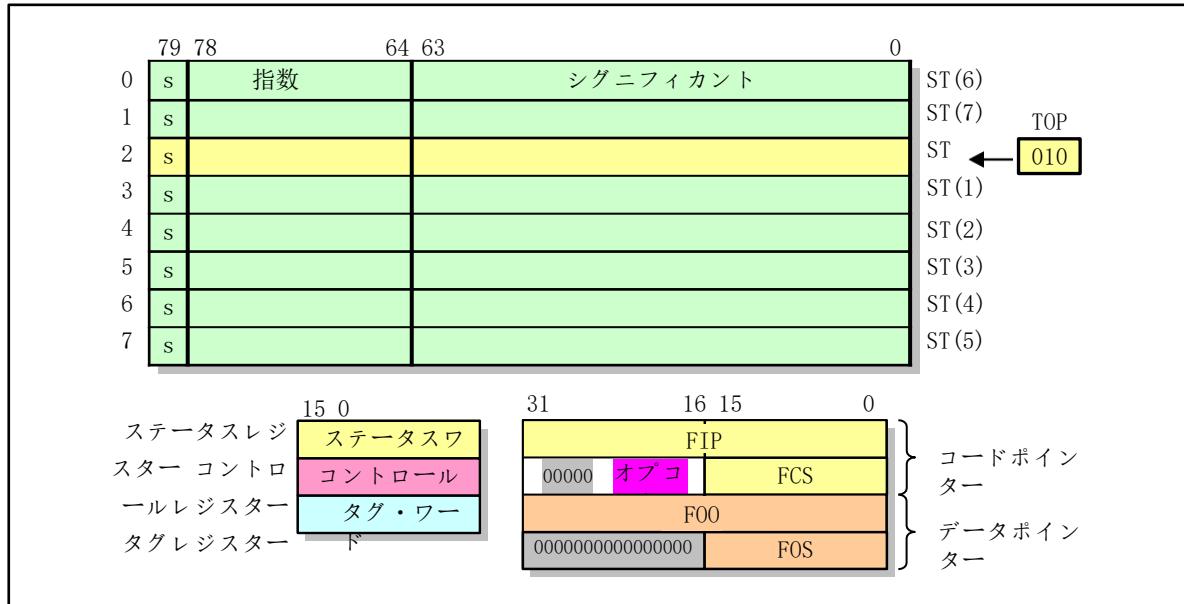


図11-5 80387のレジスタ

1. スタック式浮動小数点アキュムレータ

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（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（ $i = 1 \dots 7$ ）。どの80ビット物理レジスタが現在のスタックトップSTであるかについては、特定の演算処理によって指定される。図11-6に示すステータスワードレジスタのTOPと名付けられた3ビットフィールドには、現在のトップSTに対応する80ビット物理レジスタの絶対位置が格納されている。プッシュまたはロード操作により、TOPフィールドの値が1つデクリメントされ、新しいSTに新しい値が格納されます。プッシュ操作の後、元のSTはST(1)となり、元のST(7)は現在のSTとなります。つまり、すべてのアキュムレータの名前が、元のST(i)からST((i+1)&0x7)に変わったのです。ポップまたはストア操作は、現在のTOPレジスタSTから値を読み取ってメモリに格納し、TOPフィールドの値を1だけ増加させる。したがって、ポップ操作の後、元のST（すなわちST(0)）はST(7)となり、元のST(1)は新しいSTとなる。つまり、すべてのアキュムレータの名前は、元のST(i)からST((i-1)&0x7)へと変更されるのである。

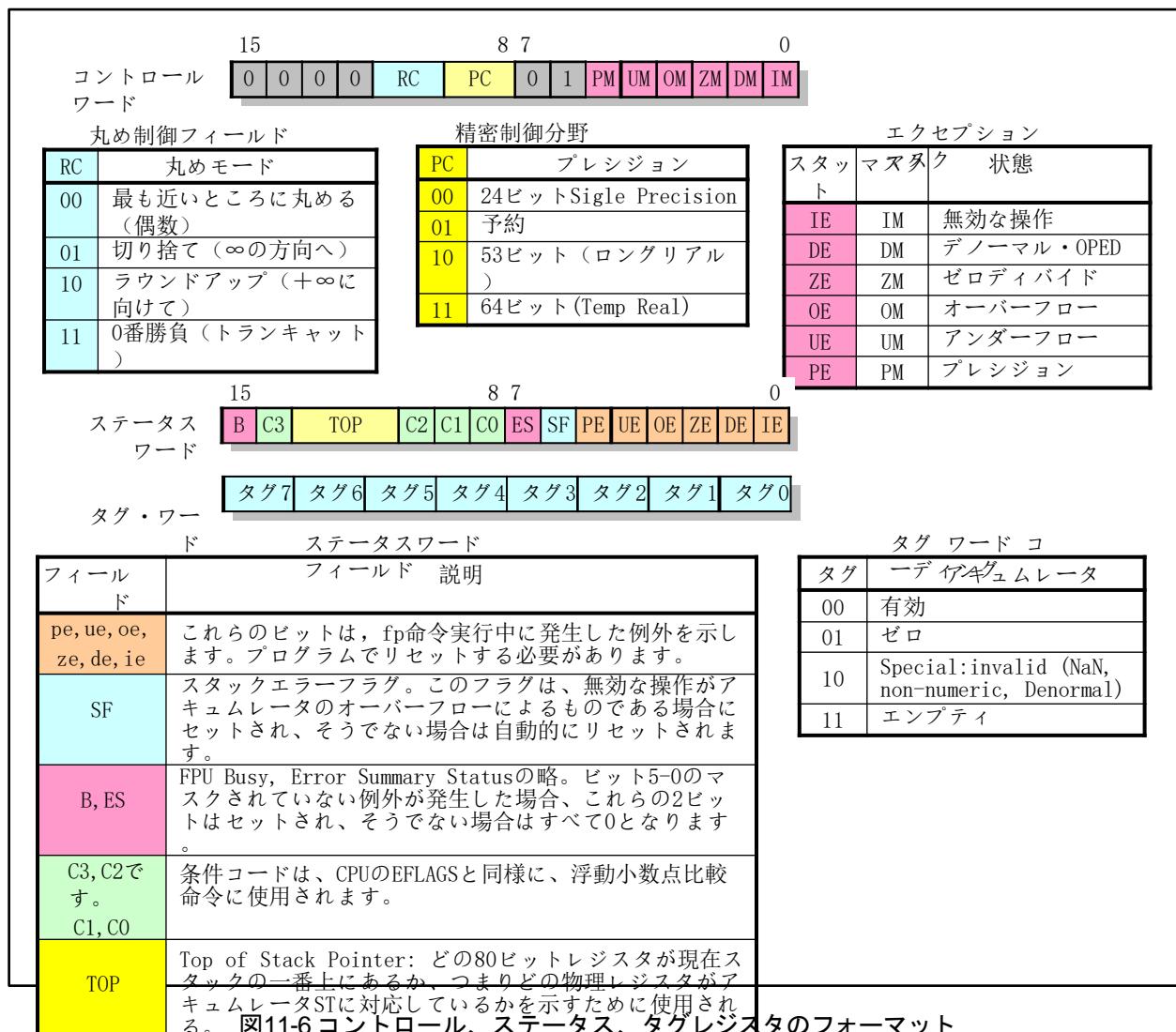
STは、すべての浮動小数点命令の暗黙のオペランドとして使用されるため、アキュムレータのように動作します。他にオペランドがある場合は、第2オペランドは残りのアキュムレータST(i)のいずれか、またはメモリオペランドとなります。スタック内の各アキュムレータは、実数の一時的な実数形式で格納された80ビットの空間を提供し、最上位ビットは符号ビット、ビット78--64は15ビットの指数フィールド、ビット63--0は64ビットの有効数フィールドとなります。

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

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

2. ステータス&コントロールレジスタ

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。



A. コントロールワード

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

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、それを個別にマスクすることができます。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。

B. ステータスワード

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

C. タグ・ワード

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。push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。

3. エラー・ポインタ・レジスタ

エラー・ポインタ・レジスタは、図11-6に示すように、最後に実行された命令と使用されたデータへの80387個のポインタを含む4つの32ビット80387レジスタです。最初の2つのレジスタFIP(FPU Instruction Pointer)とFCS(FPU Code Selector)は、最後に実行された命令の2つのオペコードへのポインターです(プレフィックスの

コード)を使用しています。FCSはセグメントセレクタとオペコード、FIPはセグメント内のオフセットです。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. 浮動小数点命令フォーマット

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、80386の通常の命令を使って、対応するエミュレーション演算を以下のように行います。

各命令の構造について演算コプロセッサ80387には70以上の命令があり、表11-3のように5つのカテゴリに分けられています。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ます。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

1	11011	OPA		1	MOD		1	OPB		R/M		SIB	DISP	
2	11011	MF		OPA	MOD		OPB		R/M		SIB	DISP		
3	11011	d	P	OPA	1	1	OPB		ST(i)					
4	11011	0	0	1	1	1	1	OP						
5	11011	0	1	1	1	1	1	OP						
	15 -- 11	10	9	8	7	6	5	4	3	2	1	0		

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）。

- a) OP (Operation opcode) は、命令のオペコードです。いくつかの命令では、2つの部分に分かれています。 OPAとOPBです。
- b) MF (Memory Format) は、メモリフォーマットです。 00～32ビットの実数、01～32ビットの整数、10～64ビットの実数、11～64ビットの整数。
- c) P(Pop)は、操作後にポップアップ処理を行うかどうかを示します。 0 - 必要なし、1 - 操作後にスタックをポップアップします。
- d) d (デスティネーション) は、演算結果を保存するアキュムレータを示す。 0 - ST(0); 1 - ST(i).
- e) MOD_ModeとR/M(Register/Memory)は、操作モードフィールドとオペランドの位置フィールドです。
- f) SIB(Scale Index Base)とDISP(Displacement)は、MODとR/Mのフィールド命令を持つオプションのフォローアップフィールドです。

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。

- a) FI FIADD、FIELDなど、整数データを操作するための命令はすべてFIで始まる。
- b) FB FBLD、FBSTなど、BCD型データを操作する命令はすべてFBで始まります。
- c) FxxP FSTPやFADDPなど、ポップ操作を行う命令はすべてPで終了します。
- d) FxxPP FCOMPP、FUCOMPPなど、2つのポップ操作を行うすべての命令は、文字PPで終了します。
- e) e) FNxx FNで始まる命令を除いて、すべての命令は実行前にマスクされていない演算例外を検出します。 FNで始まる命令は、FNINIT、FNSAVEなどの算術異常を検出しません。

2. math-emulation.c

1. 機能

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ーフェース関数math_emulate()であり、このクラスにはこのような関数が1つしかありません。 第2のタイプは、浮動小数点命令エミュレーション処理のメイン関数do_emu()であり、このクラスにはこのような関数が1つしかありません。 さらに、すべての関数は、他のCプログラムの関数を含め、シミュレーション操作の補助関数です。

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

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

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

```

11 struct info {
12     long math_ret;           // 呼び出し元(int7)のリターンアドレス。
13     long orig_eip;          // オリジナルのEIPが一時的に保存される場所です。
14     long edi; long          // int7の処理中にスタックにプッシュされたレジスタ。
15     esi; long ebp;
16     long sys_call_ret;
17     long eax;               // システムコールのリターンコードを処理します。
18     long ebx; long          // 以下は、システムコールのスタックと同じです。
19     ecx; long edx;
20     long orig_eax;
21     long fs;
22     long es; long           // システムコールでなければ-1です。
23     ds; long eip;
24     long cs; long
25     eflags; long
26     esp; long ss;           // CPUが自動的にプッシュします
27     .
28
29
30
31 };
```

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、現在のプロセスに指定された信号を送信し、スタックポインタespが割込みプロセスのmath_emulate()関数のリターンアドレス(math_ret)を指すように修正し、直ちに割込みハンドラに戻ります。

11.2.2 コードアナリティクション

プログラム 11-1 linux/kernel/math/math_emulate.c

```

1 /*
2 * linux/kernel/math/math_emulate.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * 限定的なエミュレーション 27.12.91 - ほとんどがロード/ストアで、gccはこれを望んでいます。
9 * Bruce Evans氏のパッチを使用しない限り、soft-floatであっても同様です。パッチは
10 * は素晴らしいものですが、バージョンごとに適用し直さなければならず、また
11 * soft-floatと80387ではライブラリが異なります。そのため、エミュレーションはより
12 * 遅くても実用的です。
13 */
14 * 28.12.91 - BCDでもロード/ストアが動作します。考え始めなければならないのは
15 * add/sub/mul/divについて。Urgel. 何か良いソースを見つけなければならないのですが、私は
16 * 何かを捏造するだけ。
17 */
18 * 30.12.91 - add/sub/mul/div/com はほとんど機能しているようです。私の場合は
19 * あらゆる可能な組み合わせをテストします。
20 */
21
22 /*
23 * このファイルは、醜いマクロなどでいっぱいです：一つの問題は、gccが単に
24 * は、本来あるべき構造を作ろうとはせず、次のことに挑戦しなければなりません。
25 * align them. 気持ち悪いコードですが、少なくとも私は醜いものを隠しました
26 * この1つのファイルの中では、他のファイルはこれらのことなどを知る必要はありません。
27 */
28 * 他のファイルもST(x)などは気にせず、アドレスを取得するだけです。
29 * を80ビットの一時的なリアルに変換し、それを好きなように使えるようにしました。私がやりた
30 * 387固有のものはほとんどここに隠されています。
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・オペレーションのための
33 #include <signal.h> (英語)

```

```

34
35 #define ALIGNED_TEMP_REAL 1
36 #include <linux/math_emu.h>
37 #include <linux/kernel.h> (日本語)
38 #include <asm/segment.h>
39
40 #define bswapw(x) asm ("xchgb %%al,%%ah": "=a" (x) : ((short)x)) // 2バイトの交換。
41 #define ST(x) (*st((x))) // ST(x)のシミュレーション値を得る。
42#define PST(x) ((const temp_real *) st((x))) // シミュレーションされたST(x)への
43) pointrを取得します。
44 /*
45 * これらはインライン化したくありません - マシンコードの中で混乱してしまいます。
46 */
47 // 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。
48 static void fpop(void);
49 static void fpush(void);
50 static temp_real_unaligned * st(int i);
51
52 // 浮動小数点命令のシミュレーションを行う。
53 // 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
54 // します。
55 // 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・
56 // フラグBを設定する。その後
57 // 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
58 // ポインタEIPで//を入力し、そのコードを解析して意味に応じた処理を行なうことができます。については
59 // コードの種類によって、Linusはシミュレーション処理にいくつかの異なるスイッチブロックを使
60 // 用している。
61 // 引数はtemp_feature構造体へのポインタです。
62 static void do_emu(struct info * info)
63 {...}
64
65 // 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
66 // 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
67 // 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。
68 // 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
69 // になっている場合）、次のようにになります。
70 // カーネルには浮動小数点命令を使用するコードがありますが、これは許されません。そのため
71 // カーネルのCS、EIP、「Needs Mathematical Simulation in
72 // 浮動小数点演算命令のI387 cwd & I387 swd」情報 if (I387 cwd & I387 swd) のクリア B.
73 // &0x3f) = EIP; // オリジナルのEIPを保存
74 // 0x0007 はユーザーコードトスペース。
75 // I387 cwd & 0x8000 // ビジーフラグの設定。
76 // を意味します。
77 // ユーザーコードスペースにない場合
78 if (CS != 0x000f) printk("math_emulate: %04x:%08x\n", STOP);
79 panic("数学のエミュレー カーネルに必要なもの")
80 // 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
81 // CPUは "Little endien"スタイルでデータを格納しているため、この時に取得されるコードは全く
82 // 逆になります。
83 // 命令の1バイト目と2バイト目からの//の順序を入れ替える必要があります。
84 // のバイトを「コード」にします。そして、最初のコードバイト（バイナリ11011）のESCビットをマ
85 // スクアウトします。

```

```

// 次に、浮動小数点命令ポインタEIPがTSSセグメントのfipフィールドに保存されます。
// i387構造体で、CSをfcsフィールドに保存し、わずかに処理されたフローティング
fcsフィールドの上位16ビットには//point instruction codeが入ります。これらの値は
// プログラムを保存しておくことで、擬似的に本物のコプロセッサのように処理することができます
。
// プロセッサの例外です。最後に、EIPが後続の浮動小数点演算命令を指すようにする
68 // またはaddgetfs word((unsigned short *) EIP); // 2バイトの浮動小数点コードを取得し
69     bswapw(code);                               ます。
70     code &= 0x7ff;                                // バイトを交換します。
71     I387.fip = EIP;                            // コードの中のESCの部分をマスクする
72     *(unsigned short *) & I387.fcs = CS;
73     *(1+(unsigned short *) & I387.fcs) =      // EIP、コードセレクター、コードを保
74     code; EIP += 2;                            存し次の命令コードを指す。
.

// そして、そのコードを解析し、意味に応じた処理を行います。異なるコードタイプの場合
// の値は、Linusは処理のためにいくつかの異なるスイッチブロックを使用します。
// (1) まず、命令のオペコードが固定のコード値を持つ場合（レジスタに依存しない
// のように）、以下のように処理されます。
// マクロmath_abort()は、コプロセッサのエミュレーション操作を終了させるために使用され、定義
// されています。
// ファイルlinux/math_emu.hの52行目にあります。実際の実装コードは、linux/math_emu.hの488行
75 目にあります。
76 // プログラムでcastionmath_*endスケルトンの前に記述を参照してください
77     さい。    もの。
78     case 0x1d1: case 0x1d2: case 0x1d3: // 無効なコード、シグナルを送って終了。
79         math_abort(info, 1<<(SIGILL-1));
80
81     case 0x1e0: // FCHS - STの符号ビットを変更します。ST = -ST.
82         ST(0).exponent ^= 0x8000;
83         を返すことができます。
84     case 0x1e1: // FABS - 絶対値の取得。ST = |ST|.
85         ST(0).exponent &= 0x7fff;
86         return;
87     case 0x1e2: case 0x1e3: // 無効なコード。シグナルを送って終了。
88
89     case 0x1e4:           // FTST - TSをテストし、ステータスワードにCn
90         ftst(PST(0));    を設定する。
91         return;
92     case 0x1e5:           // FXAM - TSをチェックし、ステータスワードのCnを
93         printf("fxam not implemented\n");
94         math_abort(info, 1<<(SIGILL-1));
95     case 0x1e6: case 0x1e7: // 無効なコード。シグナルを送って終了。
96         math_abort(info, 1<<(SIGILL-1));
97     case 0x1e8:           // FLD1 - 定数1.0をアキュムレータにロード ST.
98         fpush();          // FLD1です。
99         st(0)=const1;
100        return;
101    case 0x1e9:           // FLDL2T - 定数Log2(10)をSTに読み込む。
102        fpush();          // FLDL2Tです。
103        st(0) = const1t;
104        return;
105    case 0x1ea:           // FLDL2E - 定数Log2(e)をSTにロードする
106        fpush();          // FLDL2Eです。
.

```

```

107         st(0) = const12e;
108         を返すことができます。
109     case 0x1eb:           // FLDPI - 定数PiをSTにロードします。
110         fpush() です。
111         st(0) = constpi;
112         を返すことができます。
113     case 0x1ec:           // FLDLG2 - 定数Log10(2)をSTにロードする。
114         fpush() です。
115         st(0) = constlg2;
116         を返すことができます。
117     case 0x1ed:           // FLDLN2 - 定数Loge(2)をSTにロードする。
118         fpush() です。
119         st(0) = constln2;
120         を返すことができます。
121     case 0x1ee:            // FLDZ - 定数0.0をSTに読み込みます。
122         fpush() です。
123         st(0)=constz;
124         を返すことができます。
125     case 0x1ef:            // シグナルを送って終了します。
126         math_abort(info, 1<<(SIGILL-1));
127     case 0x1f0: case 0x1f1: case 0x1f2: case 0x1f3:
128     case 0x1f4: case 0x1f5: case 0x1f6: case 0x1f7:
129     case 0x1f8: case 0x1f9: case 0x1fa: case 0x1fb:
130     case 0x1fc: case 0x1fd: case 0x1fe: case 0x1ff:
131         printf("%"#04x "fxxx not implemented\n", code + 0xc800);
132         math_abort(info, 1<<(SIGILL-1));
133         case 0x2e9: // FUCOMPP - 順序比較なし。
134         fucom(PST(1), PST(0));
135         fpop(); fpop();
136         を返すことができます。
137     case 0x3d0: case 0x3d1: // FNOP - 387で。!!!0x3e0, 0x3e1でなければなりません。
138         を返すことができます。
139     case 0x3e2: // FCLEX - ステータスワードの例外フラグをクリアします。
140         I387_swd &= 0x7f00;
141         を返すことができます。
142         case 0x3e3: // FINIT - コプロセッサを初期化します。
143             I387_cwd=0x037fです。
144             I387_swd = 0x0000;
145             I387_twd = 0x0000;
146             を返すことができます。
147         case 0x3e4:           // 80387でFNOP-。
148             を返すことができます。
149         case 0x6d9:           // FCOMPP - ST(1)とSTを比較し、2回弾く。
150             fcom(PST(1), PST(0)) です。
151             fpop(); fpop();
152 // (2) 次に、2バイト目の最後の3ビットが0の意味であるという命令を処理します。つまり
153 // 11011, XXXXXXXXcfREG 0x7e09 形式 ST SW AX でスイドはズロナリで ST(AX)をレジスターに保存されま
154 ています。
155             *(short *) & EAX = I387_swd;
156             を返すことができます。
157         }

```

```

// マクロ real_to_real(a, b)は以下の間の代入に使用されます。 定義された2つの一時的なア
// ファイルlinux/math_emu.hの72行目にあります。 ル

157     switch (code >> 3) {
158         case 0x18: // FADD ST, ST(i)
159             fadd(PST(0), PST(code & 7), &tmp) です。
160             real_to_real(&tmp, & ST(0)) です。
161             を返すことができます。
162             case 0x19: // FMUL ST, ST(i)
163                 fmul(PST(0), PST(code & 7), &tmp)。
164                 real_to_real(&tmp, & ST(0)) です。
165             を返すことができます。
166             case 0x1a: // FCOM ST(i)
167                 fcom(PST(code & 7), &tmp) です。
168                 real_to_real(&tmp, & ST(0)) です。
169             を返すことができます。
170             case 0x1b: // FCOMP ST(i)
171                 fcom(PST(code & 7), &tmp) です。
172                 real_to_real(&tmp, & ST(0)) です。
173                 fpop() です。
174             を返すことができます。
175             case 0x1c: // FSUB ST, ST(i)
176                 real_to_real(& ST(code & 7), &tmp) です
177                 。
178                 tmp.exponent ^= 0x8000;
179                 fadd(PST(0), &tmp, &tmp) です。
180                 real_to_real(&tmp, & ST(0)) です。
181             を返すことができます。
182             case 0x1d: // FSUBR ST, ST(i)
183                 ST(0).exponent ^= 0x8000;
184                 fadd(PST(0), PST(code & 7), &tmp) です。
185                 real_to_real(&tmp, & ST(0)) です。
186             を返すことができます。
187             case 0x1e: // FDIV ST, ST(i)
188                 fdiv(PST(0), PST(code & 7), &tmp) です。
189                 real_to_real(&tmp, & ST(0)) です。
190             を返すことができます。
191             case 0x1f: // FDIVR ST, ST(i)
192                 fdiv(PST(code & 7), PST(0), &tmp) となり
193                 ます。
194                 real_to_real(&tmp, & ST(0)) です。
195             を返すことができます。
196             case 0x38: // FLD ST(i)
197                 fpush() です。
198                 ST(0) = ST((コード&7) +1)。
199             を返すことができます。
200             case 0x39: // FXCH ST(i)
201                 fxchg(& ST(0), & ST(code & 7)) となりま
202                 す。
203             を返すことができます。
204             case 0x3b: // FSTP ST(i)
205                 ST(コード&7) = ST(0) となります。
206                 fpop() です。
207             を返すことができます。
208             case 0x98: // FADD ST(i), ST
209                 fadd(PST(0), PST(code & 7), &tmp) です。
210                 real_to_real(&tmp, &6ST(code & 7));

```

```

208          を返すこ
209          とができ
210          ます。
211          case 0x99:           // FMUL ST(i), ST
212          fmul(PST(0), PST(code & 7), &tmp)。
213          real_to_real(&tmp, & ST(code & 7));
214          を返すことができます。
215          case 0x9a: // FCOM ST(i)
216          fcom(PST(code & 7), PST(0))です。
217          を返すことができます。
218          case 0x9b: // FCOMP ST(i)
219          fcom(PST(code & 7), PST(0))です。
220          fpop()です。
221          を返すことができます。
222          case 0x9c: // FSUBR ST(i), ST
223          ST(code & 7).exponent ^= 0x8000;
224          fadd(PST(0), PST(code & 7), &tmp)です。
225          real_to_real(&tmp, & ST(code & 7));
226          を返すことができます。
227          case 0x9d: // FSUB ST(i), ST
228          real_to_real(&ST(0), &tmp)です。
229          tmp.exponent ^= 0x8000;
230          fadd(PST(code & 7), &tmp, &tmp)。
231          real_to_real(&tmp, & ST(code & 7));
232          を返すことができます。
233          case 0x9e: // FDIVR ST(i), ST
234          fdiv(PST(0), PST(code & 7), &tmp)となります。
235          real_to_real(&tmp, & ST(code & 7));
236          を返すことができます。
237          case 0x9f: // FDIV ST(i), ST
238          fdiv(PST(code & 7), PST(0), &tmp)となります。
239          real_to_real(&tmp, & ST(code & 7));
240          を返すことができます。
241          case 0xb8: // FFREE ST(i)
242          printf("ffree not implemented\n");
243          math_abort(info, 1<<(SIGILL-1));
244          case 0xb9: // FXCH ST(i)
245          fxchg(& ST(0), & ST(code & 7))となります。
246          を返すことができます。
247          case 0xba: // FST ST(i)
248          ST(コード&7) = ST(0)となります。
249          を返すことができます。
250          case 0xbb: // FSTP ST(i)
251          ST(コード&7) = ST(0)となります。
252          fpop()です。
253          を返すことができます。
254          case 0xbc: // FUCOM ST(i)
255          fucom(PST(code & 7), PST(0));
256          を返すことができます。
257          case 0xbd: // FUCOMP ST(i)
258          fucom(PST(code & 7), PST(0));
259          fpop()です。
260          を返すことができます。

```

```

261         real_to_real(&tmp, & ST(code & 7));
262         fpop() です。
263         を返すことができます。
264     case 0xd9: // FMULP ST(i), ST
265         fmul(PST(code & 7), PST(0), &tmp)。
266         real_to_real(&tmp, & ST(code & 7));
267         fpop() です。
268         を返すことができます。
269     case 0xda: // FCOMP ST(i)
270         fcom(PST(code & 7), PST(0)) です。
271         fpop() です。
272         を返すことができます。
273         case 0xdc: // FSUBRP ST(i), ST
274             ST(code & 7).exponent ^= 0x8000;
275             fadd(PST(0), PST(code & 7), &tmp) です。
276             real_to_real(&tmp, & ST(code & 7));
277             fpop() です。
278             を返すことができます。
279     case 0xdd: // FSUBP ST(i), ST
280         real_to_real(&ST(0), &tmp) です。
281         tmp.exponent ^= 0x8000;
282         fadd(PST(code & 7), &tmp, &tmp)。
283         real_to_real(&tmp, & ST(code & 7));
284         fpop() です。
285         を返すことができます。
286         case 0xde: // FDIVRP ST(i), ST
287             fdiv(PST(0), PST(code & 7), &tmp) です。
288             real_to_real(&tmp, & ST(code & 7));
289             fpop() です。
290             を返すことができます。
291     case 0xdf: // FDIVP ST(i), ST
292         fdiv(PST(code & 7), PST(0), &tmp) となります。
293         real_to_real(&tmp, & ST(code & 7));
294         fpop() です。
295         を返すことができます。
296     case 0xf8: // FFREE ST(i)
297         printf("ffree not implemented\n") です。
298         math_abort(info, 1<<(SIGILL-1));
299         fpop() です。
300         を返すことができます。
301     case 0xf9: // FXCH ST(i)
302         fxch(& ST(0), & ST(code & 7)) となります。
303         を返すことができます。
304     case 0xfa: // FSTP ST(i)
305     case 0xfb: // FSTP ST(i)
306         ST(コード&7) = ST(0) となります。
307         fpop() です。
308         を返すことができます。
309 // (3) 次に、2バイト目の7-6がMOD、ビット2-0が--という形でコードを処理します。1)
// R/M、つまり「11011, XXX, MOD, XXX, R/M」となります。MODは各サブルーチンで処理されます。
// なので、まずコードをAND 0xe7 (つまり0b11100111)にして、MODをマスクアウトさせます。

```

```

310     switch ((code>>3) & 0xe7) {
311         case 0x22: // FST - 単精度リアル (short real) を保存
312             put_short_real(PST(0), info, code);
313             を返すことができます。
314             case 0x23: // FSTP - 単精度リアル (short real) を保存
315                 put_short_real(PST(0), info, code);
316                 fpop();
317                 return;
318             case 0x24: // FLDENV - ステータスレジスタやコントロールレジスタなどをロード
319                 する address = ea(info, code); // 効率的なアドレスを取得する。
320                 for (code = 0 ; code < 7 ; code++) {
321                     ((long *) & I387)[code] =
322                         ...
323                         get_fs_long((unsigned long *) address);
324                         address += 4;
325                 }
326             case 0x25: // FLDENV - ができます。control word. address =
327                 ea(info, code);
328                 *(unsigned short *) & I387.cwd =
329                     get_fs_word((unsigned short *) address)
330                     です。
331             case 0x26: // FSTENV - ができます。タスやコントロールレジスタなどを格納する
332                 address = ea(info, code);
333                 verify_area(address, 28) です。
334                 for (code = 0 ; code < 7 ; code++) {
335                     put_fs_long((long *) & I387)[code],
336                         (unsigned long *) address);
337                         address += 4;
338                 }
339                 を返すことができます。
340             case 0x27: // FSTCW - コントロールワードの格納 address =
341                 ea(info, code);
342                 verify_area(address, 2);
343                 put_fs_word(I387.cwd, (short *) address);
344                 return;
345             case 0x62: // FIST - 短い整数を格納する
346                 put_long_int(PST(0), info, code);
347                 を返すことができます。
348             case 0x63: // FISTP - 短い整数を格納する
349                 put_long_int(PST(0), info, code);
350                 fpop();
351                 return;
352             case 0x65: // FLD - 拡張された（一時的な）実数を読み込む fpush();
353                 get_temp_real(&tmp, info, code);
354                 real_to_real(&tmp, & ST(0));
355                 return;
356                 put_temp_real(PST(0), info, code);
357                 fpop() です。
358                 を返すことができます。
359                 put_long_real(PST(0), info, code);
360
361
362

```

```

363         を返すことができます。
364         put_long_real(PST(0), info, code);
365             fpop();
366             return;
367
368     case 0xa4: // FRSTOR - 108バイトのレジスタの内容をすべて復元します。
369         address = ea(info, code);
370         for (code = 0 ; code < 27 ; code++) {
371             ((long *) & I387)[code] = ...
372             get_fs_long((unsigned long *) address);
373             address += 4;
374         }
375         を返すことができます。
376     case 0xa6: // FSAVE - 108バイトのレジスタの内容をすべて保存します。
377         verify_area(address, 108) です。
378         for (code = 0 ; code < 27 ; code++) {
379             put_fs_long((long *) & I387[code],
380                         (unsigned long *) address);
381             address += 4;
382         }
383         I387. cwd = 0x037f;
384         I387. swd = 0x0000;
385         I387. twd = 0x0000
386         です。
387         を返すことができます。
388     case 0xa7: // FSTSW - Store status word. address =
389         ea(info, code); verify_area(address, 2);
390         put_fs_word(I387. swd, (short *) address);
391         return;
392         put_short_int(PST(0), info, code);
393         を返すことができます。
394
395
396     case 0xe3: // FISTP - 短い整数を格納します。
397         put_short_int(PST(0), info, code);
398         fpop();
399         を返すことができます。
400     case 0xe4: // FBLD - BCD型の数値をロードする fpush();
401
402         get_BCD(&tmp, info, code);
403         real_to_real(&tmp, &
404             ST(0)); return;
405
406     case 0xe5: // FILD - 長い整数をロードする fpush();
407         get_longlong_int(&tmp, info, code);
408         real_to_real(&tmp, & ST(0));
409         return;
410
411     case 0xe6: // FBSTP - BCD型の数値を格納する。
412         fpop() です。
413         を返すことができます。
414         put_longlong_int(PST(0), info, code);
415

```

```

416          fpop() です。
417          を返すことができます。
418      }
419      // (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
420      // インストラクションコードのビット10~9のMFに応じて、指定されたタイプの//が取られます。
421      //となり、OPAとOPBの合計値に応じて別々に処理されます。その
422      // つまり、「11011、MF、000、XXX、R/M」という形式の命令コードが処理されるのです。
423      // スイッチ (コード >> 9) {
424      case 0:           // MF=00、ショートリアル (32ビットリアル)。
425          get_short_real(&tmp, info, code) です。
426          ブレークします。
427      case 1:           // MF = 01、短い整数 (32ビット整数)。
428          get_long_int(&tmp, info, code) です。
429          ブレークします。
430      case 2:           // MF = 10、long real (64-bit real).
431          get_long_real(&tmp, info, code) です。
432          ブレークします。
433      case 4:           // MF = 11、長い整数 (64ビット)。
434          get_short_int(&tmp, info, code)
435      }
436      // (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。
437      switch ((code>>3) & 0x27) {
438      case 0:           // FADD
439          fadd(&tmp, PST(0), &tmp) です
440          .
441          real_to_real(&tmp, & ST(0))。
442          を返すことができます。
443      case 1:           // FMUL
444          fmul (&tmp, PST(0), &tmp) 。
445          real_to_real(&tmp, & ST(0))。
446          を返すことができます。
447      case 2:           // FCOM
448          fcom (&tmp, PST(0)) です。
449          を返すことができます。
450      case 3:           // FCMP
451          fcom (&tmp, PST(0)) です。
452          fpop() です。
453          を返すことができます。
454      case 4:           // FSUB
455          tmp.exponent ^= 0x8000;
456          fadd(&tmp, PST(0), &tmp) です
457          .
458          real_to_real(&tmp, & ST(0))。
459          を返すことができます。
460      case 5:           // FSUBR
461          ST(0).exponent ^= 0x8000;
462          fadd(&tmp, PST(0), &tmp) です
463          .
464          real_to_real(&tmp, & ST(0))。
465          を返すことができます。
466      case 6:           // FDIV
467          fdiv (PST(0), &tmp, &tmp) です
468          .
469          real_to_real(&tmp, & ST(0))。
470          を返すことができます。
471      case 7:           658 // FDIVR
472          fdiv(&tmp, PST(0), &tmp) です
473          .

```

```

464           real_to_real(&tmp, & ST(0)) です。
465           を返すことができます。
466       }
467       // プロセス 11011, XX, 1, XX, 000, R/M ”という形式の命令コードです。
468       もし ((code & 0x138) == 0x100) { // FLD, FILD
469           fpush() です。
470           real_to_real(&tmp, & ST(0)) です。
471           を返すことができます。
472       }
473       // 残りの部 は無効な命令です。
474       pintk("Unknown math-insns: %04x:%08x %04x%n%r", CS, EIP, code);
475       math_abort(info, 1<<(SIGFPE-1));
476   }
477
478   // 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.
479   // 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
480   // used_mathを入力し、80387のコントロールワード、ステータスワード、タグワードを初期化します
481   // 最後に
482   // int 7 がこの関数を呼び出したときの戻り値を、パラメータとして使用して、フローイング
483   // ポイント命令エミュレーション関数 do_emu(). パラメータfalseには_orig_eipを指定します。
484 void math_emulate(long false)
485 {
486     if (! current->used_math) {...}
487     current->used_math = 1;
488     I387. cwd = 0x037f;
489     I387. swd = 0x0000;
490     I387. twd = 0x0000;
491
492     /* & false は info-> orig_eip を指しているので、1を引いて info を得る
493      do_emu((struct info*) ((& false) - 1)).
494
495     // シミュレーションを終了します。
496     // 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
497     // 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
498     // 処理を行います。最後に、int7がこれを起動したときの戻り値をスタックポインタで指定します。
499     // 関数を使用して、直接割り込みに戻ります。この関数は、マクロの
500     // math_abort(). 詳しい説明は、linux/math_emu.hファイルの50行目をご覧ください。
501 void math_abort(struct info* info, unsigned int signal)
502 {...}
503     eip = orig_eip;
504     current->signal |= signal;
505     asm ("movl %0, %%esp ; ret": "g" ((long) info) で
506         す。)
507
508     // アキュムレータ・スタック・ポップアップ・オペレーション
509     // 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。
510 static void fpop(void)
511 {...}
512     符号付き長さのtmp
513
514     tmp = I387. swd & 0xfffffc7ff;
515     I387. swd += 0x00000800;
516     I387. swd &= 0x00003800;

```

```

502      I387.swd |= tmp;
503 }
504
// アキュムレータ・スタック プッシュ操作。
// ステータスワードのTOPフィールドを1デクリメント（つまり7を加算）し、モジュロ7とする。
505 static void fpush(void)
506 { ... 符号付き長さのtmp
508
509     tmp = I387.swd & 0xfffffc7ff;
510     I387.swd += 0x00003800;
511     I387.swd &= 0x00003800;
512     I387.swd |= tmp;
513 }
514
// 2つのアキュムレータ・レジスタの値を入れ替えます。
515 static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b)
516 {
517     temp_real_unaligned c;
518
519     c = *aで
520     す。
521     *a = *b
522 }   です。
523     *b = cで
524
// I387構造体のST(i)メモリポインタを取得する。
// ステータスワードのTOPフィールド値を取り、指定された物理データレジスタ番号を加える
// とmodulo 7を組み合わせて、最後にST(i)に対応するポインタを返すようにしました。
524 static temp_real_unaligned * st(int i)
525 {
526     i += I387.swd >> 11; // ステータスワードのTOPフィールドを取得 i &= 7;
527
528     (temp_real_unaligned *) (i*10 + (char *) (I387.st_space))を返しま
529 }   す。
530

```

3. error.c

1. 機能

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。

2. コードアノテーション

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
// シグナル操作関数のプロトタイプ。
// <linux/sched.h> のマクロと、この のリターナータイプでは取得に構造体の組み込みア
// センブリ関数マクロの記述があります。
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ギービットをクリアします。最後のタスクがコプロセッサを使用していた場合。
// そして、コプロセッサのエラー信号フラグを設定します。戻り値の後、この関数は次のようにジャ
ンプします。
13 // ret_fromasm_callrex テムコール割り込み時の場所で実行を継続すべきワードのビギービット
14 void math_error(task)      をクリアします。
15 {
16     last_task_used_math = signal 中セレクタ(使用している場合は、シグナルフラグを
17     設定します。

```

4. ea.c

1. 機能

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。

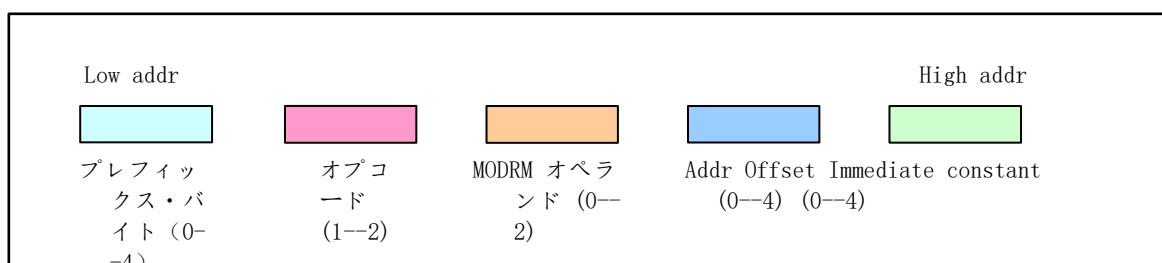


図11-7 一般命令の符号化形式

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0~4バイトで構成され、次の命令を変更するために使用されます。opcodeフィールドは、操作を示す主要なフィールドです。

命令の各命令には、少なくとも1バイトのオペコードが必要です。必要に応じて、命令のオペコードフィールドは、オペランドの種類と数を明示的に示すために使用されるMODRMオペランドインジケーターに従うかどうかを示します。メモリオペランドの場合、アドレスディスペイスマントフィールドは、オペランドのオフセットを与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、またはアドレス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。MODRMエンコーディングの中には、アドレスシングモードを示すために、2番目のアドレスシングバイト(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)スケール-スケール(S)フィールド

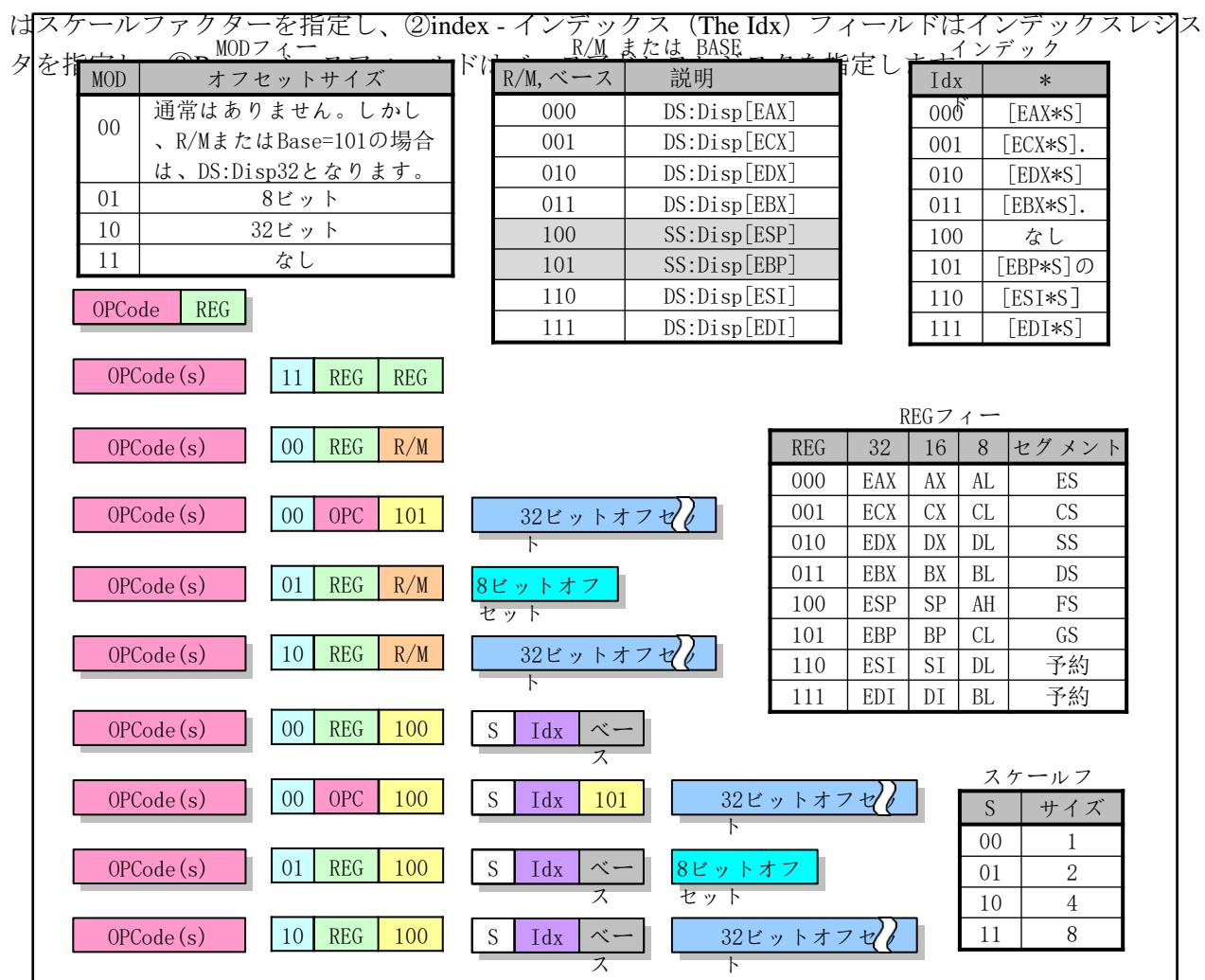


図11-8 インストラクション・エンコーディングとアドレスシング・フォーマットの概要

本プログラムのea()関数は、命令中のアドレスシング・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。もし

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

11.4.2 コードアナリティクス

プログラム 11-3 linux/kernel/math/ea.c

```

1 /*
2 * linux/kernel/math/ea.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * 実効アドレスの算出
9 */
10
// <stddef.h> 標準定義のヘッダファイルです。NULL, offsetof(TYPE, MEMBER) が定義されています。
// <linux/math_emu.h> コプロセッサのエミュレーションのためのヘッダーファイルです。コプロセッサのデータ構造と
// 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
13 #include <linux/math_emu.h>
14 #include <asm/segment.h>
15
// 情報構造体における各レジスタのオフセット位置。offsetof()を使用することで
16 // 構造体のオフセット位置。include/stddef.hファイルを参照してください。
17 // 構造体のオフセット位置。構造体情報
18 // オフセットof(構造体情報
19 static int regoffsex[セグメント].of(構造体情
20     報, edx), オフセットof(構造体
21     情報, ebx), オフセットof(構造
22     体情報, esp), オフセットof(構
23     造体情報, ebp), オフセットof(
24     構造体情報, esi), オフセット
25 };      of(構造体情報, edi)
26
// info構造体の指定された位置にあるレジスタの内容を取得します。
27 #define REG(x) (*(long *)(_regoffset[(x)]+(char *)_info))
28
// 第2オペランド指示バイトSIB(Scale, Index, Base)の値を取得します。
29 static char * sib(struct _info *_info, int mod)
30 {
31     unsigned char ss, index, base;
32     long offset = 0;
33
// SIBバイトはまずユーザーコードセグメントから取得し、次にフィールドビット値を取得します。

```

```

34     base = get_fs_byte((char *) EIP);
35     EIP++です。
36     ss = base >> 6;                      // スケールサ
                                                イズ
37     index = (base >> 3) & 7;
38 // インデックスが0の場合は、インデックスオフセット値がないことを意味します。それ
    以外の場合は、インデックスオフセット
39 // 値のオフセット = レジスタの内容 * スケールファ
40     クター if (index == 4)
41         offset = 0となります。
42     その他
43         offset = REG(index);
44     offset <= ss;

// 前のMODRMバイトのMODが0でない場合、またはBaseが0b101に等しくない場合、それは
// baseで指定されたレジスタにオフセット値があること。したがって、オフセットは
// ベースとなる対応するレジスタの内容に加算されます。MOD=1の場合、オフセット値は
44 // は1バイト(8ビット)です。その他、MOD=2、base=0b101の場合、オフセット値は4バイトとな
45     ります。 if (mod || base != 5)
46         offset += REG(base);
47     if (mod == 1) {
48         offset += (signed char) get_fs_byte((char *) EIP);
49         EIP++;
50     } else if (mod == 2 || base == 5) {
51         offset += (signed) get_fs_long((unsigned long *) EIP);
52         EIP+= 4;
53     }
53 // 最後にオフセット値を保存して返します。
54     I387.foo = offset;
55     I387.fos = 0x17;
56 }
56 return (char *) offset;
57
// 命令コードのアドレッシングモードバイトを基に、実効アドレスを算出する。
58 char * ea(struct info * info, unsigned short code)
59 {...}
60     unsigned char mod, rm;
61     long * tmp = & EAX;
62     int offset = 0;
63
// まず、命令コードのMODフィールドとR/Mフィールドの値を取る。MOD=0b11であれば
// は、オフセットフィールドのない1バイト命令を意味します。R/Mフィールド=0b100でMODが
// 0b11ではなく、2バイトアドレスモードのアドレッシングを意味するので、sib()を呼び出してオフ
    セット値を見つける
64 // と返します。code >> 6) & 3;           // MODフィール
                                            ドです。
65     rm = コード & 7;                   // R/M分野。
66     if (rm == 4 &&) mod != 3)
67 // R/Mフィールドが0b101で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を示す
    // に続いて32バイトのオフセット値を入力します。その後、ユーザーコードの4バイトのオフセット
    値を取り、保存します。
68 // 返します。
69     if (rm == 5 && !mod) {
70         offset = get_fs_long((unsigned long *) EIP);
71         EIP+= 4;
72         I387.foo = offset;

```

```

72     I387. fos = 0x17;
73     return (char *) offset;
74 }
// 残りのケースについては、MODに従って処理されます。まず、コンテンツの値
R/Mコードの対応するレジスタの//がポインタtmpとして取り出されます。MOD=0の場合。
// オフセット値はありません。MOD=1の場合は、コードの後に1バイトのオフセット値が入ります。
MOD=2の場合。
75 // コードの後に4バイトのオフセットがあります。最後に有効なアドレス値を保存して返す。 tmp =
76     & REG(rm);
77     スイッチ (MOD) {
78         case 0: offset = 0; break;
79         case 1:
80             offset = (signed char) get_fs_byte((char *) EIP);
81             EIP++;
82             ブレークします。
83         case 2:
84             offset = (signed) get_fs_long((unsigned long *) EIP);
85             EIP += 4;
86             ブレークします。
87         case 3:
88             math_abort(info, 1<<(SIGILL-1));
89     }
I387. foo = offset;
I387. fos = 0x17;
90 }
91 return offset + (char *) *tmp;
92 }
93

```

5. convert.c

1. 機能

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。

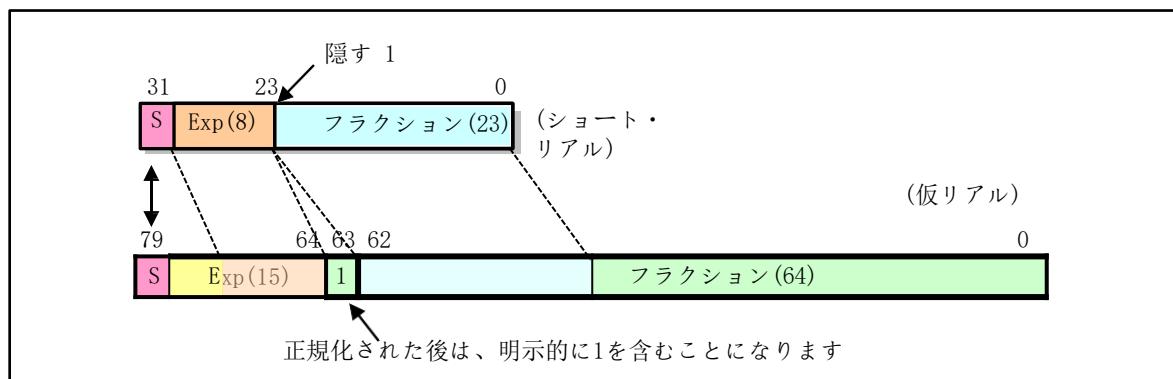


図11-9 ショートリアルからテンポラリーリアル形式への変換図

11.5.2 コードアノテーション

プログラム 11-4 linux/kernel/math/convert.c

```

1 /*
2 * linux/kernel/math/convert.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 #include <linux/math_emu.h> ←ここをクリックしてください。
8
9 /*
10 * 注意!!!temp_to_longには「明らかではない」最適化が施されています。
11 * およびtemp_to_short変換ルーチン: 以下の条件を満たさない場合は、これらに触れないでください。
12 * は、何が起こっているかを知っています。彼らは、丸めの中の1の足し算である: .
13 * オーバーフロービットは、指数に1を加算する際にも使用されます。そのため
14 * は、オーバーフローが正しく処理されないように見えますが、その原因は
15 * IEEEの数字の使い方は正しいです。
16 *
17 * 変換時のトータルオーバーフローのチェックはありませんが(つまり
18 * 臨時実数が単に短実数や長実数に收まらない場合)。
19 */
20
// 短い実数を一時的な実数に変換します。
// 短い実数は32ビット、その有効数字(仮数)は23ビットの長さです。
// 指数は8ビットで、符号ビットが1つあります。
21 void short_to_temp(const short_real * a, temp_real * b)
22 {
    // まず、短い実数が0である場合を扱います。
    // 一時的な実数bは0に設定され、一時的な実数の符号ビットは次のようになります。
    // 短い実数の符号ビットつまり指数の最上位ビットに応じて設定されます。
23    if ((*a & 0x7fffff) == ...)
        b->a = b->b = 0;                      // テンパイのシニフィカントを設定 real =
24    if (*a)                                0.
25        b->exponent = 0x8000;               // 符号ビットを
26   その他                                設定します。
27        b->exponent = 0;
28    を返すことができます。
29 }
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、整数值に偏った表現の概念を使う必要があります
// (11.1節参照)を使用しています。短い実数指数のバイアス番号は127、バイアスの一時的な実数の指数の//は16383です。したがって、指数の値を抽出した後に
// ショートリアルの場合、バイアス値を16383に変更する必要があります。これにより、指数の値を一時的な実数形式で表示します。また、短い実数が負の値の場合は
// で一時的な実数の符号ビット(ビット79)を設定します。次に、仮数を設定します。その方法は
// で短い実数を8ビット左にシフトし、最上位の23桁を
仮の実数のビット62に仮の実数の//が入ります。仮実数のビット63
// 仮数を1にする必要があるので、OR 0x80000000の演算が必要です。最後に、ロー
// テンポラリー・リアルの32ビット有効数字がクリアされます。

```

```

31     b->exponent = ((*a>>23) & 0xff)-127+16383と // 偏った値を16383に変更します。
32     なります。
33     if (*a<0)
34         b->exponent |= 0x8000; // 必要に応じて負の記号を加える
35         .
36     }
37 // long realをtemporary realに変換します。
38 // このメソッドは、short_to_temp()と全く同じですが、long realは64ビット長です。
39 // その有効数字（仮数）は52ビット、指数は11ビット、そして
40 // 1の符号ビット。また、ロングリアル指数の偏り値は1023です。
41 void long_to_temp(const long_real * a, temp_real * b)
42 {
43     if (!a->a && !(a->b & 0x7fffff)) {
44         b->a = b->b = 0;
45         if (a->b)
46             b->exponent = 0x8000; // 符号ビットを
47             その他                     設定します。
48         b->exponent = ((a->b >> 20) & 0x7ff)-1023+16383; // 偏った値を16383に変更 if (a-
49         >b<0)
50         b->exponent |= 0x8000;
51         b->b = 0x80000000 | (a->b<<11) | (((unsigned long)a->a)>>21); // 1位。 b->a
52         = a->a<<11;
53     }
54
55 // 一時的な実数を短い実数に変換します。
56 // 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。
57 // 丸めの問題。
58 void temp_to_short(const temp_real * a, short_real * b)
59 {
60     // 指数部が0の場合、短い実数は、-0または0に設定されます。
61     // 符号ビットがない場合は、表11-2を参照してください。
62     も (! (a->exponent & 0x7fff)) {
63         し
64         *b = (a->exponent)?0x80000000:0;
65         を返すことができます。
66     }
67 // まず、指数部分の処理、つまり指数の偏りの量を(16383)
68 // の場合、//は短い実数の偏った量127に置き換えられ、符号ビットが設定されます。
69 // それは負の数です。
70     *b = (((long) a->exponent)-16383+127) << 23) & 0x7f800000;
71     if (a->exponent < 0) // 負の場合は符号を設定します。
72         *b |= 0x80000000;
73
74 // 続いて、一時的な実数からシニフカントの上位23ビットを取得し、以下の処理を行います。
75 // コントロールワードの丸め設定に応じて、丸め動作を行います。
76     *b |= (a->b >> 8) & 0x007fff; // tempの上位23ビットを取得 real.
77     スイッチ (ROUNDING) {
78         ケース ROUND_NEAREST:
79             if ((a->b & 0xff) > 0x80)
80                 ++*b;
81             ブレークします。

```

```

70     ケース ROUND DOWN。
71         if ((a->exponent & 0x8000) && (a->b & 0xff))
72             ++*b;
73         ブレークします。
74     ケース ROUND UP。
75         if (!(a->exponent & 0x8000) && (a->b & 0xff))
76             ++*b;
77         ブレークします。
78     }
79 }
80

// 一時的な実数を長い実数に変換します。
// 長い実数は64ビットの長さで、その有効数字（仮数）は52ビットの長さ、指数は
// は11ビットで、符号ビットが1つあり、指数の偏り値は1023です。
81 void temp_to_long(const temp_real * a, long_real * b)
82 {
83     if (!(a->exponent & 0x7fff)) {
84         b->a = 0;
85         b->b = (a->exponent)?0x80000000:0;
86         return;
87     }
88     b->b = (((0x7fff & (long) a->exponent)-16383+1023) << 20) & 0x7ff00000;
89     if (a->exponent < 0)
90         b->b |= 0x80000000;
91     b->b |= (a->b >> 11) & 0x000fffff;
92     b->a = a->b << 21;
93     b->a |= (a->a >> 11) & 0x001fffff;
94     switch (ROUNDING) {
95         ケース ROUND NEAREST:
96             if ((a->a & 0x7ff) > 0x400)
97                 asm ("addl $1,%0 ; adcl $0,%1"
98                      :"=r"(b->a), "=r"(b->b)
99                      :"r" (b->a), "I" (b->b)) となります。
100            break;
101        case ROUND DOWN:
102            if ((a->exponent & 0x8000) && (a->b & 0xff))
103                asm ("addl $1,%0 ; adcl $0,%1"
104                    :"=r"(b->a), "=r"(b->b)
105                    :"r" (b->a), "I" (b->b)) となります。
106            break;
107        case ROUND UP:
108            if (!(a->exponent & 0x8000) && (a->b & 0xff))
109                asm ("addl $1,%0 ; adcl $0,%1"
110                    :"=r"(b->a), "=r"(b->b)
111                    :"r" (b->a), "I" (b->b)) となります。
112        ブレークします。
113    }
114 }

// 一時的な実数を一時的な整数形式に変換します。
// 一時的な整数も10バイトで表されます。下位8バイトは符号なし整数
// の値を表し、上位2バイトは指数の値と符号ビットを表します。もし、最上位の
// 上位2バイトの//バイトが1の場合は負の数を示し、0の場合は

```

```

// a ポジティブな数字です。
ボイド real_to_int(const temp_real* a, temp_int * b)
117 {
118     int shift = 16383 + 63 - (a->exponent & 0x7fff)になります。
119     unsigned longのアンダーフロー。
120
121     b->a = b->b = アンダーフロー = 0;
122     b->sign = (a->exponent < 0)。
123     if (shift < 0) {...  

124         set_OE()です。  

125         を返すことができます。  

126     }  

127     if (shift < 32) {...  

128         b->b = a->b; b->a = a->a とな  

129         ります。  

130     } else if (shift < 64) {  

131         b->a = a->b; underflow = a->a;  

132         shift -= 32;  

133     } else if (shift < 96) {  

134         underflow = a->b;  

135         shift -= 64;  

136     } else  

137         を返すことができます。  

asm ("shrdl %2,%1,%0") // 32ビットの右シフト
138         : "=r" (アンダーフロー)、 "=r" (b-
139         >a)
140         : "c" ((char) shift) , "" (underflow), "I" (b->a));
141         : "=r" (b->a), "=r" (b->b)
142         : "c" ((char) shift) , "" (b->a), "I" (b->b))。
143         asm ("shrl %1,%0")
144         : "=r" (b->b)
145         : "c" ((char) shift) , "" (b->b)) のようになります。
146         スイッチ (ROUNDING) {
147             ケース ROUND NEAREST:
148                 asm ("addl %4,%5 ; adc1 $0,%0 ; adc1 $0,%1")
149                 : "=r" (b->a), "=r" (b->b)
150                 : "" (b->a), "I" (b->b)
151                 , "r" (0x7fffffff + (b->a & 1))
152                 , "m" (*&underflow) となります。
153             ブレークします。
154             ケース ROUND UP:
155                 if (!b->sign && underflow)
156                     asm ("addl $1,%0 ; adc1 $0,%1")
157                     : "=r" (b->a), "=r" (b->b)
158                     : "" (b->a), "I" (b->b)) となりま
159                     す。
160                     ブレークします。
161                     ケース ROUND DOWN:
162                         if (b->sign && underflow)
163                             asm ("addl $1,%0 ; adc1 $0,%1")
164                             : "=r" (b->a), "=r" (b->b)
165                             : "" (b->a), "I" (b->b)) となりま
166                         ブレークします。 669
167 }

```

168

// 一時的な整数を一時的な実数形式に変換します。

169 void int_to_real(const temp_int* a, temp_real* b)

170 {

// 元の値は整数なので、一時的に実数に変換する際に、指数が

// は、偏った量である16383に加えて63を加えます。これにより、記号部には

// を 2^{63} 倍することになりますが、これは符号部も整数値であることを意味します。

171 b->a = a->a です。

172 b->b = a->b となります。

173 if (b->a || b->b)

174 b->exponent = 16383 + 63 + (a->sign? 0x8000:0)

となりますが。

175 else {

176 b->exponent = 0;

177 を返すことができます。

178 // 変換フォーマットが正規化された後の正常な実数、つまり上位の
// ビットではゼロではありません。
す。

179 一方 (b->b >= 0) {

180 b->exponent--.

181 asm ("addl %0,%0 ; adc1 %1,%1")

182 :"=r"(b->a), "=r"(b->b)

183 :""(b->a), "I"(b->b))

となります。

184 }

185 }

186

6. add.c

1. 機能

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。

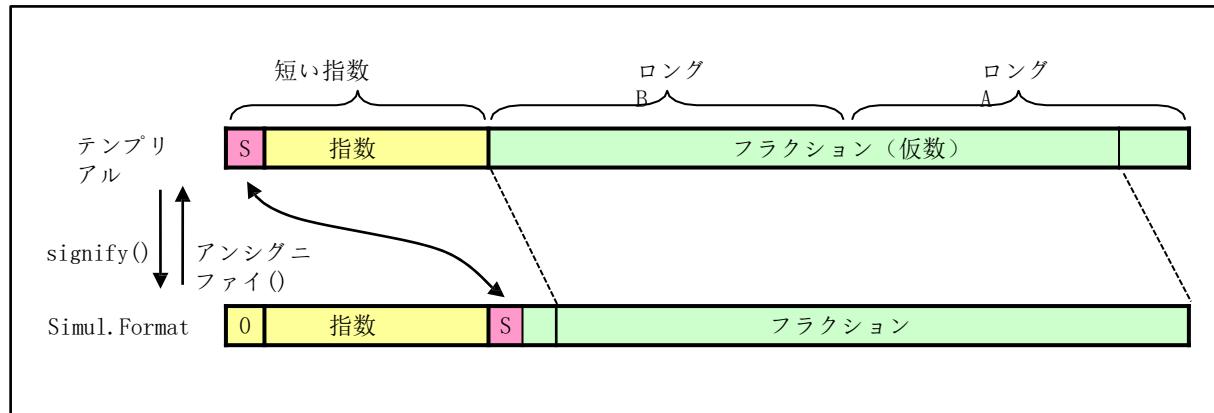


図11-10 一時的なリアルとシミュレーションのフォーマット間の変換

11.6.2 コードアノテーション

プログラム 11-5 linux/kernel/math/add.c

```

1 /*
2 * linux/kernel/math/add.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * 一時的な実数加算ルーチン。
9 */
10 * 注: これらは正確なものではありません: 62ビットの幅しかないため、以下のことはできません
11 * 正しい丸め方。高速なハックです。その理由は、右にシフトすることで
12 * オーバーフロー(1ビット)しないようにするために、*の値を2倍にして、*の値を表示します。
13 * 符号を仮数部(1ビット)に移動させるためのものです。よりシンプルなアルゴリズムです。
14 * 通常は、62ビット(実際に61ビット)の精度で十分です。そのため
15 * 奇妙なことに気づくのは、64ビットのアプリケーションを追加するときだけです。
16 * 整数をまとめて表示しています。倍精度(52ビット精度)を使用する場合は
17 * 61ビットの精度は全く発揮されません。
18 */
19
20 #include <linux/math_emu.h>を使用します。
21
// 数値の負の数(2の補数)表現を取得します。
// 演算内容は、一時的な実数の仮数(significand)を反転させて加算するものです。
// 1を指定します。パラメータ'a'は、一時的な実在の構造体へのポインタである。の組み合わせは
// そのフィールドaとbは、一時的な実数のシニフィカントです。
22 #define NEGINT(a) ♫ ♪ ♪
23 asm ("notl %0 ; notl %1 ; addl $1, %0 ; adcl $0, %1")
24     いう言葉があります。
25     :"" (a->a), "I" (a->b))
26
// 仮数を符号化しました。
// つまり、一時的な実数を指数と整数の表現に変換します。
// シミュレーションの操作を容易にするためにそのため、ここではシミュレーション・フォーマット
// と呼んでいます。
// 64ビットのバイナリの仮数を2ビット右に移動させる操作です(そのため、指数は
// 2を加算する)。指数フィールドの最上位ビットは符号ビットなので、指数値が
// が0より小さい場合、その数字は負の値になります。そこで、仮数を補数で表します。
// (マイナスを取る)して、指数をプラスに取ることができます。このとき、仮数部の
// には、2ビットシフトしたシニフィカントだけでなく、値のサインビットも含まれています。
27 static void signify(temp_real * a)
28 {.

// 30 行目: %0 は a->a; %1 は a->b です。アセンブリ命令 "shldl $2, %1, %0" は次のことを実行
// します。
// 倍精度(64ビット)の右シフトで、結合された仮数<b, a>を
29 // 2ビット右に移動します。この移動は %1 (a->b)/階級を割り下げるで、次のことも必要です
30 。      asm ("shrdl $2, %1, %0 ; shr1 $2, %1") // 仮数部は2ビット右にシフトします。
31 // を2ビット右に沿うa)し e)すb)の場合は 「=r」 と
// なります。

```

```

32          :"" (a->a), "I" (a->b))
33          となります。
34      if (a->exponent < 0)           // ネガティブになる
35          NEGINT(a)です。          // 符号ビット（もしあれば）を取り除きます。
36  } a->exponent &= 0x7fff;
37
38 static void unsignify(temp_real * a)
39 {
40 // 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
41 // 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
42 // 仮数部//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
43 // た、表
44 // (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
45 // 指数值はそれに応じてデクリメントされます。つまり、左シフトの操作を行います
46 // 仮数の最上位ビットが0にならないように（最後のa->bの値が見えるのは
47 // 負の値よりに瘤る) a->b) {
48     a->exponent = 0;           // ゼロの場合は
49     return;                   ret.
50 }
51     a->exponent &= 0x7fff;    // 符号ビットをリセットします。
52     if (a->b < 0) {          // 負の場合には、仮数を正とする。
53         NEGINT(a)です。
54         a->exponent |= 0x8000; // 符号ビットを追
55     }                         加します。
56     while (a->b >= 0) {
57         a->exponent--;        // 仮数は正規化されます。
58         asm ("addl %0, %0 ; adcl %1, %1")
59         (a->a), (a->b)の場合は [=r] とな
60         ります。
61     }                         :"" (a->a), "I" (a->b))
62     }                         となります。
63
64 void fadd(const temp_real * src1, const temp_real * src2, temp_real * result)
65 {...}
66     temp_real a, b;
67     int x1, x2, shift;
68
69     // まず、2つの数字の指数値x1, x2を取ります（符号ビットを取り除いたもの）。
70     // 次に、変数aを最大値とし、変数'shift'を
71     // x1 = src1->exponent & 0x7fff;
72     x2 = src2->exponent & 0x7fff;
73     if (x1 > x2) {...}
74         a = *src1;
75         b = *src2;
76         shift = x1-x2です。
77     } else {
78         a = *src2です。
79

```

```

70         b = *src1;
71         shift = x2-x1;
72     }
// 両者の差が2^64以上と大きすぎる場合は、無視することができます。
// 小さい数字（つまりbの値）を返します。つまり、値aを直接返すことができます。そうでなければ
// もし差が
// が2^32以上の場合は、下位32ビットの値を無視して、小
// そこで、bの上位ロングフィールド値b.bを32ビット右にシフト、つまり
// それをb.aにして、bの指数を32倍にします。つまり、指数の差は
// を32で引いたものになります。この調整の後、追加された2つの数字のマンタは次のようになります
73 .
74 // 実質的に同じ範囲である。 if
75     (shift >= 64) {
76         *result = a;
77         return;
78     }
79     if (shift >= 32) {
80         b.a = b.b;
81         b.b = 0;
82         shift -= 32;
83     }
// その後、微調整を行い、両者の指数が同じ値になるように調整します。そのためには
// 調整方法は、小数値bの仮数部を「シフト」ビット分だけ右に移動させます。
// したがって、この2つの指数は同じで、同じオーダーになります。そこで、次のように追加します
82 .
83     asm ("shrdl %4,%1,%0 ; shr1 %4,%1" // 倍精度（64ビット）の右シフト。
84 // 2つのマンチカ化を追加する前に、(A.b)、(B.b)のフォーマットに変換して
85 // 加算操作後に一時的な実数形式(扇り)ます。((char) shift))。
86     signify(&a); // フォーマットを変更する。
87     signify(&b);
88     asm ("addl %4,%0 ; adc1 %5,%1" // 通常の命令を使って追加します。
89             (a.a), (a.b)の場合は「r」となります。
90             :" (a.a), "l" (a.b), "g" (b.a), "g" (b.b));
91     unsignify(&a); // 変更 テンポリアルフォーマットに戻します。
92 }
93

```

7. compare.c

1. 機能

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。

2. コードアナリティクス

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

```

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

```

```

3 /*
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * 一時的な実数比較ルーチン
9 */
10
11 #include <linux/math_emu.h>を追加します。
12
13 // 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。
14 #define clear_Cx() (I387.swd &= ~0x4500)
15
16 static void normalize(temp_real * a)
17 {
18     int i = a->exponent & 0x7fff;           // 指数を取得します（符号ビットを無視
19     int sign = a->exponent & 0x8000;         // します）。
20     // 一時的な実数aの64ビットの有効数字（仮数）が0であれば、aは次のようにになります。
21     // そこで、aの指数をクリアして戻ります。 if
22     (! (a->a || a->b)) {
23         a->exponent = 0;
24         return;
25     }
26     // a の仮数が左端にゼロ値のビットを持つ場合、仮数を左にシフトする。
27     // と指数値を調整（デクリメント）します。のMSB（最上位ビット）まで、b
28     // フィールド（bが負の値として現れる場合）。最後にサインビットを加えます。
29     // は1
30    一方 (i && a->b >= 0) {
31         i--;
32         asm ("addl %0,%0 ; adcl %1,%1")
33         // (a->a), (a->b) の場合は [=r] となります。
34         // :"" (a->a), "I" (a->b) となります。
35         a->exponent = i | sign;
36     }
37     // 浮動小数点命令をシミュレートする FTST (Floating-point Test)。
38     // 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
39     // ステータスワードは比較結果に応じて設定されます。ST > 0.0の場合、C3, C2, C0は
40     // それぞれ 000; ST < 0.0の場合、条件ビットは 001; ST == 0.0の場合、条件ビットは
41     // は100、そうでなければ条件ビットは111となります。
42 void ftst(const temp_real * a)
43 {
44     temp_real b;
45     // まず、ステータスワードのコンディションフラグがクリアされ、比較値b (ST) が
46     // を正規化します。bが0にならず、符号ビットがセットされている（負の数である）場合は
47     // 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
48     // トされる。 clear_Cx()
49     b = *aです。

```

```

39      normalize(&b) です。
40      if (b.a || b.b || b.exponent) {
41          if (b.exponent < 0)
42              set_C0() です。
43      } else
44          set_C3() です。
45  }
46
// 浮動小数点演算命令FCOM (Floating-point Compare) をシミュレートします。
// 2つのパラメータ src1, src2 を比較し、比較結果に応じてコンディションビットを設定する
// 結果。src1 > src2 の場合、C3, C2, C0 はそれぞれ 000 となります。src1 < src2 の場合、条件
// は
// ビットは001、2つが等しい場合は条件ビットは100となります。
47 void fcom(const temp_real* src1, const temp_real* src2)
48 {...    temp_real a;
49
50      a = *src1; a.exponent
51      ^= 0x8000;                                // 符号ビットを反転させます。
52      fadd(&a, src2, &a);                  // 2つを足す(つまり引き算する)。
53      ftst(&a);                          // テスト結果、条件設定
54  }
55
56 // 浮動小数点演算命令FUCOM (無順序比較) をシミュレートします。
// この関数は、オペランドの1つがNaNである場合の比較演算に使用されます。
57 void fucom(const temp_real* src1, const temp_real* src2)
58 {...    fcom(src1, src2)
59
60 }    です。
61

```

8. get_put.c

1. 機能

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。

2. コードアノテーション

プログラム 11-7 linux/kernel/math/get_put.c

```

1 /*
2 * linux/kernel/math/get_put.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * このファイルは、ユーザ・メモリへのすべてのアクセス（取得と保存）を処理します。
9 * ints/reals/BCDなど。を気にする部分はここだけです。

```

10 * 一時的なリアル形式以外のもの。他のすべてのカルは厳密にはtemp_realです。

11 */

// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、および
// 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・オペレーションのための

12 #include <signal.h> (英語)

13

14 #include <linux/math_emu.h>.

15 #include <linux/kernel.h> (日本語)

16 #include <asm/segment.h>

17

// ユーザーメモリ内のショートリアル（単精度実数）を取得します。

// 浮動小数点演算命令コードのアドレッシングモードバイトの内容による。

// と、情報構造体の現在のレジスタの内容、有効なアドレスの

// (ファイルmath/ea.c参照) の短い実数が配置されているところを取得して

// 対応する実数値がユーザーデータエリアから読み込まれます。最後に、ユーザーの短い実

// 数値を一時的な実数に変換して (math/convert.c) 返します。

// Parameters: tmp - 一時的な実体へのポインタ; info - 情報構造体へのポインタ。

// コード - インストラクションコード

18 void get_short_real(*info, unsigned short code)

19 {

20 char * addr;

21 short_real sr;

22

23 addr = ea(info, code); // 有効なアドレスを算出する。

24 sr = get_fs_long((unsigned long *) addr); // ユーザーデータエリアから取得

25 short_to_temp(&sr, tmp); // します。

26 }

27 // tempのリアルフォーマットに変換します。

28

// ユーザーメモリのlong real (倍精度の実数) を取得します。

// この関数はget_short_real()と同じように処理されます。

29 void get_long_real(temp_real * tmp,

30 struct info * info, unsigned short code)

31 {

32 char * addr;

33 long_real lr;

34

35 addr = ea(info, code); // 有効なアドレスを取得 lr.a =

36 get_fs_long((unsigned long *) addr); // 長いリアルを取得。

37 lr.b = get_fs_long(1 + (unsigned long *) addr);

38 long_to_temp(&lr, tmp); // tempのリアルフォーマットに変換する。

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。
// 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。
// コード - インストラクションコード
41 void get_temp_real(temp_real * tmp,
42                     struct info * info, unsigned short code)
43 {
44     char * addr;
45
46     addr = ea(info, code); // 有効なアドレスを得る。
47     tmp->a = get_fs_long((unsigned long *) addr);
48     tmp->b = get_fs_long(1 + (unsigned long *) addr);
49     tmp->exponent = get_fs_word(4 + (unsigned short *) addr);
50 }
51
// ユーザーのメモリにある短い整数を取得します。
// この関数はまず、内容に応じて短い整数の実効アドレスを取得します。
浮動小数点演算命令コードのアドレッシングモードバイトの//の内容と
// 情報構造体の現在のレジスタを読み、対応する整数値を
// ユーザーデータ領域に、一時的な整数形式で保存します。最後に、一時的な
// 整数値を一時的な実数に変換します。
// 一時的な整数も10バイトで表されます。下位8バイトは符号なし整数
// の値を表し、上位2バイトは指数値と符号ビットを表します。もし、最上位の
上位2バイトのうち、// 有効なバイトが1の場合は負の数を意味し、// 有効なバイトが2の場合は負
の数を意味します。
// ビットが0の場合、正の数を意味します。
// 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。
// コード - インストラクションコード
52 void get_short_int(temp_real * tmp, unsigned short code)
53 {
54     char * addr;
55     temp_int ti;
56
57
58     addr = ea(info, code); // 命令コードで有効なものを取得 ti.a = (signed short)
59     get_fs_word((unsigned short *) addr)となります。
60     ti.b = 0;
61     if (ti.sign = (ti.a < 0)) // 負の場合は符号ビットを設定します
62         ti.a = - ti.a;
63     int_to_real(&ti, tmp); // そして、仮数部を非符号化します。
64 } // テンポラリーに変更します。
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。
// この関数は、上記のget_short_int()と同じように処理されます。
66 void get_long_int(temp_real * tmp,
67                   struct info * info, unsigned short code)
68 {
69     char * addr;
70     temp_int ti;
71
72     addr = ea(info, code); // 有効なアドレスを得る。
73     ti.a = get_fs_long((unsigned long *) addr);
74     ti.b = 0;
75     if (ti.sign = (ti.a < 0))
76         ti.a = - ti.a;

```

```

77     int_to_real(&ti, tmp)
78 }   です。
79
// ユーザーメモリ内の64ビット長整数（拡張長整数）を取得します。
// まず、浮動小数点演算命令のアドレッシング・モード・バイトの内容に応じて
// コードと情報構造体の現在のレジスタの内容、実効アドレス
64ビット長整数の // を取得し、対応する整数値を読み出す。
// をユーザーデータ領域から削除し、一時的な整数として保存します。最後に、一時的な
// 整数値を一時的な実数に変換します。
// 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。
80 void get_longlong_int(temp_real * tmp,
81             struct info * info, unsigned short code)
82 {
83     char * addr;
84     temp_int ti;
85
86     addr = ea(info, code) です。          // 有効なアドレスを得る。
87     ti.a = get_fs_long((unsigned long *) addr); // 64ビット長の整数を得る。
88     ti.b = get_fs_long(1 + (unsigned long *) addr);
89     if (ti.sign = (ti.b < 0))           // 負の場合は符号ビットを設定します
90         asm ("notl %0 ; notl %1\n"
91                 "addl $1,%0 ; adcl
92                     $0,%1"
93                 : "=r" (ti.a), "=r"
94                 (ti.b)           // 仮設リアルに変更。
95             );
96     int_to_real(&ti, tmp) です。
// 64ビットの整数（Nなど）を10倍します。
// このマクロは、以下のBCDコードの値を一時的な実数に変換する際に使用します。
// 数字の形式です。その方法は、N<<1 + N<<3 です。
97 #define MUL10(low,high) \(^o^)
98 asm ("addl %0,%0 ; adcl %1,%1\n"
99     "movl %0,%ecx ; movl %1,%ebx\n"
100    "addl %0,%0 ; adcl
101    %1, %1===="
102    "addl %%ecx,%0 ; adcl %%ebx,%1"
103    :「」(低), 「I」(高) :「
104      cx」, 「bx」) 105
// 64ビットの加算。64ビットの<high, low>に32ビットの符号なし数値valを加算します。
106 #define ADD64(val,low,high) □。
107 asm ("addl %4,%0 ; adcl $0,%1": "=r" (low), "=r" (high) &&& >,*)
108 : "低" (Low), "I" (High), "r" ((unsigned long)
(val))の順で表示されます。) 109
// ユーザーのメモリにあるBCDコードの値を取り、一時的に実数フォーマットに変換する。
// この関数は、まず、BCDコードの有効なアドレスを内容に応じて取得します。
浮動小数点演算命令コードのアドレッシング・モード・バイトに含まれる//の内容と
// 情報構造体に登録されているレジスタから、対応する10バイトのBCDコードを
// ユーザーデータ領域（1バイトが符号に使用される）で、一時的に整数に変換される
// の形で表示されます。最後に、一時的な整数値が一時的な実数に変換されます。
// 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。
// コード - インストラクションコード
110 void get_BCD(temp_real * tmp, struct info * info, unsigned short code)

```

```

111 {
112     int kです。
113     char * addr;
114     temp_int i;
115     符号なしの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）。まずBCDコード値の符号ビットを得る
// とし、一時的な整数の符号ビットを設定します。そして、9バイトのBCDコード値を変換します。
// を一時的な整数形式に変換し、最後に一時的な整数値を
// 一時的な実数です。
117     addr = ea(info, code);           // 有効なアドレスを得る。
118     addr += 9;                      // 最後の（10番目の）バイトを指し
119     i.sign = 0x80 & get_fs_byte(addr--);  ます。
120     i.a = i.b = 0;                  // 符号ビットを取得します。
121     for (k = 0; k < 9; k++) {...}    // 一時的な整数に変更します。
122         c = get_fs_byte(addr--);
123         MUL10(i.a, i.b);
124         ADD64((c>>4), i.a, i.b);
125         MUL10(i.a, i.b);
126         ADD64((c&0xf), i.a, i.b)とな
127         ります。
128     }                                // 仮設リアルに変更。
129 }      int_to_real(&i, tmp)です。
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、次に
// 一時実数の結果を短実数形式に変換して、その場所に格納する
// of addr。
// パラメータ: tmp - 一時的なリアルフォーマットの結果値, info - 情報構造体のポインタ。
// コード - インストラクションコード
131 void put_short_real(const temp_real * tmp,
132                     struct info * info, unsigned short code)
133 {
134     char * addr;
135     short_real sr;
136
137     addr = ea(info, code);           // 有効なアドレスを得る。
138     verify_area(addr, 4);          // データ領域を確認します
139     temp_to_short(tmp, &sr);       .
140     put_fs_long(sr, (unsigned long *) addr); // 短い実数に変換します。
141 }                                // addr の位置に格納する。
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。
// この関数は、上記のput_real_real()と同じように処理されます。
143 void put_long_real(const temp_real * tmp,
144                     struct info * info, unsigned short code)
145 {
146     char * addr;
147     long_real lr;
148
149     addr = ea(info, code);           // 有効なアドレスを得る。
150     verify_area(addr, 8);

```

```

151     temp_to_long(tmp, &lr);
152     put_fs_long(lr.a, (unsigned long *) addr);
153     put_fs_long(lr.b, 1 + (unsigned long *) addr)
154 } となります。
155
// 結果を一時的な実数形式でユーザーデータ領域に保存します。
// この関数は、まず結果を保存するためのアドレス addr を取得し、それを確認した後に
// そのアドレスに十分な（10バイトの）ユーチームメモリがある場合は、一時的なリアル
// 番号を addr。
// パラメータ: tmp - 一時的なリアルフォーマットの結果値, info - 情報構造体のポインタ。
// コード - インストラクションコード
156 void put_temp_real(const temp_real * tmp,
157                     struct info * info, unsigned short code)
158 {
159     char * addr;
160
161     addr = ea(info, code); // 有効なアドレスを得る。
162     verify_area(addr, 10); // 十分なユーザーメモリがあることを確認し
163     put_fs_long(tmp->a, (unsigned long *) addr); // ユーザー領域に一時的なリアル
164     を保存 put_fs_long(tmp->b, 1 + (unsigned long *) addr);
165     put_fs_word(tmp->exponent, 4 + (short *) addr);
166 }
167
// 結果を短い整数形式でユーザーデータ領域に保存します。
// この関数は、まず、結果を保存するためのアドレス addr を取得し、それを
// 一時的な実数形式の結果を一時的な整数形式に変換します。もしそれが負の
// 数、整数の符号ビットを設定します。最後に、整数をユーチームメモリに保存します。
// パラメータ: tmp - 一時的なリアルフォーマットの結果値, info - 情報構造体のポインタ。
// コード - インストラクションコード
168 void put_short_int(const temp_real * tmp,
169                     struct info * info, unsigned short code)
170 {
171     char * addr;
172     temp_int ti;
173
174     addr = ea(info, code); // 有効なアドレスを得る。
175     real_to_int(tmp, &ti); // 臨時の整数に変更します。
176     verify_area(addr, 2); // ユーザー領域の確認（2バイト必要
177     if (ti.sign) ) // 結果を否定する。
178         ti.a = -ti.a; // ユーザーデータ領域に格納
179     put_fs_word(ti.a, (short *) addr); // する。
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。
// この関数は、上記のput_short_int()と同じように処理されます。
ボイド put_long_int(const temp_real * tmp,
183                     struct info * info, unsigned short code)
184 {
185     char * addr;
186     temp_int ti;
187
188     addr = ea(info, code) です。 // 有効なアドレスを得る。
189     real_to_int(tmp, &ti); // 臨時の整数に変更します
。
```

```

190     verify_area(addr, 4);                                // ユーザーエリア(4バイト)を確認
191     if (ti.sign)                                         // します。
192         ti.a = -ti.a;                                    // 符号付きの場合は、結果をマイナスにします。
193     put_fs_long(ti.a, (unsigned long *) addr);          // データ領域に保存します。
194 }
195
196 void put_longlong_int(const temp_real * tmp,
197                      struct info * info, unsigned short code)
198 {
199     char * addr;
200     temp_int ti;
201
202     addr = ea(info, code);                             // 有効なアドレスを得る。
203     real_to_int(tmp, &ti);                            // 一時的な整数に変更します。
204     verify_area(addr, 8);                            // 8バイトが利用可能であることを確認します。
205     if (ti.sign)
206         asm ("notl %0 ; notl %1\n\t"
207               "addl $1, %0 ; adcl "
208               "$0, %1"
209               : "=r" (ti.a), "=r"
210               (ti.b))                                     // ユーザーデータ領域に保存します。
211     put_fs_long(ti.b, 1 + (ti.sign ? 1 : 0));           // 符号付きの場合は、負の値に変更します。
212     put_fs_long(ti.a, (unsigned long *) addr);          // 反転して1を加える。
213
214 // 符号なしの数値<high, low>を10で割って、その余りをremに入れる。
215 #define DIV10(low, high, rem) \
216     asm ("divl %6 ; xchgl %1, %2 ; divl %6" ) \
217     =d/ (rem)、 /=a/ (low)、 /=b/ (high) \
218     の順で表示されます。 \
219     : ""(0)、 "I"(高)、 "2"(低)、 "c"(10)
220
221     // 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。
222     // 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
223     // 10バイトのBCDコードを格納するスペース。一時的な実数のフォーマットの結果は
224     // を、BCDコード形式のデータに変換して、ユーザーメモリに保存します。負の場合は
225     // メモリの最上位バイトの最上位ビットが設定されます。
226     // パラメータ: tmp - 一時的なリアルフォーマットの結果値, info - 情報構造体のポインタ。
227     // コード - インストラクションコード
228 void put_BCD(const temp_real * tmp, struct info * info, unsigned short code)
229 {
230     int k, rem;
231     char * addr;
232     temp_int i;
233
234     符号なしのchar
235     c;
236     addr = ea(info, code);                            // 結果を格納するためのアドレスを取得
237     verify_area(addr, 10);                           // メモリスペースを確認します。
238     real_to_int(tmp, &i);                           // 一時的な整数に変更します。

```

```

229     if (i.sign)                                // 符号付きの場合は、上位バイトのMSB
230         put\_fs\_byte(0x80, addr+9) です。    ビットを設定します。
231     その他                                     // それ以外の場合は、MSBビットをリセ
232         put\_fs\_byte(0, addr+9) です。        ットします。
233     for (k = 0; k < 9; k++) {...}           // BCDコードに変更して保存します。
234         DIV10(i.a, i.b, rem);          DIV10(i.a, i.b, rem);
235         c = remです。                      c = rem;
236         DIV10(i.a, i.b, rem);          c += rem<<4;
237         c += rem<<4;                   put\_fs\_byte(c, addr++) です。
238     }
239 }
240 }
241

```

9. mul.c

1. 機能

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。

2. コードアノテーション

プログラム 11-8 linux/kernel/math/mul.c

```

1 /*
2 * linux/kernel/math/mul.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * temporary real multiplier ルーチン。
9 */
10
11 #include <linux/math_emu.h>を追加します。
12
13 // 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。
14 static void shift(int * c)
15 {
16     asm ("movl (%0), %%eax ; addl %%eax, (%0)\n"
17          "movl 4(%0), %%eax ; adc1 %%eax, 4(%0)\n"
18          "movl 8(%0), %%eax ; adc1 %%eax, 8(%0)\n"
19          "movl 12(%0), %%eax ; adc1\n"
20          %%eax, 12(%0)\n"
21          :: "r" ((長)c): "ax") となっています。
22
23 // 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。
24 static void mul64(const temp\_real* a, const temp\_real* b, int * c)
25 {

```

```

24         asm ("movl (%0), %%eax_n¥t"
25             /* ミュール (%1) i.e.
26                 "movl %%eax, (%2)in¥t"
27                 "movl %%edx, 4(%2)in¥t"
28                 "movl 4(%0), %%eax_n¥t"
29             /* ミュール 4(%1)n¥t
30                 "movl %%eax, 8(%2)in¥t"
31                 "movl %%edx, 12(%2)in¥t"
32                 "movl (%0), %%eax_n¥t"
33             /* ミュール 4(%1)n¥t
34                 "addl %%eax, 4(%2)%n¥t"
35                 /*adc1 %%edx, 8(%2)in¥t"
36                 /*adc1 $0, 12(%2)in¥t"
37                 "movl 4(%0), %%eax_n¥t"
38             /* ミュール (%1) i.e.
39                 "addl %%eax, 4(%2)%n¥t"
40                 /*adc1 %%edx, 8(%2)in¥t"
41                 /*adc1 $0, 12(%2) "
42                 :: "b" ((long) a), "c" ((long) b), "d" ((long)
43                 : "ax" ; "dx"
44             );
45 }

// シミュレーション浮動小数点演算命令 FMUL (Floating-point Multiply)。
// 仮設リアル src1 * scr2 -> result.
46 void fmul(const temp_real* src1, const temp_real* src2, temp_real* result)
47 {...
48     int i, sign;
49     int tmp[4] = {0, 0, 0, 0};
50
// まず2つの数字の掛け算の符号を決める。符号が等しいのは
// 符号ビット 両者のXOR。次に、掛けられた指数の値を計算します。指数値には
// を乗算時に加算します。しかし、指数は偏った数値形式で格納されているので
// 2つの数字の指数を足すと偏った量が2回加算されるので、それが
// sign = (src1->exponent ^ src2->exponent) & 0x8000;
51     i = (src1->exponent & 0x7fff) + (src2->exponent & 0x7fff) - 16383 + 1;
52
// 結果の指数が負になれば、2つの数値が掛けられたことになります。
// でアンダーフローが発生します。そこで、符号付きのゼロの値を直接返します。
// 結果の指数が0x7fffよりも大きい場合は、オーバーフローが発生したことを示すので
// ステータスワードのオーバーフロー例外フラグを設定して返す if
53     (i<0) {
54         result->exponent = sign;
55         result->a = result->b = 0;
56         return;
57     }
58     if (i>0x7fff) {
59         set_OE();
60         return;
61     }
// 2つの数値の仮数を掛け合わせ、その結果が0でない場合、結果の仮数は
// を正規化します。つまり、結果として得られる仮数の値は左にシフトされ、最大の
// の有効ビットが1となり、それに応じて指数が調整されます。もし、16バイトの

```

```

// 2つの数字の仮数を掛け合わせて得られる結果は0で、指数も
// 最後に、乗算結果が一時的な実数「result」に保存されます. mul64(src1, src2, tmp);
62     if (tmp[0] || tmp[1] || tmp[2] || tmp[3])
63         while (i && tmp[3] >= 0) {
64             i--;
65             shift(tmp) です。
66         }
67     その他
68     i = 0;
69     result->exponent = i | sign;
70     result->a = tmp[2];
71     result->b = tmp[3];
72
73 }
74

```

10. div.c

1. 機能

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。

2. コードアナリティクス

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
7 /*
8 * 一時的な実数分割ルーチン。
9 */
10
11 #include <linux/math_emu.h> を追加します。
12
13 // 4バイトの内容を1ビットだけ左にシフト（2倍）します。
14 static void shift_left(int * c)
15 {
16     asm volatile ("movl (%0), %%eax ; addl %%eax, (%0) \r" "movl
17                 4(%0), %%eax ; adc1 %%eax, 4(%0) \r" "movl 4(%0), %%eax
18                 ; adc1 %%eax, 4(%0) \r"
19                 "movl 8(%0), %%eax ; adc1 %%eax, 8(%0) \r"
20                 "movl 12(%0), %%eax ; adc1 %%eax, 12(%0) "
21             :: "r" ((長)c): "ax") となっています。
22
23 // ポインタcが指す4バイトの内容を1ビットだけ右にシフトする。

```

```

22 static void shift_right(int * c)
23 {
24     asm ("shrl $1, 12(%0) ; rcr1 $1, 8(%0) ; rcr1 $1, 4(%0) ; rcr1 $1, (%0)"
25         :: "r" ((長)c)のようになります。)
26 }
27
// 16バイトの減算関数。
// 最後に、借用があるかどうかによってokが設定されます。
// フラグ(CF=1)を設定します。借り入れがない場合(CF=0)はOK=1、そうでない場合はOK=0となります。
28 static int try_sub(int * a, int * b)
29 {
30     char ok;
31
32     asm volatile ("movl (%1), %%eax ; subl %%eax, (%2)in¥t"
33                 "movl 4(%1), %%eax ; sbb1 %%eax, 4(%2)in¥t"
34                 "movl 8(%1), %%eax ; sbb1 %%eax, 8(%2)in¥t"
35                 "movl 12(%1), %%eax ; sbb1 %%eax, 12(%2)in"
36                 "facility"
37                 "setae %%al": "=a" (ok): "c" ((long) a), "d" ((long)
38 }           b); return ok;
39
// 16バイトの分割機能。
// パラメータ a / b -> c. 減算を使ってマルチバイトの分割をシミュレートする方法です。
40 static void div64(int * a, int * b, int * c)
41 {
42     int tmp[4];
43     int i;
44     符号付き整数マスク =
45     0;
46     c += 4;
47     for (i = 0 ; i<64 ; i++) {...}
48     if (!(mask >>= 1)) {
49         c--;
50         mask = 0x80000000;
51     }
52     tmp[0] = a[0]; tmp[1] = a[1];
53     tmp[2] = a[2]; tmp[3] = a[3];
54     if (try_sub(b, tmp)) {
55         *c |= マスク。
56         a[0] = tmp[0]; a[1] = tmp[1]
57         .
58         a[2] = tmp[2]; a[3] = tmp[3];
59     }
60     shift_right(b);
61 }
62
// 浮動小数点命令をシミュレートする FDIV.
// 一時的な実数分割: src1 / src2 -> 結果。
63 void fdiv(const temp_real* src1, const temp_real* src2, temp_real* result)
64 {
65     int i, sign;
66     int a[4], b[4], tmp[4] = {0, 0, 0, 0};
67

```

```

68     sign = (src1->exponent ^ src2->exponent) & 0x8000;
69     if (!(src2->a || src2->b)) {...  

70         set_ZE();  

71         return;  

72     }  

73     i = (src1->exponent & 0x7fff) - (src2->exponent  

74     & 0x7fff) + 16383;  

75     if (i<0) {  

76         set_UE()を使用しています。  

77         result->exponent = signとなります。  

78         result->a = result->b = 0となります。  

79         を返すことができます。  

80     }  

81     a[0] = a[1] = 0となります。  

82     a[2] = src1->a;  

83     a[3] = src1->b;  

84     b[0] = b[1] = 0となります。  

85     b[2] = src2->a;  

86     b[3] = src2->b;  

87     while (b[3] >= 0) {...  

88         i++;  

89         shift_left(b)です。  

90     }  

91     div64(a, b, tmp)。  

92     if (tmp[0] || tmp[1] || tmp[2] || tmp[3]) {.  

93         while (i && tmp[3] >= 0) {.  

94             i--;  

95             shift_left(tmp)。  

96         }  

97         if (tmp[3] >= 0)  

98             set_DE()です。  

99     } else  

100        i = 0;  

101        if (i>0x7fff) {。  

102            set_OE()です。  

103            を返すことができます。  

104        }  

105        if (tmp[0] || tmp[1])  

106            set_PE()を使用しています。  

107        result->exponent = i | sign;  

108        result->a = tmp[2];  

109        result->b = tmp[3];  

110    }

```

11.11 まとめ

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

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 ファイルシステム(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	サ	最終更新時刻 (GMT)	降臨
Makefile	176 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	
ファイル 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	
パイプ.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 バイト	1991-12-09 21:11:34	
truncate.c	1692 バイト	1992-01-11 19:47:28	

12.1 主な機能

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。その際に生成される一時ファイルの中には

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

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

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ファイルシステム

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・デバイスの構造、その配布を図12-1に示す。

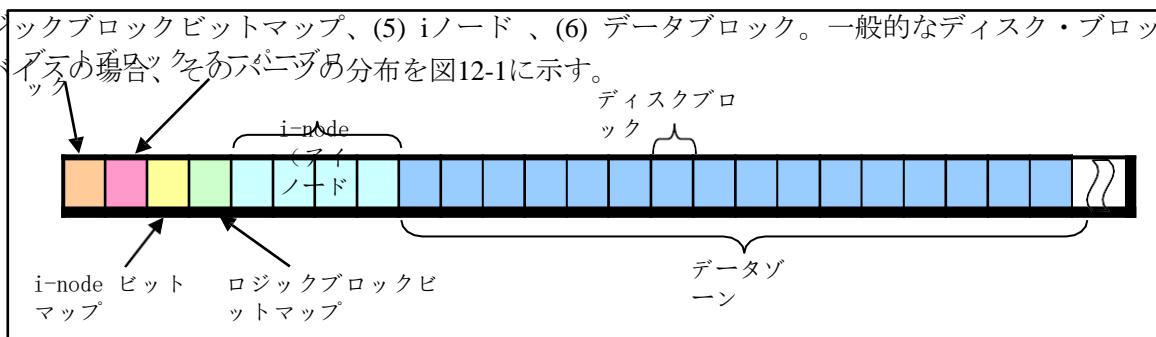


図12-1 MINIXファイルシステムの各部のレイアウトの模式図

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できます。

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。⁶⁹⁰ ただし、どのディスクブロックのデバイスも

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

大容量のハードディスクブロック装置では、通常、図12-2に示すように、その上にいくつかのパーティションを分割し、各パーティションに異なる完全なファイルシステムを格納することができます。図では、4つのパーティションがあり、FAT32ファイルシステム、NTFSファイルシステム、MINIXファイルシステム、EXT2ファイルシステムが格納されています。ハードディスクの第1セクターはマスターboot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と終了位置パラメータ、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示します。



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、スーパープロックテーブル（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。スーパープロックテーブルは、`super.c`プログラムの`mount_root()`関数で初期化されます。スーパープロックテーブルは、`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プロックがテーブルに設定され、`put_super()`関数でスーパープロックが解放されます。

フィールド名	データタイプ	説明
s_ninodes	ショート	i-nodeの数
s_nzones	ショート	ゾーンの数
s_imap_blocks	ショート	i-nodeのビットマップのブロック数
s_zmap_blocks	ショート	ロジックビットマップのブロック数
s_firstdatazone	ショート	データゾーンにあるFirst Logic Block No. です。
s_log_zone_size	ショート	$\log_2(\text{DiskBlock}/\text{LogicBlock})$
S_MAX_SIZE	ロング	最大ファイルサイズ
s_magic	ショート	FSマジックナンバー (0x137f)
s_imap[8].	buffer_head *.	キャッシュ内のi-nodeビットマップブロック配列。
s_zmap[8].	buffer_head *.	ロジックブロックのビットマップブロック配列でのキャッシュを使用しています。
s_dev	ショート	スーパーblockのデバイス番号
s_isup	m_inode *	インストールされているfsのi-node。
s_imount	m_inode *	fsがインストールしているi-node。
s_time	ロング	修正時間
s_wait	task_struct *	このスーパーblockで待っているタスク
s_lock	チャー	ロックされたフラグ。
s_rd_only	チャー	Readonlyフラグ。
s_dirt	チャー	変更フラグ (ダーティフラグ)

図12-3 MINIXファイルシステムのスーパーblock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blockの構造から、論理ブロックのビットマップは最大8個のバッファブロック(s_zmap[8])を使用し、各ブロックサイズは1024バイトで、各ビットは1つのディスクブロックの占有状態を表していることがわかります。したがって、1つのバッファブロックは8192個のディスクブロックを表すことができ、8つのバッファブロックは合計65,536個のディスクブロックを表すことができるので、MINIXファイルシステム1.0がサポートするブロックデバイスの最大容量 (長さ) は64MBとなります。

i-nodeは、ディスクデバイス上の各ファイルやディレクトリ名のインデックス情報を格納するために使用される。i-nodeのビットマップは、i-nodeが使用されているかどうかを示すために使用され、各ビットはi-nodeを表します。1Kブロックの場合、1つのディスクブロックは8192個のi-nodeの使用を表すことができる。論理ブロックのビットマップの場合と同様に、アイドル状態のi-nodeを探す関数は、全てのi-nodeが使用されている場合は0の値を返すので、最下位ビット (ビット0) のは、i-nodeビットマップの1バイト目と対応するi-node 0はアイドル状態になっており、i-node 0の対応するビットマップのビット位置は、ファイルシステムの作成時にあらかじめ1に設定されています。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。

フィールド名	データタイプ	説明
i_mode	ショート	ファイルタイプと属性 (rwxビット)
i_uid	ショート	ファイルオーナーのユーザーID。
i_size	ロング	ファイルサイズ (バイト)
i_mtime	ロング	修正時間 (1970.1.1:0, secより)
i_gid	チャー	ファイルグループID
i_nlinks	チャー	リンク数 (ディレクトリエントリの数 はこのi-nodeを指しています)。
i_zone[9].	ショート	ファイルが占有する論理ブロック番号 ゾーン[0]-ゾーン[6] ダイレクトブロックNo. ゾーン[7] 間接ブロック番号 ゾ ーン[8] 2 nd 間接ブロック番号 注: デバイスファイルの場合、zone[0]は指 定されたデバイスファイルのデバイスNo.
i_wait	task_struct *	タスクがi-nodeを待っている。
i_atime	ロング	ラストアクセス時間
i_ctime	ロング	i-nodeの修正時間。
i_dev	ショート	i-nodeが属するデバイスNo.
i_num	ショート	i-node No.
i_count	ショート	i-nodeの参照カウント。0 - アイドル。
i_lock	チャー	i-nodeはロックされています。
i_dirt	チャー	i-nodeが変更されました (ダーティフラグ)。
i_pipe	チャー	i-nodeはパイプに使用されます。
i_mount	チャー	i-nodeは他のfsと一緒にインストールされまし た。
i_seek	チャー	検索フラグ (lseek用)。
i_update	チャー	i-nodeの更新フラグ。

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。

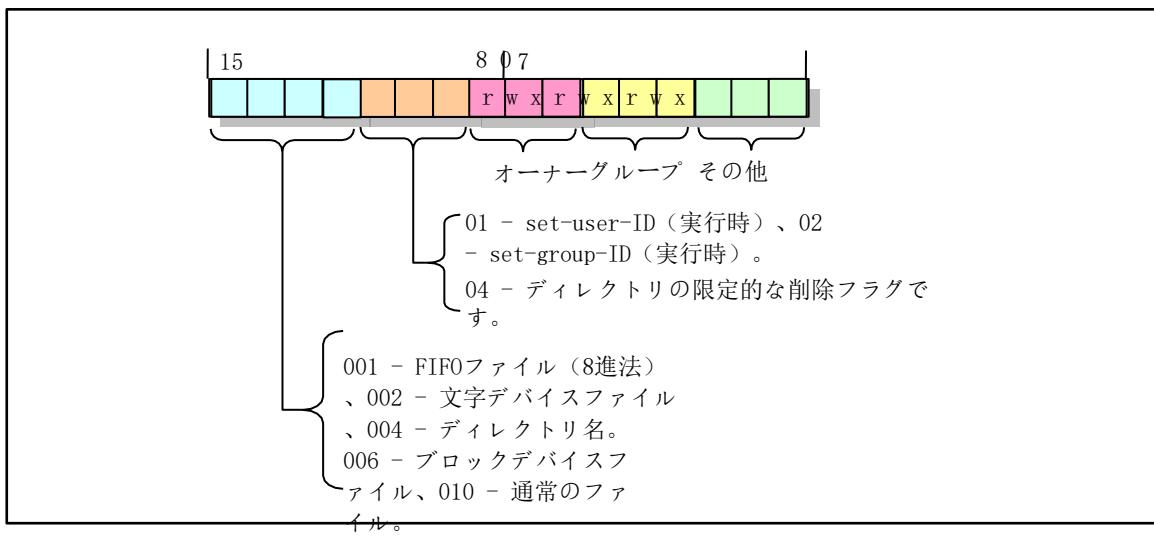


図12-5 i-nodeモードフィールドの内容

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 $(7 + 512 + 512 * 512) = 262,663\text{KB}$ 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

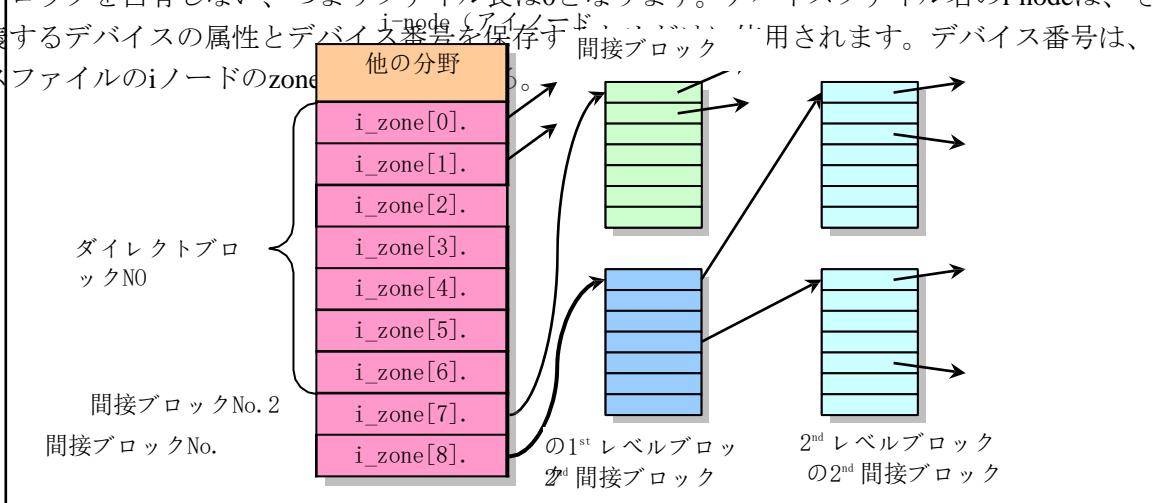


図12-6 i-nodeの論理ブロック配列の機能

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の構造はすべて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ます。

PCの場合、一般的には1セクタ(512バイト)の長さがブロックデバイスのデータブロック長として使われます。

MINIXファイルシステムでは、連続する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です。ディスクの最初のディスクブロックから数えていきます。つまり、ポートブロックは0番目のディスクブロックです。

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。

2. ファイルの種類、属性、ディレクトリの項目

1. ファイルの種類と属性

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ます。

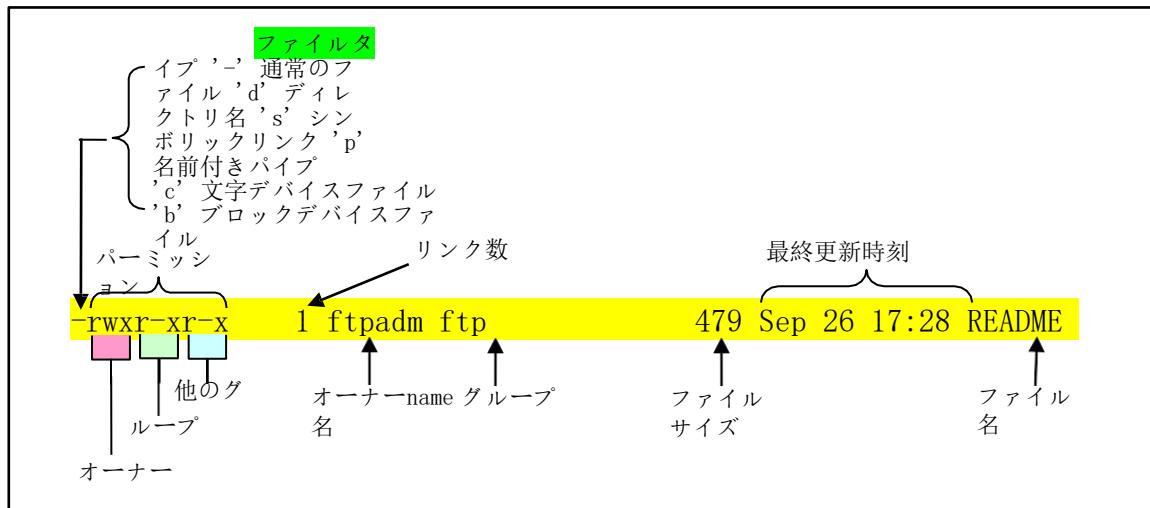


図12-7 コマンド「ls -l」で表示されるファイル情報

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、上位4ビットで表現される。ファイルシステムを操作する際、システムは以下のようないくつかのマクロを使用しています。⁶⁹⁵

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

図では、ファイルタイプ文字の後に、1セット3文字で構成されたファイル許可属性が続き、ファイルホスト、同一グループのユーザー、他のユーザーのファイルに対するアクセス権を示すために使用されている。`rwx`はそれぞれ、ファイルの読み取り、書き込み、実行の許可を示す。ディレクトリ名のファイルの場合、実行可能とは、そのディレクトリに入ることができる意味する。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／実行ができ、同じグループのユーザーなどがファイルの読み取り／実行ができることを示す。Linuxでは

0.12のソースコードでは、ファイルのパーミッション情報はi-nodeの*i_mode*フィールドにも格納されており、フィールドの下位9ビットは3組のパーミッションを表すのに使用され、しばしば変数*mode*で表されます。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を示し、`'Group'`

`name`は、そのユーザーが所属するグループ名です。

12.1.2.2 ファイルシステムディレクトリの項目構成

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、`fs`例えば`fs`で定義される名`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、`#define NAMELEN`ファイル`root/`のデータブロックの最後に格納されます。ファイルシステムのルート`#define ROOT_INO`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。

構造 `struct dir_entry {`

```
    unsigned short inode;           // i-nodeの番号。
    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、図12-8に示すように、与えられたファイル名に応じてそのi-node番号を見つけ、対応するi-node情報によってファイルが置かれているディスクブロックの位置を見つける。例えば、ファイル名`/usr/bin/vi`のi-node番号を見つけるために、ファイルシステムはまず固定のi-node番号(1)を持つルート・ディレクトリから開始する、つまり、i-node番号1のデータ・ブロックから名前を見つけるのである。`usr`のディレクトリエントリは、このようにファイル`/usr`のiノード番号を取得します。このiノード番号により、ファイルシステムはスムーズにディレクトリ「`/usr`」を得ることができ、その中にファイル名「`bin`」のディレクトリエントリを見つけることができる。このようにして、`/usr/bin`のi-node番号もわかるので、ディレクトリ`/usr/bin`のディレクトリの位置がわかり、その中の`vi`ファイルのディレクトリエントリを見つけることができます。最後に、ファイルパス名「`/usr/bin/vi`」のi-node番号が分かる⁶⁹⁶ので、i-node構造の情報を取得するために

ディスクを使
用しま
す。

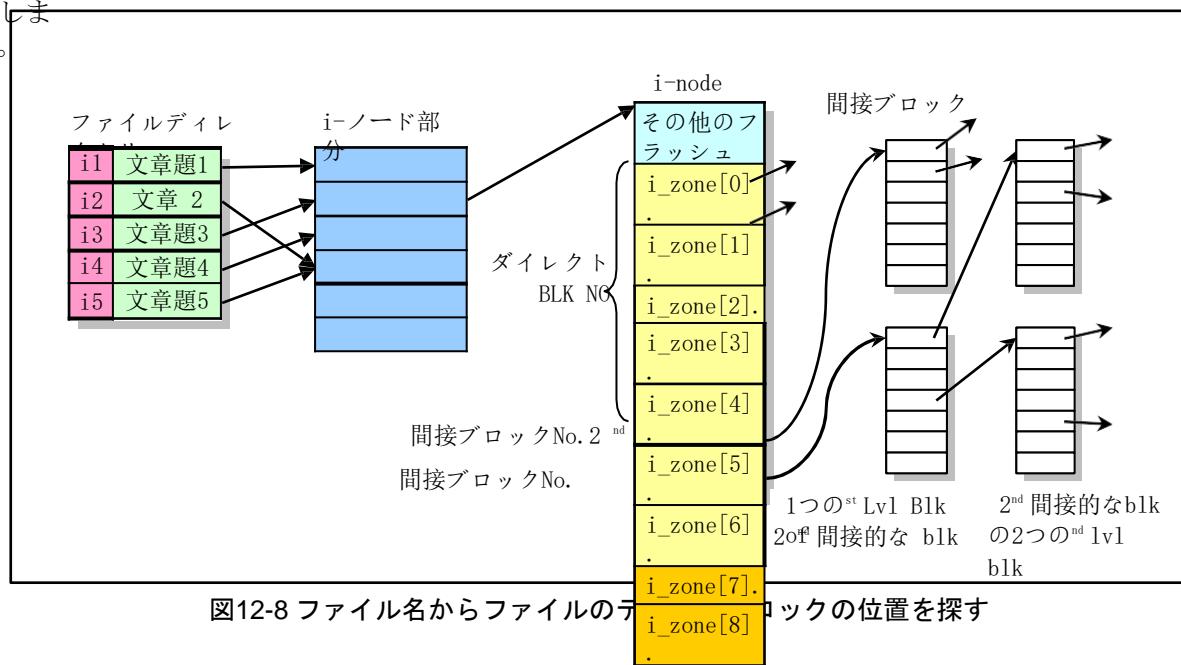


図12-8 ファイル名からファイルのデータブロックの位置を探す

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）。

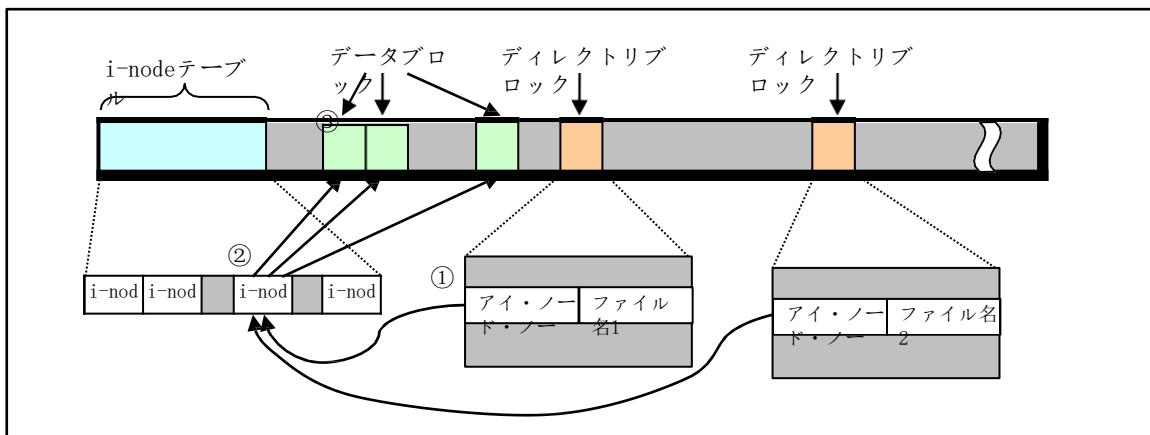


図12-9 ファイル名からそのデータブロックを取得する

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*nlinks*があり、そのi-nodeを指す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・エントリを使って、他のファイル・システムのi-nodeを指すことはできない、ということになる。

は、ハードリンクがファイルシステムを越えられません。

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

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ます。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うことができる。

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「...」サブディレク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、親ディレクトリのi-node番号56とサブディレクトリのi-node番号123の間のリンクは図12-10

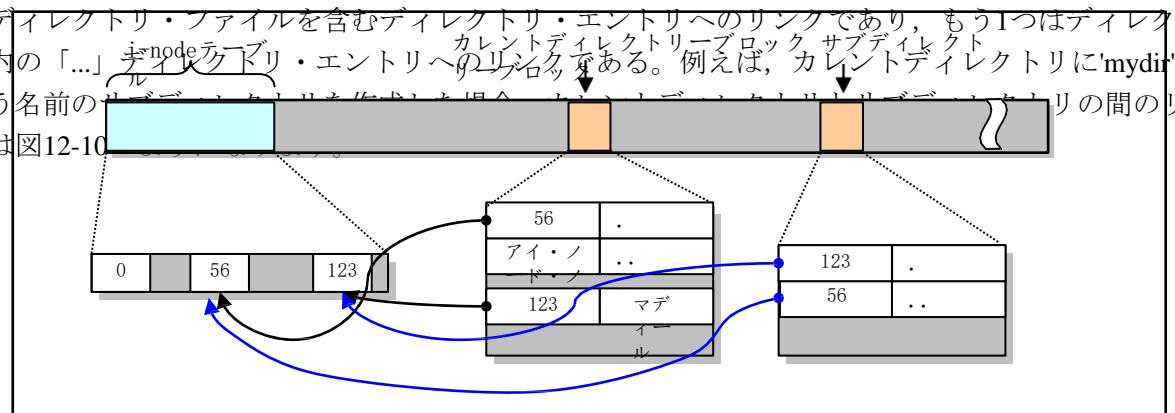


図12-10 ディレクトリファイルのエントリとサブディレクトリのリンク

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 ディレクトリ構造の例

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

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

ドルブルク	10 ルート	ルート	176 3月21日 2004 .
スルクス			
ドルブルク	10 ルート	4096	176 3月21日 2004 ..
スルクス			
ドルブルク	2 ルート	4096	912 3月21日 2004年のピン
スルクス			
ドルブルク	2 ルート	ルート	336 3月21日 2004年デビュー
スルクス			
ドルブルク	2 ルート	ルート	224 3月21日 2004年など
スルクス			
ドルブルク	8 ルート	ルート	128 3月21日 2004年のイメージ
スルクス			
ドルブルク	2 ルート	ルート	32 3月21日 2004年mnt
スルクス			
ドルブルク	2 ルート	ルート	64 3月21日 2004 tmp
スルクス			
ドルブルク	10 ルート	ルート	192 3月29日 2004年 米国
スルクス			
ドルブルク	2 ルート	ルート	32 3月21日 2004年秋
スルクス			

[/]# hexdump .

00000000 0001 002e 0000 0000 0000 0000 0000 0000	// .
00000010 0001 2e2e 0000 0000 0000 0000 0000 0000	// ..
00000020 0002 6962 006e 0000 0000 0000 0000 0000	// ピン
00000030 0003 6564 0076 0000 0000 0000 0000 0000	// デブ
00000040 0004 7465 0063 0000 0000 0000 0000 0000	// その他

hexdump. 以下を実行する i-node のノードをロックするすべてのディレクトリ・エントリがリストされます。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 番号で、次の4バイトが i-node の名前である。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・エントリが使われていないか、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番であることは、システムのノードディレクトリ構造の特殊性である。他のサブディレクトリ構造とは異なるものである。

etc のディレクトリでも 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。

ドルブルク	-2 ルート	ルート	224 3月21日 2004 .
etc	-fa		
ドルブルク	10 ルート	ルート	176 3月21日 2004 ..
スルクス			
-rw-r--r--	1 ルート	ルート	137 3月4日 2004年グループ。
.			
-rw-r--r--	1 ルート	ルート	11801 3月4日 2004年のマジック
.			
-rw-r--r--	1 ルート	ルート	11 1月22 18:12 mtab
.			
-rw-r--r--	1 ルート	ルート	142 3月5日 2004 mtools
.			
-rw-r--r--	1 ルート	ルート	266 3月4日 2004 passwd
[/]# hexdump etc			
00000000 002e 0000 0000 0000 0000 3月4日 2004年のプロ			
.			
-rw-r--r--	1 ルート	ルート	57 3月4日 2004年RC 699
.			
-rw-r--r--	1 ルート	ルート	1034 3月4日 2004年ターム キャップ。
.			
-rwx--x--x	1 ルート	ルート	10137 1月15 日 1992年更新

```

0000010 0001 2e2e 0000 0000 0000 0000 0000 // ..
0000020 0007 6372 0000 0000 0000 0000 0000 // rc
0000030 000b 7075 6164 6574 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 // not used.
0000070 007C 616D 6967 0063 0000 0000 0000 // マジック
0000080 0016 7270 666f 6c69 0065 0000 0000 // プロフィール
0000090 007e 6170 7373 6477 0000 0000 0000 // passwd
00000a0 0081 7267 756f 0070 0000 0000 0000 // グループ
00000B0 01EE 746D 6f6f 736c 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。

3. 高速バッファ（バッファキャッシング）

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（バッファキャッシング）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、バッファキャッシングは、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。バッファキャッシングには、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ングを検索します。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ングにない場合は、ブロックデバイスのリードコマンドが発行され、データがキャッシングに読み込まれます。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ング内の空きバッファブロックを要求し、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。

4. ファイルシステムの下位機能

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。

- bitmap.cプログラムには、i-nodeビットマップとロジックブロックビットマップのリリースと占有のハンドラが含まれています。i-nodeビットマップを操作する関数はfree_inode()とnew_inode()であり、ロジックブロックビットマップを操作する関数はfree_block()とnew_block()です。
- truncate.cプログラムには、データファイルの長さを0に切り詰める関数truncate()が含まれています。i-nodeで指定されたデバイスのファイル長を0に切り、ファイルデータが占有していたデバイスの論理ブロックを解放します。

- inode.cプログラムには、i-node関数を割り当てる関数`iput()`と、メモリのi-nodeを戻す関数`iput()`、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る関数`bmap()`が含まれています。
- namei.cプログラムには、主に関数`namei()`が含まれています。この関数は、与えられたファイルのパス名を、`iget()`、`iput()`、`bmap()`を使って、そのi-nodeにマッピングします。
- super.cプログラムは、ファイルシステムのスーパー・ブロックを扱うように設計されており、関数`get_super()`、`put_super()`、`free_super()`が含まれています。また、いくつかのファイルシステムのロード/アンロードハンドラーや、`sys_mount()`などのシステムコールも含まれています。

これらのファイルの機能間の階層関係を図12-11に示します。



図12-11 ファイルシステムの低レベル機能階層

12.1.5 ファイルデータアクセス操作

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。

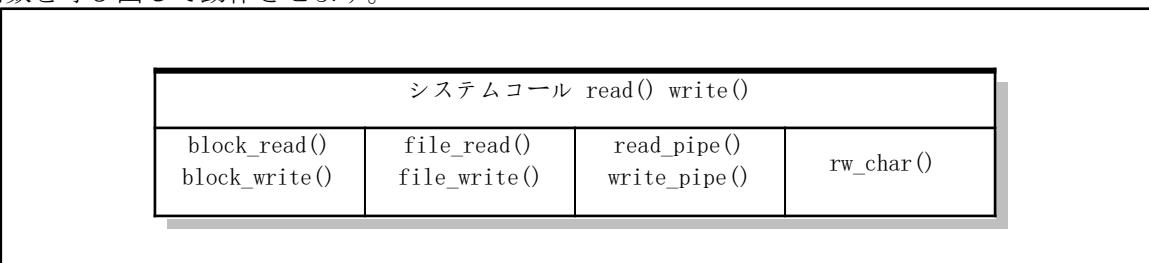


図12-12 ファイルデータアクセス機能

`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。

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

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

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

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

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

~~また、カーネルは、ファイル構造、ファイルテーブルfile_table[]、メモリ内i-nodeテーブルinode_table[]を用いてファイル操作アクセスを管理します。これらのデータ構造やテーブルの定義は、struct file inode include/linux/fs.hに記載されています。アビゲイル構造は以下のよう~~に定義されています。
off_t f_pos; // ファイルハンドラー数
}; // メモリ内のi-node (v-node) を指します。
struct file file_table[NR_FILE]. // 現在のリード/ライトポジション。
// ファイルテーブルの配列、合計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、次のようなものがある。

// open()とfcntl()で使用されるファイルアクセスモード。同時に使用できるのは3つのうち1つだけです。	8 #define O_RDONLY 00 // オープンフラグをリードオンリーモードで表示します。
	9 #define O_WRONLY 01 // オープンフラグを書き込み専用モードにします。
	10 #define O_RDWR 02 // オープンフラグリード/ライトモードです。
// 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。	11 #define O_CREAT 00100 // 存在しない場合は作成します (fcntlでは使用しません)。
	12 #define O_EXCL 00200 // ファイルフラグを使用して除外することができます。
	13 #define O_NOCTTY 00400 // no control tty.
	14 #define O_TRUNC 01000 // ファイルが存在し、書き込みモードであれば、0にファイル構造体のファイル参照回数フィールドf_lnk_countます。ファイルが何回
	15 #define O_APPEND 02000 // アpendモードで開くと、EOFを指します。
	16 #define O_NONBLOCK 04000 // はノンブロックモードで開きます。
	17 #define O_NDELAY 0 // NDELAY

メモリ i-node 構造体フィールド `f_inode` は、このファイルの対応する i-node テーブル内のメモリ i-node 構造体アイテムを指している。ファイルテーブルは、カーネル内のファイル構造項目の配列である。

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

カーネル内の i-node テーブル `inode_table[NR_INODE]` は、メモリ i-node 構造体の配列であり、`NR_INODE=32` であるため、現時点では、カーネルは同時に 32 個のメモリ i-node 情報しか格納できません。プロセスがオープンしたファイルとカーネルのファイルテーブルの関係と、それに対応するメモリ

i-node は図 12-13 のように表すことができます。図中の 1 つ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 1 つ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）。

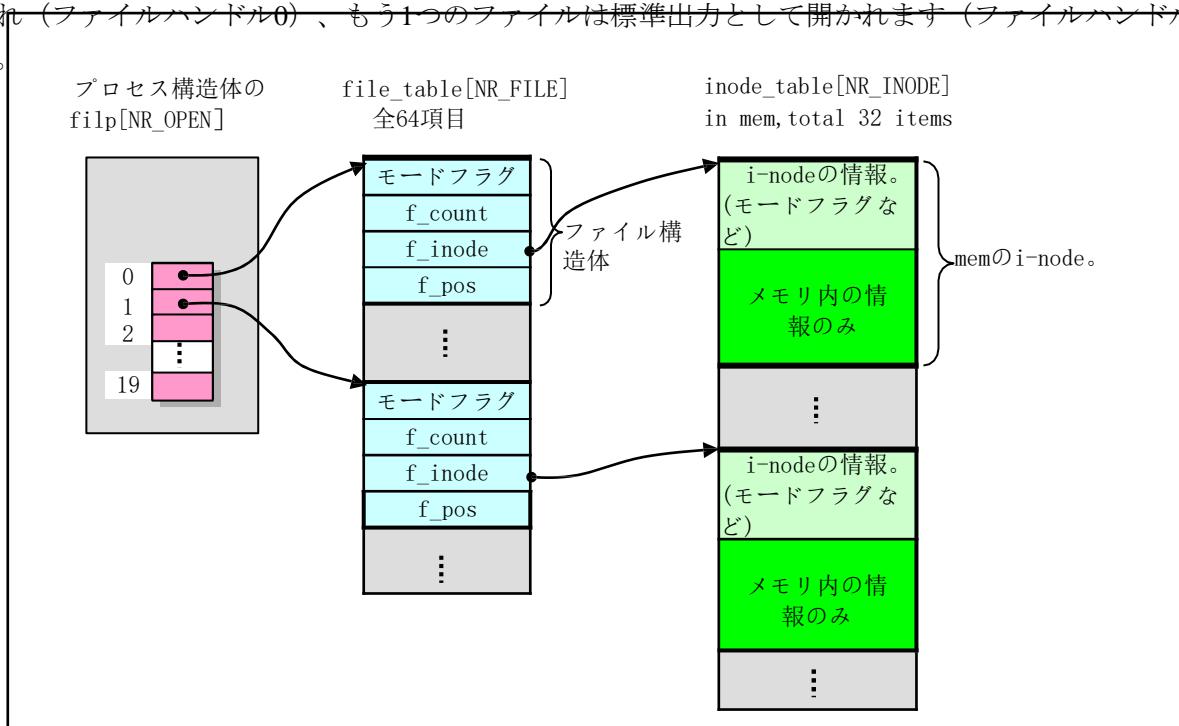


図 12-13 ファイルを開くためにプロセスが使用する構造体

12.1.6 ファイルおよびディレクトリ管理システム・コール

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。

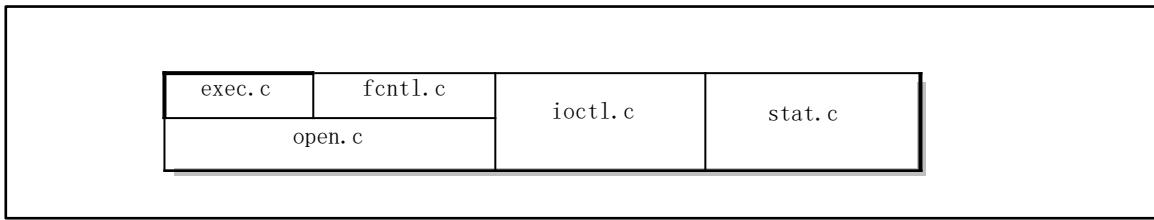


図12-14 ファイルシステム上位層の動作プログラム

`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)の関数番号のCハンドラです。

`NR_execve()`で、`exec()`関数群の主要な実装関数です。

`Fcntl.c`は、ファイル制御システムコールである`fcntl()`と、2つのファイルハンドル複製システムコールである`dup()`と`dup2()`を実装しています。`dup2()`は新しいハンドルの値を指定する必要がありますが、`dup()`は現在の値が最も小さい未使用的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 360KBフロッピーディスクのファイルシステムの解析

図12-1に示したファイルシステム構造の理解を深めるために、Linuxの0.12システムでは、360KBのフロッピーイメージにMINIX 1.0のファイルシステムを作成し、`hello.c`というファイルのみを保存していました。

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

```

[~/usr/root]# mkfs /dev/fd1 360          // 2nd fdに360KBのファイルシステムを作成する
120 inodes
360ブロック
Firstdatazone=8 (8)
Zonesize=1024
Maxsize=268966912

[~/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           // 3つのディレクトリエントリがあります。
[~/usr/root]# ll -a /mnt
トータル3
drwxr-xr-x 2 root root   48 Feb 23 17:48 .
ドルブルク 10 ルート ルート   176 2004年3月21日 ...
スルクス
-rw----- 1 ルート ルート   74 Feb 23 17:48 hello.c

[~/usr/root]# umount /dev/fd1           // ファイルシステムのアンマウントを行います。

```

```
[/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
ライプ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。bochsシステムを終了して見ることもできます
notepad++などのバイナリを変更できる編集プログラムを使用した場合。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）。

```
0000000 44eb 4d90 6f74 6c6f 2073 0020 0102 0001 // 0x0000～0x03ff(1KB)がブートブロックです。
0000010 e092 4000 F00B 0009 0012 0002 0000 0000
[usr/root]# hexdump /dev/fd1 more
0000020 0000 0000 0000 0000 0000 0000 0000 0000
*
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 ff00 // 0x0800～0xbfff(1KB)はi-nodeのビットマップ
                                                // です。
0000810 ffff ffff ffff ffff ffff ffff ffff ffff
*
0000c00 0007 0000 0000 0000 0000 0000 0000 0000 // この範囲のデータは1です。
0000c10 0000 0000 0000 0000 0000 0000 0000 0000 // 0xc00～0xffff(1KB),論理的なblkのビットマ
                                                // ップ。
0000c20 0000 0000 0000 0000 0000 fffe ffff
0000c30 ffff ffff ffff ffff ffff ffff ffff ffff
*
0001000 41ED 0000 0030 0000 c200 421c 0200 0008 // 0x1000～0x1fff(4KB)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
*
0002000 0001 002e 0000 0000 0000 0000 0000 0000 // 0x2000～0x23ff(1KB)、ルートi-nodeのデータ
0002010 0001 2e2e 0000 0000 0000 0000 0000 0000
0002020 0002 6568 6C6C 2E6F 0063 0000 0000 0000
0002030 0000 0000 0000 0000 0000 0000 0000 0000
*
0002400 6923 636e 756c 6564 3c20 7473 6964 2e6f // 0x2400～0x27ff(1KB)、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と等しい。

	コンテンツや値
s_ninodes	i-nodeの数 0x0078 (10進数で120)
s_nzones	ディスクにおける論理ブロック数 0x0008 (10進数で8)
s_imap_blocks	i-nodeのビットマップで占められているブロック数 0x0001
s_zmap_blocks	ゾーンビットマップのブロック数 0x0001
s_firstdatazone	最初のデータブロック番号 0x0008
s_log_zone_size	Log2(ブロック/ゾーン) 0x0000
S_MAX_SIZE	最大ファイルサイズ 0x10081c00 = 268966912 バイト
s_magic	FSマジックナンバー 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。ディスクの容量が少ないので360KBの場合、ファイルシステムが実際に使用するのは360ビットであり、 $360/8=45$ バイトとなる。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個です。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、使用されている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、残りの未使用ビット0を加えて、ビットマップは合計353ビットを必要とします。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blockは $nr + 8 - 1$ 、すなわち $block = nr + s_{\text{firstdatazone}} - 1$ となります。

i-nodeビットマップと同様に、論理ブロックビットマップの第1バイトの最初の3ビットは既に占有されている。 第1ビット（ビット0）は予約されており、第2ビットと第3ビットはディスクデータゾーンで2つのディスクブロック（論理ブロック）が使用されていることを示している。 実際には、ビット1で表されるディスク・データ・ゾーンの最初のディスク・ブロックは、ルートのi-nodeがデータ情報（ディレクトリ・エントリ）を保存するために使用され、ビット2で表される2番目のディスク・ブロックは、ナンバー2のi-nodeが関連するデータ情報を保存するために使用される。 なお、ここでのデータ情報とは、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は、i-nodeの構造情報を格納するために使用される。 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）、 $120 \times 32 = 3840$ バイト、つまり4つのディスクブロックが必要となる。 上記のデータから、最初の32バイトはルートi-nodeの内容を保持し、それ以降の32バイトはノード2の内容を保持していることがわかります。

表12-2および表12-3に示すとおりです。

フィールド名	説明	値
i-mode	ファイルモード（表12-2 No.1ルートのi-node構造の内容）	0x100 (rw-rw-rwx)
i_uid	ファイルのオーナーID	0x0000
i_size	ファイルサイズ	0x00000030 (48バイト)
i_mtime	修正時間	0x421cc200 (2月23日 17:48)
i_gid	ファイルグループID	0x00
i_nlinks	リンク数	0x02
i_zone[9].	論理ブロック番号の配列	zone[0] = 0x0008、残りの項目はすべて0です。

表12-3 No.2 i-nodeの構造の内容

フィールド名	説明	値
i-mode	ファイルのモード	0x8180 (-rw-----)
i_uid	ファイルのオーナーID	0x0000
i_size	ファイルサイズ	0x0000004a (74バイト)
i_mtime	修正時間	0x421cc200 (2月23日 17:48)
i_gid	ファイルグループID	0x00
i_nlinks	リンク数	0x01
i_zone[9].	論理ブロック番号のアラエイ	zone[0] = 0x0009、残りの項目はすべて0です。

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

いいえ 。	I-node No.	ファイル名
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 機能

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glblk()とブロック読み出し関数bread()について詳しく説明します。最後に、システムの観点から、バッファキャッシュのアクセス処理と同期動作の方法を説明します。

1. バッファキャッシュの物理的位置

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。

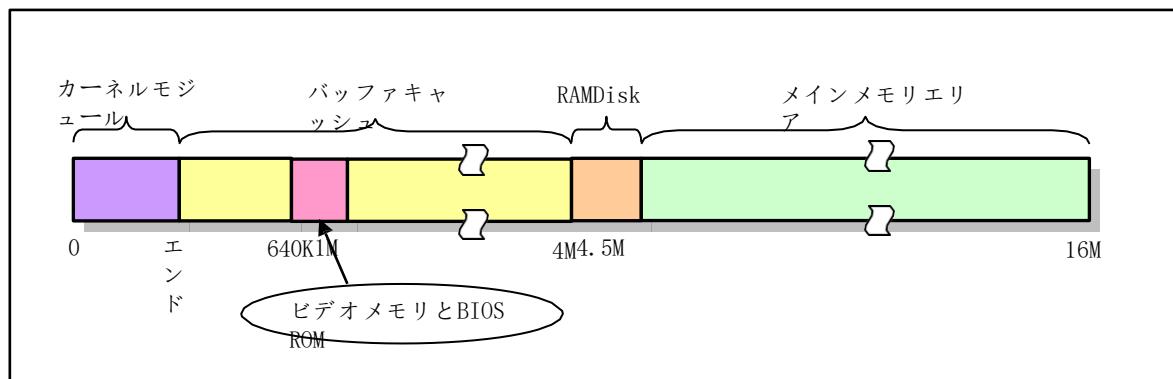


図12-15 物理メモリ内バッファキャッシュの位置

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クId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。リンク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。さらにリンクは、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.バッファキャッシュの初期化

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。バッファの高端は、1024バイトのバッファブロックに分割され、バッファヘッダ

各バッファブロックに対応する `buffer_head` 構造体 (include/linux/fs.h, line 68) がそれぞれローエンドに設けられています。このヘッダ構造は、対応するバッファブロックの属性を記述するために使用され、すべてのバッファヘッダをリンクリストに結合するために使用されます。この設定操作は、バッファブロックを分割するのに必要なバッファのメモリが不足するまで続けられます。

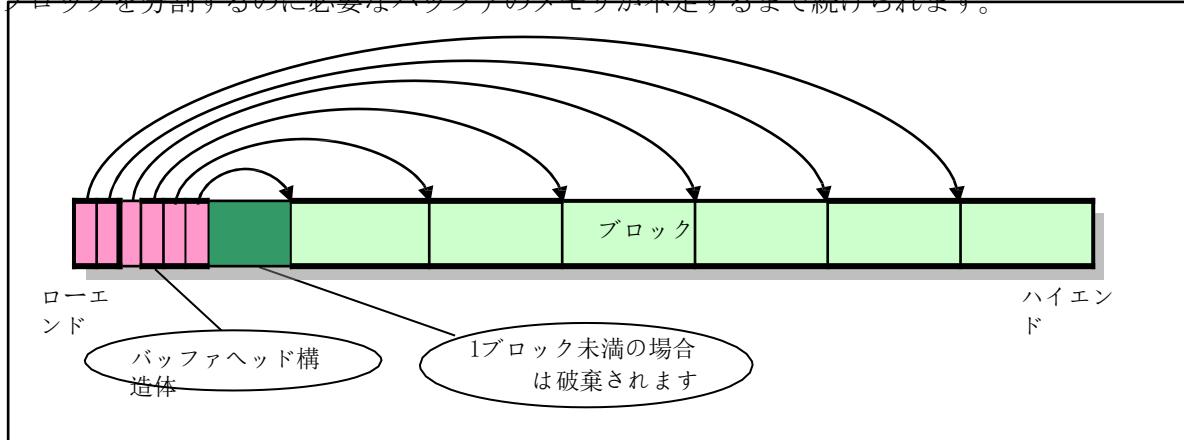


図12-16 バッファキャッシュの初期化

3.バッファキャッシュ構造とリンクリスト

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を指しており、つまり、今までに

は、最近使用されたものです。バッファブロックヘッダのデータ構造は

```
struct buffer_head {
    char * b_data; // バッファブロックのデータ領域を指す (1024バイト)
    符号付きロング b_blocknr; // Block No.
    符号付きショート b_dev、符 // データソースのデバイス番号 (0=フリー)。
    号付きチャ一 b_uptodate、 // データが更新されたかどうかを示す。
    符号付きチャ一 b_dirt、符 // 修正フラグ: 0 -クリーン、1 -修正 (ダーティ)
    号付きチャ一 b_count、符号 // ブロックを使用しているユーザーの数です。
    付きチャ一 b_lock。 // ブロックがロックされているかどうか。0-OK, 1-
    struct task_struct * b_wait; // ブロックを待っているタスクを指します。
};
```

```

struct buffer_head * b_prev;           // ハッシュキューの前のブロック。
struct buffer_head * b_next;         // ハッシュキューの次のブロックです。
struct buffer_head * b_prev_free;    // フリーリストの前のブロックです。
struct buffer_head * b_next_free;   // フリーリストの次のブロックです。
};

。
```

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、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で使用されるカウントで、何回目の対応するバッファブロックは各プロセスで使用（参照）されているため、このフィールドはバッファブロックのプログラム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、データ

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update（valid）フラグです。両方のフラグは、ブロックが初期化またはリリースされたときに0に設定され、この時点では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。

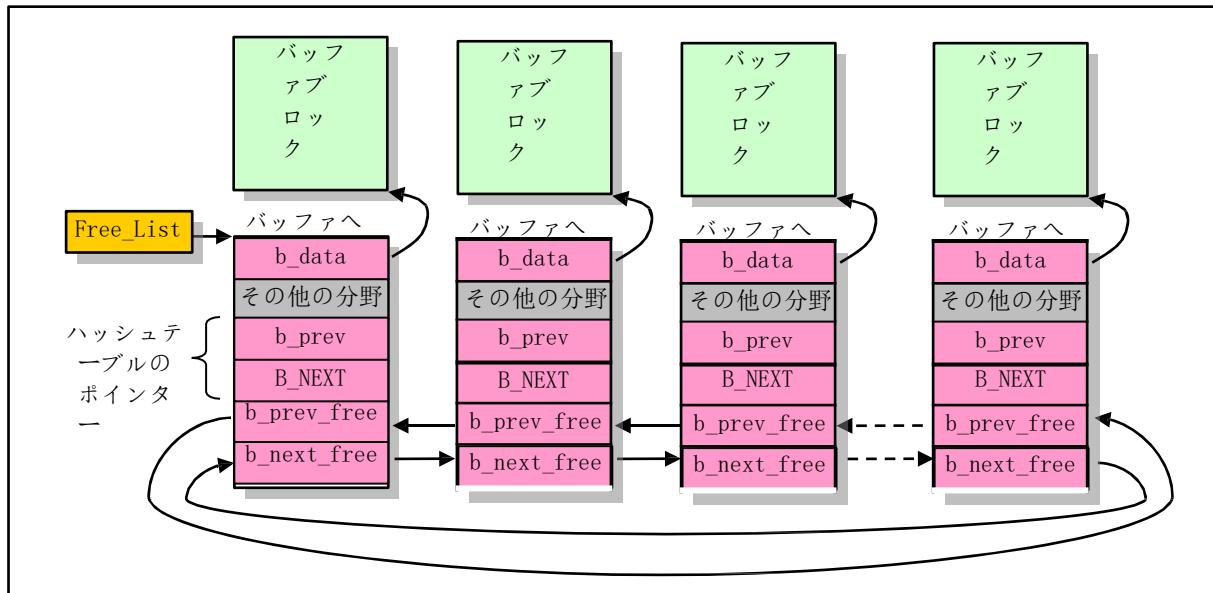


図12-17 全バッファブロックの双向향サーキュラーリンクされたフリーリスト

図のバッファヘッダ構造の「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getblk()を使用して、一致するブロックや空きブロックを探します。

また、すべてのバッファブロックで使用します。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のデータアクセスと管理機能の呼び出し階層は、図12-18で説明できます。

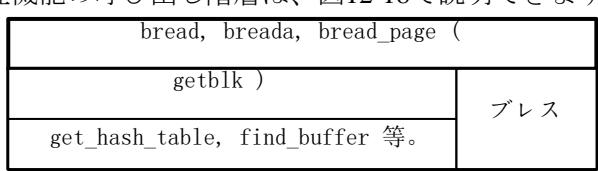


図12-18 バッファーマネジメント機能の階層的な関係

4. バッファキャッシュのハッシュキュー

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 \wedge 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向リンクであり、すなわち、ハッシュ関数によって算出された同じハッシュ値を持つバッファブロック

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

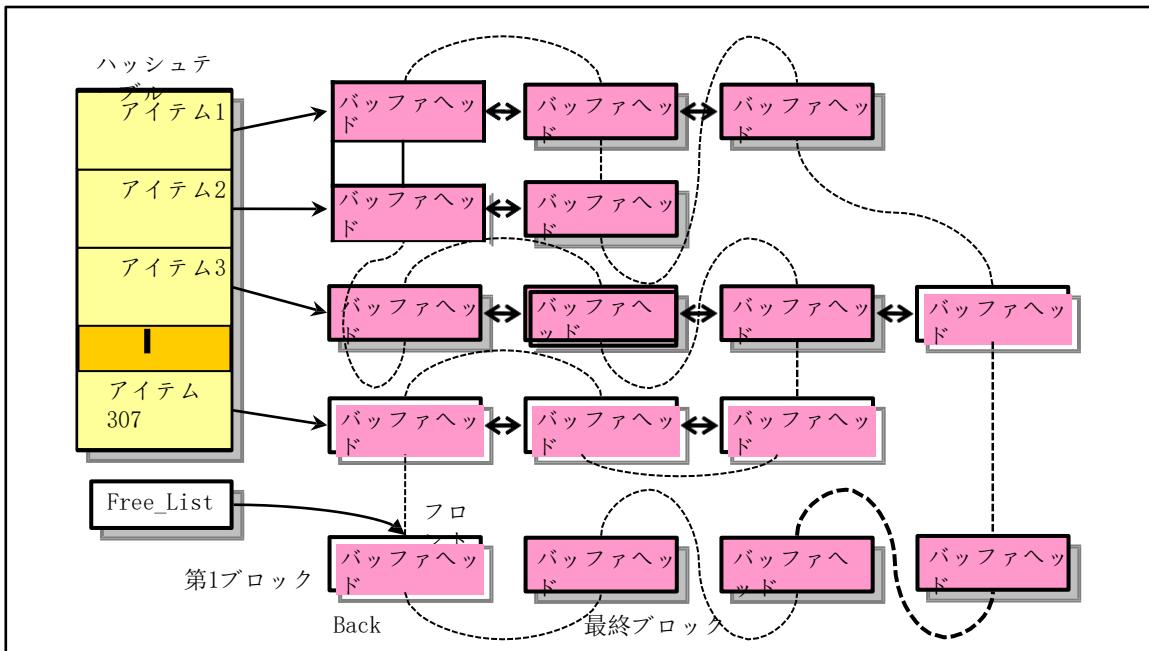


図12-19 ある瞬間のバッファブロックのハッシュキューの模式図

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（Least 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。

5. バッファブロック取得機能

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ドフラグもされている場合は、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（ロックドライバ）のロック解除を待つ必要がある。スリープ待機中にバッファブロックが他のプロセスに占有されていた場合は、再度バッファブロックの検索を開始すればよい。それ以外の場合は、バッファブロックが変更されているかどうかを判断し、変更されていれば、そのブロックをディスクに書き込み、ブロックのロックが解除されるのを待ちます。この時、バッファブロックが再び他のプロセスに占有されると、再び完全に放棄されてしまうので、再びgetblk()の実行を開始しなければなりません。

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

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

図12-20をご覧ください。

以上の分析から、この関数は新しい空きバッファブロックを取得するたびには、それを free_list ヘッダポインタが指すリンクリストの最後に移動させます。つまり、バッファブロックがリンクリストの末尾に近いほど、使用される時間が短くなります。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。

- 指定されたバッファブロックがハッシュテーブルに存在する場合は、利用可能なバッファブロックを取得したことを示すので、そのまま戻ります。
- そうでなければ、リンクリストのfree_listヘッダーから検索を開始する必要があります。つまり、最も最近使用されたバッファブロックから検索を開始します。

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。デバイスの操作には時間がかかることが多いので、これを増やす必要がある。

を b_dirt の重量として計算する。そして、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。

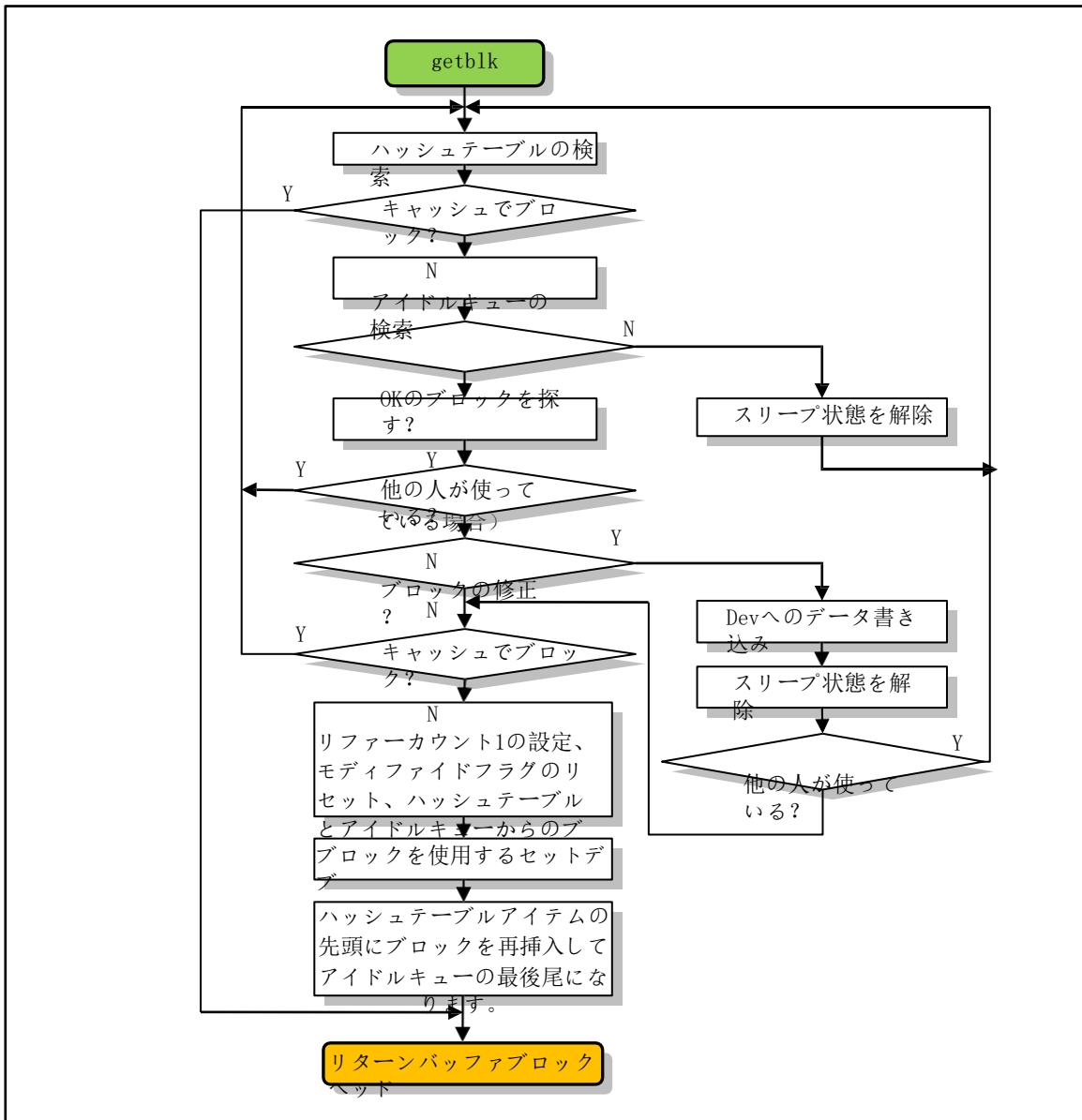


図12-20 Getblk()関数の実行フローチャート

6.バッファブロックリード機能

以上の処理から、getblk()が返すバッファブロックは、新たなフリー ブロックである場合もあれば、必要なデータを含む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())の際には、バッファブロックの更新フラグを判定して、次のように判断する必要があります。

は、含まれているデータが有効かどうかを判断します。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戻すことができる。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、同時に自分自身をスリープ状態にして、バッファブロックにデータが読み込まれるのを待ち、起きた後にデータが有効かどうかを判断する必要があります。有効であれば、このデータをアプリケーションプログラムに返すことができ、そうでなければ、デバイスへの読み出し動作は失敗し、データは取り出せない。したがって、バッファブロックは解放され、NULL値が返されます。図12-21は、bread()関数のブロック図である。breada()関数とbread_page()関数は、bread()関数と似ている。

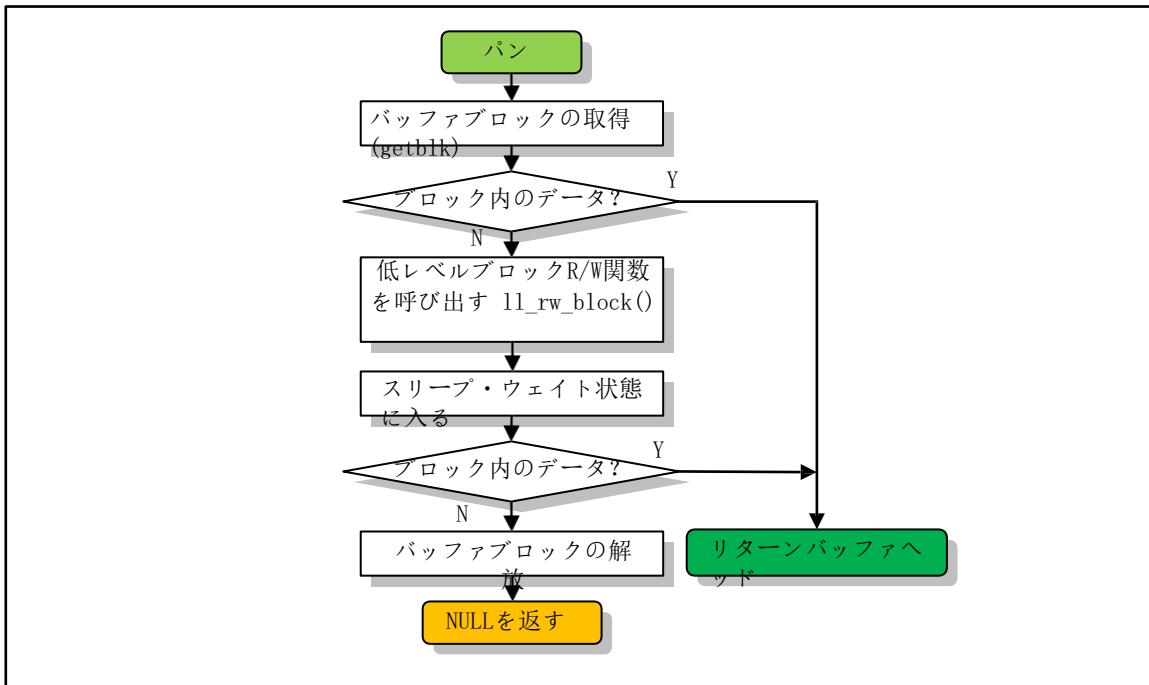


図 12-21 Bread()機能の実行フロー図

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。

7. バッファアクセス方法と同期動作

以上のように、高速バッファは、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、キャッシング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、キャッシングマネージャの`bread()`関数と、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()`を通じてキャッシング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ングにある場合は、キャッシング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、キャッシングマネージャは`ll_rw_block()`を通じて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。図12-22のように、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ングに入れた後、キャッシングマネージャは上位プログラムにデータを返します。

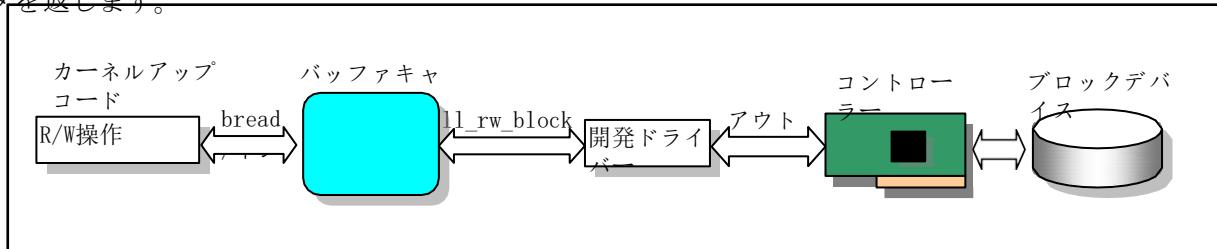


図12-22 カーネル・ロック・デバイス・アクセス動作

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。

- (1) データ構造情報は、キャッシュバッファ内のバッファブロックと同期しており、ドライバーには独立した責任があります。
- (2) バッファキャッシュ内のデータブロックとディスクの対応ブロックとの同期問題は、ここではバッファ管理プログラムによって処理される。

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違うのかを知るためにには、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_inodes()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 コードアノテーション

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 /*
7 *
8 * 'buffer.c' は、バッファキャッシュ機能を実装しています。レースコンディションは
9 * 割り込みでは絶対にバッファを変更させないようにすることで回避できます。
10 * もちろん、データも) ではなく、呼び出し側に任せることにしました。注意! 割り込みとして
11 * をチェックするには、いくつかの cli-sti シーケンスが必要です。
12 * スリープ・オン・コール。これらは非常に素早くできるはずです(期待しています)。
13 */
14
15 /*
16 * 注意! ここで一つ不協和音があります: フロッピーチェックは
17 * ディスクの変更。これが一番しつくりくるのは、当然のことですが
18 * 変更されたフロッピーディスクのキャッシュを無効にします。
19 */
20// 形式のマクロがあります。主に、vsprintf、vprintf、vfprintf関数の1つの型 (va_list)
// <stdarg.h> の標準的な va_s のヘビーリングアーベルで変数説明してもありますと定義します。
// <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/initrd.h> のマクロと、記述子のソラーメータ設定を取得に構成体を組み込みア
// リングアーベルでセンブリ関数マクロの記述があります。
// <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/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リの形式です。
21 #include <stdarg.h>
22
23 #include <linux/config.h>
24 #include <linux/sched.h>
25 #include <linux/kernel.h> (日本語)
26 #include <asm/system.h>
27 #include <asm/io.h>
28
// 以下の変数'end'の値は、リンカーライドによってコンパイル時に生成され、以下を示します。
// 図12-15に示すように、カーネルモジュールの終了位置です。この値を求めるには
// カーネルがコンパイルされたときに生成されたSystem.mapファイルのことです。ここでは、次のこと
// を示すために使われています。
// バッファキャッシュは、カーネルコードの最後に始まります。
// 33行目のbuffer_wait変数は、眠っているタスクのキューヘッドポインタである
// フリーのバッファブロックを待っている間。b_wait ポインタとは異なる効果があります。
// をバッファブロックのヘッダ構造に追加します。タスクがバッファブロックを要求したときに、
// たまたま
// 利用可能な空きバッファブロックが不足している場合、タスクは buffer_wait スリープに追加さ
// れます。
// wait queueです。b_waitは、待機しているタスク専用の待機キューヘッダであり
31 // (nr指定されたheadがHash table[NR_HASH]b_wait に対応するhashノードロック)。
32 static inline buffer_head * free_list;           // フリーのバッファブロックのリンクリストのヘッ
33 static buffer_head * start_buffer_wait(struct buffer_head *) end; // 空きブロックを待つためのキューです
// システムバッファキャッシュに含まれるバッファブロックの数を定義します。ここでは
// NR_BUFFERSは、linux/fs.hファイルの48行目で定義されている定数シンボルで、以下のように定義
// されています。
// fs.hの172行目でグローバル変数として宣言されているnr_buffersという変数として
// ファイルを作成しました。Linus氏は、暗にnr_buffersを示すようにコードを書いています。
// カーネルバッファが初期化された後も変更されない「定数」です。設定されるのは
// 初期化関数buffer_init() (371行目)の中の
34 int NR_BUFFERS = 0; // バッファブロックの数です。 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
    // をバッファキューブの 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、それぞの
    // プロセスは、レジスタEFLAGSの値を自身のTSSセグメントに保存し、現在の
    // CPUのEFLAGSは、プロセスが切り替わると変化します。sleep_on()を使用しているプロセスが
    // sleepはwake_up()で明示的に目覚めさせる必要があります。
36 static inline void wait_on_buffer(struct buffer_head * bh)
37 {
    cli(); // intを無効に
38 // ロックされてもbhがロックアプロセスはスリープ状態になり、ロックが解除されるのを待ちます。
39     sleep_on(&bh->b_wait)
40     // intを有効に
41     // となります。
42 }     sti(); // する。
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ヤンされます。
//となり、変更されたバッファブロックに対してライトディスククリクエストが発生して
// バッファリングされたデータがディスクに書き込まれることで、キャッシュ内のデータが同期して
// デバイスになります。
44 int sys_sync(void)
45 {
46     int i;
47     struct buffer_head * bh;
48
49     sync_inodes()を使用しています。          /* 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します。
51     for (i=0 ; i< NR_BUFFERS ; i++, bh++)
52     {...   wait_on_buffer(bh);           // バッファのロックが解除されるのを待つ（ロックされている場合）。
53         if (bh->b_dirt)               。}
54         11_rw_block(WRITE, bh); // 書き込みデバイスブロック要求を生成します
55     }
56     0を返す。
57 }
58
//// キャッシュされたデータを、指定されたデバイス上のデータと同期させる。
// この関数は、まずバッファキャッシュ内のすべてのバッファブロックを検索し、ディスクに書き込む
// 指定されたデバイスのバッファのデータが変更された場合（同期操作）を行います。そして、インメモリのi-nodeテーブルデータをバッファキャッシュに書き込みます。その後、実行する
// 指定されたデバイスに対して、上記と同じ書き込み操作を行う dev again.
59 int syncdev(int dev)
60 {
61     struct buffer_head * bh;
62
63     // で指定されたデバイスに対して、データの同期操作を行います。
64     // パラメータを使用して、デバイス上のデータとバッファキャッシュ内のデータを同期させます。
65     // その方法は、キャッシュ内のすべてのバッファブロックをスキャンして、バッファブロックが
66     // 指定されたデバイス dev の // をロックし、ロックされている場合はスリープしてロック解除を待つ。
67     // 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。
68     // 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
69     // を表示します。の間にバッファブロックが解放されたり、他の目的に使用されたりした可能性があるからで
70     // す。bh = start_buffer。 // キャッシュの先頭を指します。
71     // のバッファブロックを再度確認する必要があります。
72     // 指定されたデバイスを選択してください。 // dev.に属さないブロックは無視する。
73     if (bh->b_dev != dev)
74         continue; // バッファのロックが解除されるのを待つ。
75     if (bh->buffer->dev == bh->b_dev && bh->b_dirt)
76         11_rw_block(WRITE, bh);
77
78     // 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。
79     // 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
80     // 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
81     // カーネルの実行効率を向上させるために、//を使用しています。バッファ同期のファーストパスは

```

```

//は、メモリ内の多くの「ダーティブロック」をクリーンな状態にすることができますので、同期動作
のための
// i-nodeを効率的に行うことができます。この2回目のバッファ同期では
// i-nodeのデータとの同期操作によってダーティになったバッファブロック
72 デバイスの中に //。
73     sync_inodes()を使用しています。
74     bh = start_buffer。
75     for (i=0 ; i< NR_BUFFERS ; i++,bh++) {...}
76         if (bh->b_dev != dev)
77             continue;
78         wait_on_buffer(bh);
79         if (bh->b_dev == dev && bh->b_dirt)
80             ll_rw_block(WRITE,bh);
81     }
82 }      0を返す。
83
84 ///////////////////////////////////////////////////////////////////
84 // 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。
85 // キャッシュ内のすべてのバッファブロックをスキャンします。指定されたデバイスのバッファブロ
86 // ックに対して、リセット
87 // その有効(更新)フラグと修正フラグ。
88 void inline invalidate_buffers(int dev)
89 {.    int i;
90     struct buffer_head * bh;
91
92     bh = start_buffer。
93     for (i=0 ; i< NR_BUFFERS ; i++,bh++)           // devに属していないブロックは無視する。
94         if (bh->b_dev != dev)
95             continue;          // バッファのロックが解除されるのを待
96         // プロセスがストリップ待機を行っているので、バッファのます。
97         // が指定されたデバイスに属していることを示します。
98
99 /*
100 * このルーチンは、フロッピーが変更されたかどうかをチェックして
101 * その場合、すべてのバッファキャッシュエントリが無効になります。この
102 * は比較的時間のかかるルーチンなので、できるだけ
103 * it.したがって、「マウント」や「オープン」の時にのみ呼び出されます。この
104 * は、スピードと実用性を兼ね備えた最高の方法だと思います。
105 * 操作の途中でディスクを交換する人がいること。
106 * to loose :-)
107 *
108 * 注! 現在はフロッピーのみの対応となっていますが、アイデアとしては
109 * リムーバブルブロックデバイスが追加された場合、このルーチンが使用されます。
110 * また、mount/openは、フロッピーやその他のものを知らなくても構いません。
111 * スペシャル。
112 */
113 ///////////////////////////////////////////////////////////////////
113 // ディスクが交換されたかどうかを確認し、交換された場合はバッファブロックを無効にします
114
114 void check_disk_change(int dev)
114 {

```

```

115     int i;
116
117 // 最初に、それがフロッピーデバイスかどうかをチェックします。
118 // メディアそうでない場合は終了します。次に、フロッピーディスクが交換されたかどうかをテス
119 // トし、そうでなければ終了します。このようにして
120 // 関数 floppy_change() は blk_drv/floppy.c の 139 行目にありま
121    す if (MAJOR(dev) != 2)
122         を返すことができます。
123     if (! floppy\_change(dev &
124         0x03)) return;
125
126 // フロッピーディスクが交換されたので、i-nodeのビットマップで占められているバッファブロック
127 // と
128 // 対応するデバイスの論理ブロックビットマップが解放され、デバイスのi-nodeが
129 // データブロック情報で占有されていたキャッシュブロックを無効にします。
130     if (super\_block[i].s_dev == dev)
131         put\_super(super\_block[i].s_dev);
132     invalidate\_inodes(dev) です。
133 } invalidate\_buffers(dev) です。
134
135 // 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。
136 // ハッシュテーブルの主な役割は、要素を探すのにかかる時間を短縮することです。を確立すること
137 // で
138 // 要素の格納場所とキーワード（ハッシュ関数）を対応させる。
139 // 関数の計算によって、指定された要素をすぐに直接見つけることができます。
140 // 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
141 // 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
142 // ます。
143 // 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
144 // を主に使用しています。
145 // 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
146 // 号
147 // '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
148 // があります。
149 // 2つのキーの値。ここで2つのキーワードのXOR演算は、単なる計算方法の1つです。
150 // のキーバリューを使用しています。キーの値にモジュロ演算(%)を行って、その値が
151 // 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。
152 #define hashfn(dev, block) (((unsigned)(dev ^ block))%NR_HASH)
153 #define hash(dev, block) hash\_table[ hashfn(dev, block) ]。
154
155 ///////////////////////////////////////////////////////////////////
156 // 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。
157 // ハッシュキュー(bh->b_next)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
158 // 双方向のサーキュレーションリスト構造
159 static inline void remove\_from\_queues(struct buffer\_head * bh)
160 {
161     if (bh->b_prev)
162     {
163         bh->b_prev->b_next = bh->b_next となります。
164     /* ハッシュテーブルから削除 */
165     // ハッシュテーブルがキューの次のバッファブロックを指す if
166     if (hash(bh->b_dev, bh->b_blocknr) == bh)
167         hash(bh->b_dev, bh->b_blocknr) = bh->b_next;
168     /* フリーリストから削除 */
169     if (!(bh->b_prev_free) || !(bh->b_next_free))
170         panic("Free block list corrupted");
171     bh->b_prev_free->b_next_free = bh->b_next_free;
172     bh->b_next_free->b_prev_free = bh->b_prev_free
173     となります。
174 }

```

```

// 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た場合、次のバッファブロックに誘導さ
145      れる if (free_list == bh)
146          free_list = bh->b_next_free となります。
147 }
148
149 /////////////////////////////////////////////////////////////////// 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。
150 static inline void insert_into_queues(struct buffer_head * bh)
151 {
152     /* フリーリストの最後に置く */
153     bh->b_next_free = free_list;
154     bh->b_prev_free = free_list->b_prev_free;
155     free_list->b_prev_free->b_next_free = bh;
156     free_list->b_prev_free = bh となります。
157     /* バッファにデバイスがあれば、新しいハッシュキューに入れる */ (注)
158     // 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
159     // 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
160     // で
161     // bh->b_next が NULL でない場合にのみ、b_prev に bh の値を割り当てるべきです。つまり、163
162     // 番目の
163     // の行は「bh->b_if(bh==NULL, bh->b_next)」という文を追加する必要があります。この誤りはカーネルの後に修
164     // 正されましたが、bh->b_next = NULL;
165     // 0.96バージョンより b_dev)
166     // 返すことができます。
167     bh->b_next = hash(bh->b_dev, bh->b_blocknr);
168     hash(bh->b_dev, bh->b_blocknr) = bh;
169     bh->b_next->b_prev = bh; // この前に「if (bh->b_next)」を追加する必要があります。
170 }
171
172 /////////////////////////////////////////////////////////////////// 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
173 // を調べる
174 // バッファキャッシュ内の見つかった場合はバッファブロックポインタを、そうでない場合は NULL
175 // を返します。
176 static struct buffer_head * find_buffer(int dev, int block)
177 {
178     //for(指定したdeviceのblock番号)ループ
179     // 169 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する for
180     (tmp = hash(dev, block) ; tmp != NULL ; tmp = tmp->b_next)
181         if (tmp->b_dev==dev && tmp->b_blocknr==block)
182             return tmp;
183     // NULLを返す。
184 }
185
186 /*
187 * なぜこんなことになったのかというと理由は人種的な条件です。
188 * バッファをロックすることはできません（読み込み中でない限り）。
189 * 私たちが寝ている間に何かが起こるかもしれません（例えば、読み取りエラー
190 * はそれを強制的に悪くします）。これは現在、実際には起こらないはずですが
191 * コードの準備ができました。
192 */
193 /////////////////////////////////////////////////////////////////// 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。
194 // 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
195 // 。見つかった場合は
196 // ブロックがロックされ、ブロックヘッダポインタが返されます。
197 struct buffer_head * get_hash_table(int dev, int block)
198 {

```

```

185     struct buffer_head * bh;
186
187     for (;;) {
188         // 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。
189         // 見つからなかった場合は、NULLを返して終了します。
190         if (!(bh=find_buffer(dev,block)))
191             return NULL;
192
193         // 目的のブロックが見つかれば、その参照カウントを1つ増やして、待機する
194         // ブロックがロックされている場合は）ロック解除のための //。スリープ状態を経ているので
195         // ブロックの正しさを検証し、バッファヘッドポインタを返す必要があります。
196         // 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
197         // トが取り消さ と言って、再び検索をかけます。
198
199         bh->b_count++; // バッファのロックが解除されるのを待つ（ロックされてい
200         // る場合）。
201         if (bh->b_dev == dev && bh->b_blocknr == block)
202             // bhを返す。
203             bh->b_count--;
204     }
205 }
206 */
207 * OK一スコープ内にgetblkで、あまり正確ではあがまかんかに使わんせんのより繰り返し)。
208 * なので、見た目よりもずっと効率的なはずです。
209 *
210 * アルゴリズムが変更されました: うまくいけば、より良いものになり、つかみどころのないバグ
211 * が取り除かれます。
212 */
213 // 以下のマクロでは、ブロックのモディファイドフラグとロックフラグを
214 // 同時に、修正フラグの重みがロックフラグよりも大きくなるように定義されています。
215 #define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)

216 /**
217 * 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。
218 * 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
219 * うかをチェックします。
220 * 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
221 * が返されて終了します。そうでない場合は、デバイスNo.とブロックNo.に対応する新しいエントリ
222 * が作成されます。
223 * がキヤウド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。
224 struct buffer_head * getblk(int dev, int block)
225 {
226     // 繰り返す。
227     // ハッシュテーブルを検索し、指定されたブロックがすでにバッファキャッシュに存在する場合は、
228     // そのブロックを返します。
229     // 対応するブロックヘッドのポインタを指定して
230     // 終了する。 if (bh =
231         get_hash_table(dev,block))
232         bhを返す。
233 // そうでなければ、フリーリストをスキヤンしてフリーブロックを探す。まず、'tmp' には
234 // フリーリストの最初のフリーブロックのヘッダーを、その後の操作を行うためにループします。
235     do {
236         // 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）、スキヤンを続
237         // ける
238         // 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
239         // しては
240         // をキャッシュに入れた場合、必ずしもクリーン（b_dirt=0）やアンロック（b_lock=0）ではない。
241         // したがって、我々は
242         // といった判断や選択を続けていく必要があります。例えば、次のようなブロックの場合

```

```

// b_count=0の場合、タスクがブロックの内容を書き換えてからリリースすると、ブロックb_countの
タスクがbreada()を実行して、いくつかのファイルを事前に読み込んだとき、 // はまだ0ですが、
b_lockは0にはなりません。
// ブロックの場合は、ll_rw_block()が呼ばれるとすぐにb_countをデクリメントします。ただし、こ
215 のときに
216 // ハードディスクへ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, but
    b_count=0. if (tmp->b_count)
        を続けています。
// 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()
217 // = 0) の場合は、指定されたデバイスで対応するキャッシュブロックが取得されたことを意味しま
218 す。
219 // となり、ループは終了します。そうでなければ、このループを実行し続けて
220 // BADNESS()の値が最も小さいバッファを見つけます。
221         if (!bh || BADNESS(tmp) < BADNESS(bh))
222 良いものが見つかるまで繰り返します /*。
223     } while ((tmp = if (!bnext_free) = free_list);
        ブレークします。
// ループチェックで全てのバッファブロックが使用されていると判断された場合（全てのバッファブ
ロックのヘッダー参照
// の数が >0）、sleep はバッファの空きブロックが利用可能になるのを待ちます。このプロセスは
、明示的に
224 空いているバッファブロックが利用可能になると、 // ウォークアップされ、次のように先頭にジャ
ンプします。
225 // 空きバッファブロックを再検索する関数 if (!bh)
226     {
227         .
sleep_on(& buffer_wait) です。
        goto repeat; // 210行目にジャンプします。
// を持 をクリックすると、再び上記の検索プロセスを繰り返すことがで
ここででありますと、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
あлокが解除されるのを待ちます
228 // (ロ wait_on_buffer(bh) です)スリープ期間中にバッファが他のタスクによって使用された場合は
229     if (bh->b_count) // また占領された?
230 // バッファが変更された場合。データをディスクに書き込み、バッファのロックが解除されるのを待
つ
// を再び使用します。同様に、バッファが再び他のタスクによって使用された場合、上記の検索処理
231 は
232 //を繰り返した。
233     while (bh->b_dirt) {
234         sync_dev(bh->b_dev);
235         wait_on_buffer(bh);
236     } if (bh->b_count) // また占領された ?? goto repeat;
237 /* NOTE!私たちがこのブロックを待っている間に、他の誰かが */。
238 /* "this" ブロックは既にキャッシュに追加されています。
// スリープ中に指定されたバッファブロックがハッシュテーブルに追加されたかどうかをチェックし
ます。
239 // そうであ find_bhの検索(成功)をもう一度繰り返します。
240         goto repeat;
241 /* OK、ついにこのバッファが唯一のものであることがわかりました。
242 /* そして、未使用(b_count=0)、アンロック(b_lock=0)、クリーンであること */。
// 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。

```

```

// bh->b_count=1...となります。
243     bh->b_dirt=0となります。
244     bh->b_uptodate=0;
245 // バッファブロックのヘッダは、まずハッシュキューとフリーブロックリストから削除され、次に
// バッファブロックは、指定されたデバイスとその上の指定されたブロックに使用されます。バッフ
アは
// ブロックは、フリーリストに再挿入され、ハッシュキューの新しい位置に置かれ、最後に
// remove_from_queues(bh)...バッファブ

246     ロックのヘッダを返す。
247     bh->b_dev=dev;
248     bh->b_blocknr=block;
249     insert_into_queues(bh);
250     return bh;
251 }

252 /**
253 ///////////////////////////////////////////////////////////////////
254 * brelse - バッファブロックを解放します。
255 * @buf: バッファヘッダ
256 *
257 * 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
258 * され、空きバッファブロックを待っているプロセスが明示的にアウェイクしている。
259 */
260 void brelse(struct buffer_head * buf)
261 {
262     if (!buf) // ブロックヘッドポインタが無効な場合に返します
263         return;
264     wait_on_buffer(buf); // バッファブロックが空き状態になったら、この行を実行
265     if (!buf->b_count--) // バッファブロックの参照カウントが0になったら
266         panic("Trying to free free buffer");
267     wake_up(&buffer_wait);
268 }

269 /**
270 * bread() - バッファブロックを読み込みます。
271 * @dev: デバイス番号
272 * @block: ブロック番号
273 *
274 * バッファブロックを読み込みます。戻り値がNULLの場合は、カーネルの
275 * //エラーが発生します。その後、利用可能なデータがあるかどうかを判断します。データがあ
276 * るかどうかを返します。
277 * バッファブロック内の // が有効（更新されている）であり、直接使用す
278 * ることができます。 if (!(bh=>getblk(dev, block)))
279 *     panic("bread: getblk returned NULL");
280 * if (bh->b_uptodate)
281 *     bhを返す。
282 * 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
283 * す。
284 * 使用して、リードデバイスブロック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
285 * クを待って
286 * 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
アが
287 * 更新された場合は、バッファヘッドポインタを返して終了します。それ以外の場合は、次のように
288 * 示します。
289 * デバイスの読み込み操作に失敗したので、バッファを解放し、NULLを返して終了する。

```

```

275      11_rw_block(READ, bh) です
276      。
276      wait_on_buffer(bh) です。
277      if (bh->b_uptodate)
278          bhを返す。
279      ブレルス (bh) です。
280      NULLを返す。
281  }
282
282      // ブロックをコピーフォーマタを、「from」アドレスから「to」位置にコピーします
282      。
283 #define COPYBLK(from, to)
284     __asm__ __volatile__ ("movsl n\n\t" : : "c"(BLOCK_SIZE/4), "S"(from), "D"(to));
285     : "cx", "di", "si")
286
287     /* bread_page は、4つのバッファを目的のアドレスのメモリに読み込みます。それは
288     * 機能    読むことで得られるスピードがあるので、それ自体は
289     * すべては 1つが読まれるのを待たずに、次の1つが読まれるのを待つことなく、同時に
290     * など
291     */
292
293     // 読んでみ 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
294     // へ書き込みます。address はデータを保存するアドレス、'dev' は指定した
295     // 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
296     // mm/memory.c ファイルの do_no_page() 関数（428行目）で使用されています。
297 void bread_page(unsigned long address, int dev, int b[4])
298 {
299     struct buffer_head * bh[4];
300     int i;
301
302     // この関数は4回実行されます。配列b[]に置かれた4つのブロック番号に応じて。
303     // デバイスdevからページが読み込まれ、指定されたメモリロケーション'address'に置かれます。
304     // 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
305     // 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
306     // が消えてしまうと
307     // 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
308     // 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
309     // に対応しています。したがって、この関数は、1~4個のデータブロックを自由に読むことができます。
310
311     // 指定されたb[]内のブロック番号 for
312     (i=0 ; i<4 ; i++)
313         if (b[i]) { // ブロック番号が有効な場合 if (bh[i] =
314             getblk(dev, b[i]))
315         } else      if (!bh[i]->b_uptodate)
316             bh[i] = NULL;  11_rw_block(READ, bh[i]) です。
317
318     // 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。
319     // 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
320     // ロックされている場合は）ロックを解除するための//。さらに、眠っている可能性があるので、同じ
321     // コピーする前に、バッファブロック内のデータが有効かどうかをチェックします。また
322     // コピー後のバッファブロック。
323     for (i=0 ; i<4 ; i++, address += BLOCK_SIZE)

```

```

309         if (bh[i]) {
310             wait_on_buffer(bh[i])と // ブロックがロックされている場合は、ロック
311             なります。 // 解除を待ちます。
312             if (bh[i]->b_uptodate) // データが有効であれば、コピーする。
313                 COPYBLK((unsigned long) bh[i]->b_data, address)です。
314             brelse(bh[i]); // バッファブロックを解放する。
315         }
316     }
317 /* OK、breadaはパンとして使われますが、さらに他のマークにも使えます。
318 * 読書のためのブロックもあります。引数リストの最後には、否定的な
319 * の番号です。
320 */
321 /**
322     //// 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。
323     // パラメータの数は可変で、指定されたブロック番号を連ねたものです。が表示されると
324     // 関数が成功した場合は、最初のブロックのブロックヘッダを返し、そうでない場合はNULLを返します。
325 struct buffer_head * breada(int dev, int first, ...)
326 {...
327     va_list args;
328     struct buffer_head * bh, *tmp;
329
330     // 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
331     // 指定されたデバイスとブロック番号のブロックをキャッシュから削除します。もし、バッファブロ
332     // ックデータが
333     va_start(args, first); // 無効（更新フラグが設定されていない）の場合、デバイスデータ
334     // ブロックの読み取り要求を発行します。
335     if (!(bh=getblk(dev, first)))
336         panic("bread: getblk returned NULL\n");
337     if (!bh->b_uptodate)
338         11_rw_block(READ, bh)です。
339
340     // 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
341     // ます。
342     // 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
343     // 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
344     // は事前に読み込まれた後続のデータブロックで、キャッシュに読み込まれるだけで使用されません
345     // 一方 ((first=va_arg(args, int))>=0) {
346     // そのため、337行目のgetblk(dev, first)では、その参照カウントをデクリメントしてリリースする必
347     // 要があります。 if (tmp) {
348     // ブロックのことです (getblk(tmp->b_data)の参照カウントが増えるため)。
349     // 11_rw_block(READA, bh)で // ここでは'bh'は'tmp'でなけれ
350     // す。 // 先読みブロックを解除します
351
352     // この時点での変数パラメータテーブルのすべてのパラメータが処理されます。そして、次のように
353     // 待ちます。
354     // ロックが解除された最初のバッファブロック（ロックされていた場合）。待ち時間が終わると、も
355     // し
356     // バッファend(args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
357     // す。 wait_on_buffer(bh)です
358     // そうでなければ、バッファは解放され、NULLを返します。
359     if (bh->b_uptodate)
360         bhを返す。
361     ブレルス(bh)です
362     を返す (NULL) 。

```

346 }

347

//// バッファ初期化機能。
 // パラメータ buffer_end は、バッファキャッシュメモリの終端です。16MBのシステムでは
 // メモリの場合、バッファキャッシュエンドは4MBに設定されます。8MBのメモリを持つシステムでは
 // 、バッファエンドの
 // には2MBが設定されています。この関数は、ブロックヘッダ構造体を設定（初期化）し、対応する
 // バッファキャッシュの開始位置であるstart_bufferとbuffer_endからのデータブロック
 バッファキャッシュのすべてのメモリが割り当てられるまで、それぞれのバッファの最後に//をつける。

348 void buffer_init(long buffer_end) start_buffer;

349 { . void * b;

350 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、次のようにになります。
 // ビデオメモリーやBIOSが640KB～1MBの範囲で使用されている場合、実際に使用できるバッファメモリーは
 // は、ハイエンドで640KBとします。それ以外の場合は、バッファ・キャッシュ・メモリの最上位が
 354 // 1MBを超えていません。

355 if (buffer_end == 1<<20)

356 b = (void *) (640*1024);

357 その他

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
 るために使用されます。

// とし、システム内のブロック数を取得します。演算処理は、バッファを分割して
 // バッファの最上位から1KBサイズのブロックを構築し、同時に
 // バッファの下端にあるバッファブロックを記述して、buffer_headを形成する
 // を二重に連結したリストにします。

// 'h' はブロックヘッダ構造体へのポインタ、'h+1' は次のブロックヘッダアドレス
 メモリアドレスを連続して指している//の最後を指しているとも言えます。

// h ブロックヘッダ。ブロック・ヘッダを格納するのに十分なメモリを確保するために

358 // 構造体の場合、'b' が指すメモリブロックのアドレスは、以下の値以上である必要があります。

359 // 'h' のブロック h->b_dev の終わりに等しい、つまりバイ~~b~~h.h+1' が必要です。 while360 ((b == BLOCK_SIZE) 0= ((void *) (h+1)) 汚れフラグ（ブロック修正フラグ）。

361 h->b_count = 0; // ブロック参照数。

362 h->b_lock = 0; // ブロックロックフラグ。

363 h->b_uptodate = 0; // ブロック更新フラグ（またはデータ有効フラグ

364 h->b_wait = NULL;)。365 h->b_next = NULL; // 待機中のキューをブロックします。366 h->b_prev = NULL; // 同じハッシュ値を持つ次のヘッダを指します。

367 h->b_data = (char *) b; // 固定タグ値を持つ前のヘッダを指します。

368 h->b_prev_free = h-1; // フリーリストの前の項目を指します。

369 h->b_next_free = h+1; // フリーリストの次の項目を指します。

370 h++; // h は次の新しいバッファヘッダの位置を指します

371 NR_BUFFERS++です。)。

// bが1MBにデクリメントされた場合、384KB（ビデオメモリ又はデータ）のカ~~b~~hが0xA0000 (640KB) を
 372 指すようにする。

373 b = (void *) 0xA0000;

374 }

// 次に、h に最後の有効なバッファブロックヘッダを指定し、フリーリストヘッダには

// 最初のバッファブロックのヘッダ；フリーリストのヘッダのb_prev_freeは、前の

```

// hの次のポインタは、最初のヘッダを指すので
// フリーリストは双方向のリング構造を形成します。最後に、ハッシュテーブルを初期化し、すべて
の
375 テーブル内の // ポインタを NULL にします。// hは最後のブロックヘッダを指します。
376     free_list = start_buffer;           // フリーリストのヘッダは、最初のブロックのヘッ
ダを指します。
377     free_list->b_prev_free = h;
378     h->b_next_free = free_list;
379     for (i=0;i< NR_HASH;i++)
380         hash_table[i]=NULLです
            。
381 }
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block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block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で説明します。

12.3.1 機能

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blockやi-node構造の使用状況に応じて、論理block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block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blockを解放するために使用されます。この関数は

具体的な動作としては、指定された論理blockに対応する論理block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blockを取得し、スーパーblockに記載されているデバイスデータの論理blockの範囲に応じて、論理block番号の有効性を判断しています。次に、指定された論理block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block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block番号を計算し、論理block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blockビットマップを含むバッファblockにおいて、論理block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、blockデバイスに論理blockを要求し、論理block番号を返し、そのblockに対応する論理blockビットマップビットを設定するために使用されます。まず、指定されたデバイスdevのスーパーblockを取得します。⁷²⁸次に、論理blockのビットマップ全体を検索して、最初に0になるビットを探します。

ディスク装置のスペースがなくなった場合、この関数は0を返します。それ以外の場合は、最初に見つかった0ビットの位置に1が設定され、対応するデータ論理ブロックが占有されていることを示します。同時に、このコードは、そのビットを含む論理ブロックのビットマップが配置されているバッファブロックのモディファイア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 コードアナリシス

プログラム 12-2 linux/fs(bitmap.c)

```

1 /*
2 * linux/fs(bitmap.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /* bitmap.c には、inode および block のビットマップを処理するコードが含まれています */。
8 // <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
9 // います。初期タスク0のデータと、記述子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
10 // <linux/sched.h> 関数スタックの記述があいまづいファイルでは、タスク構造体task_structや
11 // <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。
12 // #include <linux/sched.h>
13 // #include <linux/kernel.h>
14 // (日本語)
15 // 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。
16 // 入力: eax = 0; ecx = ブロックの長さをロングワードで表したもの(BLOCK_SIZE/4); edi = ブ
17 // ロックの長さを指定します。
18 // 開始アドレス addr.
19 #define ifeJustOneBlock(addr) \(^o^) // 明確な方向性
20 "rep\#"\(0\); "stosl"\(0\), "c"\(BLOCK_SIZE/4\), "d"\((long)(addr)):\ "cx",
21 "di"
22 // 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。
23 // 'nr' は32より大きくてOK! このマクロは、元のビット値を返します。
24 // 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。
25 // 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
26 // EAXレジスターを使用することで、効率的なアクセスと操作が可能になります。この変数の定義方
27 // 法は、主に
28 // インラインのアセンブリプログラムで使用されます。マクロ全体はステートメント式で、その値は
29 // は、最後の「res」の値です。
30 // 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
31 // 。
32 // ベース・アドレス(%3)とビット・オフセット(%2)をキャリー・フラグCFにセットして

```

```

// SETB命令は、キャリーフラグに応じてオペランド (%al) を設定するために使用されます。
// CFです。CF=1の場合は%al =1、そうでない場合は%al =0となります。
19 #define set_bit(nr,addr) ({})
20 register int res asm ("ax"); ^w^)
21 asm volatile ("btst %2,%3\#tsetb %%al":# 22 "=a"
(res):"0" (0), "r" (nr), "m" (*(addr)); ^w^)
23 res;})
24
//// 指定されたアドレスの先頭からnrビット目のオフセットのビットをリセットします。
// 元のビット値の逆数を返します。
// 入力: %0 -eax(戻り値); %1 -eax(0); %2 -nr, ビットオフセット; %3 -(addr), addrの内容。
// 27行目のBTRL命令は、ビットのテストとリセットに使用されます。コマンドSETNBが使用される
// キャリーフラグCFに応じてオペランド(%al)を設定します。CF = 1の場合、%al = 0、そうでない場
合は%al
// = 1.
25 #define clear_bit(nr,addr) ({# 26
register int res asm ("ax"); ^w^)
27 asm volatile ("ctrl %2,%3\#tsetnb %%al":# 28 "=a"
(res):"0" (0), "r" (nr), "m" (*(addr)); ^w^)
29 res;})
30
//// アドレス「addr」から最初のゼロビットを探し、「addr」からのビットオフセットを返します。
// 入力: %0 - ecx (戻り値); %1 - ecx(0); %2 - esi(addr).
// 「addr」で指定されたアドレスから始まるビットマップの中で、最初に0になるビットを探します
。
// と、アドレス addr からのビットオフセットを返します。addr」はのデータ領域のアドレスです。
// バッファブロックで、スキャンの範囲は1024バイト (8192ビット) です。
// 36行目のBSFL命令は、EAXレジスタの最初の非ゼロビットをスキャンするために使用されます。
// そのオフセットを EDX に配置しま
31 #define y
32 int res; ^w^
33 asm find_first_zero(addr){`'·ω·`}
34     // 演出フラグをクリアします。
35     // [esi] → [eax] を取得
36     // eaxの各ビットを反転させる
37     // 最初の非ゼロビットのオフセット → edx
38     // ラベル2にジャンプする if eax = 0.
39     // オフセットはecx (ビットマップの最初の0ビット)
40     // に追加されます。
41     // ラベル3 (エンド) にジャンプフォワードします。
42     // ecxは、ビット0が見つからない場合、32ビットのオ
43     // フセットを追加します。
44     // 8192ビット (1024バイト) をスキャンしたのか?
45     // 3= (res): "c" (0), "S" (addr)/ まだラベルになつていなければ、後ろにジャンプし
46 res;}) "si";;;;;;;;;
47     // 終了です。このとき、ecxには0ビットのオフセット
48
//// ロジックブロックをデバイスのデータソースに入れてみます。
// 指定された論理ブロックに対応する論理ブロックビットマップのビットをリセットします。
// 成功すれば1を、そうでなければ0を返します。
49 int free_block(int dev, int block)
50 {
51     struct super_block * sb;
52     struct buffer_head * bh;
53
// 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に従ってチェックされます
。
```

```

// の数と、ファイルシステム内の論理ブロックの総数を表しています。指定されたデバイスが
// スーパーブロックが存在しない場合はエラーになります。論理ブロック番号が
// ディスク上のデータゾーンの最初の論理ブロックのブロック番号、またはそれ以上
// デバイス上の論理ブロックの合計数を超えた場合、システムも停止します。
52     if (!(sb = get_super(dev))) // fs/super.c, 56行目. panic ("try to free
53         block on nonexistent device");
54     if (block < sb->s_firstdatazone || block >= sb->s_nzones)
55         panic ("try to free block not in datazone");
56     bh = get_hash_table(dev, block) です。

// 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。この時、参照数が1以上であれば、それは
// 他の誰かがバッファブロックを使用しているので、brelse()が呼び出され、b_countがデクリメン
トされます。
// を1だけ増やして終了します。そうでなければ、修正フラグと更新フラグをクリアして、データを
57 解放します。
58 // ブロックを作成します。このコードの主な目的は、現在のロジックブロックが
59 // バッファキャッシュに存在する場合、brelse()が呼び出され、b_countが解放されています。
60
61     if (bh->b_count > 1) {...}
62     bh->b_dirt=0となり
63     ます。
64     bh->
65         b_uptodate=0; if           // b_count=1の場合、brelse()を
66     }                           呼び出す。
67
68 // 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
69 。最初に計算する          bh) です。
70 // 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。
71 // 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
72 // 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。
73 // これにより、「block / 8192」は、論理ビットマップ内のどのブロックが指定されたブロックであ
74 るかを計算することができます。clear(block / 8191, sb->s_zmap[block/8192]->b_data)) {
75 // と 'block & 8191' は、現在の物理地址の、最後の8位を0にすることができます。
76 // ロジックブロックのビットマップをクリアする。"Free Block: bit already cleared\n";
77     block -= sb->s_firstdatazone - 1; // block = block - (s_firstdatazone -1);
78 // sb->s_zmap[block/8192]->b_dirt = 1. 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
79     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。
80
81 // を返します。
82 }
83
84 ///// デバイスに論理ディスクブロックを要求します。
85 // この関数は、まずデバイスのスーパーべロックを取得し、最初のゼロ値を探します。
86 論理ブロックビットマップの中の // (フリーブロックを表す)。このビットは、次のように設定さ
れます。
87 // 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
88 // 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。
89 // その更新フラグと修正フラグがセットされ、論理ブロック番号が返されます。もし
90 // 実行が成功すると、この関数は、論理ブロック番号（ディスクブロック
91 // 数）を、そうでなければ0を返します。
92
93 int new_block(int dev)
94 {

```

```

78     struct buffer_head * bh;
79     struct super_block * sb;
80     int i, j;
81
82 // 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
83 // のビットマップをスキヤンします。
84 // 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。
85 // 論理ブロックが配置されます。8ブロックの論理ブロックのビットマップの全てのビットがスキャ
86 // ンされると
87 // (i >= 8 or j >= 8192)の場合、0値のビットが見つからないか、ビットマップのバッファ・ブロッ
88 // ク・ポインタが
89 // が無効 (bh = NULL) であれば、0で終了します。 (空きロジックブロックがない)
90     if (!(sb = get_super(dev)))
91         panic ("try to get new block from nonexistent device"); j
92     = 8192;
93     for (i=0 ; i<8 ; i++)
94         if (bh=s->s_zmap[i])
95             if ((j=find_first_zero(bh->b_data))<8192)
96                 break;
97     if (i>=8 || !bh || j>=8192)
98         return 0;
99
100 // 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
101 // 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
102 // ていることを示す
103 // 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
104 // します。
105 // 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
106 // すだけなので
107 // ディスク上のデータゾーンの//、つまりロジックブロックビットマップのビットオフセット値が示す
108 // のは
109 // データゾーンの先頭からのブロック番号なので、論理ブロック番号の
110 // jを論理ブロック番号に変換するために、データゾーンの最初のブロックをここに追加する必要が
111 // あります。
112 // 新しい論理ブロック番号がシステム上の論理ブロックの合計数よりも大きい場合は
113 // デバイス上にブロックが存在しません。アプリケーションが失敗したので、0を返して終了します
114     . if (set_bit(j, bh->b_data))
115         panic ("new_block: bit already set");
116     bh->b_dirt = 1;
117     j += i*8192 + sb->s_firstdatazone-1;
118     if (j >= sb->s_nzones)
119         0を返す。
120
121 // 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
122 // 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
123 // 取得しただけでは、参照カウントが1 (getblk())で設定) でなければならず、1でない場合は
124 // 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
125 // セットします。その後、対応するバッファブロックを解放し、論理ブロック番号を返す。 if
126     (! (bh=getblk(dev, j)))
127         panic ("new_block: cannot get block");
128     if (bh->b_count != 1)
129         panic ("new_block: count is != 1")を実行します。
130
131 // 指定したi-nodeをリリースします。
132 // この関数は、最初に
133 // bh->b_uptodate = 1;
134 // パラメータです。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
135 // きは、i-nodeのbh->b_dirt = 1;
136 // または、スレーブブロック情報、i-nodeのビットマップの対応するビットを使用して操作されます。
137     return j;

```

i-node番号への//がリセットされ、i-node構造がクリアされます。

```

108 void free_inode(struct m_inode * inode)
109 {
110     struct super_block * sb;
111     struct buffer_head * bh;
112
// まず、リリースする必要のあるi-nodeの有効性や正当性を判断します。もし、そのi-nodeが
// i-nodeポインタ=NULLの場合は終了します。i-nodeのデバイス番号フィールドが0であれば、ノード
// は使用しません。そして、対応するi-nodeが占有しているメモリ領域をクリアし、リターンします
// 。
113 // Memset()はファイルinclude/string.hの395行目で定義されていますが、これはメモリ領域
114 inodeポインタで指定された//は0で埋められ、sizeof(*inode)バイトが埋められます。 if (!inode)
115         を返すことができます。
116     if (!inode->i_dev) {...}
117         memset(inode, 0, sizeof(*inode));
118         return;
119     }
// このi-nodeに他のプログラム参照がある場合、それはリリースできないことを示しています。
// カーネルコードに問題があり、システムがダウンしている場合。もし、ファイルリンクの数が
// // 複数でなければ解釈命令なので復元操作を実行するアイルディレクトリエントリがあることを意
120     //味しますので(inode->i_count>1) {...}
121         printk("try to free inode with count=%d\n", inode->i_count)とな
122         ります。
123         panic("free_inode")です。
124     }
125     if (inode->i_links)
126         //となります。
127         // 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。
128         // そのスーパーblockの情報です。まず、i-nodeがあるデバイスのスーパーblockを取ります
129         // 。
130         // 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
131         // 判断します。
132         // //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
133         // // 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
134         // //トマップ
135         // i-nodeに対応する//が存在しない場合はエラーになります。のi-nodeビットマップが存在しないので
136         // // バッファブロックは8192ビットなので、i_num>>13（つまり、i_num/8192）は、s_imap[]エントリ
137         // //を取得することができます。
138         // 現在のi-node番号、つまりディスクブロックです。
139         if (!(sb = get_super(inode->i_dev)))
140             panic("try to free inode on nonexistent device")が発生しました。
141         if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
142             panic("try to free inode 0 or nonexistent inode");
143         if (!(bh=sb->s_imap[inode->i_num>>13]))
144             panic("nonexistent imap in superblock")
145        となります。
146         // 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
147         // でに
148         // が0になると、エラー警告メッセージが表示されます。最後に、i-nodeが入っているバッファの
149         // // ビットマップの位置が変更され、i-node構造体が占有するメモリ領域は
150         // // がクリアされます。
151         if (clear_bit(inode->i_num&8191, bh->b_data))
152             // // 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。
153             // // メモリのi-nodeアルゴリズムでアリーナのi-nodeエントリを取得し、i-nodeからフリーのi-nodeを見つけ
154             // // る
155             // // memset(inode, 0, sizeof(*inode));

```

```

// bitmap.
137 struct m_inode * new_inode(int dev)
138 {
139     struct m_inode * inode;
140     struct super_block * sb;
141     struct buffer_head * bh;
142     int i, j;
143
// 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block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blockが
// まだ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
144 // i-node未てが返されempty nodeが返されて終了node idleが無いifとを示す
    // NULLを返す。
146     も (! (sb = get_super(dev))) // fs/super.c, line 56.
    j = 8192;
147     panic ("new_inode with      デバイス/)を使用しています。
        unknown")
148     for (i=0 ; i<8 ; i++)
149         if (bh=sb->s_imap[i])
150             if ((j=find first zero(bh->b_data))<8192)
151                 ブレークします。
152             も (!bh || j >= 8192 || j+i*8192 > sb->s_ninodes) {
    j = 8192;
153             // 使用されているi-node番号が見つかったので、対応する
154             // jに対応するi-nodeNOを返すアップの//ビット（既に設定されている場合は
155             // 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blockのmodifiedフラグ
156             // を設定する。
// が配置されます。最後に、i-node構造が初期化されます（i_ctimeは、コンテンツの
157             i-nodeの//を変更する）。
158             if (set_bit(j, bh->b_data))
159                 panic ("new_inode: bit already set");
160             bh->dircount=1; // 参照数
161             inode->i_nlinks=1; // ファイルディレクトリのエントリーリンク
162             inode->i_dev=dev; // の数。
163             inode->i_uid=current->euid; // i-nodeが配置されているデバイス番号。
164             inode->i_gid=current->egid; // i-nodeのユーザーID。
165             inode->i_dirt=1; // グループID。
166             inode->i_num = j + i*8192; // 修正フラグを設定します。
167             inode->i_mtime = inode->i_atime = inode->i_ctime;i_ctimeはOPENED TIMEに
168             return inode;
169 }
170

```

4. truncate.c

1. 機能

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。

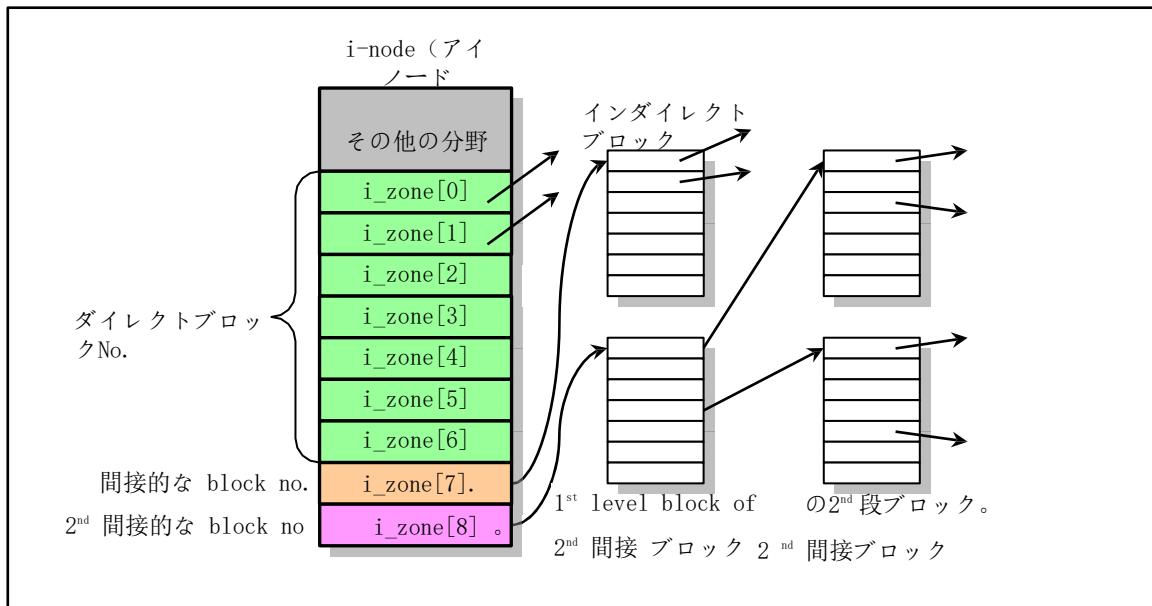


図12-23 i-nodeの論理ブロック接続の模式図

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 $(1024 \div 2) = 512$ 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ます。

12.4.2 コードアノテーション

プログラム 12-3 linux/fs/truncate.c

```

1 /*
2 * linux/fs/truncate.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6 // <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ

```

```

// の初期タスク0と、記述子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
// 数のマクロ文が含まれています。
// <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
造体を含む stat{}。
// そして、定数です。
7 #include <linux/sched.h>
8
9 #include <sys/stat.h> (日本語)
10
11 ///////////////////////////////////////////////////////////////////
12 // 全ての1間接ブロックを解放する。
13 // 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。
14 // 成功すれば1を、そうでなければ0を返します。
15 static int free_in(int dev, int block)
16 {
17     struct buffer_head * bh;
18    符号なしショート * p;
19     int i;
20     int block_busy; // ロジックブロックが解除されていません。
21
22 // 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。
23 // 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
24 // この一次間接ブロックのバッファブロックが解放されます。関数 free_block() は
25 // 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
26 // fs(bitmap.c
27     if (!block)
28         を返します。
29     block_busy = 0;
30     if (bh=bread(dev, block)) {...}
31         p = (unsigned short *) bh->b_data; // ブロックデータ領域を指す。
32         for (i=0;i<512;i++,p++) // 1つのブロックには512個のblock nr.
33             if (*p)
34                 if (free_block(dev,*p)) {...}
35                 *p = 0;
36                 bh->b_dirt = 1; // 修正フラグを立てる。
37             } else
38                 block_busy = 1; // ブロックが解放されない。
39         brelse(bh); // 間接ブロックが占有していたバッファブロックを解放する。
40     // 最後にデバイス上の間接ブロックを解放します。しかし、論理ブロックがあっても
41     // リリースされていない場合は0(失敗)を
42     // 返します。if (block_busy)
43         0を返す。
44     その他
45         return free_block(dev,block); // 成功すれば1、そうでなければ0を返す。
46 }
47
48 ///////////////////////////////////////////////////////////////////
49 // すべてのセカンダリ間接ブロックを解除します。
50 // パラメータdevはデバイス番号、'block'はセカンダリの論理ブロック番号
51 // 間接的なブロックです。
52 static int free_dind(int dev, int block)
53 {
54     struct buffer_head * bh;
55     unsigned short * p;
56     int i;

```

```

43     int block_busy; // ロジックブロックが解放されていない。
44
45     // 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。
46     // その後、コードは2次間接ブロックの第1レベルのブロックを読み込み、すべての
47     // その上の論理ブロックが使用されたことを示し、バッファブロックを解放します。
48     第一階層のブロックの // if
49         (!block)
50             を返します。
51             block_busy = 0;
52             if (bh=bread(dev, block)) {...}
53                 p = (unsigned short *) bh->b_data; // ブロックデータ領域を指す。
54                 for (i=0;i<512;i++,p++) // 512個の第2階層ブロック if (*p)
55
56                 if (free_ind(dev, *p)) // プライマリ間接ブロックを解除す
57                     {...}
58                     *p = 0; // 修正フラグを設定
59                     bh->b_dirt = 1;
60                 } else // 解放されていないブ
61                     brelse(bh); // 二次間接ブロックが占有してあるタッファブロックを解放する。
62     }
63
64     // 最後に、デバイス上の2次間接ブロックが解放されます。しかし、もし、ロジック
65     // ブロックが解放されていない場合は、0(失敗)を返す
66     if (block_busy)
67         0を返す。
68     その他
69         free_block(dev, block)を返します。
70
71     //// ファイルデータを切り捨てます。
72     // 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
73     // をリリースしました。
74     void truncate(struct m_inode * inode)
75     {
76         int i;
77         int block_busy;
78
79         // まず、指定されたi-nodeの有効性を判断します。通常のファイルでない場合は返します。
80         // ディレクトリ、またはリンクのエントリです。その後、i-nodeの7つのダイレクトブロックを解放
81         // し、7つの論理的なすべての
82         // ブロックアイテムをゼロにする。ブロックがビジー状態でリリースされていない場合は、
83         // block_busy フラグが設定されます。 if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode-
84         >i_mode) || S_ISLNK(inode->i_mode)))
85         // 繰り返す。
86         block_busy=0;
87         for (i=0;i<7;i++)
88             if (inode->i_zone[i]) // ブロック nr がゼロでなければリリ
89                 {... if (free_block(inode->i_dev, inode->i_zone[i]))
90                     inode->i_zone[i]=0;
91                 その他
92                     block_busy = 1; // リリースされていない場合に設定します。
93                 }
94         if (free_ind(inode->i_dev, inode->i_zone[7])) // プリモリーの間接ブロックを解放す
95             る inode->i_zone[7] = 0;
96     }

```

```

84      その
85      他      block_busy = 1; // リリースされていない場合に設定します。
86      if (free\_dind(inode->i_dev, inode->i_zone[8]))// 二次間接ブロックを解放する inode-
87          >i_zone[8] = 0;
88      その他
89          block_busy = 1;

// その後、i-node modifiedフラグが設定され、まだ論理ブロックがある場合は
// "busy"のためにリリースされない場合、現在のプロセス実行中のタイムスライスを0に設定して切
り替える
// を別のプロセスに渡して実行させ、しばらく待ってからリリース操作を再実行します。
// 最後に、ファイルの修正時間とi-nodeの変更時間が現在の時間に設定されます。
// マクロ CURRENT_TIME は、ヘッダファイル linux/sched.h の 142 行目で定義されており、以下の
90 ように定義されています。
91 // 1970:0:0:0からの秒数の値を(startup_time + jiffies/HZ)として。 inode-
92     >i_dirt current->counter = 0;           // 現在のプロセスのタイムスライスは0に設定
93     if (blockschedule());
94         goto リピート。
95     }
96     inode->i_size = 0; // ファイルサイズが0に設定されます。
97     inode->i_mtime = inode->i_ctime = CURRENT\_TIME
98 }          となります。
99
100

```

5. inode.c

1. 機能

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、指定されたノード番号`nr`のi-nodeを見つけ、ノードの参照カウント・フィールド値`i_count`をインクリメントする。

i-nodeの参照番号です。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ない場合、またはノード番号が指定されたノードと等しくない場合、スキャンを続ける。そうでなければ、指定されたデバイス番号とノード番号を持つi-nodeが見つかり、そのノードがロック解除されるのを待っている（もしも今ロックされているならば）。i-nodeがアンロックされるのを待っている間に、ノードテーブルが変更される可能性があります。そのため、現在、i-nodeのデバイス番号が指定されたデバイス番号と等しくない場合や、ノード番号が指定されたノード番号と等しくない場合は、i-nodeテーブル全体を再度スキャンする必要がある。続いて、i-nodeの参照カウント値を1インクリメントし、i-nodeが他のファイルシステムのマウントポイントであるかどうかを判断します。

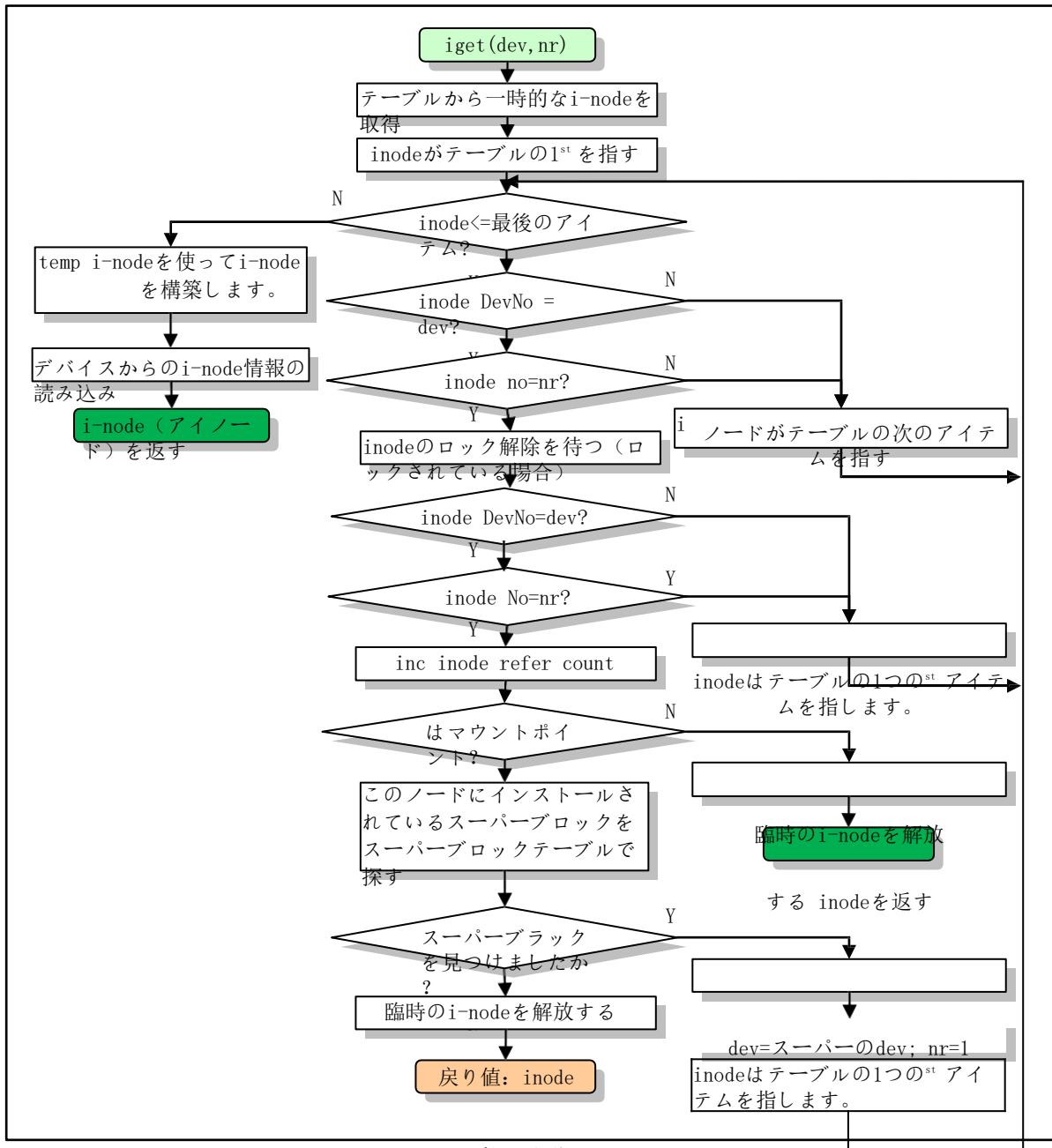


図12-24 i_get()関数の動作フローチャート

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します。

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ない場合は、対応するi-nodeが見つかったことを示すので、一時的に適用されているアイドルi-nodeはこの時点で破棄でき、見つかったi-nodeポインタが返されます。

指定されたi-nodeがi-nodeテーブルに見つからない場合は、前のアプリケーションのアイドルi-nodeを使用してi-nodeテーブルにノードを確立し、対応するデバイスからi-node情報を読み取ります。

i-nodeポインタが返されます。

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。

ある時刻にプロセスが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させるべきです。そのため、以下のいずれかの処理を行った後、カーネルコードは

通常は、iput()関数を呼び出します。

- i-nodeの参照カウントフィールドi_countの値を1つ増やします。
- namei()、dir_namei()、またはopen_namei()関数を呼び出しました。
- iget()、new_inode()、get_empty_inode()関数を呼び出しました。
- 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。
- ファイルシステムのアンマウント時（デバイスファイルのi-nodeの置き換えなど）。

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、プロセスが現在の作業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内で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

また、プロセスのルートと実行可能フィールドを指定するには、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置かれ、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る。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[]を処理して

は、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します。12.1節の図12-6を参照してください。前述のとおり、i_zone[0]～i_zone[6]にはファイルの直接論理ブロック番号が、i_zone[7]には間接論理ブロック番号が、i_zone[8]には二次間接論理ブロック番号が格納される。ファイルサイズが小さい（7K以下）場合は、ファイルが使用するディスクブロック番号をi-nodeの7つの直接ブロック項目に直接格納することができ、ファイルが少し大きい（7K+512K以下）場合は、プライマリ間接ブロック項目i_zone[7]を使用する必要があり、ファイルが大きい場合は、セカンダリ間接ブロック項目i_zone[8]が必要となる。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。
740

12.5.2 コードアノテーション

プログラム 12-4 linux/fs/inode.c

```

1 /*
2 * linux/fs/inode.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
// そして初期数を定め、記述子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
// <linux/sched.h> が含まれていますのヘッダーファイルでは、タスク構造体task_struct、データ
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネルの機能を利用しています。
// <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。
// ディスクリプタ/割り込みゲートなどが定義されています。 7 #include <string.h>
8 #include <sys/stat.h>
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h> (日本語)
12 #include <linux/mm.h>
13 #include <asm/system.h>
14
// デバイスデータブロックの総数を示すポインターの配列。各ポインタの項目は
// 与えられたメジャーデバイス番号の総ブロック数に、hd_sizes[].の各項目は
// ブロックアレイの合計数は、データブロックの合計数に対応します。
// サブデバイス番号で決まるサブデバイス。
15 extern int *blk_size[];
16
17 struct m_inode inode_table[NR_INODE]={{0,},}; // インメモリのi-nodeテーブル(NR_INODE=32)
// です。 18
18 static void read_inode(struct m_inode * inode); // 297行目のi-nodeの情報を読みます。
19 static void write_inode(struct m_inode * inode); // i-nodeの情報をキャッシュに書き込む、
324行目。 21
20
21 ///////////////////////////////////////////////////////////////////
22 // 指定したi-nodeが利用可能になるのを待ちます。
23 // i-nodeがロックされている場合、現在のタスクは中断されない待機状態になります。
24 // そして、i-nodeがロックを解除して明示的にタスクをウェイクアップするまで、i-waitキューに追
25 // 加されます。 25
26 static inline void wait_on_inode(struct m_inode * inode)
27     while (inode->i_lock)
28     {...           sleep_on(&inode->i_wait)           // kernel/sched.c, line 199.
29         }
30
31 ///////////////////////////////////////////////////////////////////
32 // i-nodeをロックする(与えられたi-nodeをロ
33 // ックする)。

```

```

// i-nodeがロックされている場合、現在のタスクは中断されない待機状態になり
// i-waitキューに追加される i_wait.i-nodeがロックを解除して明示的にタスクを起こすまで。
// そして、ロックします。
30 static inline void lock_inode(struct m_inode* inode)
31 {...
32     cli()です。
33     while (inode->i_lock)
34         sleep_on(&inode->i_wait);
35     inode->i_lock=1;                                // ロックされたフラグを設定します。
36     sti()です。
37 }
38
39     //// 指定された i ノードのロックを解除します。
40     // i-nodeのロックフラグをリセットし、i-waitで待っているすべてのプロセスを明示的にウェイクアップする
41     // queue i_wait.
42 static inline void unlock_inode(struct m_inode* inode)
43 {。    inode->i_lock=0となります。
44     wake_up(&inode->i_wait); // kernel/sched.c, line 204.
45 }
46
47     //// 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。
48     // メモリ上のi-nodeテーブル配列をスキャンし、指定されたアイテムが使用された場合はリリースします。
49     // デバイスです。
50 void invalidate_inodes(int dev)
51 {。    int i;
52     struct m_inode* inode;
53
54     // 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
55     // 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
56     // 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。
57     // もしイエスであれば
58     // が解除され、すなわち、i-nodeのデバイス番号フィールドi_devが0に設定され、修正された
59     // フラグもリセットされます。その間に、まだ使用されているかどうかもチェックされます、つまり
60     // その参照カウントが0でないかどうかを確認し、0の場合は警告メッセージを表示します。
61     // 50行目の「inode_table[0]」は、「inode_table」が開発初期のアイテムを指します
62     // "&inode_table[0]" MR MODE; の方がわかりやすいかもしれません。
63     { wait_on_inode(inode); // i-nodeが利用可能になる（アンロックされる
64         if (inode->i_dev == dev) {...} の待ちます。
65             if (inode->i_count) // referencesが0でない場合、エラー警告。
66                 printk("inode in use on removed disk\n");
67             inode->i_dev = inode->i_dirt = 0; // i-nodeをリリースし
68         } ます。
69     }
70
71     //// すべてのi-nodeを同期させます。
72     // 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。
73 void sync_inodes(void)
74 {
75     int i;

```

```

64     struct m_inode* inode;
65
66 // 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。
67 // 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
68 // ことで
69 // ロックが解除され（現在ロックされている場合）、i-nodeが修正されたかどうかをチェックして
70 // はパイプノードではありません。この場合、i-nodeはバッファキャッシュに書き込まれます。バ
71 // ッファはinode = 0+inode_table; // テーブルの最初のアイテムを指します。
72 // マネーfor(i=0; i<inode_table; i++)でディスクに書き込んでくれます。
73     { wait_on_inode(inode); // i-nodeが利用可能になるのを待ちま
74         if (inode->i_dirt && !inode->i_pipe) // 変更されており、パイプノードでは
75             write_inode(inode); // ありません。
76     }
77
78 ///// 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)。
79 // 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。
80 // 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
81 // 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
82 // グ
83 // 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
84 // されます。
85 // 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
86 // 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
87 // チェック、(2)ダイレクトブロック処理、(3)プライマリー間接ブロック処理、(4)セカンダリー
88 // 間接的なブロック処理です。
89
90 static int bmap(struct inode * inode, int block, int create)
91 {
92     int i;
93
94     // (1) まず、パラメータの有効性を確認します。ファイルのデータブロック番号が以下の場合
95     // 0の場合、カーネルは停止します。ブロック番号が (直接ブロック nr + 間接ブロック nr) より大
96     // きい場合は、カーネルは停止します。
97     // ブロック nr + 2 番目の間接ブロック nr) では、ファイルシステムの範囲外です。
98     // panic ("_bmap: block<0");
99     if (block >= 7+512+512*512)
100         panic ("_bmap: block>big")となります。
101
102 // (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
103 // 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
104 // 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
105 // 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
106 // る。
107 // と、ディスク上の論理ブロック番号が論理ブロックフィールドに記入されます。次にセット
108 // i-node change time and modified flag, and finally return the logical block number. 関数は
109 // new_block()は、bitmap.cプログラムの76行目で定義されています
110     if (block<7) {
111         if (create && !inode->i_zone[block])
112             if (inode->i_zone[block]=new_block(inode->i_dev)) {...}
113                 // inode->i_dtime=CURRENT_TIME, フラグを設置する変更時間
114             }
115         return inode->i_zone[block]となります。
116     }

```

```

// (3) ブロック番号が>>7で、(7 + 512)より小さい場合は、間接的に使用していることを意味する
// ブロックを作成します。そのため、以下では間接ブロックが処理されます。もしそれが作成操作で
// i-nodeの間接ブロックフィールドi_zone[7]が0の場合、そのファイルが
// 間接ブロックを初めて使用することになります。そのため、ディスクブロックを申請するためには
// 間接的なブロック情報を格納するために、 // 実際のディスクブロック番号を
// 間接ブロックフィールド。そして、i-node modified flagとmodification timeを設定します。も
// し、ディスクの
// ブロックの作成時に失敗すると、i-nodeの間接ブロックフィールドi_zone[7].
// はこの時点では0であり、0が返されます。または、パラメータ作成フラグは0だが、i_zone[7]の
// も0で、i-nodeに間接ブロックがないことを示しているので、マッピングディスク
91 // ブロックは失敗し、0を返して終了しま
92     ブロック -= 7;
93     if (block<512) {...}
94         if (create && !inode->i_zone[7])
95             if (inode->i_zone[7]=new block(inode->i_dev)) {
96                 inode->i_dirt=1;
97                 inode->i_ctime=CURRENT TIMEです。
98             }
99         if (!inode->i_zone[7])
100            0を返す。
101 // 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
102 // 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
103 // 合
104 // 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
105 // 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。
106 // 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
107 // の操作では、'i'はマッピング（検索）する必要のある論理ブロック番号です。最後に
108 // 間接ブロックが占有していたバッファブロックが解放され、論理ブロック番号が
109 // 新しいアプリケーションを返すまたはディスク上の対応するブロックが返されます。
110 // if((bh->high>>1&0x00000001)&bh->b_data[7]))
111     if (create && !i)
112         if (i=new block(inode->i_dev)) {...}
113             ((unsigned short *) (bh->b_data))[block]=i;
114             bh->b_dirt=1となります。
115         }
116     ブレルス(bh)です。
117     i.を返します。
118 }

```

// (4) プログラムがこの場所まで走ると、そのデータブロックがセカンダリに属することを示します。

```

//間接ブロックです。その処理は、主な間接ブロックと同様です。以下は
// 二次間接ブロックの処理について。まず、このブロックは数値から差し引かれて
// プライマリ間接ブロックに収容されているブロック（512）の//を作成または検索して
// 作成フラグが設定されているかどうかで判断します。作成操作で、セカンダリの
// iノードの間接ブロックフィールドが0の場合、プライマリを格納するために新しいディスクブロッ
// クが必要となる
// セカンダリ間接ブロックのブロック情報で、実際のディスクブロック番号が記入されている
// を2次間接ブロックフィールドに設定します。その後、i-node modified flagを設定して
// 修正時間です。同様に、ディスクブロックの作成時に失敗すると
// その場合、i-nodeの2次間接ブロックフィールドi_zone[8]は0であり、0が返されます。あるいは、
111 それが
112 // 作成操作ではないが、i_zone[8]が0になり、間接的なものがないことが判明する
113 // i-nodeにブロックがあるので、マップされたディスクブロックは失敗し、0を返して
114 終了する。 block -= 512;
115     if (create && !inode->i_zone[8])    744

```

```

113         if (inode->i_zone[8]=new_block(inode->i_dev))
114             {...     inode->i_dirt=1;
115                 inode->i_ctime=CURRENT_TIME
116                     です。
117             if (!inode->i_zone[8])
118                 0を返す。
// 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
セカンダリー・インダイレクト・ブロックの // 第1レベルのブロックをリリースします。
119 // ブロックを作成します。もし作成されていなければ、' i' は見つける必要のある論理ブロック番号
120         です。 if (!(bh=bread(inode->i_dev, inode->i_zone[8])))
121             0を返す。
122         i = ((unsigned short *)bh->b_data)[block>>9];
123         if (create && !i)
124             if (i=new_block(inode->i_dev)) {...  

125                 ((unsigned short *) (bh->b_data))[block>>9]=i;
126                 bh->b_dirt=1;
127             }
プレルス (bh) です。
// 2次間接ブロックの第2レベルのブロック番号が0の場合は
// アプリケーションのディスクブロックが故障するか、元の対応するブロック番号が0であれば、0は
// を返します。それ以外の場合は、2次間接ブロックの第2レベルのブロックを読み込んで
128 // デバイスで、2段目のブロックのブロックアイテムにある論理ブロック番号を取る。 if (!i)
129         0を返す。
130         if (!(bh=bread(inode->i_dev, i)))
131             return 0;
132         i = ((unsigned short *)bh->b_data)[block&511]; // 511(0x1ff)に制限。
// 作成フラグが設定されていて、第2階層のブロックの論理ブロック番号が
// ブロックが0であれば、データ情報を最終的に格納するブロックとしてディスクブロックを適用し
// ます。
// そして、第2レベルのブロックのブロックアイテムを、新しい論理ブロック番号「i」と同じにしま
// す。
// そして、第2レベルのブロックの修正フラグを設定します。最後に、第2レベルのブロックの
133 // セカンダリ間接ブロックが解放され、新たに適用された論理ブロック番号の
134 // またはディスク上の元の対応するブロックが返される if
135     (create && !i)
136         if (i=new_block(inode->i_dev)) {...  

137             ((unsigned short *) (bh->b_data))[block&511]=i;
138             bh->b_dirt=1;
139         }
140     } brelse(bh);
141     return i;
//// デバイス上のファイルデータブロックの論理ブロック番号を取得します。
// パラメータ: inode - ファイルのi-nodeポインタ、block - ファイルのデータブロック番号。
// 操作が成功した場合は、論理ブロック番号を返し、そうでない場合は0を返します。
142 int bmap(struct m_inode * inode, int block)
143 {

```

```

144         return bmap(inode, block, 0);      // 作成フラグ=0。
145 }
146
147     //// 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。
148     // デバイス上のファイルデータブロックの対応するディスクブロックを探します。存在していない場合
149     // 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。
150     // パラメータ: inode - ファイルのi-nodeポインタ、block - ファイルのデータブロック番号。
151     // 操作が成功した場合は、論理ブロック番号を返し、そうでない場合は0を返します。
152     int create_block(struct m_inode* inode, int block)
153     {。      return bmap(inode, block, 1);
154
155     //// i-nodeを戻す（デバイスに書き戻す）。
156     // 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
157     // パイプi-node、待機中のプロセスを起こす。ブロックデバイスファイルのi-nodeであれば、デバイスの
158     // が更新され、i-nodeのリンクカウントが0であれば、すべてのディスク論理ブロックが占有されます。
159     // i-nodeによる//が解除され、i-nodeが解除されます。
160     void input(struct m_inode* inode)
161     {。
162
163         // 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
164         // 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
165         // はすでにアイドル状態になっています。カーネルは次に、それに対してput back操作を要求しま
166         if (!inode)
167             // カーネル内のfree_inode問題があることを示しています。その後、エラーメッセージが表示され、マ
168             wait_on_inode(inode);
169             // がシャットダウンされます
170             panic("input: trying to free free inode")が発生します。
171             // パイプのi-nodeであれば、パイプを待っているプロセスを起こして、参照の数
172             // が1デクリメントされ、それでも参照カウントが0にならない場合は、関数が返されます。
173             // それ以外の場合は、パイプラインが占有していたメモリページが解放され、参照回数が増えます。
174             // ノードのモディファイドフラグ、パイプフラグがリセットされてから、関数が返される。パイプの
175             // 場合
176             // i-nodeのi_node->i_ispipeは、メモリページアドレスが格納されます。get_pipe_inode()の231, 237
177             行目を参照してください。
178             wake_up(&inode->i_wait)です。
179             wake_up(&inode->i_wait2)です          //
180             。
181             if (--inode->i_count)
182                 を返すことができます。
183             free_page(inode->i_size)です
184             。
185             inode->i_count=0です。
186             inode->i_dirt=0;
187             inode->i_pipe=0;
188             // i-nodeのデバイス番号アドレスが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
189             // を1つずつ返していくきます。例えば、パイプライン運用のi-nodeは、デバイス番号が
190             i-nodeの場合は、//0です。さらに、ブロックデバイスファイルのi-nodeであれば、その中の
191             // 論理ブロックフィールド0(i_zone[0])がデバイス番号の場合、デバイスをリフレッシュして待つ
192             のためにi-nodeのロックを解除し
193             //。 ます。
194            もし (!inode->i_dev) {
195                 inode->i_count--
196             。

```

```

172         を返す
173     }    ことが
174     if (S_ISDIR(inode->i_mode)) {
175         sync_dev(inode->i_zone[0]);
176         wait_on_inode(inode);
177     }
178 // 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
179 // 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
180 // i-nodeの参照カウント値は1（157行目でチェックされているので
181 // それは0である）。i-nodeのリンク数が0であれば、対応する
182 // ファイルが削除されます。その後、i-nodeの全ての論理ブロックがリリースされ、i-nodeがリリースされます。
183 // 関数 free_inode() は、実際に i-node の操作を解放するために使用されます、つまり、リセット
184 // 繰り返す。// i-node の bitmap マップを返すと、および i-node 構造のコンテンツをクリアします。
185     inode-> i_count--
186     ; return;
187     }
188     if (!inode->i_nlinks) {...           // この時点で、i_count = 1です
189         truncate(inode);                 .
190         free_inode(inode);             // bitmap.c, line 108.
191         を返すことができます。
192     // 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
193     // ロックを解除する（ロックされている場合）。i-node を書き込むときにスリープする
194     // ことのある場合があります。この時点では i-node は // になっていたので、上述のチェック手順（ラベルリピート）が必要です。
195     // は、プロセスが起動された後、再び繰り返されます。
196     // write_inode(inode) です。 /* we can sleep - so do again */
197     // wait_on_inode(inode) です
198     .
199 // 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
200 // が 0 ではなく、内容が変更されていないことを示しています。そのため、i-node の参照がある限り
201 // count を 1 だけデクリメントすれば、戻すことができます。このとき、i_count=0 となっているのは
202 // i-node がリリースされました。inode->i_count-
203     -; return;
204 }
205
206 // 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。
207 // 参照カウントが 0 の i-node を探し、クリアしてディスクに書き込み、それを返します。
208 // ポインタです。これで、参照カウントが 1 になりました。
209 struct m_inode* get_empty_inode(void)
210 {
211     struct m_inode* inode;
212     static struct m_inode* last_inode = inode_table; // 最初のアイテムを指します
213     int i;
214
215     // 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。
216     // 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
217     // のテーブルをループし続けるために、i-node テーブルの先頭に再ポイントさせます。
218     // テーブルになります。last_inode が指す i-node の参照カウントが 0 であれば、それは

```

```

// アイドル状態のi-nodeアイテムが見つかった場合、inodeをi-nodeに指します。もし、修正された
// フラグとi-nodeのロック・フラグの両方が0であれば、i-nodeを使用することができるので、終了
// しません()のループになっています。
203     do {
204         inode = NULL;
205         for (i = NR_INODE; i ; i--) { // NR_INODE = 32。
206             if (++last_inode >= inode_table + NR_INODE)
207                 last_inode = inode_table;
208             if (!last_inode->i_count) {...}
209                 inode = last_inode;
210                 if (!inode->i_dirt && !inode->i_lock)
211                     ブレークします。
212             }
213         }
214         // フリーのi-nodeが見つからない場合 (inode = NULL) 、デバッグ用にi-nodeテーブルが表示されます。
215         // となり、システムを停止します。
216         for (i=0 ; i< NR_INODE ; i++)
217             printf ("%04x: %6d¥t", inode_table[i].i_dev,
218                         inode_table[i].i_num) となります。
219             panic ("No free inode in mem") が発生します。
220         }
221         // そうでなければ、i-nodeのロックが解除されるのを待つ（再びロックされた場合）。i-nodeがフラグを変更した場合
222         // がセットされると、i-nodeはリフレッシュ（同期）されます。リフレッシュ時にはスリープする可能性があるため
223         // wait_on_inode(inode);
224         while (inode->i_dirt) {
225             write_inode(inode);
226             wait_on_inode(inode);
227         }
228         } while (inode->i_count);
229     }
230
231     ///// パイプのi-nodeを取得します。
232     // 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。
233     // をパイプラインが使用するようにします。そして、取得したi-nodeの参照カウントを2にする（
234     // reader
235     // とライター）、パイプの頭と尻尾が初期化され、パイプ型表現の
236     // i-nodeを設定します。i-nodeへのポインタを返し、失敗した場合はNULLを返します。
237     struct m_inode* get_pipe_inode(void)
238     {
239         struct m_inode* inode; 234
240         // 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
241         // 。
242         // 1にします。そして、i-nodeのためにメモリのページを申請し、ノードのi_sizeフィールドが指すようになります。
243         // をページに追加します。空きメモリがない場合48、i-nodeは解放され、NULLが返されます。

```

```

235      も (! (inode = get_empty_inode()))
236      し
237      も (! (inode->i_size=get_free_page())の場    // i_size はバッファページを指しま
238      し 合) {
239          inode->i_count = 0;
240          // そして、i-nodeの参照カウントを2にして、パイプの先頭と末尾のポインタをリセットします。
241          // 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
242          // パイプのヘッダとパイプのテールポインタをそれぞれ指定します。最後に、i-nodeフラグを
243          // i-nodeのタイプをパイプして、i-nodeを返します。
244          inode->i_count = 2; /* 読者/ライターの合計 */。
245          PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0; // パイプの頭と尻をリセットする。
246          inode->i_pipe = 1; // パイプのi-node.
247      }
248
249      ///////////////////////////////////////////////////////////////////
250      // デバイスからi-nodeを取得します。
251      // パラメータ: dev - デバイス番号、nr - i-node番号。
252      // 指定したi-node番号のi-node構造体の内容をデバイスから読み込んで
253      // 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
254      // 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-
255      // nodeの
256      // ポインタはいくつかのチェック処理の後に返されます。それ以外の場合は、デバイスから i-node
257      // を読み込みます。
258      // そして、i-nodeテーブルに配置され、i-nodeポインタが返されます。
259      struct m_inode * i_get(int dev, int nr)
260  {
261      ///////////////////////////////////////////////////////////////////
262      // まず、i-node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
263      // カーネルコードに問題がある場合は、エラーメッセージを表示して停止します。そうでなければ、
264      // アイドル状態のi-nodeが取られます。
265      // バックアップのためにi-nodeテーブルからあらかじめ//を入れておきます。その後、i-nodeテーブル
266      // 全体をスキャンし、以下を探します。
267      // 指定されたノード番号nrを持つi-nodeを//し、参照番号を1つ増やします。
268      // ただし、現在スキャンされているi-nodeのデバイス番号が、指定された
269      // デバイス番号またはノード番号が、指定されたノード番号と等しくない場合、スキャン
270      // を継続しています。
271      if (!dev)
272          inode = inode_table; // i-nodeテーブルの最初の項目を指します。 while (inode <
273          NODE_TABLE_NODE) {
274              if (inode->i_dev != dev || inode->i_num != nr) {
275                  inode++;
276              }
277              // 続けています。
278          }
279
280          // 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
281          // 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
282          // ルが変更される可能性があります。そのため
283          // コードが継続して実行される際にも、上記と同じチェックが必要です。もし変更が
284          // wait_on_inode(inode);
285          if (inode->i_dev != dev || inode->i_num != nr) {...}
286          inode = inode_table;
287          // 続けています。
288      }
289
290      // 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blockテーブルが検索されます。
// このi-nodeにインストールされています。スーパーblockが見つからない場合は、エラーメッセージが表示されました。
// と、関数の最初に取得したアイドルノードemptyを戻して、i-node

265         inode->i_count++です。
266         if (inode->i_mount) {...}
267             int i;
268
269             for (i = 0 ; i< NR_SUPER ; i++)
270                 if (super_block[i].s_imount==inode)
271                     ブレークします。
272             if (i >= NR_SUPER) {...}
273                 printk ("Mounted inode hasn't got sb\n")
274
275             If (empty)
276                 iput(empty)です。
277 // ここまでこのコード実行では、マウントされたファイルシステムノードのスーパーblockにインストールされた
278 // i-node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
279 // デバイス番号を、i-nodeにインストールされているスーパーblockから取得したものとし、i-
280 node
281 // 必要な数はROOT_INOに再設定され、1となります。
282 // マウントされたファイルシステムのルートi-node情報を取得します。
283         iput(inode)です。
284         dev = super_block[i].s_dev;
285         nr = ROOT_INO;
286         inode = inode_table;
287         continue;
288     }
289
290     // 最後に対応するi-nodeを見つけました。したがって、アイドルのi-nodeを破棄することができます
291     // この関数の最初に一時的に適用され、見つかったi-nodeを返す。 if (empty)
292         iput(empty);
293         return inode;
294     }
295
296     // 指定されたi-nodeがテーブルに見つからない場合、そのi-nodeは以下の方法でテーブルに作成さ
297     // れます。
298     // 前のアプリケーションのアイドルi-node 'empty'を使用して、i-nodeの情報は
299     // 対応するデバイスから読み込んで、i-nodeのポインタを返す if (!empty)
300         return (NULL);
301
302         inode=empty;
303         inode->i_dev = dev;           // 新しいi-nodeのデバイスを設定しま
304         inode->i_num = nr;          // す。
305         read_inode(inode);        // i-nodeの番号を設定します。
306         return inode;
307     }
308
309     //// 指定されたi-nodeの情報を読み取る。
310     // 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
311     // 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
312     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blockの
313     // を計算するためのINODES PER BLOCKの情報を得るために、最初に読み込まなければなりません。

```

```

// ブロック番号です。i-nodeが配置されている論理ブロック番号を計算した後に
// の論理ブロックがバッファブロックに読み込まれ、その時点でのi-nodeの内容は、次のようになり
// ます。
// バッファブロックの対応する位置が、指定した位置にコピーされます。
297 static void read_inode(struct m_inode * inode)
298 {...    struct super_block * sb;
299     struct buffer_head * bh;
300     int block;
301
302 // 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blockを取得します。
303     if (!(sb=get_super(inode->i_dev)))
304         panic("try to read inode without dev")となります。
305 // i-nodeが配置されているデバイスの論理ブロック番号 = (ブートブロック + super
// ブロック) + iノードのビットマップブロック + 論理ブロックのビットマップブロック + (iノード
// 番号-1) / iノード
// ブロックごとに、図12-1を参照してください。i-nodeの番号は0から順番になっていますが、最初
// の
// ノード0は使用されておらず、対応するノード0構造はディスクに保存されていません。そのため
// i-nodeを格納する最初のディスクブロックは、i-node番号を持つi-node構造を保持します。
// 0--31ではなく1--32です。を計算する際には、1だけデクリメントする必要があります。
// i-node番号に対応するi-nodeのディスクブロック番号。ここでは、論理的な
// デバイスからi-nodeがあるブロックを取り出し、i-nodeの内容をコピーして
306 // inode ポインタで示される場所です。
307     block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
308             (inode->i_num-1)/INODES_PER_BLOCK;
309     if (!(bh=bread(inode->i_dev, block))
310         panic("unable to read i-node block")
311     となります。)
312     *(struct d_inode *)inode =
313         ((struct d_inode *)bh->b_data)
314             [(inode->i_num-1)%INODES_PER_BLOCK]となります。
315 // 最後に、読み込まれたバッファブロックが解放され、i-nodeのロックが解除されます。ブロックデ
316 バイスの場合
317 // 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。
318     brelse(bh);
319     if (S_ISBLK(inode->i_mode)) {...}
320         if i = inode->i_zone[0]; // i_zone[0]にはデバイスファイルのdev nrが入っ
321         then inode->i_size = 0xffffffff;
322         }                     inode->i_size = 1024*blk_size[MAJOR(i)][MINOR(i)]。
323     unlock_inode(inode)
324 }      です。
325
326 // // i-nodeの情報をバッファブロックに書き込む。
327 // この関数は、指定されたi-nodeをキャッシュのバッファブロックに書き込んで
328 // キャッシュが更新されたときにディスクへの論理ブロック番号を決定するためには
329 // i-nodeがあるデバイスでは、まずそのデバイスのスーパーblockを読む必要があります。
330 // 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
331 // がバッファブロックに読み込まれ、i-nodeの内容が対応する
332 // バッファブロックの位置。
333 static void write_inode(struct m_inode * inode)
334 {
335     struct super_block * sb;
336     struct buffer_head * bh;

```

```

328      intブロック
329      ク。
// i-nodeを最初にロックします。i-nodeが変更されていない場合や、デバイス番号が
// i-nodeが0に等しい場合、i-nodeはロック解除されて終了します。されていないi-nodeについては
// 修正された場合、その内容はバッファやデバイスにあるものと同じです。その後、スーパーブロック
// を取得
330 i-node の // 。
331     lock(inode)で!inode->i_dev) {
    し。
332         unlock_inode(inode)です。
333         を返すことができます。
334     }
335     も (! (sb=get_super(inode->i_dev)))
    し panic("try to write inode without device")が発生します。
// そして、i-nodeが配置されている論理ブロックをデバイスから読み込んで、コピーします。
// i-nodeの情報を、i-nodeアイテムの位置に対応する論理ブロックに格納する。block = 2 + sb-
337     >s_imap_blocks + sb->s_zmap_blocks +
338         (inode->i_num-1)/INODES_PER_BLOCK;
339     if (! (bh=bread(inode->i_dev, block)))
340         panic("unable to read i-node block");
341     ((struct d_inode *)bh->b_data)
342         [(inode->i_num-1)%INODES_PER_BLOCK] =
            *(struct d_inode *)inode;
// その後、バッファモディファイドフラグを設定し、i-nodeのコンテンツが既に一致しているので
// バッファに格納されているので、i-node dirtyフラグは0になります。最後に、i-nodeを含むバッ
// ファは
344 //が解除されb_dinodeがなり戻されます。
    す。
345     inode->i_dirt=0;
346     ブレルス(bh)です。
347 }
348     unlock_inode(inode)
    です。
349

```

12.5.3情報

なし :)

6. super.c

1. 機能

super.cプログラムには、ファイルシステムのスーパーブロックを操作する関数が含まれている。これらの関数はファイルシステムの低レベル関数に属し、主にget_super()、put_super()、read_super()といったファイル名やディレクトリ、パスを扱う上位関数で使用されます。また、ファイルシステムのロード/アンロードのシステムコールsys_umount()とsys_mount()、そしてルートファイルシステムのロード関数mount_root()があります。他のいくつかの補助関数はbuffer.cのものと似ています。

スーパーブロックは、主にファイルシステム全体の情報を格納する。その構造については、12.1項「全体的な機能の説明」の図12-3を参照してください。

get_super()関数は、指定されたデバイスの条件で、メモリのスーパーブロック配列の中から該当するスーパーブロックを検索し、そのポインタを返す関数である。そのため、指定されたデバイスで

関数が呼び出された場合、対応するファイルシステムがマウントされているか、少なくともスーパー ブロックがスーパー ブロック配列のエントリを占めていなければならず、そうでなければNULLを返します。

`put_super()`は、指定されたデバイスのスーパー ブロックを解放するための関数である。スーパー ブロックに対応するファイルシステムのi-nodeビットマップと論理ブロックビットマップが占有するバッファブロックを解放し、スーパー ブロックテーブル（配列）`super_block[]`の対応する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われ、`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われます。プログラムの最後の関数である`mount_root()`は

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。具体的には、`mount_root()`を図12-25に示す。

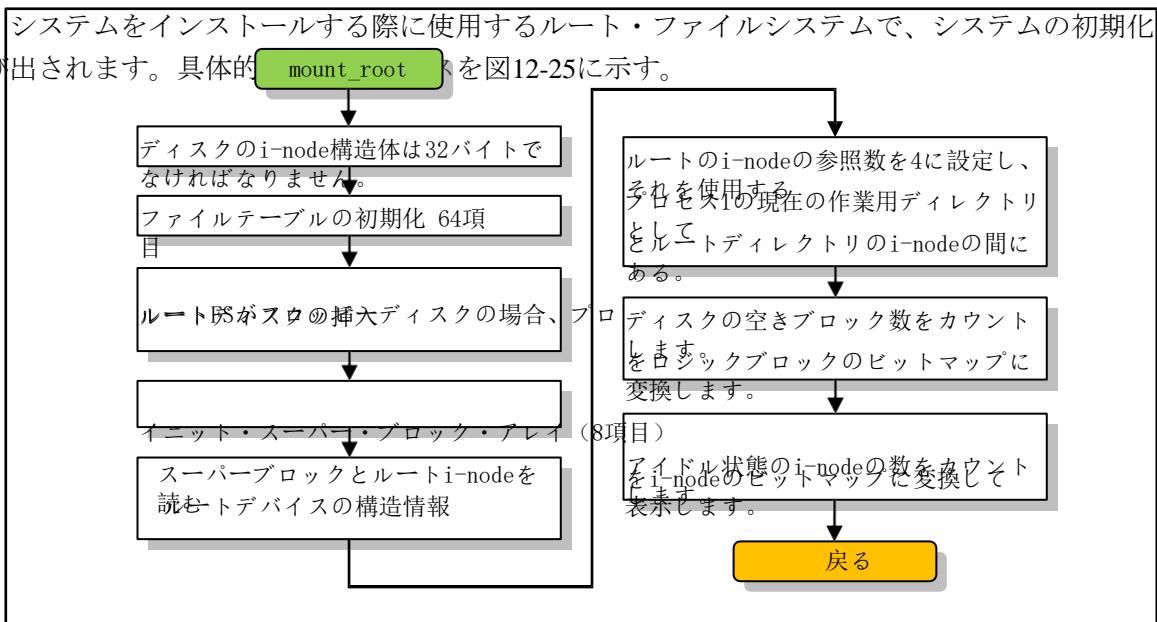


図12-25 `mount_root()`関数の目的

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 コードアノテーション

プログラム 12-5 `linux/fs/super.c`

```

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

```

```

3 /*
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * super.c には、スーパー ブロックのテーブルを扱うコードが含まれています。
9 */
// <linux/config.h> カーネル設定用のヘッダーファイルです。キーボード言語やハードディスクを定義する
// 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。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネルの使用する機能
// <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ーファイルです。システムの様々なエラー番号を含みます。
// <sys/stat.h> ファイル状態のヘッダーファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
// そして、定数です。
10 #include <linux/config.h>
11 #include <linux/sched.h>
12 #include <linux/kernel.h> (日本語)
13 #include <asm/system.h>
14
15 #include <errno.h>
16 #include <sys/stat.h>
17
    // 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)
18 int sync_dev(int dev);
19 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、以下のようになっています。
    // マクロは、ビット設定をテストして返すだけで、ビットには何の変更も加えていません（そのため、名前は
    // test_bit()の方が適切かもしれません）。
    // 入力: %0 - ax(res); %1 - eax(0); %2 - bitnr, ビットオフセット値; %3 - (addr), 開始アドレス。
    // 23行目では、ローカルレジスタ変数「res」を定義しています。この変数はeaxに保存されます。
    // レジスターへの効率的なアクセスと操作を実現します。マクロ定義全体は、コメントで示されています。
    // 値が最後の「res」の値である式。24行目のBT命令は
    // ビットのテストに使われる//。これは、アドレス addr (%3) で指定されたビットの値を
    // ビットオフセット bitnr (%2) をキャリーフラグ CFに入れます。SETB命令を使用すると
    // オペランド %al をキャリーフラグ CFに従います。CF = 1の場合、%al = 1、そうでない場合は
    %al = 0となります。
27 struct super_block super_block[NR_SUPER]; // スーパーブロックテーブル配列(NR_SUPER =
28 #define set_bit(bitnr,addr) 23 /* これは init/main.c で初期化されます */ 8)。
register int res asm("ax");
29 int ROOT\_DEV = 0;
30 asm ("bt %2,%3;setb %%al":= "a" (res): "a" (bitnr), "r" (*addr));
31 res; })
25 //////////////////////////////////////////////////////////////////
32 //////////////////////////////////////////////////////////////////
33 // スーパーブロックをロックします。

```

```

// スーパーブロックがロックされている場合、現在のタスクは中断されない待機状態になります
// 状態になり、スーパーブロックの待ち行列に追加される スーパーブロックのロックが解除される
まで s_wait
// タスクが明示的に起動されます。その後、ロックします。
// 以下の3つの関数 (lock_super()、free_super()、wait_on_super()) には
// inode.cファイルの最初の3つの関数と同じ効果がありますが、オブジェクトが
// ここで操作されるのは、スーパーブロックに置き換
えられます。 31 static void lock_super(struct
super_block *sb)
32 {           while (sb->s_lock)           // 割り込み禁止
33             sleep_on(&(sb->s_wait));    // スーパーブロックがすでにロックされてい
34         sb->s_lock = 1;                 // たら、スリープします。
35         sti();                         // kernel/sched.c, line 199.
36     }                                     // lock it!
37                                         // 割り込みを有効にする
38 }
39
//// 指定したスーパーブロックのロックを解除します。
// スーパーブロックのロックフラグをリセットし、以下を待っているすべてのプロセスを明示的にウ
エイクアップします。
// この待ち行列 s_wait (unlock_super()) という名前を使えば、より適切かもしれません)。
40 static void free_super(struct super_block * sb)
41 {...   cli() です。                   // ロックをリセットします。
42     sb->s_lock = 0;                  // kernel/sched.c, line 188.
43     wake_up(&(sb->s_wait))を行いま
す。
44     sti() です。
45 }
46
47 //// スーパーブロックが解除されるのを待つ眠り。
// スーパーブロックがロックされている場合、現在のタスクは中断されない待機状態になります
// 状態になり、スーパーブロックの待ち行列 s_wait に追加されます。スーパーブロックがロックを
解除するまで
// と明示的にタスクを起こします。
48 static void wait_on_super(struct super_block * sb)
49 {...   cli() です。
50     while (sb->s_lock)           // スーパーブロックがすでにロックされてい
51             sleep_on(&(sb->s_wait))    // たら、スリープします。
52        となります。
53     sti() です。
54 }
55
//// 指定されたデバイスのスーパーブロックを取得します。
// 指定したデバイスdevのスーパーブロック構造をスーパーブロックテーブルで検索する
// (配列)。見つかった場合はスーパーブロックへのポインタを、そうでない場合はNULLポインタ
を返します。
56 struct super_block * get_super(int dev)
57 {
58     struct super_block * s; // スーパーブロックのデータ構造ポインタ。
59 // まず、パラメータで与えられたデバイスの有効性をチェックします。デバイス番号が0の場合
// nullが返されます。次に、スーパーブロック配列の先頭を指定し、次のように開始します。
// スーパーブロック配列全体を検索して、指定したデバイスdevのスーパーブロックを見つける if
60     (!dev)
61         NULLを返す。
62     s = 0+super_block;
63     while (s < NR_SUPER+super_block)
// 現在の検索語が、指定されたデバイスのスーパーブロックである場合、すなわちデバイス番号

```

スーパー・ブロックの // フィールド値が、関数パラメータで指定されたものと同じであれば
// スーパー・ブロックは、(他のプロセスによってロックされていた場合は) ロック解除されるのを待ちます。その間
スーパー・ブロックアイテムは、他のデバイスで使用される可能性があるため、// 待機時間
// スリープが解除された後、指定されたスーパー・ブロックであるかどうかを再度確認します。
// デバイスです。そうであれば、スーパー・ブロックのポインタを返し、そうでなければ、スーパー・ブロックを検索します。

```

64 // if (s->s_dev == dev) {
65     // wait_on_super(s); if
66     // (s->s_dev == dev)
67     //     return s;
68     //     s = 0+super_block;
69
70 // 現在の検索語がない場合は、次の項目をチェックします。最後に、もし指定されたスーパー・ブロックが
71 // が見つからない場合は、NULLポインタが返されます。
72 } else
73     s++;
74
75 // NULLを返す。
76
77 ///////////////////////////////////////////////////////////////////
78 // 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。
79 // 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
80 // デバイスのinodeビットマップとロジックブロックビットマップによって占有されるバッファキャッシュブロック。もし
81 // スーパー・ブロックに対応するファイルシステムは、ルートファイルシステム、または他のファイル
82 // システムである
83 // がそのiノードのいずれかにすでにインストールされている場合、スーパー・ブロックを解放するこ
84 // とはできぬ。super_block * sb;
85 void put_super(int dev)
86 {
87
88 // 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
89 // 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
90 // イルを探して
91 // 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
92 // ブロックが
93 // ヌルイの関数を呼び出す前に、NULLポインタを参照かたが処理されていないことを示す、警告メッセー
94 // ジ
95     if (dev == ROOT_DEV) {
96
97 // が表示されて返されます。root disk reattached? prepare for damaged mount には
98         printk("root disk reattached? prepare for damaged mount");
99
100 // を返すことができます。
101 }
102
103 if (!(sb = get_super(dev)))
104     // を返すことができます。
105 if (sb->s_imount) {
106     printk("Mounted disk changed - tssk, tssk\n");
107
108 // を返すことができます。
109 }
110
111 // 指定されたデバイスのスーパー・ブロックを見つけたら、まずそれをロックして、次にその
112 // 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
113 // ロックを解放する。
114 // 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。
115 // 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
116 // ックを
117 // 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
118 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。
119 // 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
120 // 容が
121 // が変更された場合、バッファにデータを書き込むための同期操作が必要になります。
122 // ブロックをデバイスに送信します。この関数は最後にスーパー・ブロックのロックを解除して戻りま
123 // す。
```

```

89      lock_super(sb) です。
90      sb->s_dev = 0;                                // スーパーブロックをアイドル状態
91      for(i=0;i< I_MAP_SLOTS;i++)                  にする。
92      brelse(sb-> s_imap[i]) となります
93      for(i=0;i< Z_MAP_SLOTS;i++)
94      brelse(sb-> s_zmap[i]);
95      free_super(sb) です。
96      を返すことができます。
97 }/// 指定されたデバイスのスーパーブロックを読み込む。
98 // 指定されたデバイスdevのファイルシステムのスーパーブロックが、すでにスーパーブロックテーブルにある場合。
// スーパーブロックエントリへのポインタが直接返されます。それ以外の場合は、スーパーブロックが読み込まれます
// デバイスデブからバッファブロックに入り、スーパーブロックテーブルにコピーされます。最後に
// スーパーブロックのポインタを返します。
100 static struct superblock*read_super(int dev)
101 {...    struct buffer_head * bh;
102     int i,block;
103
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
// (つまりフロッピーデバイスかどうか)を確認します。ディスクが交換された場合、ディスクの中
// のすべてのバッファブロックが
// バッファキャッシュが無効であり、無効化する必要がある、つまりロードされたファイルシ
105 ステムの
106 //が解除されます。
107     if (!dev)
108         return NULL;
109     check_disk_change(dev);
110 // 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
111 // へのポインタ
112 // が直接返されます。そうでない場合は、スーパーブロックの配列の中で、空いている項目を探しま
113 // す(つまり
114 // フィールドを持つアイテム s_dev=0).配列がすでにいっぱいになっている場合は、ヌルポイ
115 // ンターを返します。
116 // を返します。
117     for (s = 0+super_block;; s++) {
118         if (s >= NR_SUPER+super_block)
119             return NULL;
120         if (!s->s_dev)
121             s->s_dev = dev; // デバイスが登録され、その情報がデバイスからバッファに読み込まれる
122         // 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。
123         // 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
124         // イテムは
125         // 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
126         // ルポインタ
127         // 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
128         // から
129         // 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

```

```

// 情報を公開します。
122     lock_super(s)
123     if (! (bh = bread(dev, 1))) {
124         s->s_dev=0;
125         free_super(s);
126         return NULL;
127     }
128     *((struct d_super_block *) s) =
129     *((struct d_super_block *) bh->b_data)
    となります。
130 // デバイスから既存の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
131 // 上記で選択された s->magic[0..1] == SUPER_MAGIC の配列が解除され、アイテムのロックが解除されて、空の
// ポインタが返されます。このバージョンの Linux カーネルでは、MINIX ファイルシステムのバ-
132 ジョンのみが s->s_dev = 0;
133 // 1.0 がサポートする s->dev の(s)マジックナンバーは 0x137f です。
134         NULL を返す。
135     }

// 以下、デバイス上の i-node ビットマップとロジックブロックビットマップデータの読み込みを開始
します。
// まず、メモリスーパー ブロック構造のビットマップ空間を初期化します。次に、i-node のビットマ-
ップ
// と論理ブロックのビットマップ情報は、デバイスから読み込まれて
// スーパー ブロックの対応するフィールドです。i-node のビットマップは、論理ブロックの
136 // デバイス内の Z_MAP_BLOCKS に i+ 個計で s->s_imap_blocks ブロックを占有します。ロジックの
137 // ブロック毎に s->s_zmap_blocks[i] が [i] 後続のブロックである s->s_zmap_blocks のブロックを占有します。
138     for (i=0; i < Z_MAP_SLOTS; i++)
139         s->s_zmap[i] = NULL;
140     block=2 です。
141     for (i=0 ; i < s->s_imap_blocks ; i++)           // デバイス内の i-node のビットマップ
142         if (s->s_imap[i]=bread(dev, block))          を読み取る。
143             block++;
144     その他
145         ブレークします。
146     for (i=0 ; i < s->s_zmap_blocks ; i++) // デバイス内のロジックブロックのビットマッ-
147     プを読み込む(s->s_zmap[i]=bread(dev, block))
148         block++;
149     その他
150         ブレークします。

// 読み込んだビットマップブロックの数が、論理ブロックの数と一致しない場合は
// ビットマップが占めるべきであることから、ファイルシステムのビットマップに問題があること
を示しており
// スーパー ブロックの初期化に失敗します。そのため、以前に適用されていたすべてのリソース
// と占有されているものを解放することができます。つまり、i-node ビットマップで占有されている
キャッシュブロックと
151 // 論理ブロックのビットマップが解除されると、上記で選択したスーパー ブロックの配列項目が解除
152 されます。
153 // スーパー ブロックアイテムのロックが解除され、NULL ポインタが返されま
す。. if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {
    for(i=0;i<I_MAP_SLOTS;i++) // ビットマップが使用しているバッファブロック
        を解放します。

```

```

154         for(i=0;i< Z_MAP_SLOTS;i++)
155             brelse(s->s_zmap[i]) と
156            なります。
157             s->s_dev=0;           // 選択されたスーパー ブロックアイテムを解
158             // それ以外はすべてOKです。さらに、アイドル状態のi-nodeを要求する関数では、もし
159             // デバイス上のすくてのi-nodeが使用された場合、ルックアップ関数は0の値を返します。
160             // 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
161             // ファイルシステムが0番目のiノードを割り当てるのを防ぐために、//は1に設定されています。同じ理
162             // ロジックブロックのビットマップの最下位ビットも1に設定されます。最後の関数では、ロックを
163             // 解除してs->s_imap[0]->b_data[0] |= 1;
164             // スーパーブロックi-nodeを返します。
165             free_super(s) です。
166             を返します。
167         }
168         ///////////////////////////////////////////////////////////////////
169         // ファイルシステムをアンマウントする system-call.
170         // 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。
171         // 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
172         // に
173         // 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
174         // する
175         // スーパーブロックとビットマップによって占有されます。最後に、バッファキャッシュを同期させ
176         // て
177         // デバイス上のデータを表示します。アンマウント操作が成功した場合は0を、そうでない場合は
178         // エラーを返します。fs/namei.cでsys_umount()の実装を確認してください。
179         // エラーを返す
180         int sys_umount(char *dev_name);
181         {
182             int dev;
183             // 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
184             // 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
185             // 参照
186             fs/namei.cプログラムのシステムコールsys_mknod()の実装を確認してください。また、ファイルシステム
187             // がただ戻され、エラーコードが返されます。
188             // ブロックデバイスに格納される必要があり、ブロックデバイスのファイルでない場合は、i-node
189             dev_i          return -ENOENT
190             dev = inode->i_zone[0];
191             if (! S_ISBLK(inode->i_mode)) {...}
192                 ifput(inode); // fs/inode.c, line 150.
193                 return -ENOTBLK;
194             }
195             // OK、デバイス番号を取得するために取得したi-nodeがミッションを完了したので、元に戻します
196             .
197             // ここではi-nodeです。そして、ファイルシステムをアンマウントするための条件が満たされている
198             // かどうかをチェックします。
199             // 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、ビージェラーが返されます。
200             // 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
201             // 場合。
202             // 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
203             // れます。もし、i-node
204             // がマウントされている // にフラグ i_mount が設定されていない場合、警告メッセージが表示されま
205             // す。すると
206             // 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
207             // 、もし
208             // ifput(inode);
209             if (dev==ROOT_DEV)

```

```

181         return -EBUSY;
182    もし (!sb->get_super(dev)) || !(sb->s_imount))
183     return -ENOENT
184    もし (!sb->s_imount->i_mount)
185     printf ("Mounted inode has i_mount=%h")となります。
186     for (inode=inode_table+0 ; inode< inode_table+NR_INODE ;
187         inode++) if (inode->i_dev==dev && inode->i_count)
188         return -EBUSY;
189 // 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
190 // 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
191 // トします。
192 // マウントされている場合は、i-nodeをリリースし、スーパー・ブロックのマウントされたi-nodeフィ
193 // ールドを次のように設定します。
194 // を NULLにします。マウントされていないシステムのルート i-node を戻す。のポインタを設定します
195 // となります。
196 // スーパーブロックから戻すされたファイルシステムのルートiノードをNULLにする。
197     sb->s_imount = NULLとな
198     ります。
199 // 最後にput_super(sb->s_isup)です。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
200 // 放します。
201 // キャッシュとデバイス上のデータとの間の同期操作を実行するための//と
202 // ならば0（アンマウント成功）を返す。
203     // パラメータ dev_name はデバイスのファイル名、dir_name はマウントされるディレクトリ名。
204     // また、rw_flagはマウントされたファイルシステムの読み取り/書き込みフラグです。マウントされ
205     // る場所は
206     // 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
207     // いけません。
208     // 成功すれば0を、そうでなければエラー番号を返します。 199
209 int sys_mount(char * dev_name, char * dir_name, int rw_flag) 200 {
210     struct super_block * sb;
211     int dev;
212
213     // まず、デバイスのファイル名に従って、対応するi-nodeを見つけて、デバイスの
214     // 番号です。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
215     // i_zone[0]にあります。
216     // まだ取得した情報はアシズレコラクデモアトスナシカヌハナリハ、そうでない場合は、i-
217     node dev_if (!(dev_i=>name_i(dev_name)))
218         return -ENOENT
219     dev = dev_i->i_zone[0] となります。
220     if (! S_ISBLK(dev_i->i_mode)) {...}
221         put(dev_i)です。
222         return -EPERM;
223     }
224
225     // 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。
226     // 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
227     // リ名が
228     // 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
229     // 。
230     // 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
231     // ここで参照されています）、またはノード番号がルートファイルシステムの1である場合は、次の
232     // ように戻します。

```

```

// iノードでは、エラーコードを返す。また、そのノードがディレクトリファイルノードでない場合
は
//のみnodeは元に痕跡でマウントされますが、このノードは既にマウントされています。
212     input(dev_i)です。
213     if (!(dir_i->name_i == dir_name))
214         return -ENOENT
215     if (dir_i->i_count != 1 || dir_i->i_num == ROOT_INO) {
216         input(dir_i)です。
217         return -EBUSY;
218     }
219     if (!S_ISDIR(dir_i->i_mode)) { // マウントポイントは、ディレクトリエントリである必
要があります。
220         input(dir_i)です。
221         return -EPERM;
222 // マウントポイントがチェックされたので、新しいファイルシステムのスーパーべロックの読み込み
を開始します。
// ファイルシステムのスーパーべロックは、最初にスーパーべ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べロックテーブルにない場合は新しいファイルのスーパーべロックを取得した
後
223 // ファイルシステムをマウントしていないことを確認するために、まずそれをチェックします。新し
いファイルシステムがマウントされていないことを確認するために
224 // 他の場所で、マウントされるi-nodeが（他のファイルシステムによって）占有されていないこと、
225 そうでなければ
226 // エラーが発生して戻ってきたました。
227     if ((sb->s_imount) && // 新しいfsが他の場所でマウントされてい
228         !(sb->read_super(dev))) {           る場合。
229         input(dir_i);
230         return -EBUSY;
231     }
232     if (dir_i->i_mount) { // マウントポイントが占有されてい
233         input(dir_i);           る場合
234         return -EPEERM;
235     }
236 // 最後に、新しいファイルシステムのスーパーべロックの「マウントされたi-node」フィールドの「
s_imount」を次のように設定します。
// マウントされているディレクトリのi-nodeを指し、マウントフラグとnodeを設定する
237 // マウント位置のi-nodeのフラグを修正し、0を返す（インストールが成功した場合）。
238     dir_i->i_mount=1となりま
239     /* 注意! input(dir_i)はしません
240     dir_i->i_dirt=1となりま
241     0を返す。          /* umount で行います */
242 }
243 // 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
244 // ルートファイルシステムをマウントする。マウントするには、マウントフラグとマウント
245 // ポイントを設定する。最後に、マウントフラグを修正し、0を返す（インストールが成功した場合）。
246 void mount_root(void)
247 {
248     int i, free;
249     struct super_block * p;
250     struct m_inode * mi;

```

```

// まず、ディスクのi-node構造のサイズが要件（32バイト）を満たしているかどうかをチェックして
// コードを修正する際の不整合を防ぎます。続いて、ファイルテーブルの初期化（64項目）
// とスーパーblockテーブル（8項目）があります。ここで、すべてのファイル構造における参照
カウントは
// 0（アイドルを示す）に設定され、スーパーblockテーブルの各構造体のデバイスフィールド
// は0に初期化されます（アイドル状態であることも示します）。ルートファイルシステムが配置さ
247 れているデバイスが
248 // がフロッピーディスクの場合、「ルートフロッピーを挿入して、ENTERを押してください」とプロ
249     for(i=0; i<NR_FILE; i++) {
250         if (32 != sizeof(struct d_inode)) {
251             if (MAJOR(ROOT_DEV) == 2) {
252                 printk("Insert root floppy and press ENTER");
253                 wait_for_keypress();
254             }
255             for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++)
256             {...      p->s_dev = 0;                                // スーパーブロックテーブル(8)を初期
257               p->s_lock = 0;                                化します。
258               p->s_wait = NULL
259           }                                              となります。
260 // 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。
261 // その後、ファイルシステムのスーパーblockがルートデバイスから読み込まれ、ルートのi-node
へのポインタが
262 // メモリのi-nodeテーブルのファイルシステムの//(ノード1)を取得します。上のスーパーblockが
263 // デバイスが故障するか、ルートノードが故障すると、メッセージが表示され、デバイスがダウン
264     if (!(p=read_super(ROOT_DEV)))
265     if (!(mi=page_get(ROOT_DEV,ROOT_INO))) {
266         if (!mi)
267             if (mi->i_count += 3 /* 注意! 論理的には1回ではなく、4回使用されている */ .
268             p->s_isup = p->s_imount = mi;
269             current->pwd = mi;
270             current->root = mi;
271             i=p->s_nzones;
272             while (--i >= 0)
273                 if (! set_bit(i&8191, p->s_zmap[i>>13]-
274                               >b_data))
275             // デバイス上の空き論理free++数を表示した後、次に数を数えます。
276             // 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。
277             // スーパーブロックで示されたデバイスを+1(0のノードもカウントするために1を加える)して

```

```

// 対応するビットの占有率に応じて、アイドル状態のi-nodeの数を計算する
//をi-nodeのビットマップに表示します。最後に、空いているi-nodeの数と、i-nodeの合計数
デバイスで利用可能な // が表示されます。
273     printk ("%d/%d free blocks\n", free, p->s_nzones);
274     free=0;
275     i=p->s_ninodes+1となります。
276     while (-- i >= 0)
277         if (! set_bit(i&8191, p->s_imap[i>>13]-
278             >b_data)) free++;
279     printk ("%d/%d free inodes\n", free, p->s_ninodes).
280 }
281

```

7. namei.c

1. 機能

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。については

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2byteのi-node番号のみが含まれています。

```

36 #define NAME_LEN 14                                // ファイル名の最大長。
37 #define ROOT_INO 1                               // ルートのi-node番号。
38
39 struct unsigned short inode;                      // i-nodeの番号。
40     char name[NAME_LEN];                           // ファイル名の文
41
42 };
```

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、「/usr/bin/vi」のi-node番号を取得する。

ファイル'usr'を取得する。i-node numberファイルシステムによれば、ディレクトリ'usr'を正常に取得することができ、その中にファイル名'bin'のディレクトリ・エントリを見つけることができる。これもi-node numberで'usr/bin'がわかるので、ディレクトリ'usr/bin'の位置がわかり、その中にある'vi'ファイルのディレクトリ・エントリを見つけることができます。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ます。例えば、「.../kernel/Makefile」を探すには、まず、カレント・ディレクトリの「...」ディレクトリ・エントリから親ディレクトリのi-node番号を取得し、その後、上述のプロセスに従ってルックアップ操作を行えばよい。

プログラムのいくつかの主要な機能は、その前に詳細なコメントがあり、各機能やシステムコードの使用方法が明確であるため、ここでは説明しません。

12.7.2 コードアノテーション

プログラム 12-6 linux/fs/namei.c

```

1 /*
2 * linux/fs/namei.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * tytsoによる若干の修正。
9 * 初期タスクなど、記述子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が含まれています。
10 // <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネルの使用する機能
// <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・オペレーションのため
// <string.h> 文字列のヘッダーファイルです。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ーファイルです。演算制御定数記号の定義は
// ファイルとそのディスクリプターに使用されます。
// <errno.h> エラー番号のヘッダーファイルです。システムの様々なエラー番号を含みます。
// <const.h> 定数シンボルファイルでは、現在、i-nodeのi_modeフィールドのフラグを定義しています。
// <sys/stat.h> ファイル状態のヘッダーファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
// そして、定数です。
11 #include <linux/sched.h>
12 #include <linux/kernel.h> (日本語)
13 #include <asm/segment.h>
14
15 #include <string.h>
16 #include <fcntl.h>
17 #include <errno.h>
18 #include <const.h>
```

```

19 #include <sys/stat.h> (日本語)
20
21 static struct m_inode * namei(const char * filename, struct m_inode * base.),
22 int follow_links); 23
23
24 // 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
25 // 配列名で表される配列項目 (a[b]など) の値と
26 // 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。
27 // 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。
28 // 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
29 // は、「LoveYou」+2と同等です。また、文字列「LoveYou」がある位置は
30 // 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
31 // 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
32 // 8進法で0166。C言語では、文字は次のようにしてASCIIの値で表すことができます。
33 // 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
34 // は「\x76」または「\166」と表現します。そのため、表示できない文字（例.
35 // /0x00--0x1fの範囲のASCIIコード値を持つ制御文字）は次のように表現できます。
36 // そのASCIIコード値
37 //
38 // 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。
39 // 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。
40 // 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。
41 // 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。
42 // 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。
43 // xの値: 0~3。例えば、x が 2 の場合、マクロは 006 という 8 進法の値を返し、次のことを示します。
44 // 読み込み可能性と書き込み可能性(rw)を示しています。また、0_ACCMODE = 00003の場合は、マスクの
45 // インデックス値 x.
46
47 #define ACC_MODE(x) ("XXXX"[(x)& 0_ACCMODE])
48
49 /*
50 * 名前 > NAME_LEN の文字数を増やしたい場合は、この行をコメントアウトしてください。
51 * 切り捨てられます。それ以外の場合は、許可されません。
52 */
53 /* #define NO_TRUNCATE */ (英語)
54
55
56 /*
57 * パーミッション()
58 *
59 * は、ファイルの読み取り/書き込み/実行の権限をチェックするために使用されます。
60 * euidだけを見るべきか、euidと両方を見るべきかはわかりません。
61 * uidですが、これは簡単に変更できるはずです。
62 */
63
64 ///////////////////////////////////////////////////////////////////
65 // ファイルのアクセス権を検出します。
66 // 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。
67 // 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。
68 static int permission(struct m_inode * inode, int mask)
69 {
70     ...
71     int mode = inode->i_mode; // ファイルアクセスモード。
72 }
```

```

46
47 /* 特別な場合: ルートでも削除されたファイルの読み書きはできない */。
48 // 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。
49 // 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
50 // の
51 // プロセスは、i-nodeのユーザーID、ファイル所有者のアクセス許可と同じです。
52 // が取られます。それ以外の場合は、プロセスの有効なグループID (egid) がグループ
53 // i-nodeの file->i_node && !inode のアクセス権が取られます。
54     return 0;
55     else if (_current->euid==inode->i_uid)
56         mode >>= 6;
57     else if (_in_group_p(inode->i_gid))
58         mode >>= 3;
59
60 // 最後に、アクセス権がマスクコードと同じである場合、またはスーパー
61 // ユーザーの場合は1を、そうでない場合は0を返します。
62     if (((mode & mask & 0007) == mask) || _suser())
63     return 1;
64     0を返す。
65 }
66 /*
67 * ok, we can't use strncmp, because the name is not in our data space.
68 * したがって、matchを使用する必要があります。大きな問題はありません。また、マッチは
69 * いくつかのサンタリーテスト
70 */
71 * 注意! strncmpとは異なり、matchは成功すると1、失敗すると0を返します。
72 */
73
74 // 文字列の照合・比較機能。
75 // 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。
76 // de - ディレクトリエントリの構造です。
77 // 戻ります。_static int match(int len, const char * name, struct
78 dir_entry * de) _67_ .
79
80 register int same asm ("ax"); // 変数を登録します。_69_
81
82 // 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triが
83 // ポインタが空であるか、ディレクトリエンtriの i-node が NULL であるか、比較する長さが
84 // ファイル名の長さ NAME_LEN を超える。比較対象の長さが 0 に等しい場合でも
85 // ディレクトリエンtriのファイル名が「...」であれば、同じと判断し、1(マッチ)を返します。
86 // 比較される長さ 'len' が NAME_LEN よりも小さく、ファイル名の長さが
87 // が len を超える場合は、0(一致しない)も返されます。
88 // 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定する方法は
89 // name[len]がNULLかどうか。長さが len を超える場合は、name[len] は通常の文字です。
90 // NULLではないもの。長さ len の文字列名の場合、name[len] という文字は NULL でなければなりません。
91
92    もし (!de || !de->inode || len > NAME_LEN)
93     0を返す。
94     /* ""は、"を意味します —> そのため、"/usr/lib//libc.a"のようなパスは動作し
95     ます。
96     もし (!len && (de->name[0]=='.') && (de->name[1]=='\0'))
97     を返します。
98     もし (len < NAME_LEN && de->name[len]) // 与えられた名前の長さ > len.
99     // 次に、埋め込み0を返すブリストメントを使用して、すばやく比較します。これは、文字
100    列
101   // ユーザーデータスペース(fsセグメント)での比較です。

```

```

// %0 - eax (比較結果、同じ); %1 - eax (eaxの初期値0); %2 - esi (名前)。
// %3 - edi (ディレクトリ名ポインタ); %4 - ecx (比較されたバイト) len。
77     asm ("cld##"           // 明確な方向性
78         "fs ; repe ; cmpsb##t" // ユーザ空間で[esi++]と[edi++]を比較する。
79         "setz %%al"          // 同一の場合はal = 1とする (same = eax)。
80         :"=a" (同)
81         :""(0), "S" ((long) name), "D" ((long) de->name), "c"
82         (len)
83         "cx", "di", "si")。
84 }      return same; // 結果を返す。
85
86 /*
87 * find_entry()
88 *
89 * 指定されたディレクトリの中から、指定された名前のエントリを見つけます。それは
90 * エントリーが発見されたキャッシュバッファを返し、エントリーの
91 * 自分自身(パラメータとしてres_dir)を読みます。の inode を読むことはありません。
92 * エントリーしたい方はご自身で行ってください。
93 *
94 * これは、「... トラバーサル」によるいくつかの特殊なケースにも対応しています。
95 * 疑似ルートとマウントポイントの上にある。
96 */
97
98     //// 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
99
100    // 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
101    // 名の長さ。
102
103    // 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
104    // 与えられたファイル名に応じて特別な処理を行います。
105
106    // ファイル名が「...」の場合の設定です。の151行目の前のコメントをご覧ください。
107    // 関数の引数におけるポインタの役割については、// linux/sched.c を参照してください。
108
109    // 戻り値: 成功した場合はキャッシュブロックポインター、返された場合はディレクトリエントリポ
110    // インター
111    const char * name, int namelen, struct dir_entry ** res_dir)
112    // at *res_dir. 失敗すると null ポインタが返されます。
113
114    static struct buffer_head * find_entry(struct m_inode ** dir,
115
116        int entries,
117        int block, i;
118        struct buffer_head * bh;
119        struct dir_entry * de;
120        struct super_block * sb;
121
122
123    // 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
124    // のパラメータを設定します。上記の30行目でシンボル定数NO_TRUNCATEを定義した場合、ファイル
125    // 名前の長さが最大長のNAME_LENを超えると、処理されません。もし、NO_TRUNCATE
126    // が定義されていない場合、ファイル名の長さが最大長のNAME_LENを超えると切り捨てられます。
127
128    #ifdef NO_TRUNCATE
129        if (namelen > NAME_LEN)
130            NULLを返す。
131
132    #else
133        if (namelen > NAME_LEN)
134            namelen = NAME_LEN;
135
136    #endif
137
138
139    // 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
140    // ディレクトリの i-node には、このディレクトリに含まれるデータのサイズが含まれていますので

```

```

// をディレクトリエントリのサイズ（16バイト）で割ると、ディレクトリエントリの数になります。
// それ null 返すべきディレクトリエントリへのポインタ。
113 から     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は、関数パラメータで指定されたディレクトリと同じです。
// これは、このプロセスにとって、このディレクトリが擬似的なルートディレクトリであることを意
味し、つまり
// プロセスは、ディレクトリ内のアイテムにのみアクセスでき、親ディレクトリにフォールバックす
ることはできません。
// つまり、このプロセスでは、このディレクトリがファイルシステムのルートディレクトリになっ
て
いるということです。そのため
// ファイル名を「...」に変更する必要があります。
// そうでなければ、ディレクトリのi-node番号がROOT_INO（番号1）に等しい場合、それは確かに
// ファイルシステムのルートi-nodeなので、ファイルシステムのスーパーblock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。
118 // ひとりずつ。つまり(*dir)の状況を受けて、私たちは静かに「盗み」を実行したのです。
119 // コラム」プロジェクト namelen=1;
120 /* ... をチェックすると、何らかの "魔法" をかけなければならないかもしれませんからです。
121 if (namelen==2 && get_ls_byte(name)==ROOT_INO) { 117 /* 疑
122 /* アクションポイント上の操作ですが、dir がマウントされたものと交換されま
123 /* ルート内のディレクトリ inode です。注意! mounted を設定しているのは、新しいdirを
    // inputできるようにするために。
124     sb=get_ifsu(str->*imount->dev)となります。// マウントされた
125         input(*dir);           ポイント
126         (*dir)=sb->
127             s_imount; (*dir)-
128                 >i_count++;
129     }
130 }

// では早速、指定したファイル名のディレクトリエントリがどこにあるかを調べてみましょう。
// したがって、ディレクトリのデータを読み取る、つまり、データブロックを取り出す（論理
// ブロック）を、ディレクトリi-nodeに対応するブロックデバイスのデータゾーンに配置します。ブ
ロックは
i-node構造体のi_zone[]配列には、これらの論理ブロックの // 番号が格納されています。
// まず最初に保持されているデータブロックのブロック番号を取得し、次に指定されたディレクトリを読み
込みます
131     if (!(block = (*dir)->i_zone[0]))
132         NULLを返す。
133     if (!(bh = bread((*dir)->i_dev, block)))
134         NULLを返す。

// この時点で、与えられたファイル名に一致するディレクトリ・エントリを検索し、read
// データブロック。まず'de'にバッファブロックのデータブロック部分を指示示すようにして、ル
ープで
// のディレクトリエントリ数を超えないように検索します。ここで i は
135 // ディレクトリエントリのインデ
136     ツクス i = 0;
        de = (struct dir_entry*) bh->b_data;

```

```

137     while (i < entries) {
138         // 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
139         // 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
140         // 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
141         // 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
142         // 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。
143         // を入れることができます。141行目のi/DIR_ENTRIES_PER_BLOCKでは、データブロック番号を
144         // ディレクトリ項目が現在検索されているディレクトリファイルと、bmap()関数(inode.c,
145         // 142行目)は、デバイス上の対応する論理ブロック番号を計算できます。 if ((char
146             *)de >= BLOCK_SIZE+bh->b_data) {
147                 brelse(bh);
148                 bh = NULL;
149                 if (!block = bmap(*dir, i/DIR_ENTRIES_PER_BLOCK)) ||
150                     !(bh = bread((*dir)->i_dev, block))) {
151                         i += DIR_ENTRIES_PER_BLOCK;
152                         を続けています。
153                     }
154                     de = (struct dir_entry *) bh->b_data;
155                 }
156             // 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
157             // ディレクトリエントリのi-nodeポインタ'*dir' とディレクトリエントリのデータブロックポインタ
158             'bh' は
159             // を返し、この関数を終了します。そうでなければ、次のディレクトリエントリの比較を続けます
160             // ディレクトリエントリ(add_entry(name,len,name,de)) {
161                 .
162                 *res_dir = de;
163                 bhを返す。
164             }
165             de++」です。
166             i++;
167         }
168         // 指定されたディレクトリ内のすべてのディレクトリエントリが検索され、かつ
169         // 対応するディレクトリエントリが見つからなかった場合、そのディレクトリのデータブロックが解
170         // 放されます。
171         // そして最後にNULLが返される(失敗)。
172         NULLを返す。
173     }
174 }
175 */
176 * add_entry()
177 *
178 * 指定されたディレクトリにファイル・エントリを追加するには、同じ
179 * セマンティクスは find_entry()と同じです。失敗した場合は NULLを返します。
180 *
181 * 注意!!!de'のinode部分は0のままで。
182 * これを呼び出してから、何かを入力するまでの間、*は眠らないかもしれません。
183 * あなたが寝ている間に誰かが使っていたかもしれないからです。
184 */
185 ///// 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。
186 // 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
187 // さ。
188 // 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。
189 static struct buffer_head* add_entry(struct m_inode* dir,
190 const char * name, int namelen, struct dir_entry** res_dir) 171 {

```

```

172     int block, i;
173     struct buffer_head * bh;
174     struct dir_entry * de;
175
// 同様に、この機能でも判断して、妥当性を検証する必要があります
// のパラメータを設定します。上記の30行目でシンボル定数NO_TRUNCATEを定義した場合、ファイル
// 名前の長さが最大長のNAME_LENを超えると、処理されません。もし、NO_TRUNCATE
// が定義されていない場合、ファイル名の長さが最大長のNAME_LENを超えると切り捨てられます。
176 *res_dir = NULL; // 結果のディレクトリのエントリポインタです。 177#endif
NO_TRUNCATE
178     if (namelen > NAME_LEN)
179         NULLを返す。
180 #else
181     if (namelen > NAME_LEN)
182         namelen = NAME_LEN;
183 #endif
// それでは早速、指定されたファイル名のディレクトリエントリを、指定された
// のディレクトリです。そのため、ディレクトリのデータを読む、つまりデータを取り出す必要があ
// るのですが
ディレクトリに対応するブロックデバイスのデータゾーン内の // ブロック（論理ブロック）です。
// i-node（アイノード）です。これらの論理ブロックのブロック番号は、i_zone[]配列に格納されて
// います。
// i-nodeの構造。まず、その中に保存されている最初のダイレクトブロック番号を取得し、それを読
184 み取って
185 // 指定されたディレクトリアイテムのデータブロックをデバイスから取り出します。また、ファイル
186 名の長さが
187 // パラメータで指定された値が0の場合はNULLで返す if (!namelen)
188         NULLを返す。
189     if (!(block = dir->i_zone[0]))
        return NULL;
    if (!(bh = bread(dir->i_dev, block)))
        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
190 // 最後の未使用の空のディレクトリ・エントリです。まず、ディレクトリ・エントリ・ポインタに、
191 データ de = (struct dir_entry *) bh->b_data;
192 バッファ whlFle191 // ブロック部分、つまり最初のディレクトリエントリです。ここでiは、インデ
クス
// 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 // ポインタが空であれば、それは
// 論理ブロックは存在しないため、新たに作成された空のブロックである可能性があります。そこで
我々は
// ディレクトリエントリ数に対するディレクトリエントリインデックス値 DIR_ENTRIES_PER_BLOCK
論理的なブロックが保持できることを示す // ブロックをスキップして検索を続行します。そうでな
ければ、新たに
// 読み込みブロックにはディレクトリエントリデータがあるので、ディレクトリエントリ構造ポイン
タは
193 // ブロックのデータ部分をバッファリングして、そこから検索を続けます。
194 // 196行目のi/DIR_ENTRIES_PER_BLOCKは、ブロック番号を計算するために
195 // 現在検索されているディレクトリ・エントリiが存在するディレクトリ・ファイルで、関数
196 // create_block()(inode.c, line 147)を使用して、デバイス上のロジックブロックを読み込んだり
    、作成したりすることができます。
            brelse(bh);           770
            bh = NULL;
            block = create_block(dir, i/DIR_ENTRIES_PER_BLOCK);

```

```

197         if (!block)
198             NULLを返す。
199         if (!(bh = bread(dir->i_dev, block))) {           // 空の場合はスキップします。
200             i += DIR_ENTRIES_PER_BLOCK;
201             を続けています。
202         }
203         de = (struct dir_entry *) bh->b_data;
204 // 現在、ディレクトリエントリ番号iにディレクトリエントリサイズを乗じたサイズである場合
// ディレクトリのiノードが示すディレクトリデータサイズi_sizeを超える場合は
// ファイルが削除されたことにより、空のエントリが残っていないことを示しています。そのため、新しい
// ディレクトリデータファイルの最後に追加する必要があるディレクトリエントリです。したがって
// 私たちは
// このディレクトリのディレクトリ・エントリを設定する必要があるので、ディレクトリのi-node番
号を設定する
205 // エントリを空にして、ディレクトリファイルのサイズを更新します（さらに、ディレクトリの長さ
206 を
de->inode=0となります。
207 // エントリを設定し、変更するsizeをこののi-nodeを設定して、変更を更新する
// このディレクトリの時刻を現在の時刻に変換します。
208     if (i->size>=st.st_size && st.st_ctime) {...}
209     dir->i_ctime = CURRENT_TIME;
210 // 現在検索されているディレクトリエントリ'de'のi-node番号が空の場合、それは次のことを意味し
ます。
// 使用されていないフリーのディレクトリ・エントリ、または追加された新しいディレクトリ・エ
ントリが見つかります。
// したがって、ディレクトリの修正時刻を現在の時刻に更新して、ファイル
ユーザーデータスペースからディレクトリエントリのファイル名フィールドに // 名前を設定し
211 // 対応するディレクトリエントリのname[0]を修正 CURRENT TIMEに設定します。
212 // ディレクトリエントリへのポインタとCURRENT TIMEへのポインタです。
213     for (i=0; i < NAME_LEN; i++)
214         de->name[i]=(i<namelen)? get_fs_byte(name+i):0;
215         bh->b_dirt = 1;
216         *res_dir = de;
217         bhを返す。
218     }
219     de++; // エントリーが使用中の場合は、引き続き次のエントリーをチェックし
ます。
220     i++;
221 // この関数はここでは実行できません。これは、Linus氏がコードをコピーしたためと思われます。
// 上記のfind_entry()関数を修正して、この関数にしました :). brelse(bh);
222     NULLを返す。
223
224 }
225
226 ///////////////////////////////////////////////////////////////////
227 // シンボリックリンクのi-nodeを探す。
228 // 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。
229 // 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。
230 static struct m_inode * follow_link(struct m_inode * dir, struct m_inode * inode)
231 {
232     unsigned short fs;           // FSセグメントレジスタを一時的に保存するために
使用します。
233     struct buffer_head * bh;

```

```

// まず、関数のパラメータの有効性を判断します。ディレクトリのi-nodeが与えられていない場合。
// プロセス・タスク構造のルートi-nodeセットを使用し、リンク数を増やす
// を1つずつ表示します。ディレクトリエントリのi-nodeが与えられていない場合、ディレクトリのi-
nodeは戻されて
// NULLが返されます。指定されたディレクトリエントリがシンボリックリンクでない場合は、直接
231 // ディレクトリエントリに対応するi-node. if (!dir)
232     {. .
233         dir = current->root;
234         dir->i_count++;
235     }
236     if (!inode) {
237         i_put(dir);
238         return NULL;
239     }
240     if (! S_ISLNK(inode->i_mode))
241         { i_put(dir);
242         return inode;
243     }
// 続いて、FSセグメントレジスタの値を取得します。FSは通常、0x17を指すセレクタを保持しています。
// タスク（ユーザー）データセグメントを指します。FSがユーザーデータセグメントを指していない場合や、最初の
// 与えられたディレクトリ・エントリi-nodeのダイレクト・ブロック・ナンバーが0に等しいか、エラーが発生した場合
// 最初のダイレクトブロックを読み取る際の名文字列inodeのノードを戻して、NULLを返す。
244 // そうであればFSが0x17で各セグメントの内容を取得します。読み取りに成功したことになります。if (fs !=0x17 || !inode->i_zone[0] ||)
245 // このシンボリックリンク名のデータセグメントのノードは、すでに
246 // bhで指定されたi_put(dir)です。i_put(inode)です。
247         i_put(inode);
248         NULLを返す。
249     }
// この時点で、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
250 // がある
251 // FS = 0x10とし、関数の後に元のFSを復元する。
252 // namei()が呼び出されます。最後に、バッファブロックが解放され、指示されたファイルのi-
nodeが
253     namei()で取得したシンボルリンクによる // i_put(inode)が
254     戻されます。
255     asm ("mov %0, %%fs": "r" ((unsigned short) 0x10));
256 }     inode = namei(bh->b_data, dir, 0);
257     asm ("mov %0, %%fs": "r" (fs));
258 /*     brelse(bh);
259 *     get_dir();
260 */
return inode;

```

```

261 * Getdir は、パス名が最上位のディレクトリに到達するまで走査します。
262 * 失敗するとNULLを返します。
263 */
264 static struct m_inode * get_dir(const char * pathname, struct m_inode * inode)
265 {
266     チャーC
267     const char * thisname;
268     struct buffer_head * bh;
269     int namelen, inr;
270     struct dir_entry * de;
271     struct m_inode * dir;
272
273 // まず、パラメータの有効性を判断します。もし、与えられたi-nodeポインタのinodeが
274 // ディレクトリが空の場合は、現在のプロセスの作業ディレクトリi-nodeが使用されます。
275 // 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
276 // 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。
277 // 現在のプロセスのタスク構造ではそこで、指定されたディレクトリi-nodeを戻す必要があります。
278 // 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。
279 // i-nodeの // 参照カウントと、パス名の最初の文字'/'を削除します。これにより
280 // 現在のプロセスは、自分が設定したルートのi-nodeを開始点としてのみ使用できることを
281 // を検索し(既述) {
282     inode = current->pwd;           // 作業ディレクトリのi-ノード。
283     inode->i_count++;             // i-node>1_count++です。
284 }
285     も ((c=get_fs_byte(パス名))== '/') {
286         input(inode)です。          // は、元のi-nodeに戻します。
287         inode = current->root;      // ルートプロセスに指定されたi-node。
288         // 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
289         // inode>1_count++です。
290         // 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。
291         // 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。
292         // 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
293         // がディレクトリタイプでない場合、またはディレクトリに入るためのアクセス許可がない場合は
294         // i-nodeが戻され、NULLが返されます。もちろん、ループに入るとときには、i-nodeの
295         // カレントディレクトリの' inode'は、プロセスルートのi-nodeまたはカレントディレクトリのi-
296         // nodeです。
297 // 作業ディレクトリ、またはパラメータで指定された検索開始ディレクトリのi-nodeです。
298     thisname = pathname;
299     if (! S_ISDIR(inode->i_mode) || ! permission(inode, MAY_EXEC)
300     {...}
301         input(inode)です。
302         NULLを返す。
303 // 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
304 // 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
305 // 現在のパス名ポインタ' pathname'から末尾の文字になるまで
306 // (NULL) または'/'文字です。この時点で、変数「namelen」は、正確には、「NULL」の長さになります。
307 // 現在処理されているディレクトリ名の部分を、変数' thisname'が指しているので

```

```

// ディレクトリ名部分の先頭です。このとき、その文字が終了文字であれば
// NULLの場合は、パス名の最後まで検索したことを示し、最後に指定した
// ディレクトリ名やファイル名に到達した場合は、i-nodeポインタを返します。
// 注意: パス名の最後の名前がディレクトリ名でもあり、'/'文字が
// が付加されていない場合は、最後のディレクトリ名のi-nodeを返すことはありません!
// 例えばパス名「/usr/src/linux」の場合、この関数はi-nodeのみを返します。
src/' ディレクトリ名の//。

289     for (namelen=0; (c=get_fs_byte(pathname++))&&(c!= '/') ;namelen++)
290         /* nothing */ ;
291     if (!c)
292         return inode;
// カレントディレクトリ名部分（またはファイル名）を取得した後、ルックアップディレクトリを呼び出す
// エントリ関数 find_entry() で、指定された名前のディレクトリエントリを
// 現在処理中のディレクトリ。（見つからなかった場合は、i-nodeを戻してNULLを返す。）その後。
// 見つかったディレクトリにi-node番号「inr」とデバイス番号「idev」が取り出される。
// エントリーがあると、そのディレクトリ・エントリーを含むキャッシュ・ブロックが解放され、i-
// nodeが置かれます。
// を返します。そして、ノード番号' inr' のi-node 'inode'を取り、次のループを続けます。
// ディレクトリエントリをカレントディレクトリとするパス名の中の // ディレクトリ名部分。もし
293 // 現在処理されているディレクトリエントリがシンボリックリンク名であれば、follow_link() を使
294 用する
295 // が指し示すディレクトリエントリ名のi-nodeを取得します。
296     if (!(bh = find_entry(&inode, thisname, namelen, &de))) {
297         inr = deiput(inode);    // カレントディレクトリ名部分のi-node番号。
298         brelse(bNULLを返す。
299         dir = inode;
300         if (!(inode = iget(dir->i_dev, inr))) {           // i-nodeのコンテンツ
301             iput(dir)です。                                を取得します。
302             NULLを返す。
303         }
304         if (!(inode = follow_link(dir, inode)))
305             NULLを返す。
306     }
307 }

308 /*
309 */
310 * dir_namei()
311 *
312 * dir_namei()は指定したディレクトリのinodeを返します。
313 * 指定された名前と、そのディレクトリ内の名前。
314 */
315 static struct m_inode * dir_namei(const char * pathname,
316         int * namelen, const char ** name, struct m_inode * base)
317 {
318     チャーC
319     const char * basename;
320     struct m_inode * dir;

```

321

```
// まず、指定されたパス名のトップレベルディレクトリのi-nodeを取得し、次に検索
// とパス名を検出し、最後の'/'文字の後の名前文字列を見つけ、それを計算します。
// 長さを指定し、最上位ディレクトリのi-nodeポインタを返します。なお、最後の
// パス名の//文字がスラッシュ文字'/'の場合、返されるディレクトリ名は
// 空であり、長さは0です。しかし、返されたi-nodeポインタはまだi-nodeを指しています。
// ディレクトリ名の最後の'/'文字より前の部分です。
```

322 if (!(dir = get_dir(pathname, base))) // baseは、starting dirのi-nodeです。323 return NULL;

324 basename = パス名。

325 while (c=get_fs_byte(pathname++))

326 if (c=='/')

327 basename=パス名。

328 *namelen = pathname-basename-1;

329 *name = basename;

330 return dir;

331 }

332

```
//// 指定されたパス名のi-nodeを取得する（内部機能）。
```

```
// Parameters: pathname - パス名; base - 検索開始ディレクトリ i-node;
```

```
// follow_links - シンボリックリンクをフォローするかどうか、1 - はい、0 -
// いいえ。
```

```
// 対応するi-nodeを返します。
```

333 struct m_inode*followlink(const char * pathname, struct m_inode* base,

334 {

335 const char * basename;

336 int inr, namelen;

337 struct m_inode * inode;338 struct buffer_head * bh;339 struct dir_entry * de;

340 }

341

```
// まず、指定されたパス名の最上位ディレクトリのディレクトリ名を見つけて
// そのi-node。存在しない場合は、NULLを返して終了します。もし、一番上の
// 返された名前が0の場合は、パス名がディレクトリの最後の項目にちなんだ名前になっていること
// を意味します。
```

```
// したがって、対応するディレクトリのi-nodeを見つけたので、それを返すことができます。
```

```
// i-nodeを直接入力します。返された名前の長さが0でなければ、dir_namei()を呼び出します。
```

```
// 新たに指定した開始位置を持つトップレベルのディレクトリ名を検索するために、再度関数を実行
```

311 する もし (!(dir = dir_namei(pathname, &namelen, &basename)))312 //ディレクトリペNULLを返された情報をもとに同様の判断をする。

313 もし (!namelen) /* 特殊なケース: '/usr/'など */.

314 dirを返す。

315 もし (!(base = dir_namei(pathname, &namelen, &basename, base)))316 NULLを返す。

317 もし (!namelen) /* 特殊なケース: '/usr/'など */.

318 リターンベース。

```
// 続いて、指定されたファイル名のディレクトリエントリのi-nodeを、返されたトップレベルの
// ディレクトリです。もう一度注意してください!もし、最後にもディレクトリ名ですが、そこには
// の後に「/」が付いていない場合は、最後のディレクトリのi-nodeを返しません。例えば、以下の
// ようになります。
```

```
// '/usr/src/linux' は 'src/' ディレクトリ名の i-node のみを返します。関数
```

```
// dir_namei()は、「/」で終わらない最後の名前をファイル名として扱いますが、これは必要なことです。
```

```
// この状況を処理するために、i-nodeのfind directory entry関数find_entry()を使用します。
```

```
// を別々に表示します。この時点では'de'にはディレクトリ・エントリ・ポインタが見つかり、
// 'dir'には
```

```

// ディレクトリエントリを含むディレクトリのi-nodeポインタ。
346     bh = find_entry(&base, basename, namelen, &de);
347     も (!bh) {
348         input(base)です。
349         NULLを返す。
350 // 次に、}ディレクトリ・エントリのi-node番号を取り、以下を含むキャッシュ・バッファ・ブロック
    を解放します。
// ディレクトリエントリを、ディレクトリのi-nodeに戻します。その後、i-nodeを取り
ノード番号に対応する // アクセスタイムを現在時刻に修正して
// 修正されたフラグです。最後に、i-nodeポインターinodeが返されます。さらに、現在の
// 処理されたディレクトリエントリがシンボリックリンク名であれば、follow_link() を使って
351 // ポイントするi-nodeをトリエントリのi-node。
352     プレルス(bh)です。
353     if (!(inode = iget(Base->i_dev, inr))) {
354         input(base)です。
355         NULLを返す。
356     }
357     if (follow_links)
358         inode = follow_link(base, inode);
359     その他
360         input(base)です。
361     inode->i_atime=CURRENT_TIMEです。
362     inode->i_dirt=1;
363     return inode;
364 }
365
    //// なく、指定されたパス名のi-nodeを取る。          シンボリックリンクの後に
    // パラメータ: pathname - パス名。
    // 対応するi-nodeを返します。
366 struct m_inode* lnamei(const char * pathname)
367 {
368     return namei(pathname, NULL, 0);
369 }
370
371 /*
372 * namei()
373 *
374 *は、ほとんどのシンプルなコマンドで、指定した名前のinodeを取得するために使
    用されます。
375 * 開く、リンクするなどは独自のルーチンを使用しますが、これで十分です。
376 chmod/などの*。
377 */
    //// 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。
    // パラメータ: pathname - パス名。
    // 対応するi-nodeを返します。
378 struct m_inode* namei(const char * pathname)
379 {
380     return namei(pathname, NULL, 1)
381 }    となります。
382
383 /*
384 *      open_namei()
385 *

```

386 * openのnamei - これは実際にはほとんどすべてのopen-routineです。

387 */

```
    //// ファイルを開くために使用される名目上の関数です。
    // パラメータ: filename - ファイルのパス名, flag - ファイルを開く際のフラグで、以下の値を
    取ることができます。
    // O_RDONLY (読み取り専用)、O_WRONLY (書き込み専用)、O_RDWR (読み取りと書き込み)、O_CREAT
    などの値があります。
    // (作成)、O_EXCL (ファイルが存在してはならない)、O_APPEND (ファイルの最後に追加) などの
    // Mode - 指定されたファイルの許可属性。
    // 新しいファイルを作成します。これらの属性は、S_IRWXU (ファイルの所有者が読み取り、書き込
    み、および実行の権限を持つ
    // パーミッション)、S_IRUSR (ユーザーがファイルの読み取りパーミッションを持っている)、
    S_IWUSR (グループメンバーが読み取り。
    // 書き込み、および実行権限)などがあります。新規に作成されたファイルの場合、これらの属性は
    // 将来のファイルへのアクセスにのみ使用され、読み取り専用のファイルを作成するオープンコール
    は
389   // また、読み書き可能なファイルハンドルを返します。インクルードファイルsys/stat.h, fcntl.h
390   を参照してください。const char * basename;
391   // 戻り値。正常に返された場合は0、そうでない場合はエラーコードが返されます; res_inode - 戻
392   り値
393   // 対応するファイルバス名のi-nodeポインタです。388 int
394   open(namei, const char * pathname, int flag, int mode,
395       struct dir_entry * de;
```

396 // まず、関数のパラメータが合理的に処理されます。もし、ファイルアクセスモードフラグが
 // 読み込み専用(0)だが、ファイル切り捨てフラグO_TRUNCが設定され、書き込み専用フラグO_WRONLY
 が
 // をファイルオープンフラグに追加しています。その理由は、切り捨てフラグO_TRUNCが
 // ファイルが書き込み可能な場合にのみ有効です。そして、現在のファイルのアクセス許可マスクを
 使用して
 // 処理し、与えられたモードで対応するビットをマスクし、通常のファイルフラグI_REGULARを追加
 します。
 // このフラグは、開いているファイルがない場合、新規ファイルのデフォルトプロパティとして使用
 されます。
 // 存在しない場合は、以下A11行目の注釈を参照してください。[400](#) mode |= I_REGULAR;
 // そしてmodeに指定された&^S_IWENT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。この時、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
 // 该当レコードを返します。操作が開いています。そのため、直接ディレクトリのi-nodeを返したり
 // 0を返す（終了しまだdir_name以外の場合、namei, &basename, NULL）i-nodeを戻して、エラー
 .
 も
 し
402 return -ENOENT
403 もし (!namelen) { /* 特殊なケース: '/usr/'など */。
 し
404 if (!(flag & (O_ACCMODE|O_CREAT|O_TRUNC))) {
405 *res_inode=dir;
406 0を返す。
407 }
 // そして、上記で取得したトップレベルのディレクトリ名のi-nodeに従って、ディレクトリの
 // パス名の最後のファイル名に対応する // エントリ構造が検索されます。
408 return EISDIR;
409 }
410 }

```

// ディレクトリエンtriesのキャッシュブロックが同時に取得されます。もし、キャッシュブロックの
// ポインタ
// がNULLの場合は、ファイル名に対応するディレクトリエンtriesが見つからないことを意味するので
// ファイルを作成することしかできません。このとき、ファイルを作成しない操作の場合は
// または、ユーザーがディレクトリに書き込む権利を持っていない場合は、i-nodeを戻します。
411 // bh = find_entry(&dir, basename, namelen, &de)とする。
412     if (!bh) {
413         if (!(flag & O_CREAT)) {
414             iput(dir);
415             return -ENOENT;
416         }
417         if (! permission(dir, MAY_WRITE))
418             { iput(dir);
419             return -EACCES;
420         }
421     // ここで、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
422     // パス名で指定されたファイル名を使用するために、デバイス上の新しいi-nodeを申請します。
423     // そして、新しいi-nodeの初期設定を行います：ユーザーIDの設定、アクセスモードの設定、モディ
424     // フアイラクの設定指定されたディレクトリdirに新しいディレクトリエンtriesを追
425     // てます。その 加します。
426     // 後      inode = new_inode(dir->i_dev);
427     //       if (!inode) {
428         //         iput(dir) です。
429         //         return -ENOSPC;
430     }
431     //       inode->i_uid = current->euid;
432     //       inode->i_mode = mode;
433     //       inode->i_dirt = 1;
434     //       bh->add_entry(dir, basename, namelen, &de)です
435     //       if (!bh) {
436     //         // キャッシュブロックを解放し、i-nodeを戻します。そして、i-nodeポインタの
437     //         // iput(inode) です。
438     //         // iput(dir) です。
439     //         //         return -ENOSPC;
440     //       }
441     //       de->inode = inode->i_num;
442     //       bh->b_dirt = 1;
443     //       // ブレルス (bh) です。
444     //       // iput(dir) です。
445     //       *res_inode = inode;
446     //       0を返す。
447   }
448
449 // 上記(411行目)でファイルに対応するディレクトリエンtriesの取得に成功した場合は
450 // 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
451 // を示します。その後は
452 // ディレクトリ・エントリーの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
453 // 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。

```

```

// 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
443 了しますinr = de->inode;
444      dev = dir->i_dev;
445      ブレルス (bh) です。
446      if (flag & O_EXCL) {...}
447          input(dir) です。
448          return -EEXIST;
449      }
450
// そして、ディレクトリ・エントリのi-nodeの内容を読みます。i-nodeがディレクトリのi-nodeであ
// れば
451 // そして、アクセスモードが書き込み専用または読み書き兼用、またはアクセス許可がない場合は、
452 // 元に戻します。
453 // if (!(inode = follow_link(dir, iget(dev, inr))))
454 //     return -EACCES;
455 //     if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
456 //         ! permission(inode, ACC_MODE(flag))) {
457 //         input(inode);
458 //         return -EPERM;
459 //     }
460
// そして、i-nodeのaccess timeフィールドを現在の時刻に更新します。もし、切り捨てフラグが
// が設定されている場合は、ファイルサイズが0に切り捨てられます。最終的には、iノードへのポイ
461 // ンタを返します。
462 // inode->i_atime = CURRENT_TIME;
463 //     if (flag & O_TRUNC)
464 //         truncate(inode) です。
465 //     *res_inode = inode;
466 //     return 0;
467
468
//// デバイスの特殊ファイルまたは通常のファイルノードを作成します。
// ファイルシステムノード（コモンファイル、デバイススペシャルファイル、名前付きパイプ）を作
469 成します。
// モードとdevで指定された名前のファイル名。
// Parameters: filename - ファイル名のパス名; mode - 使用する権限を指定して
// 作成されたノードのタイプ、dev - デバイス番号。
// 戻り値: 成功した場合は0、そうでない場合はエラーコードを返します。
470 int sys_mknod(const char * filename, int mode, int dev)
471 {。    const char * basename;
472     int namelen;
473     struct m_inode * dir, * inode;
474     struct buffer_head * bh;
475     struct dir_entry * de;
476
// 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
477 クセス許可を返します。
// エラーコードです。パス名のトップレベルディレクトリに対応するi-nodeが
// 見つかった場合は、エラーコードが返されます。トップファイル名の長さが0の場合は、与えられ
478 た
479 // パス名の最後にアフターバックスラッシュ(名が指定されていないか、プロセスに書き込み権限がない場合
480 // ディレクトリの中 returnがありません)場合は、ディレクトリの i-node を戻して、エラーコードを返します
481 。   もし (!(dir = dir_namei(filename, &namelen, &basename, NULL)))
482     return -ENOENT
483    もし (!namelen) {

```

```

477         iput(dir);
478         return -ENOENT;
479     }
480     も (! permission(dir, MAY_WRITE))
481     し {
482         iput(dir) です。
483         return -EPERM;
484 // 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。
485 // 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
486 // パス名に対応する最後のファイル名が既に存在する場合、そのファイルを含むバッファブロックは
487 // 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
488 // bh = find_entry(&dir, basename, namelen, &de); if
489     (bh) {
490         brelse(bh);
491         iput(dir);
492         return -EEXIST;
493     }
494 // ファイル名のディレクトリエントリが見つからない場合は、新しいi-nodeを申請して
495 // i-nodeの属性モードを設定します。ブロックデバイスファイルを作成している場合や、キャラクタ
496 // デバイスファイルでは、i-nodeのダイレクト論理ブロックポインタ0をデバイス番号と同じにする
497 // つまり、デバイスファイルの場合、i-nodeのi_zone[0]には、デバイス番号が格納されています。
498 // デバイスファイルで定義されたデバイスです。の修正時間とアクセス時間を設定します。
499 // i-nodeを現在の時間に変更し、i-nodeの修正フラグを設定する。
500     inode = new_inode(dir->i_dev);
501     if (!inode) { // inodeを戻し、失敗したらエラーコードを返す。
502         iput(dir) です。
503         return -ENOSPC;
504     }
505     inode->i_mode = mode;
506     if (S_ISBLK(mode) || S_ISCHR(mode))
507         inode->i_zone[0] = dev;
508     inode->i_mtime = inode->i_atime = CURRENT_TIMEとなります。
509     inode->i_dirt = 1;
510 // 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（
511 // キャッシュブロックポインタが
512 // ディレクトリのエントリーを含む // がNULLの場合）、ディレクトリのi-nodeを戻し、リセットしま
513 // す。
514 // bh = add_entry(dir, basename, namelen, &de);
515     if (!bh) {
516         iput(dir) です。
517         inode->
518         i_nlinks=0;
519         iput(inode);
520         return -ENOSPC;
521     }
522 // ここ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
523 // ディレクトリ・アイテムです。ディレクトリ・エントリのi-nodeフィールドを新しいi-node番号と
524 // 同じにします。inode = inode->i_num;
525 // そしてbh-キャッシュブロック修正フラグを設定し、ディレクトリと新しいi-nodeを戻し、リリース
526 // します。iput(dir) です。
527 // そして最後に0(成功)を返します。
528 // IPUT(inode) です。
529 // ブレルス (bh) です。

```

```

512          0を返す。
513 }
514
515     //// ディレクトリを作る (system-callで使用)。
516     // 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。
517     // 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。
518     int sys_mkdir(const char * pathname, int mode)
519     {
520         const char * basename;
521         int namelen;
522         struct m_inode * dir, * inode;
523         struct buffer_head * bh, *dir_block;
524         struct dir_entry * de;
525
526         // まず、操作権限とパラメータの有効性を確認し、i-nodeの
527         // 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
528         // i-nodeが
529         // 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
530         // ル名が
531         // 長さが0の場合、与えられたパス名の最後にファイル名が指定されていないことを意味します。
532         // プロセスがディレクトリに書き込み権限を持っていない場合は、i-nodeを戻します。
533         // ディレクトリを作成し、エラーコードを返します。
534         if (!(dir = dir_namei(pathname, &namelen, &basename, NULL)))
535             return -ENOENT;
536         if (!namelen) {
537             iput(dir);
538             return -ENOENT;
539         }
540         if (! permission(dir, MAY_WRITE))
541             { iput(dir);
542             return -EPERM;
543         }
544
545         // 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
546         // 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
547         // ディレクトリ
548         // 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
549         // ックは
550         // 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
551         // 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
552         // て
553         bh = find_entry(&dir, basename, namelen, &de)です。
554         // i-nodeもしくは(bh){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
555         // バイト(2つのディレクトリエントリのサイズ)、ノードの修正フラグを設定し、ノードの
556         // 修正時間とアクセス時間です。2つのディレクトリエントリは、「...」と「...」のディレクトリ
557         // に使用されます。return -EEXIST;
558         // それぞれ
559         inode = new_inode(dir->i_dev);
560         if (!inode) { // inodeを戻し、失敗したらエラーコードを返す。
561             iput(dir)です。
562             return -ENOSPC;
563         }
564         inode->i_size = 32;
565         inode->i_dirt = 1;
566         inode->i_mtime = inode->i_atime = CURRENT_TIMEとなります。

```

```

// 次に、新しいi-nodeのディレクトリ・エントリ・データを保存するためのディスク・ブロックを申請します。
// そして、i-nodeの最初のダイレクトブロックポインタをブロック番号と同じにします。もし、その
// ような
// アプリケーションが失敗したら、ディレクトリのi-nodeを戻し、新しく要求されたi-nodeをリセットする
547 // リンクカウント；新しいi-nodeを戻し、スペースエラーコードを返さない。そうでなければ、新し
548 // い
549 // i-node modified flag.
550     if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
551         input(dir);
552         inode->i_nlinks--;
553         ; input(inode);
554         return -ENOSPC;
555     }
556     inode->i_dirt = 1;
557 // 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
558 // 対応するブロックをバッファキャッシュに入れる）。同様に、エラーが発生した場合は、i-nodeを
559 戻す
560 // ディレクトリの // 要求されたディスクブロックを解放し、新たに要求された i-node リンクをリセ
561 ットする。
562 // count; 新しいi-nodeを戻し、no spaceエラーコードを返して終了。 if
563     (! (dir_block=bread(inode->i_dev, inode->i_zone[0]))) {
564         input(dir) です。
565         inode->i_nlinks--;
566         ; input(inode);
567         return -ERROR;
568     }
569 // 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。
570 // ファイルをバッファブロックに格納します。まず、'de'にディレクトリ名を設定します。
571 // de->inode=inode->i_num など
572 // エントリーの i-node-number フィールドを、新しく適用された
573 // i-node 番号、で name フィールドには".."が設定されます。すると'de'は次のディレクトリ・エントリ
574 を指す de->inode = dir->i_num; // '..' のディレクトリエントリを設定します。
575 // 構造体に 親ディレクトリの i-node 番号と名前「..」を格納します。
576 // そして inode->i_nlinks=2; ブロックモディファイドフラグをセットし、バッファブロックをリリースしま
577 // す。モードを再初期化します。
578 // de = (struct dir_entry *) dir_block->b_data;
579     inode->i_mode = I_DIRECTORY | (mode & 0777 & ~current->umask);
580 // 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。
581 // 新しく作成されたディレクトリの名前。失敗した場合（バッファブロックポインタの
582 // ディレクトリ・エントリーがNULLの場合）、ディレクトリのi-nodeを戻します；要求されたi-node
583 bh = add_entry(dir, basename, namelen, &de); // 参照リンクカウントはリセットされてi-node
584 // に戻される。エラーコードを返して終了する。bh =
585     add_entry(dir, basename, namelen, &de);
586     if (!bh) {
587         input(dir) です。
588         inode->i_nlinks=0 です。
589         input(inode);
590         return -ENOSPC;
591     }
592 // 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
593 // 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
782

```

```

// ブロックが解放され、最後に0（成功）が返される。 de-
578     >inode = inode->i_num;
579     bh->b_dirt = 1;
580     dir->i_nlinks++;
581     dir->i_dirt = 1;
582     iput(dir);
583     iput(inode) とな
584     ります。
585     brelse(bh);
586 }      return 0;
587
588 /*
589 * 指定されたディレクトリが空であるかどうかをチェックするルーチン
(rmdir 用)
590 */
591 static int empty_dir(struct m_inode * inode)
592 {...    int nr,block;
593     int len;
594     struct buffer_head * bh;
595     struct dir_entry * de;
596
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
598 // .
599 // 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。少なくとも
600 // 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
601 // の数が2未満であるか、またはディレクトリのiノードの最初のダイレクトブロックが
602 // 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、警告の
603 // のメッセージが表示され、0が返されます。(失敗)となります。
604     len = inode->i_size / sizeof (struct dir_entry); // ディレクトリ・エントリの数。
605     if (len<2 || !inode->i_zone[0] ||)
606         !(bh=bread(inode->i_dev, inode->i_zone[0])) {
607             printk ("warning - bad directory on dev %04x\n", inode->i_dev).
608             0を返す。
609 // 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
610 // させています。
611 // 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
612 // す。については
613 // 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
614 // 同じでなければなりません。
615 // 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
616 // ィールド
617 // 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
618 // 最初のディレクトリ・エントリの番号がカレント・ディレクトリの番号と一致しない場合、または
619 // 2番目のディレクトリ・エントリのi-node番号が0であるか、または2つのディレクトリのnameフィ
620 // ールドが
621 // de = (struct dir_entry *) bh->b_data;
622     if (de[0].inode != inode->i_num || !de[1].inode ||
623         strcmp(".", de[0].name) || strcmp("..", de[1].name))
624     {
625         printk ("warning - bad directory on dev %04x\n", inode->i_dev);
626         return 0;
627     }
628 // 次に、'nr'をディレクトリのエントリ番号(0から数えたもの)とし、'de'を
629 // の残りの(len - 2)のディレクトリ・エントリをすべてループします。
630 // i-node numberフィールドが0(使用済み)でないかどうか、ディレクトリを確認します。

```

```

610      nr=2としました
611      .
612      de += 2である。
613 // 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
614 // 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
615 // 、次の
616 // 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。
617 // 読み込みの
618 // 方法は、対応するデータブロック番号 (nr/DIR_ENTRIES_PER_BLOCK) の
619 // 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
620 // 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
621 // 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
622 // バッファブロック内のデータがまだ読み込まれていません (volid * bh->data が BLOCK_SIZE ではない)
623 // (あるいは使用されたことがない、そのような)
624 // ファイルが削除されたなどの理由で)、次のブロックの読み取りを続行し、読み取れない場合は
625 // の場合、エラーは0を返します。bmap()が例外の場合 (BLOCK_SIZE が DIR_ENTRIES_PER_BLOCK 内の最初のディレクトリエントリを指定します。
626 // を指定します。
627 if (!block) {
628     nr += DIR_ENTRIES_PER_BLOCK;
629     を続けています。
630 }
631     if (!bh=bread(inode->i_dev, block))
632         0を返す。
633 // 現在のディレクトリ・エントリ (struct dir_entry *) bh->bh_data フィールドが0に等しくない場合、そ
634 // れは次のことを意味します。
635 // 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
636 // 合は
637 // バッファブロックが解放され、0が返されます。それ以外の場合は、すべてのディレクトリエント
638 // リ
639 // ディレクトリ内のがクエリされていない場合、ディレクトリエントリ番号「nr」がインクリメント
640 // されて
641 // 'de' は次のディレクトリエントリを指しており、検出は継続されます。 if (de-
642     >inode) {
643         brelse(bh);
644         return 0;
645     }
646     de++;
647     nr++
648     .
649 return 1; // ディレクトリは空です!
650 // ここでコードが実行された場合、使用されたディレクトリエントリが
651 // ディレクトリ (もちろん最初の2つを除く) に移動した後、バッファブロックが解放され、1
652 // //を返します。
653 // パラメータ: name (hh) - ディレクトリ名 (パス名) です。
654 // 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
655 // す。
656 int sys_rmdir(const char * name)
657 {
658     const char * basename;
659     int namelen;
660     struct m_inode * dir, * inode;
661     struct buffer_head * bh;
662     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
// 長さが0の場合、与えられたパス名の最後にファイル名が指定されていないことを意味します。
// プロセスがディレクトリに書き込み権限を持っていない場合は、i-nodeを戻します。
// ディレクトリを作成し、エラーコードを返します。
```

643

```
if (!(dir = dir_namei(name, &namelen, &basename, NULL)))
```

644

```
return -ENOENT
```

645

```
if (!namelen) {
```

646

```
iput(dir)です。
```

647

```
return -ENOENT
```

648

```
}
```

649

```
if (! permission(dir, MAY_WRITE)) {...
```

650

```
iput(dir)です。
```

651

```
// そして、指定されたディレクトリのi-nodeとディレクトリ名に応じて、関数
```

652

```
find_entry()は、ディレクトリ・エントリを見つけるために、そのエントリを含むバッファ・ブロ
ック'bh'を使用します。
```

```
// ディレクトリ・エントリ、ディレクトリのi-node 'dir'、ディレクトリ・エントリ 'de' が返され
ます。
```

```
// そして、ディレクトリ・エントリ「de」のi-node番号にしたがって、対応するi-nodeの
```

```
// は、iget()関数を使って取得します。特定の操作の過程で、もしディレクトリ
```

```
// パス名の最後のディレクトリ名のエントリーが存在しない場合、そのエントリーを含むバッファアブ
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
```

653

```
// のファイルがすでに存在している場合は、その旨が返されます。ディレクトリエントリのi-nodeを
取得する操作が
```

654

```
// が間違っていると、ディレクトリのi-nodeが戻され、ディレクトリを含むバッファアブロックが
```

655

```
// bh = find_entry(&dir, basename, namelen, &de)となりま
```

```
す。
```

656

```
if (!bh) {
```

657

```
iput(dir);
```

658

```
return -ENOENT;
```

659

```
}
```

660

```
if (!(inode = iget(dir->i_dev, de->inode))) {
```

661

```
iput(dir);
```

662

```
brelse(bh);
```

663

```
return -EPERM;
```

```
}
```

```
// この時点では、ディレクトリのi-node 'dir'、削除されるディレクトリ・エントリ 'de' があり
ます。
```

```
// とそれに対応するi-nodeがあります。以下では、削除の実行可能性を確認するために
```

```
// この3つのオブジェクトの情報を
```

```
//
```

664

```
// ディレクトリ(dir->i_mode & S_ISVX)が設定されているかかつ、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
```

665

```
// i-nodeのinode->ID値と現在のプロセスIDもしくは削除する権限を持っていないことを意味します。
```

666

```
// ディレクトリ名のi-nodeでなければ、そのディレクトリのi-nodeとエントリーのi-nodeを戻して、次に
```

667

```
// バッファアブロック iput(inode)を返します。
```

668

```
brelse(bh)です。
```

669

```
return -EPERM;
```

```

// ディレクトリエントリのi-nodeのデバイス番号が、i-nodeのデバイス番号と等しくない場合。
// このエントリーを含むディレクトリ、または削除するディレクトリの参照リンク数が
// が1より大きい場合（シンボルリンクがあることなど）、そのディレクトリは削除できません。
// そのため、削除されるディレクトリ名を含むディレクトリのi-nodeと、エントリのi-nodeを
// if (inode->i_dev != dir->i_dev || inode->i_count>1) {。

670         iput(dir)です。
671         iput(inode);
672         brelse(bh);
673         return -EPERM;
674     }
675 // ディレクトリエントリのi-nodeがディレクトリのi-nodeと等しい場合、そのことを示します。
// ". "ディレクトリを削除しようとしていますが、これは許されません。すると、ディレクトリ・エント
// リのi-node
// と削除するディレクトリのi-nodeを戻し、バッファロックをリリースする。
// となつてのエラーコードが返されます。
// います。
676     もし (inode == dir) { /* 「...」は削除できないが、「.../dir」はOK */。
677         iput(inode)です。
678         iput(dir)です。
679         ブレルス(bh)です。
680         return -EPERM;
681     } // 削除されるディレクトリのi-nodeの属性が、それがないことを示している場合は
// ディレクトリには、この削除操作の前提となるものが全く存在しません。すると、i-nodeの
// 削除されるディレクトリ・エントリの // とそのディレクトリの i-node が戻されると、バッファ
// S_ISDIR(inode->i_mode)) {...

682         iput(inode);
683         iput(dir)で
684         す。
685         brelse(bh);
686         return -ENOTDIR;
687     }

// 削除するディレクトリが空でない場合は、削除できません。すると、ディレクトリのi-node
// 削除されるディレクトリ名と削除されるディレクトリのi-nodeを含む
// if (! empty_dir(inode)) {
688         iput(inode);
689         iput(dir);
690         brelse(bh)で
691         す。
692         return -ENOTEMPTY;
693     }

// 空のディレクトリの場合、ディレクトリ・エントリ・リンクの数は 2 (上部の
// ディレクトリと自身) を削除します。削除するi-nodeのリンク数が、以下の値に等しくない場合
// 2の場合は、警告メッセージが表示されますが、削除操作は続行されます。その後、i-nodeの
// 削除するディレクトリのディレクトリエントリの // number 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、バッファロックモディファイドフラグが設定されて
// いることを示す
694 // バッファをnodeを解放します) そして、削除されたi-nodeのリンク数を設定します。
695 // ディレクトリを開放して、i-nodeを空にします。link = 0でnodeを設定します。
// なります。
696     de->inode = 0となります。
697     bh->b_dirt = 1;
698     ブレルス(bh)です。
699     inode->i_nlinks=0です。

```

```

700     inode->i_dirt=1;
    // 次に、削除されたディレクトリ名を含むディレクトリのi-nodeリンクカウントを減らす
    // を1ずつ加算し、変更時刻と修正時刻を現在の時刻に修正し、修正した
    // ノードのフラグです。最後に、削除されるディレクトリ名を含むディレクトリi-ノード
    // と削除するディレクトリのi-nodeが戻され、0が返されます（削除の
    // 操作が成功した場合）。
701     dir->i_nlinks--;
702     dir->i_ctime = dir->i_mtime = CURRENT_TIME
    となります。
703     dir->i_dirt=1;
704     iput(dir)です。
705     iput(inode)です。
706 }
707     0を返す。
708
    //// ファイル名に対応するディレクトリエントリを削除（解除）します。
    // ファイルシステムから名前を削除します。それがファイルへの最後のリンクであり、どのプロセ
    // スも
    // ファイルを開くと、そのファイルも削除され、占有していたデバイスの領域が解放されます。
    // パラメータ: name - ファイル名（パス名）です。
    // 戻り値: 成功した場合は0、そうでない場合はエラーコードを返します。
709 int sys_unlink(const char * name)
710 {
711     const char * basename;
712     int namelen;
713     struct m_inode * dir, * inode;
714     struct buffer_head * bh;
715     struct dir_entry * de;
716
    // まず、操作権限とパラメータの有効性を確認し、i-nodeの
    // パス名にトップレベルのディレクトリが含まれています。トップレベルのディレクトリに対応する
    i-nodeが
    // パス名に含まれるディレクトリが見つからない場合は、エラーコードが返されます。トップファイ
    ル名が
    // 長さが0の場合、与えられたパス名の最後にファイル名が指定されていないことを意味します。
717 // プロセスがディレクトリに書き込み権限を持っていない場合は、i-nodeを戻します。
718 // ディレクトリを作成し、エラーコードを返します。
719     if (!(dir = dir_namei(name, &namelen, &basename, NULL))
        return -ENOENT;
720
721     if (!namelen) {
722         iput(dir);
723         return -ENOENT;
724     }
725     if (! permission(dir, MAY_WRITE))
        { iput(dir);
        return -EPERM;
    }
726
    // そして、指定されたディレクトリのi-nodeとディレクトリ名に応じて、関数
    // find_entry()は、ディレクトリ・エントリを見つけるために、そのエントリを含むバッファ・ブロ
    ック'bh'を使用します。
727 // ディレクトリ・エントリ「de」、ディレクトリ・エントリ「dir」が返され
    ます。      です。
728 // それでも、バッファ・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。
729     iput(dir)です。
730     return -ENOENT
731 }
```

```

732      も (! (inode = iget(dir->i_dev, de->inode))) {
733          し
734              input(dir) です。
735              ブレルス(bh) です。
736              return -ENOENT
737      } // 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。
738      // とそれに対応するi-nodeがあります。以下では、削除の実行可能性を確認するために
739      // この3つのオブジェクトの情報を
740      //
741      // 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
742      // 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。
743      // 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。
744      // 現在のプロセスがディレクトリを削除する権限を持っていないことを示します。その後、戻して
745      // 削除されるディレクトリ名を含むディレクトリのi-nodeと
746      // if ((dir->i_mode & S_ISVTX) && ! suser() && )
747          current->euid != inode->i_uid &&
748          current->euid != dir->i_uid) {
749              input(dir);
750              input(inode);
751              brelse(bh)で
752              す。
753              return -EPERM;
754      }
755      // 指定されたファイル名がディレクトリの場合は、削除できません。そのときは、i-nodeの
756      // ディレクトリのi-nodeとファイル名のディレクトリ・エントリが戻されると、その中のバッファ・
757      ブロックは
758      // if (S_ISDIR(inode->i_mode)) {
759          input(inode) です。
760          input(dir);
761          brelse(bh);
762          return -EPERM;
763      }
764      // 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に修正される。 if
765      (!inode->i_nlinks) {
766          printk("Deleting nonexistent file (%04x:%d), %d\n",
767                  inode->i_dev, inode->i_num, inode->i_nlinks);
768          inode->i_nlinks=1です。
769      }
770      // ここで、ファイル名に対応するディレクトリエントリを削除します。すると、i-nodeの番号
771      // ファイル名ディレクトリエントリの // フィールドが 0 に設定されている場合、そのディレクトリエ
772      ントリがde->inode = 0と
773      // 解放されません。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ます。
774      bh->b_dirt = 1;
775      // 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。
776      // 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
777      // 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
778      // ファイル、つまり、i-nodeのリンク数から1を引いた値が0になり、どのプロセスも開いていない
779      状態
780      // この時点では、input()がファイルを戻すために呼び出されたときに、ファイルは削除されます。
781      // i-nodeと占有していたデバイス空間を解放する。fs/inode.c, line 183 を参照。
782      inode->i_nlinks--;
783      inode->i_dirt = 1;
784      inode->i_ctime = CURRENT_TIMEです。

```

```

762     iput(inode);
763     iput(dir);
764     return 0;
765 }
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。
    // 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。
767 int sys_symlink(const char * oldname, const char * newname)
768 {.
    struct dir_entry * de;
769     struct m_inode * dir, * inode;
770     struct buffer_head * bh, * name_block;
771     const char * basename;
772     int namelen, i;
773     チャー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i-nodeが
    // パス名のディレクトリを戻し、エラーコードを返します。また、ユーザーが
    // を新しよ:て新しレタ名の書き込み権限を持つodをを渡し、場合其ノシトを確立せません。
776     から     dir = dir_namei(newname, &namelen, &basename, NULL);
777     if (!dir)
778         return -EACCES;
779     if (!namelen) {
780         iput(dir)です。
781         return -EPERM;
782     }
783     if (!permission(dir, MAY_WRITE)) {....
784         iput(dir)です。
785         return -EACCES;
786     }
787
    // ここで、ディレクトリで指定されたデバイスに新しいi-nodeを申請し、i-nodeを設定します。
    // モードをプロセスが指定したシンボリックリンクタイプとモードマスクコードに設定して
    // i-node modified flag.
788     も (! (inode = new_inode(dir->i_dev))) {
789         iput(dir)です。
790         return -ENOSPC;
791     }
792     inode->i_mode = S_IFLNK | (0777 & ~current->umask);
793 // シンボリックリンクのパス名の文字列を保存するためには、ディスクブロックを
    // i-nodeで、最初のダイレクトブロック番号i_zone[0]を、得られた論理的な
    // ブロック番号を指定して、i-node modified flagを設定します。アプリケーションが失敗した場合
    // は、元に戻して
    // //戻されるi-nodeはトタのi-node:ラ新かに要素返されま戻i-nodeのリンクカウントをリセットし、元に
    // 戻す もし (! (inode->i_zone[0]=new_block(inode->i_dev))) {
794         iput(dir)です。
795         inode->i_nlinks--;
796         iput(inode)です。

```

```

797         return -ENOSPC;
798     }
799     inode->i_dirt = 1;
// 新たに要求されたディスクブロックは、デバイスから読み込まれます（目的は、ディスクブロックに
// ブロックをバッファキャッシュに格納します）。エラーが発生した場合は、ディレクトリのi-node
800 を戻します；reset
801 // 新たに要求されたi-nodeのリンクカウント；新しいi-nodeを戻し、エラーコードを返す。 if
802     (!(name_block=bread(inode->i_dev, inode->i_zone[0]))){
803         input(dir)です。
804         inode-> i_nlinks--
805         ; input(inode);
806         return -ERROR;
}
// これで、シンボリックリンク名の文字列をこのディスクブロックに入れることができます。ディスクブロックの長さは
// は1024バイトなので、デフォルトのシンボルリンク名の最大長は1024バイトにしかなりません。私たちは
// ユーザ空間のシンボリックリンク名の文字列を、ディスクブロックのあるバッファブロックにコピーする。
// が配置されており、バッファブロックモディファイドフラグを設定しています。ユーザーが入力した文字列が
806     while (i < 1023 && (c=get_fs_byte(oldname++)))
807 // ヌルを入れずに終了した場合、バッファブロックのデータ領域の最後のバイトにヌルを入れる必要があります。
808     name_block->b_data[i+f] = c;
809 // その後、バッファブロックを解放し、対応するファイルのデータのサイズを
810     name_block->b_size = i+f;
811 // シンボリックリンク名の文字列の長さと同じになるように // i-nodeを設定します。
812 // 修正されたフラグ。
813     inode->i_size = i;
814     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
// シンボリックリンクのファイル名が既に存在する場合、ディレクトリエントリを含むバッファブロックは
814 // リリースされ、新たに要求されたi-nodeのリンクカウントがリセットされ、ディレクトリのi-node
815 が
816 // bh = find_entry(&dir, basename, namelen, &de);
817     if (bh) {
818         inode-> i_nlinks--
819         ; input(inode);
820         brelse(bh);
821         input(dir);
822         return -EXIST;
}
// 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。
// 新たに作成されたディレクトリが戻されますが、ファイル名の // 番号とディレクトリ名を指定します。失敗
822 した場合(= add_entry(dir, basename, namelen, &de))
// 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
823 // が戻され、(bh)が戻されたi-nodeの参照リンクカウントがリセットされ、i-nodeが戻されます。
// し
824     inode->i_nlinks--;
825     input(inode)です。
826     input(dir)です。
827     return -ENOSPC;
828 }

```

```

// 最後に、新しいディレクトリ・エントリのi-nodeフィールドは、新しいi-node番号に等しく、そ
の
// バッファブロック修正フラグが設定されると、バッファブロックが解放され、ディレクトリと新
829 // i-nodeが戻され、=最後の0(成功)が返されます。
830     bh->b_dirt = 1;
831     ブレス(bh)です。
832     input(dir)です。
833     input(inode)です。
834     0を返す。
835 }
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。
// 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。
837 int sys_link(const char * oldname, const char * newname)
838 {
839     struct dir_entry * de;
840     struct m_inode * oldinode, * dir;
841     struct buffer_head * bh;
842     const char * basename;
843     int namelen;
844

// まず、元のファイル名を検証します。ディレクトリ名ではなく、存在している必要があります。そ
こで、まず
// 元のファイルパス名に対応するi-node 'oldinode' を取る。0であれば、それは
// エラーが発生し、エラーコードを返します。元のパス名がディレクトリに対応する場合は
845 //名の場合はinodeが戻されません。コードも返されます。
846     if (!oldinode)
847         return -ENOENT
848     if (S_ISDIR(oldinode->i_mode))
849     {...}
850         input(oldinode)です。
851         return -EPERM;
852 // 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
853     .
854     // 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
855     // 元のパス名が戻され、エラーコードが返されます。もし、ファイル名が
856     // 新しいパス名に含まれる、元のパス名のi-nodeと新しいパス名の
857     // dir = dir_namei(newname, &namelen, &basename, NULL); if
858     (!dir) {
859         input(oldinode);
860         return -EACCES;
861     }
862     if (!namelen) {
863         input(oldinode);
864         input(dir);
865         return -EPERM;
866     }
867     // 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
868     // 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
869     // 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
870     // のエラーコードが返されます。また、ユーザーが書き込み権限を持っていない場合は

```

```

// 新しいディレクトリでは、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は
// となって 元のパス名のi-nodeが戻され、エラーコードが返されます。
862   います。 (dir->i_dev != oldinode->i_dev) {
    もし
863     input(dir)です。
864     input(oldinode)です。
865     return -EXDEV;
866   }
867   も (! permission(dir, MAY_WRITE)) {
    し
868     input(dir)です。
869     input(oldinode)です。
870 // 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
871 // リンクを設定します。その後、既存のディレクトリエントリを含むバッファブロックが解放されて
// 新しいパス名のディレクトリのi-nodeと、元のパス名のi-nodeが戻される。
// とエラーコードが返されます。
872   bh = find_entry(&dir, basename, namelen, &de);
873   if (bh) {
    brelse(bh);
874     input(dir);
875     input(oldinode)
    。
876     return -EXIST;
  }
// 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
879 // 元のパス名のi-node番号と同じでない場合、修正フラグが
// 新たに追加され、ディレクトリ・エントリを含む // がセットされ、バッファ・ブロックがリリース
880 されてもし (!bh) {
881 // ディレクトリのinput(dir)が戻される。
882   input(oldinode)です。
883   return -ENOSPC;
884 }
885   de->inode = oldinode->i_num;
886   bh->b_dirt = 1;
887   ブレルス(bh)です。
888 // その後、元のノードのリンクカウントを1増やし、変更時間を現在のものに修正します。
// の間に合わせて、i-node modified flagを設定する。最後に、元のパスのi-nodeを戻します。
// 名前を付けて0(成功)を返す
889   . oldinode-
890   >i_nlinks++;
891   oldinode->i_ctime = CURRENT_TIMEとなります。
892   oldinode->i_dirt = 1;
893   input(oldinode);
894 } return 0;
895

```

8. file_table.c

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。

2. コードアノテーション

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ーファイル。ファイルテーブル構造を定義する (file,
buffer_head,
// m_inode など)。
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ー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。

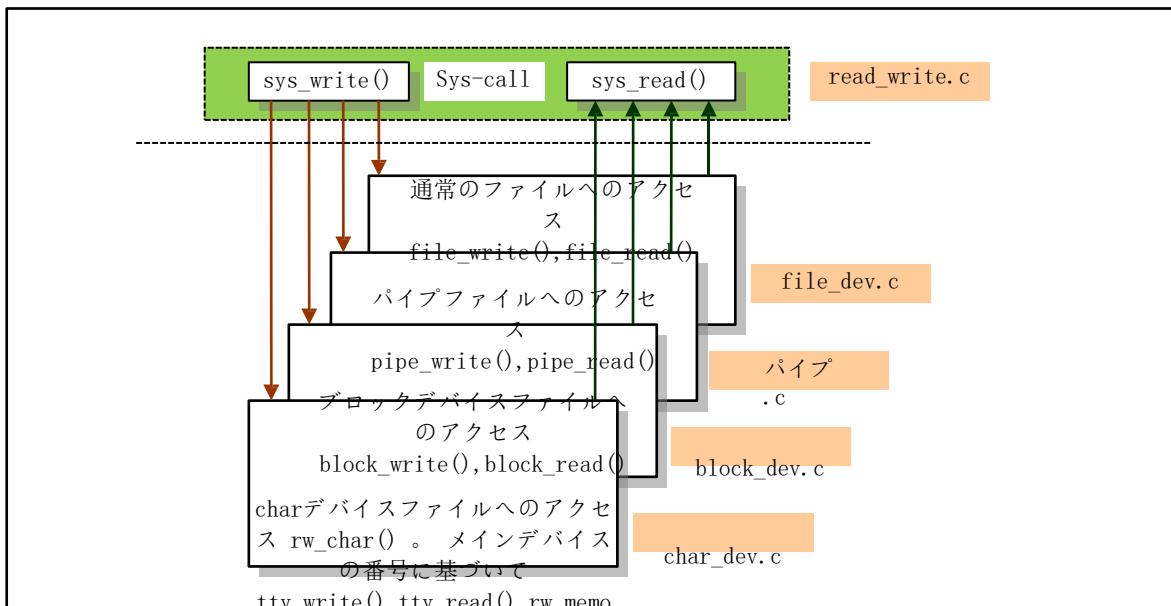


図12-26 各種ファイルとシステムコールのインターフェース機能

12.9.1 機能

`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。

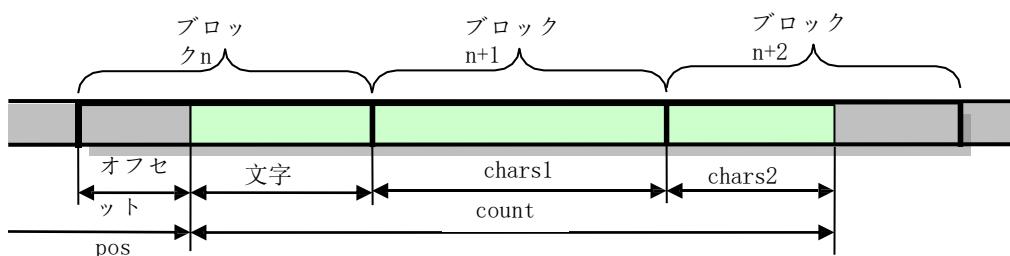


図12-27 ブロックデータ読み書き動作ポインタ位置図

ユーザーのバッファは、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、または動的に適用される

を実行します。この関数を呼び出す前に、システムは、ユーザー・バッファが使用する仮想リニアアドレスを、メインメモリ領域の対応するメモリページにマッピングします。

block_read()関数は、データがバッファからユーザーが指定した場所にコピーされることを除けば、block_write()と同じように動作します。

12.9.2 コードアノテーション

プログラム 12-8 linux/fs/block_dev.c

```

1 /*
2 * linux/fs/block_dev.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <errno.h> エラー番号のヘッダーファイルです。システムの様々なエラー番号を含みます。
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// の初期タスク0と、記述子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
// のマクロ文が含まれています。
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネルの使用する機能
// <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が定義されています。
// セグメントレジスタの操作。
// <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。
// ディスクリプタ/割り込みゲートなどが定義されています。 7 #include <errno.h>
8
9 #include <linux/sched.h>
10 #include <linux/kernel.h> (日本語)
11 #include <asm/segment.h>
12 #include <asm/system.h>
13
// デバイスデータブロックの総数を示すポインターの配列。そのポインタのそれぞれは
// 指定されたメジャー・デバイス番号の総ブロックの配列であるhd_sizes[]に各項目は
// ブロックアレイの合計数は、データブロックの合計数に対応します。
// サブデバイス番号で決まるサブデバイス。
14 extern int *blk_size[]; // blk_drv/ll_rw_blk.c, line 49. 15
// ブロックライト機能 - 指定されたサイズのデータを、デバイス上の指定されたオフセットに書き込みます。
// パラメータ: dev - デバイス番号、 pos - デバイスファイル内のオフセットポインタ。
// buf - ユーザースペースのバッファ； count - 転送するバイト数。
// 書き込んだバイト数を返します。データが書き込まれていない場合やエラーが発生した場合は、エラーコードを返します。
// カーネルにとっての書き込み操作とは、キャッシュにデータを書き込むことです。データが最終的に
// デバイスに書き込まれた//をキャッシュマネージャーが判断して処理します。また、今回は
// ブロックデバイスは、ブロック単位で読み書きを行いますが、書き込み位置が
// ブロックの端に位置する場合は、データがあるブロック全体を先に読み出す必要があります。
// とし、書き込み位置からデータを埋めていきます。その後、データの完全なブロックを書き込む
// をディスクに送信します（つまり、バッファキャッシュに処理します）。
16 int block_write(int dev, long * pos, char * buf, int count)
17 {
// まず、ファイルの位置'pos'をブロック番号'block'に変換し、その中に
// のディスクブロックの読み書きが開始されるオフセット位置「offset」と

```

```

// ブロックに書き込まれる最初のバイトを取得します。
18     int block = *pos >> BLOCK_SIZE_BITS;           // posが設置されているブロック番号
19     int offset = *pos & (BLOCK_SIZE-1);           // データブロック内のオフセット値です。
20     int chars;
21     int written = 0;
22     int size;
23     struct buffer_head * bh;
24     register char * p;                           // ローカルレジスタ変数。
25

// ブロックデバイスファイルの書き込み時には、書き込みに必要なデータブロック数の合計が
// は、もちろん、指定されたデバイスで許容されるデータブロックの最大数を超えてはなりません。
// そのため、まず指定されたデバイスの総ブロック数を取り出して比較します
// 関数のパラメータで指定された書き込みデータの長さを // 制限します。長さが指定されない場合は
// システム内のデバイスについては、デフォルトの長さである0x7fffff (2GBブロック) が使用
26     されます。 if (blk_size[MAJOR(dev)])
27         size = blk_size[MAJOR(dev)][MINOR(dev)] となります。
28     その他
29         size = 0x7fffff;

// そして、書き込むバイト数「count」に対して、以下の処理をループさせます。
// データが完全に書き込まれるまで、 // 。ループの実行中に、データのブロック番号が
// 現在書き込まれているデータが、指定されたブロックの合計数以上である場合
// デバイスに書き込まれたバイト数を返して終了します。書き込むことができるバイト数は
// 現在処理されているデータブロックに書き込まれるバイト数が計算されます。もし、バイト数が
// が1ブロックに満たない場合は、カウントバイトだけを書けばよいのです。
// 1ブロック分のデータを書き込みたい場合は、直接バッファブロックを申請し、ユーザーデータを
// 置く
// を読み込んでいます。そうでない場合は、パーシャルに書き込まれるデータブロックを読み込む必
// 要があります。
// のデータを読み、次の2ブロックのデータを先読みします。その後、ブロック番号をインクリメン
30 トし、準備方 (count>0) {
31 // 次の操作のために(block & size)操作が失敗した場合、書き込まれたバイト数は
32 // が返され、バイトが書き込まれません。場合によっては-EIOコード（負の数）が返されます。
33     chars = BLOCK_SIZE - offset; // このブロックに書き込めるバイト数。
34     if (chars > count)
35         chars=countです。
36     if (chars == BLOCK_SIZE)
37         bh = getblk(dev, block); // buffer.cの206行目と322行目です。
38     その他
39         bh = breada(dev, block, block+1, block+2, -1) です。
40     block++;
41     if (!bh)
42         return written?written:-EIO;

// 次に、ポインタ p は、まず、データがあるバッファブロック内の位置に向けられます。
// が書きされることになります。最後のループで書き込まれたデータが1ブロックに満たない場合
// は
// ブロックの先頭から必要なバイトを埋めて（修正して）いくため、オフセットは
// はデフォルトでゼロに設定されます。その後、ファイル内のオフセットポインタ 'pos' が転送さ
// れます。
// を今回の書き込みバイト数で割って、書き込みバイト数を
// 統計値'written'に蓄積されます。その後、バイト数の'count'を引く
// 今回書き込むべきバイト数 (chars) から、書き込む必要のある//を選択します。その後、私たちは
// ユーザーバッファからの'chars'バイトを、バッファブロック内の
43 // 書き込みを開始します。コピーが完了すると、バッファブロックモディファイドフラグがセットさ
// れて
// p = offset + bh->b_data;

```

```

44         offset = 0となります。
45         *pos += chars;
46         written += chars;           // 書き込まれた総バイト数。
47         count -= chars;
48         while (chars-->0)
49             *(p++) = get_fs_byte(buf++); となります。
50         bh->b_dirt = 1;
51         brelse(bh);
52     }
53     のリターンが書                                // 書き込まれたバイト数を返します。
54 } かれています。
55
56     //// ブロックリード機能 - 指定されたデバイスからユーザーバッファにデータを読み込みます。
57     // 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。
58     // count - 転送するバイト数。
59     // 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
60     // 返されます。
61     int block_read(int dev,unsigned long *pos,BLOCK_SIZE_BQS, char * buf, int count)
62     {
63         int offset = *pos & (BLOCK_SIZE-1);
64         int chars;
65         int size;
66         int read = 0;
67         struct buffer_head * bh;
68         register char * p;                // ローカルレジスタ変数。
69
70         // ブロックデバイスファイルを読み出す際には、読み出すのに必要なブロックの総数が
71         // 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
72         // デバイスのブロックの//を最初に取り出して比較し、与えられたリードデータの長さを制限します。
73         // を関数パラメータで指定します。システム内のデバイスに長さが指定されていない場合は、デフォ
74         // ルトの
75         // 0x7fffff (2GBブロック) の長さが使用される
76         . if (blk_size[MAJOR(dev)])
77             size = blk_size[MAJOR(dev)][MINOR(dev)]となります。
78         その他
79             size = 0x7fffff;
80
81         // 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。
82         // 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
83         // 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
84         // 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
85         // 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
86         // 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
87         // 関数を呼び出します。
88         // 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。
89         // もし
90         // 読み取り操作にエラーがあった場合は、読み取ったバイト数を返します。バイトが読み込まれてい
91         // ない場合は
92         // エラーコードが返されます。次に、ブロック番号を1つ増やして、次の
93         // //の操作を行います。
94         while (count>0) {
95             if (block >= size)
96                 return read? read:-EIO;  chars = BLOCK_SIZE-
97             offset;
98             if (chars > count)
99                 chars = count; 797
100            if (!(bh = breada(dev,block,block+1,block+2,-1)))
101                return read? read:-EIO;

```

78 block++。
 // 次に、ポインタpは、まず、読み込んだディスクのバッファブロック内の位置を指し示す
 データを読み込むブロックを指定します。最後のループリード操作のデータが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出す必要があるため、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。

```

79                   p = offset + bh->b_data;
80                   offset = 0;
81                   *pos += chars;
82                   read += chars;                                  // 読み込まれた総バイト数。
83                   count -= chars;
84                   while (chars-->0)
85                         put_fs_byte(*(p++), buf++);
86                   brelse(bh);
87         }
88         readを返す。                                          // 読んだバイト数を返します。
89 }
90

```

10. ファイル_dev.c

1. 機能

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ます。この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。

2. コードアノテーション

プログラム 12-9 linux/fs/file_dev.c

```

1 /*
2 * linux/fs/file_dev.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <fcntl.h> ファイル制御のヘッダファイルです。演算制御定数記号の定義は
// ファイルとそのディスクリプターに使用されます。

```

```

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// の初期タスク0と、記述子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
// 数のマクロ文が含まれています。
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプ
// ロトタイプ定義が含まれています。
// カーネルの使用する機能
// <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が定義され
7 // <errno.h>
8 // <file.h> ファイル操作。
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h> (日本語)
12 #include <asm/segment.h>
13
14 #define MIN(a, b) (((a)<(b))?(a):(b)) // a, b の最小値を得る。
15 #define MAX(a, b) (((a)>(b))?(a):(b)) // a, bで最大値を得る。
16
17 ///// 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
18 //す。
19 // i-nodeからはデバイス番号を知ることができます、ファイル構造からは現在の
20 // ファイル内の読み書き可能なポインタの位置を指定します。Buf」は、ユーザーのバッファの位置
21 //を指定します。
22 // スペース、「count」は読み取る必要のあるバイト数です。戻り値は
23 // 実際に読み込まれたバイト数、またはエラーコード（0未満）を表します。
24 int fileread(struct char *left, struct inode *inode, struct file *filp, char *buf, int count)
25 {。 struct buffer_head *bh;
26
27 // まず、パラメータの有効性を判断します。読み込むバイト数が以下の場合は0を返します。
28 // 0以下であること。読み込むバイト数が0にならない場合は、次のようにになります。
29 // データが完全に読み込まれるか、問題が発生するまで、//ループ操作が行われます。
30 // 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
31 // 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'
32 // 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、
33 // ループを終了します。
34 // 読み取り操作が失敗します。nrが0の場合は、指定されたデータブロックが存在しないことを意味
35 // します。
36 // そのため((left->count)=0)クポインタはNULLに設定されています。(filp->f_pos)/BLOCK_SIZE を
37 // 用いて 0を返す。
38 // ファイルのleft)トポインタが置かれているデータブロック番号。
39 if (nr = bmap(inode, (filp->f_pos)/BLOCK_SIZE)){ // inode.c, line 140.
40         if (! (bh=bread(inode->i_dev, nr)))
41            ブレークします。
42     } else
43         bh = NULL。
44
45 // 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
46 // 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。
47 // 今読み取る必要のあるバイト数を'左'にして、小さい値は数字の
48 // この操作で読み込まれるバイト'chars'の // 数。（BLOCK_SIZE - nr）> left」の場合は、ブロック
49 // が読み込まれるべき最後のデータで、そうでなければ次のブロックのデータを読み込まなければな
50 // りません。
51 // 次に、ファイルの読み取りと書き込みのポインタを調整します。ポインタはバイト数を進める
52 // 'chars'
53 // 今回読み込まれるバイト数です。残りのバイト数「left」も差し引く必要があります。
54 // 「chars」バイトで。
55     nr = filp->f_pos % BLOCK_SIZE;

```

```

31         chars = MIN( BLOCK_SIZE-nr , left );
32         filp->f_pos += chars;
33         left -= chars;
// 上記のデータがデバイスから読み込まれた場合、バッファブロック内の以下の場所をpに指定します。
// データが読み込まれると、'chars' バイトをユーザー「buf」にコピーし、そうでなければ
34 // ユーザーバッファを'chars' バイトのゼロで埋める if
35     (bh) {
36         char * p = nr + bh->b_data;
37         while (chars-->0)
38             put fs byte(*(p++), buf++);
39         brelse(bh);
40     } else {
41         while (chars-->0)
42             put fs byte(0, buf++);
43     }
}
// 最後に、i-nodeのアクセス時間が現在の間に修正され、いくつかの
// 読んだバイト数が返されます。読んだバイト数が0の場合は、エラーコードが返されます。
// CURRENT_TIME は、include/linux/sched.h の 142 行目で定義されたマクロで、以下の計算に使用
44 されます。
45 // UNIXの時間。つまり、1970年1月1日の0:00から現在の時刻までです。単位は秒です。 inode-
46 }      >i_atime = CURRENT_TIME;
47 return (count-left)?(count-left):-ERROR;
//// ファイル書き込み機能 - i-nodeとファイル構造に基づいて、ユーザーデータをファイルに書き
込みます。
// i-nodeからはデバイス番号がわかり、ファイル構造からは
// ファイル内の現在の読み書き可能なポインタの位置を指定します。'buf' は、バッファの位置を指
定します。
ユーザー空間での//、'count' は書き込まれるバイト数です。戻り値は
// 実際に書き込まれたバイト数、またはエラーコード（0より小さい）です。
48 int filewrite(struct m_inode* inode, struct file* filp, char * buf, int count)
49 {
50     int block,c;
51     struct buffer_head * bh;
52     char * p;
53     int i=0;
54
55 /*
56 * ok, append は多くのプロセスが同時に書き込んでいる場合には動作しないかもしれません。
57 * しかし、それがどうした。その道はいずれにしても狂気につながる。
58 */
59 */
// まず、データをファイルのどこに書き込むかを決めます。にデータを追加したい場合は
// ファイルの場合、ファイル書き込みポインタをファイルの最後に移動させ、そうでない場合は
// ファイルの現在の読み取りおよび書き込みポインタ。
60     if (filp->f_flags & O_APPEND)
61         pos = inode->i_size;
62     その他
63         pos = filp->f_pos;
// そして、'i' (最初はゼロ) を書き込んだバイト数が少なくなると
// 指定された書き込みバイト数'count' よりも多い場合、以下の処理を行います。
// を周期的に行います。ループ操作では、まず、論理的な（ディスク）ブロック番号「block」を取
得します。
// デバイス上のファイルデータブロック(pos/BLOCK_SIZE)に対応しています。もし、ディスクブロッ
クが
// が存在しない場合は、作成します。作成されたディスクのブロック番号が0の場合、作成は失敗し
ます。

```

```

// となり、ループが終了します。それ以外の場合は、デバイス上のディスクブロックを以下のように
読み込みます。
64 // ディスクのブロック番号を入力し、エラーが発生したらループ
65     を抜ける while (i < count) {
66         if (!(block = create_block(inode, pos/BLOCK_SIZE))
67             break;
68         if (!(bh=bread(inode-
69             >i_dev, block))) 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
69 ンタに対して
70 // 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
71 // をブロックの最後まで書き込みます。cが書き込まれる残りのバイト数より大きい場合は
72 c = pos % BLOCK_SIZE... // (count - i) とすると、今度は単純にc =
73             (count - i) バイトを書き込む。
    p = c + bh->b_data;
    bh->b_dirt = 1;
    c = BLOCK_SIZE-cです。
    if (c > count-i) c = count-i;
// データを書き込む前に、次のループで書き込むファイル内の位置をあらかじめ設定する
// の操作を行います。そこで、書き込まなければならぬバイト数だけ「pos」ポインタを前進させ
ます。
74 // 「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
75 // i-nodeのファイル長フィールドが修正され、i-nodeの修正フラグが設定されます。その後
76 // 今回書き込まれるバイト数は、書き込まれたバイト数に加算される 'i' for loop judgment.
77 // 続いて、'c' バイトがユーザーバッファ'buf'から、'buf' が指す開始位置にコピーされる。
78 // キャッシュバッファブロックの'p'。バッファブロックはコピー後に解放されます
79     .
80     if (pos > inode->i_size) {...}
81         inode->i_size = pos;
82         inode->i_dirt = 1;
83     }
    i += c.
    while (c-->0)
        *(p++) = get_fs_byte(buf++);
    brelse(bh)となります。
}
// 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
。
84 // 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。
85 // とし、ファイルの読み書きポインタを調整します。つまり、操作によってデータが追加されない場
86 合は
87 // をファイルの末尾に移動させると、ファイルの読み書きポインタが現在の読み書きの
88 // 位置'pos'、ファイルi-nodeの修正時間が現在の時間に変更されます。
89 // 最後に、書き込んだバイト数を返します。書き込まれたバイト数が0の場合は
90 // エラーコード-1が返されます。
91     inode->i_mtime = CURRENT_TIMEです。
    if (!(filp->f_flags & O_APPEND)) {
        filp->f_pos = pos;
        inode->i_ctime = CURRENT_TIMEです。
    }
(i?i:-1)を返します。

```

11. パイプ.c

1. 機能

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ーポインタが、*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。

図12-28に示すように、パイプラインの読み込みでは、データがパイプラインの終端から読み込まれ、パイプのテールポインタが読み込まれたバイト数だけ前方に移動し、パイプラインへの書き込みでは、データがパイプのヘッダに書き込まれ、ヘッドポインタが(ヌルバイトを指して)何箇所か前方に移動します。

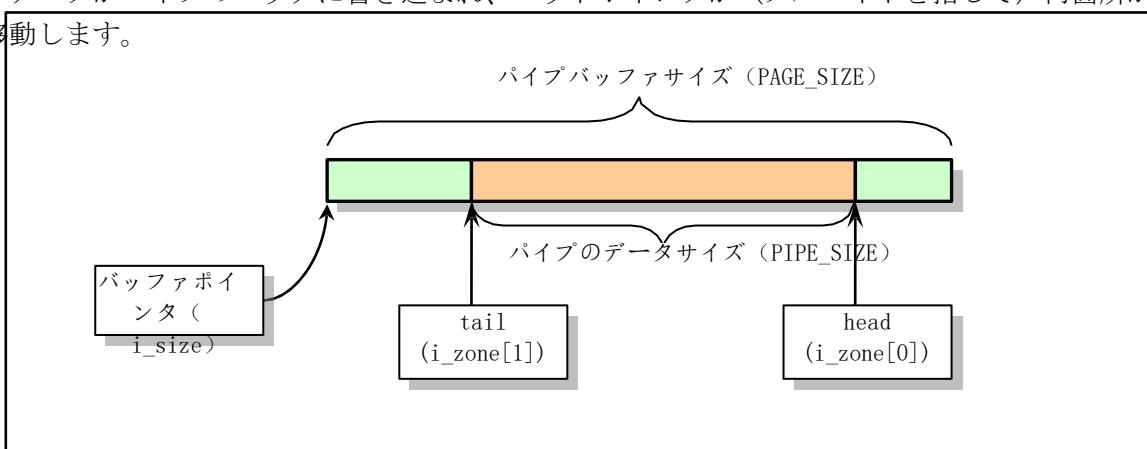


図12-28 パイプバッファ動作図

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ーテーブルから2つの未使用的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 コードアノテーション

プログラム 12-10 linux/fs/pipe.c

```

1  /*
2  * linux/fs/pipe.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナルの定義
// 操作関数のプロトタイプ。
// <errno.h> エラー番号のヘッダーファイルです。システムの様々なエラー番号を含みます。
// <termios.h> 端末入出力機能のヘッダーファイル。主に端末の入出力機能を定義しています。
// 非同期通信ポートを制御するインターフェースです。
// <linux/fs/pipe.h> など、記述子の DMA メモリ設定と取得に関する構造体が組み込まれた関数
// のマクロ文が含まれています。
// <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/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義され
// ています。
// セグメントレジスタの操作。
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプ
7  ロトタイプ定義が含まれています。
8 // include リンク用の機能
9 #include <termios.h>
10
11 #include <linux/sched.h>
12 #include <linux/mm.h>           get_free_page */ のため
13 #include <asm/segment.h>         の /*。
14 #include <linux/kernel.h>
(日本語)
15
16 // バッファの'buf'はパイプの i-node、'buf' はユーザーデータのバッファポインタであり
// 'count' は読んだバイト数です。
17 int read_pipe(struct m_inode * inode, char * buf, int count)
18 {
19     int chars, size, read = 0;
20
21     // 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。
22     // の操作を行います。ループ操作中、現在のパイプにデータがない場合 (size=0)。
23     // そのノードを待っているプロセスが起こされます。これは通常、書き込みパイプラインプロセスで
24     // 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
25     // 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
26     // 場合は
27     // 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
28     // 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
29     // の場合は
30     // 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
31     // include/linux/fs.hで定義され、PIPE_WAIT(*inode) は system 内の任意障害を取得は wake_up (&
32     // kernel/signal.c プログラム PIPE_WRITE_WAIT (*inode));
        while (count > 0)
    {

```

```

23         if (inode->i_count != 2) /* ライターはいますか? */
24             readを返す。
25             if (current->signal & ~current->blocked)
26                 return read? read:ERESTARTSYS;
27             interruptible_sleep_on(& PIPE_READ_WAIT(*inode)) となって
28         }    います。
// 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
読まなければならぬバイト数の'count'と同じにします。chars'が次の値より大きい場合
// 現在のパイプラインのデータの'サイズ'と同じになります。すると、その数は
// 読むべきバイト数'count'は、この時までに読めるバイト数から減算され
// 既に読んだバイト数に追加される'read'。
29         chars = PAGE_SIZE-PIPE_TAIL(*inode);
30         if (chars > count)
31             chars = count;
32         if (chars > size)
33             chars = size;
34         count -= chars;
35         read += chars;
// そして、'size'にパイプのテールポインタを指定させ、現在のパイプのテールポインタを調整しま
す(forward)
// 'chars'バイト)。テールポインタがパイプの終端を超えると折り返します。のデータは
// のパイプがユーザーバッファにコピーされます。パイプのi-nodeでは、パイプバッファのブロック
36 ポインタ      size = PIPE_TAIL(*inode)です。
37 // は i_size フィー PIPE_TAIL(*inode) += chars;
38         PIPE_TAIL(*inode) &= (PAGE_SIZE-1);
39         while (chars-->0)
40             put_fs_byte((char *)inode->i_size)[size++], buf++)と
なります。
41 // 読み込みパイプ操作が終了すると、パイプラインを待っているプロセスが起こされて
// wake_up(& PIPE_WRITE_WAIT(*inode));
42         return read;
43
44 }
45
//// パイプの書き込み機能。
// パラメータの' inode'はパイプのi-node、'buf'はデータバッファのポインタ、'count'は
// はパイプに書き込まれるバイト数です。
46 int write_pipe(struct m_inode * inode, char * buf, int count)
47 {
48 int chars, size, written = 0; 49

// 書き込まれるバイト数'count'がまだ0より大きい場合は、ループします。
// 以下の操作を行います。ループ操作の間、現在のパイプが満杯の場合（空きスペース
// size = 0）の場合、パイプラインを待っているプロセスが起こされ、通常はリードパイププロセス
が
// が起こされます。パイプリーダがない場合、つまり、i-nodeの参照カウント値が小さい場合は
// 2よりも大きい場合、SIGPIPEシグナルが現在のプロセスに送信され、書き込まれたバイト数が
// を返します。0バイトが書き込まれた場合は、-1が返されます。それ以外の場合は、現在のプロセ
スをスリープさせます
50 // パイプのプロセスがデータを読み取るのを待って、パイプを解放します。
51 // 空間にります。マクロのPIPE_SIZE()、PIPE_HEAD()などは、include/linux/fs.hで定義されてい
52     ます。
        while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
            wake_up(& PIPE_READ_WAIT(*inode));

```

```

53         if (inode->i_count != 2) /* 読者がいない
54             */
55             current->signal |= (1<<(SIGPIPE-1));
56             written?written:-1を返します。
57         }
58         sleep_on(& PIPE_WRITE_WAIT(*inode))となりま
59         す。
60     // 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
61     // 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
62     // 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
63     // 大きい場合
64     // 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'
65     // 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
66     // 書き込まれるバイト数を「count」から「this」に書き込まれたバイト数を引いた値にします。
67     // chars = PAGE_SIZE-PIPE_HEAD(*inode) とする。
68     if (chars > count)
69         chars = count;
70     if (chars > size)
71         chars = size;
72     count -= chars;
73     written += chars;
74
75     // 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
76     // します。
77     // (前方のcharsバイト)。ヘッドポインタがパイプの終端を超えると折り返します。その後
78     // ユーザーバッファから' chars' バイトをパイプの先頭ポインタにコピーします。のために
79     // size = PIPE_HEAD(*inode);
80     PIPE_HEAD(*inode) += chars;
81     PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
82     while (chars-->0)
83         ((char *)inode->i_size)[size++] = get_fs_byte(buf++);
84     }
85
86     // 書き込みパイプ操作が終了すると、パイプを待っているプロセスが起こされて
87     // wake_up(& PIPE_READ_WAIT(*inode));
88
89     return written;
90 }
91
92 }
93
94 }
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805

```

```

86          (f[j++]=i+file_table)->f_count++;
87      もし (j==1)
88          f[0]->f_count=0;
89      もし (j<2)
90          1を返す。
// 取得した2つのファイルテーブル構造項目のそれぞれに、ファイルハンドル番号が割り当てられる
。
// と、プロセスファイル構造配列の2つのアイテムがそれぞれ2つのファイルを指しています。
// 構造体であり、ファイルハンドルはその配列のインデックス番号である。同様に、もし
// フリーのファイルハンドルが 1 つしかない場合、そのハンドルは解放されます。2 つのフリーの
91 ハンドルが見つからない場合は
92 // 上記で取得したファイル構造の2つの項目を解放して-1を返す j=0;
93     for(i=0;j<2 && i< NR_OPEN;i++)
94         if (! current->filp[i])
95             {...}
96             current->filp[ fd[j]=i ] = f[j];
97             j++;
98         }
99     if (j==1)
100        current->filp[fd[0]]=NULL;
101    if (j<2) {...}
102        f[0]->f_count=f[1]->f_count=0;
103        return -1;
104    }
// 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。
105 // 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。失敗しない場合にc, 2次元配列の231.ファイルハンドル
106 とファイル      (! (inode=get_pipe_inode(
107 // 構造体のアイオノードはcurrentに応じて解放され、-1が返されます。
108           >filp[fd[0]] =
109           current->filp[fd[1]] = NULL
110           .
// パイプi-nodeのfd[0]がcurrent指向が成功すると、fd[1]のファイル構造が初期化されるので
111 // 両方が同じパイプのi-nodeを指していることを確認し、readとwriteのポインタを0に設定します。
112 // 1つ目のファイル構造のファイルモードが読み取りに設定され、2つ目のファイル構造のファイルモードが
113 // ファイル構造が書き込みに設定されます。最後に、ファイルハンドル配列がユーザ空間にコピーさ
114 れる
115 // 配列を作成します。f[0]->f_inode = f[1]-
116     >f_inode mode inodeとf[0]->f_pos = f[1]/* 読んでく
117     >f_pos = 0 となります。          ださい。
118     f[1]->f_mode = 2;            /* 書き込ん
119     でください
120     .
121     put_fs_long(fd[0], 0+fildes) です
122     .
123     put_fs_long(fd[1], 1+fildes) です
124
125 // パラメータを返すpino - パイプiノードポインタ; cmd - 制御コマンド; arg - 引数。
126 // 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。
127 int pipe_ioctl(struct m_inode *pino, int cmd, int arg)
128 // // パイプioの制御機能。
129 // パイプラインの現在の読み取り可能なデータ長を取得するコマンドの場合は、パイプデータを
130 // 長さの値をユーザー引数で指定された場所に格納し、0を返します。それ以外の場合は
131 // 無効なコマンドのエラーコードが返されます。

```

```

120     スイッチ (cmd) {
121         case FIONREAD:
122             verify_area((void *) arg, 4);
123             put_fs_long(PIPE_SIZE(*pino), (unsigned long *) arg);
124             return 0;
125             のデフォルトです。
126             return -EINVAL;
127     }
128 }
129

```

12. char_dev.c

1. 機能

char_dev.cプログラムには、主にrw_ttxy()、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_ttxy()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_ttxy()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機能で、メジャー・デバイス番号は1です。メモリイメージに対するバイト操作を実装していますが、Linux 0.12カーネルでは、その操作を実装していません。

マイナー・デバイス番号0、1、2の読み書きを、0.96バージョンで実装し始めるまで。

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。

2. コードアノテーション

プログラム 12-11 linux/fs/char_dev.c

```

1 /*
2 * linux/fs/char_dev.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <sys/types.h> 型のヘッダファイル。基本的なシステムデータ型が定義されています。
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// の初期タスクと、記述子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
// のマクロ文が含まれています。

```

```

// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネルの使用する機能
// <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が定義されています。
// セグメントトレジスタの操作。
7 //include<io.h> ヘッダーファイルです。の形で、ioポートを操作する関数を定義します。
8 //include<sys/types.h> システムインターフェース用の組み込みアセンブリ言語。
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h> (日本語)
12
13 #include <asm/segment.h>
14 #include <asm/io.h>
15
16 extern int tty_read(unsigned minor, char *buf, int count); // 端末の読み取り
17 extern int tty_write(unsigned minor, char *buf, int count); // 端末の書き込み
18
19 // 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。
20 typedef (*crw_ptr)(int rw, unsigned minor, char *buf, int count, off_t *pos);
21
22 ///////////////////////////////////////////////////////////////////
23 //シリアル端子のリード/ライト操作機能。
24 // パラメータ: rw - リード/ライトコマンド、 minor - ターミナルサブデバイス番号、 buf - バッファ。
25 // cout - 読み書きされたバイト数; pos - 読み書き操作の現在のポインタ、このポインタ
26 //は端末操作には使えません。
27 // 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。
28 static int rw_ttyx(int rw, unsigned minor, char *buf, int count, off_t *pos)
29 { . . . return ((rw==READ)?
30             tty_read(minor,buf,count):
31             tty_write(minor,buf,count));
32 }
33
34 ///////////////////////////////////////////////////////////////////
35 //端末のリード/ライト操作機能。
36 // 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。
37 static int rw_tty(int rw, unsigned minor, char *buf, int count, off_t *pos)
38 { . . .
39 // プロセスが対応する制御端子を持たない場合は、エラーコードが返されます。
40 // そうでない場合は、端末が読み書き可能な関数 rw_ttyx() を呼び出して、実際の
41 // 読み取りまたは書き込みバイト数。
42     if (_current->tty<0)
43         return -EPERM;
44     return rw_ttyx(rw,_current->tty,buf,count,pos);
45 }
46
47 ///////////////////////////////////////////////////////////////////
48 //メモリデータの読み書き。実装されていません。
49 static int rw_ram(int rw, char *buf, int count, off_t *pos)
50 { . . .
51     return -EIO;
52 }
53
54 ///////////////////////////////////////////////////////////////////
55 //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。
56 static int rw_mem(int rw, char *buf, int count, off_t *pos)
57 {

```

```

41         return -EIO;
42 }
43
44 ///////////////////////////////////////////////////////////////////
45 // 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
46 ///////////////////////////////////////////////////////////////////
47 static int rw_kmem(int rw, char * buf, int count, off_t * pos)
48 {。       return -EIO;
49 }
50
51 // ポートのリード/ライト操作機能。
52 // パラメータ: rw - 読み取りまたは書き込みコマンド; buf - バッファ; cout - 読み取り/書き込みのバイト数。
53 // pos - ポートのアドレス。戻り値: 実際に読み書きしたバイト数。
54 static int rw_port(int rw, char * buf, int count, off_t * pos)
55 {
56     int i=*pos; 52
57
58     // 読み書きに必要なバイト数で、ポートアドレスが
59     // 64kでは、1バイトのリードまたはライトの操作がサイクリックに実行されます。コマンドがリードの場合
60     // 1バイトのコンテンツがポートiから読み込まれ、ユーザーバッファに置かれます。もし、コマンド
61     // が
62     // while (count-->0 && i<65536) { 。
63     //     if (rw==READ)
64             put_fs_byte(inb(i),buf++); 53
65     //     その他
66     //     i++; // outb(go) / fs-byte(bufを増やす)i){??}
67     // }
68
69     // そして、読み書きされたバイト数をカウントし、対応する読み書きポインタを調整します。
70     // i -= *pos...と、読み書きしたバイト数を返します
71
72     // す。
73     *pos+= i;
74     return i;
75
76 }
77
78 ///////////////////////////////////////////////////////////////////
79 // メモリの読み書き操作機能です。メモリのマスターナンバーは1で、処理のみ
80 // 0~5のサブデバイスの//がここに実装されています。
81 static int rw_memory(int rw, unsigned minor, char * buf, int count, off_t * pos)
82 {
83
84     // 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。
85     // スイッチ (マイ
86     // ナー) {           // return rw_ram(rw,buf,count,pos);
87     // case 0:          // return rw_mem(rw,buf,count,pos);
88     // case 1:          // return rw_kmem(rw,buf,count,pos);
89     //     return (rw==READ)?0:count; /* */
90     // case 2: rw_null */ (rw==READ)?0:count; // デバイスファイル名は/dev/null
91     // case 3:          // デバイスファイル名は
92     //                 /dev/port return rw_port(rw,buf,count,pos);
93     // case 4:
94
95     //     のデフォ
96     //     ルトですreturn -EIO;
97 }
98
99

```

```

81 }
82
83 // システム内のデバイスの数を定義します。
84 #define NRDEVS ((sizeof (crw_table))/(sizeof (crw_ptr)))
85
86 // 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。
87 static crw_table /*nodev */{
88     rw_memory,      /* /dev/mem など */
89     NULL,           /* /dev/fd */
90     NULL,           /* /dev/hd */
91     rw_ttyx,        /* /dev/ttyx */
92     rw_tty,         /* /dev/tty */
93     NULL,           /* /dev/lp */
94     NULL} とな    /* 無名のパイプ */
95     ります。
96
97     //// キャラクターデバイスのリード/ライト機能。
98     // パラメータ: rw - 読み取りまたは書き込みコマンド、 dev - デバイス番号、 buf - バッファ、
99     count - 数
100    // pos - 読み取りまたは書き込みポインタ。
101    // 戻り値: 実際に読み書きしたバイト数。
102    int rw_char(int rw, int dev, char *buf, int count, off_t *pos)
103    {
104        crw_ptr call_addr;
105
106        // デバイス番号がシステムデバイスの数を超える場合は、エラーコードが返されます。もし
107        // デバイスに対応するリード/ライト機能がない場合は、エラーコードが返されます。
108        // それ以外の場合は、対応するデバイスのリード/ライト操作機能が呼び出され
109        // 実際に読み書きされたNRDEVS数が返されます。
110        return -ENODEV;
111       もし (!call_addr=crw_table[MAJOR(dev)])
112        return -ENODEV;
113        return call_addr(rw, MINOR(dev), buf, count, pos);
114    }
115

```

13. read_write.c

1. 機能

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・オペレーションを実行し、読み取ったバイト数を返します。

write()システムコールの実装は read()と似ていますが、file_dev.c の file_read() というファイル読み取り関数を呼び出します。

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 コードアナリティクス

プログラム 12-12 linux/fs/read_write.c

```

1 /*
2 * linux/fs/read_write.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
// そして、定数です。
// <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。
// <sys/types.h> 型のヘッダファイル。基本的なシステムデータ型が定義されています。
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーの樹構成用アセンブリ記述子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
// <linux/sched.h> 文字列が含まれてないマップヘッダーファイルでは、タスク構造体task_struct、データ
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
// セグメントレジスタの操作。 7
#include <sys/stat.h>
8 #include <errno.h>
9 #include <sys/types.h>
10
11 #include <linux/kernel.h> (日本語)
12 #include <linux/sched.h>
13 #include <asm/segment.h>
14
// fs/char_dev.c, line 95.
15 extern int rw_char(int rw, int dev, char *buf, int count, off_t *pos);
// バイブ操作関数の読み込み fs/pipe.c, line 16.
16 extern int read_pipe(struct m_inode *inode, char *buf, int count);
// バイブ操作関数の記述 fs/pipe.c, line 46.
17 extern int write_pipe(struct m_inode *inode, char *buf, int count);
// fs/block_dev.c の 56 行目です。 18 extern int block_read(int
dev, off_t *pos, char *buf, int count);
// ブロックデバイスの書き込み機能 fs/block_dev.c, line 16.
19 extern int block_write(int dev, off_t *pos, char *buf, int count);
// ファイル操作関数の読み込み fs/file_dev.c, line 17.
20 extern int file_read(struct buf *buf, struct m_inode *inode, struct file *
filp); // ファイル操作関数を書く fs/file_dev.c, line 48.
21 extern int file_write(struct m_inode *inode, struct file *
filp),
22 char *buf, int count); 24
/// ファイルの読み取り/書き込みポインタを再配置する system-call.

```

```

// パラメータ fd はファイルハンドル、offset は新しいファイルの読み取りまたは書き込みポイン
タのオフセットです。
// で、origin はオフセットの開始位置です。3つのオプションがあります。SEEK_SET (0,
// ファイルの先頭から)、SEEK_CUR (1、現在の読み書き位置から)、SEEK_END
// (2、ファイルの最後から)。
25 int sys_lseek(unsigned int fd, off_t offset, int origin)
26 {
27     struct file * file;
28     int tmp;
29
// まず、関数が提供するパラメータの有効性を判断します。もし、ファイルハンドル
// の値が、プログラム NR_OPEN(20)で開くファイルの最大数よりも大きい場合、または
// ハンドルのファイル構造ポインタが空であるか、対応するファイルのi-nodeフィールドが
// 構造体が空であるか、指定されたデバイスファイルのポインタが位置決めできない場合は、エラー
// コード
// が返されて終了します。ファイルに対応するi-nodeがパイプノードの場合は、エラーコード
30 // がe Sheilaに遅れれば OPEN せなれば (file->f_inode->i_node->f_type) が自由で動けないからです
31         || !IS_SEEKABLE(MAJOR(file->f_inode->i_dev)))
32         return -EBADF;
33    もし (file->f_inode->i_pipe)
34     return -ESPIPE;
// そして、設定された位置決めフラグに従って、ファイル読み書きポインタを再配置する。
// Origin = SEEK_SET、つまり、ファイルの読み書きポインタの設定が必要になる
// をファイルの先頭の原点として設定します。オフセット値がゼロより小さい場合は、エラー
// コードを返し、そうでなければファイルの読み書きポインタをオフセットと同じにしま
35    す。
36     case 0:
37         if (offset<0) return -EINVAL;
38         file->f_pos=offset;
39         ブレークします。
// Origin = SEEK_CUR であることから、読み書き可能なポインタは、次のように再配置される必要が
あります。
// ファイルの現在の読み取り/書き込みポインタの原点。もし、ファイルのカレントポインタに加え
て
40 // オフセット値が0より小さい場合は、エラーコードが返されます。そうでない場合は、オフセット
41 値が
42 // 現在の読み取り/書き込みポインタに追加されます。
43     case 1:
44         if (file->f_pos+offset<0) return -EINVAL;
45         file->f_pos += offset;
46         ブレークします。
// 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
47 // ファイルの末尾からのオフセットです。このとき、ファイルサイズにオフセット値を加えた値が
48 // 0の場合はエラーコードを返します。それ以外の場合は、読み取り/書き込みポインタをファイル長
49 に再配置する
50 // にオフセット値を加えたものです。
51     case 2:
52         if ((tmp=file->f_inode->i_size+offset) < 0)
53             return -EINVAL;
54         file->f_pos = tmp;
55         break;
56 // オリジンの設定が無効な場合は、エラーコードが返されます。最後に、ファイルの読み取り/書き
57 込みの
58 // 再配置後のポインタの値を返します。
59     のデフォルトです。
60     return -EINVAL;
}
return file->f_pos;

```

```

54 ///////////////////////////////////////////////////////////////////
55 // Read file system-call.
56 // パラメータの'fd'はファイルハンドル、'buf'はユーザーバッファ、'count'は数を表す。
57 // 読むべきバイト数。
58 int sys_read(unsigned int fd, char * buf, int count)
59 {
60     struct file * file;
61     struct m_inode * inode;
62
63     // 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
64     // プログラムの最大オープンファイル番号NR_OPENよりも大きい、またはバイトカウント値を
65     // 読み込んだ値が0より小さい場合や、ハンドルのファイル構造ポインタがNULLの場合は、エラー
66     // コードが
67     // を返して終了します。読み込むバイト数が0の場合は、0を返します。
68     return -EINVAL;
69     if (! count)
70         0を返す。
71
72     // 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
73     // i-nodeを
74     // 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。
75     // 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
76     // 作は
77     // 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
78     // コード
79     // が返されます。文字ファイルであれば、文字デバイスの読み取り操作を行い
80     // 読んだ文字数を返します。ブロックデバイスのファイルであれば、ブロックデバイスの読み取り
81     verify_area(buf, count); // 演算が行われ、読み込んだバイト数が返さ
82    れます。
83     inode = file->f_inode;
84     if (inode->i_pipe)
85         return (file->f_mode&1)? read_pipe(inode, buf, count):-
86             EIO; if (S_ISCHR(inode->i_mode))
87             return rw_char(READ, inode->i_zone[0], buf, count, & file->f_pos);
88     if (S_ISBLK(inode->i_mode))
89         return block_read(inode->i_zone[0], & file->f_pos, buf, count);
90
91     // 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
92     // します。
93     // 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足したものがあれば、それを調
94     // 整する。
95     // ファイルサイズよりも大きい場合は、ファイルサイズを差し引いた値を読み取りバイト数として設
96     // 定します。
97     // 現在の読み取り/書き込みポインタの値。読み取り番号が0に等しい場合は0が返されます。その後
98     // if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
99
100        if (count+file->f_pos > inode->i_size)
101            count = inode->i_size - file->f_pos;
102        if (count<=0)
103            0を返す。
104        return file_read(inode, file, buf, count);
105    }
106
107    // ここで実行されると、ファイルの属性を判断できないということになります。その後、プリント
108    // printf("(Read) inode->i_mode=%06o\n", inode->i_mode);
109    // Write_file_system-call.
110    return -EINVAL;
111
112    // パラメータの'fd'はファイルハンドル、「buf」はユーザーバッファ、「count」は数を表す。
113    // 書き込まれるバイト数

```

```

83 int sys_write(unsigned int fd, char * buf, int count)
84 {
85     struct file * file;
86     struct m_inode * inode;
87
// 同様に、まず、関数のパラメータの有効性をチェックします。もし、ファイルハンドルの値が
// がプログラムの最大オープンファイル番号NR_OPENより大きい場合、またはバイトカウント値を
// 読み込んだファイルが0より小さい場合や、ハンドルのファイル構造ポインタがNULLの場合は、エ
// ラーコード
88 // が返され 終了NR_OPEN。|読み込むバイト数が0の場合は0を返す。
89         return -EINVAL;
90    もし (! count)
91         0を返す。
// 次に、ファイルのi-nodeを取得します。これは、対応する書き込み操作を呼び出すために使用され
// ます。
// i-nodeのモードに応じて機能します。パイプファイルで、書き込みパイプにある場合は
// モードでは、パイプの書き込み操作が実行されます。成功した場合、書き込まれたバイト数は
// それ以外の場合はエラーコードを返し、キャラクターデバイスファイルの場合は
// 書き込まれた文字数を返す；ブロックの場合
// デバイスファイルでは、ブロックデバイスの書き込み操作が行われ、書き込まれたバイト数が
// が返されます。通常のファイルであれば、ファイルの書き込み操作を行って
92 // inode=file->f_inode;
93     if (inode->i_pipe)
94         return (file->f_mode&2)? write_pipe(inode, buf, count):-
95             EIO; if (S_ISCHR(inode->i_mode))
96                 return rw_char(WRITE, inode->i_zone[0], buf, count, & file-
97             >f_pos); if (S_ISBLK(inode->i_mode))
98                 return block_write(inode->i_zone[0], & file-
99             >f_pos, buf, count); if (S_ISREG(inode->i_mode))
100             return file_write(inode, file, buf, count)。
// ここで実行されると、ファイルの属性を判断できないということになります。その後、プリント
// printf("(Write)inode->i_mode=%06o\n", inode->i_mode);
101         return -EINVAL;
102
103 }
104

```

3. ユーザープログラムの読み出し/書き込み操作

上記のプログラムを読めば、ユーザー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、ユーザー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。

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。

- A. システムコールのインターフェース機能。
- B. メモリの割り当て管理機能。
- C. 信号処理機能のセット。
- D. 文字列処理機能。
- E. 標準的な入出力機能。

F. その他の機能セット（BSD機能、暗号化・復号化機能、算術演算機能、端末操作機能、ネットワークソケット機能セットなど）。

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ます。関数ライブラリの移植性を高め、ユーザープログラムのシステム独立性を高くすることができます。新しいライブラリのソースコードでは、システムコールの部分（システムインターフェース部分）を新しいOSのものに置き換えるだけで、基本的に関数ライブラリの移植が完了します。

ライブラリのサブルーチンは、アプリケーションとカーネルシステムの間の中間層と考えることができます。サブルーチンの主な役割は、アプリケーションがシステムコールを実行するための「ラッパー機能」を提供することであり、それに加えて

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。このようにすることで、呼び出しインターフェースが簡素化され、インターフェースがよりシンプルで覚えやすくなり、また、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機能で行うことができ、プログラムの信頼性や安定性が向上します。

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は、まずファイルのオープン操作を行い、開始する動作をOSに通知する必要がある。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ません。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。書き込みの場合は、実際に書き込まれたバイト数が返される。この値が第3引数で指定した値と一致しない場合は、書き込み時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。

_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)
{
    ロングレスって
    asm volatile
        ("int $0x80")
        :"=a" ( res )
        :"" ( NR_read ), "b" ((long)(fd)), "c" ((long)(buf)), "d" ((long)(count)) ;
    if ( res>=0 )
        return int res;
    errno=- res;
    return -1;
}
```

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つて実行され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、実際に読み込んだバイト数を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するので、エラーコードを反転させてグローバル変数errnoに格納し、その値をもとに

呼び出したプログラムに-1が返されます。

Linuxカーネルでは、読み取り操作はファイルシステムのread_write.cファイルに実装されています。上記のシステムコール割込みが実行されると、read_write.cファイルの55行目から始まるsys_read()関数が呼び出されます。sys_read()関数のプロトタイプは以下のように定義されています。

int sys_read(unsigned int fd, char *buf, int count)

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ければ（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inodeを取得し、そのnodeのフラグ情報に基づいてファイルを分類・判定し、次のタイプの読み出し操作関数を呼び出し、実際に読み込んだバイト数を返す。

- ファイルがパイプファイルの場合は、リードパイプ関数 read_pipe() (fs/pipe.c で実装)を呼び出す。
- キャラクタデバイスファイルの場合は、キャラクタデバイス読み込み関数 rw_char()が呼び出される (fs/char_dev.c にて実装)。この関数は、キャラクタデバイスドライバを呼び出したり、特定のキャラクタデバイスサブタイプに基づいてモリキャラクタデバイスを操作する。
- ブロックデバイスファイルの場合は、ブロックデバイスリード関数block_read()が呼び出されます（実装は

`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。

- ファイルが通常のレギュラーファイルの場合、通常のファイルリード関数 `file_read()` (`fs/file_read.c` で実装)が呼び出され、データのリード操作が行われる。この関数は、リードブロックデバイスの操作と同様である。最後に、ファイルシステムが配置されているブロックデバイスの下層ドライバアクセス関数`ll_rw_block()`も 呼び出すが、`file_read()`は、ファイルのカレントポインタの移動など、内部のファイルテーブル構造の情報を保持する必要がある。

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、エラーアンダーフローを反転させてグローバル変数`errno`に格納し、-1の値をアプリケーションに返します。ユーザーが`read()`関数を実行してから、カーネル内で実際に動作するまでの全体の流れは図12-29を参照してください。

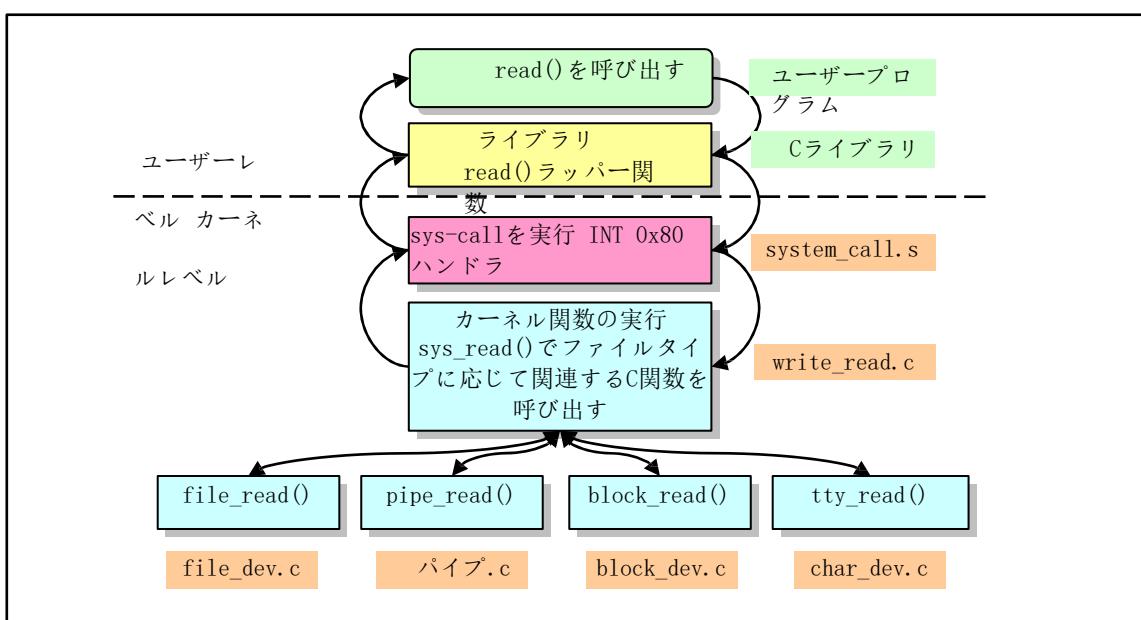


図12-29 `Read()`関数呼び出しの実行手順

12.14 open.c

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んでいる。`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()`を含んでいる。`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している。`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。

1. 機能

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。

2. コードアナリシス

プログラム12-13 linux/fs/open.c

```

1  /*
2  * linux/fs/open.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <fcntl.h> ファイル制御のヘッダファイルです。演算制御定数記号の定義は
// ファイルとそのディスクリプターに使用されます。
// <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。アクセス時間と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ーファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
// そしの初期数値と、記述子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
// <linux/sched.h> 由文が含まれてあります。ヘッダーファイルでは、タスク構造体task_struct、データ
// <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/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネルの使用する機能
// <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が定義されています。
// セグメントレジスタの操作。    7
#include <string.h>
8 #include <errno.h>
9 #include <fcntl.h>
10 #include <sys/types.h>.
11 #include <utime.h>
12 #include <sys/stat.h>
13
14 #include <linux/sched.h>
15 #include <linux/tty.h>
16 #include <linux/kernel.h> (日本語)
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。
// 成功して、ファイルシステムの空きブロックとアイドルノードの合計数が追加される

```

ubufが指すustat構造体に // を入力します。ustat構造体は、以下で定義されています。

```

// include/sys/types.hです。
20 int sys_ustat(int dev, struct ustata * ubuf)
21 {。 return -ENOSYS; // エラーコードを返す、まだ実装されていない。
22 }
23 }
24
25 ///////////////////////////////////////////////////////////////////
26 // ファイルのアクセス時間と修正時間を設定します。
27 // 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
28 // 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
29 // 構造体は、ファイルのアクセス時間と修正時間を設定するために使用されます；もし times ポイ
30 // ンタ
31 // がNULLの場合は、システムの現在の時刻がアクセス時刻と修正時刻に使われます。
32 // 指定されたファイルのドメイン。
33 int sys_utime(char * filename, struct utimbuf * times)
34 {。 struct m_inode * inode;
35     long actime, modtime;
36
37 // 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
38 // ファイル名です。与えられたアクセス時間と修正時間の構造体ポインタの時間が
39 // NULLの場合、ユーザーが設定した時刻の値が構造体から読み込まれます。それ以外の場合は、現在
40 // の時刻
41 // システムの // 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。 if
42     (!inode=namei(filename)))
43         return -ENOENT;
44     if (times) {
45         actime = get_fs_long((unsigned long *) & times->actime);
46         modtime = get_fs_long((unsigned long *) & times->modtime)
47         となります。
48     } else
49         actime = modtime = CURRENT_TIME となります。
50
51 // その後、i-nodeのアクセス時間フィールドと修正時間フィールドを修正し、i-nodeに
52 // i-node modified flag, put back the i-node, and return 0.
53     inode->i_atime = actime;
54     inode->i_mtime = modtime;
55     inode->i_dirt = 1;
56     iput(inode);
57
58 // 0を返す。
59 /*
60 * XXX 本当のuidと実効uidのどちらを使うべきですか？BSDは本当のuidを使います。
61 * この呼び出しが setuid プログラムに役立つようにするためです。
62 */
63
64 ///////////////////////////////////////////////////////////////////
65 // ファイルのアクセス権を確認してください。
66 // 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
67 . R_OK
68 // 値4)、W_OK (2)、X_OK (1)、F_OK (0) のいずれかで、チェック対象のファイルが
69 // 読み込み可能、書き込み可能、実行可能、またはファイルが存在する。アクセスが許可されていれば 0 を、そうでなければ
70 // エラーコードを返します。
71 int sys_access(m_inode * inode, const char * filename, int mode)
72 {。 int res, i_mode;
73
74 // ファイルのアクセスパーミッションは、ファイルのi-nodeにも格納されているので、まずは

```

```

// ファイル名に対応するi-nodeを取得します。検出されたアクセス属性モードは
下位3ビットの//なので、すべての上位ビットをクリアするには「AND」の8進数07が必要です。もしi-
nodeが
    ファイル名に対応する // が存在しない場合は、許可されていないエラーコードが返されます。
    // i-nodeが存在する場合は、ファイル属性コード、i-nodeが戻されます。さらに
53 // 57行目inode->mode = 00000007ode); "という文は、61行目以降に配置するのが望ましいです。
54     if (!(inode->name == filename))
55         return -EACCES;                                // エラーコードです。アクセス権
56         // がありません。
57     i_mode = res = inode->i_mode & 0777;
58 // 現在のinodeで本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。
59 // そうでなければ、現在のプロセスユーザーがファイル所有者と同じグループに属している場合、
60 // ファイルの
61 // group属性が取られ、そうでない場合はresの下位3ビットが「その他」のアクセス権となる
62 // ファイルの
63     if (current->uid == inode->i_uid)
64         res >>= 6;
65     else if (current->gid == inode->regid) & 3; 」とすべき [?]
66 // この時、resの下位3ビットは、以下の方法で選択されたアクセス属性ビットです。
67 // 現在のプロセスとファイルの関係。では、この3つのビットを確認してみましょう。
68 // ファイルがパラメータで照会された属性ビットモードを持っていれば、アクセスが許可されて
69 // 0を返します。
70     if ((res & 0007 & mode) == mode)
71         return 0;
72 /**
73 * このテストを最後に行うのは、本当は
74 * 有効なユーザーIDと本当のユーザーIDを（一時的に）交換します。
75 * を呼び出してから、suser()ルーチンを呼び出します。仮に
76 * suser()ルーチンは、最後に呼ばれる必要があります。
77 */
78 // 現在のユーザーIDが0（スーパーユーザー）で、マスク実行ビットが0の場合や、ファイルが
79 // 誰でも実行・検索できる場合は0を、そうでない場合はエラーコードを返します。
80     (! (mode & 1) || (i_mode & 0111)))
81     return 0;
82
83     return -EACCES; // エラーコード。アクセス権がありません。
84 }

85
86 /////
87 // 現在の作業ディレクトリを変更します。
88 // パラメータfilenameは、ディレクトリ名です。
89 // 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。
90 int sys_chdir(const char * filename)
91 {
92     struct m_inode * inode;
93
94     // 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
95     // に
96     // プロセススタスク構造の//は、与えられたディレクトリ名のi-nodeを指します。そこで我々は
97     // まず、与えられた名前のi-nodeを取ります。i-nodeがディレクトリのi-nodeでない場合、i-nodeの
98 // が戻され(! ode == ameが返されない)
99     return -ENOENT;                                // エラーコード：ファイルまたはディレクトリが存
100    在しません。
101
102   もし (! S_ISDIR(inode->i_mode)) {
103        put(inode)です。
104

```

```

84         return -ENOTDIR; // エラーコード: ディレクトリ名ではありません。
85     }
86     // そして、プロセスの元の作業ディレクトリのi-nodeを解放して、それを指します。
87     // iput(current->pwd);
88     current->pwd = inode;
89     return (0);
90 }

91 ///////////////////////////////////////////////////////////////////
92 // ルートディレクトリの変更。
93 // 指定されたディレクトリ名をカレントプロセスのルートディレクトリ'/'に設定します。
94 // 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。
95 int sys_chroot(const char * filename)
96 {
97     struct m_inode * inode; 94

98 // 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
99 // るために使用されます。
100 // 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
101 // が存在する場合は、エラーコードが返されます。i-nodeがディレクトリi-nodeでない場合、i-node
102 // は
103 // 戻してエラーコードを返す if
104     (!(inode=namei(filename)))
105     {
106         return -ENOENT;
107         if (!S_ISDIR(inode->i_mode))
108             { iput(inode);
109             return -ENOTDIR;
110         }
111     // その後、現在のプロセスの元のルートのi-nodeを解放し、それをリセットして
112     // iput(current->root);
113     // current->root = inode;
114     // を返します(0)。
115 }

116 ///////////////////////////////////////////////////////////////////
117 // ファイルの属性(モード)を変更する。
118 // 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。
119 // 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。
120 int sys_chmod(const char * filename, int mode)
121 {
122     struct m_inode * inode; 109

123 // この関数は、指定されたファイルに新しいアクセスモードを設定します。なぜなら、ファイルのモ
124 // ードは
125 // がファイルのi-nodeにある場合、まずファイル名に対応するi-nodeを取得する必要があります。
126 // 現在のプロセスの実効ユーザーIDがファイルのユーザーIDと異なる場合
127 // i-nodeがスーパーユーザーではない場合、ファイルのi-nodeは戻され、エラーコードが返されます
128 .
129 // (アクセス権はありません)。
130     if
131         (!(inode=namei(filena
132             me))return -ENOENT;
133     if ((current->euid != inode->i_uid) && !suser())
134         { iput(inode);
135             return -EACCES;
136         }
137     // 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-
138     // nodeは修正されます。
139     // フラグがセットされ、i-nodeが戻され、0が返されます。

```

```

116     inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777);
117     inode->i_dirt = 1;
118     iput(inode);
119     return 0;
120 }
121
122 ///////////////////////////////////////////////////////////////////
123 // ファイルの所有者を変更する。
124 // 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。
125 // 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。
126 int sys_chown(const char * filename, int uid, int gid)
127 {
128     struct m_inode * inode; 125
129
130 // この呼び出しへは、ファイルのi-nodeにユーザーとグループのIDを設定するために使用されますので
131 // 最初に
132 // 与えられたファイル名のi-nodeを。現在のプロセスがスーパーユーザでない場合は、与えられたフ
133 // リル名の
134 // i-nodeでエラーコード（アクセス権なし）を返す。 if
135 // !(inode=namei(filename))
136 //      return -ENOENT;
137 //      if (! suser()) {
138 //          iput(inode);
139 //          return -EACCES;
140 //      }
141 // それ以外の場合 uid=metaで指定された値を使用して、ユーザーIDとグループIDを設定します
142 // です。
143 // ファイルのi-nodeを設定してフラグを修正し、i-nodeを戻して0を返します。
144 // となります。
145     inode->i_dirt=1;
146 } iput(inode)です。
147 // 0を返す。
148
149 ///////////////////////////////////////////////////////////////////
150 // 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。
151 // この補助関数は、次のシステムコールsys_open()でのみ使用されます。使用されるのは
152 // 現在のプロセスの属性とシステムのttyテーブルの変更と設定を確認する
153 // オープンファイルがtty端末のキャラクターデバイスの場合。
154 // 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
155 // 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。
156 static int check_char_dev(struct m_inode * inode, int dev, int flag)
157 {
158     struct tty_struct *tty;
159     int min; // サブデバイスの番号。
160
161 // 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。
162 // 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
163 // プロセスに
164 // 制御端末、/dev/ttyはプロセス制御端末装置の同義名。
165 // つまり、/dev/tty デバイスは仮想デバイスであり、/dev/ttyxx のいずれかに対応します。
166 // プロセスが実際に使用するデバイス。プロセスが制御端末を持っている場合、そのプロセスのために
167 // に tty
168 // そのタスク構造の//フィールドは、デバイスNo.4のサブデバイス番号になります。
169 // 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
170 // 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。
171 // 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。
172 // デバイスNo.4の番号。それ以外の場合は、パラメータにデバイスNo.4のサブデバイスが与えられて
173 // いる場合。
174 // サブデバイス番号を直接取得します。取得したサブデバイス番号が、デバイスNo.

```

```

// 4が0より小さい場合は、プロセスに制御端末がないか、デバイス番号が
// が正しくない場合は、-1が返されます。
144     if (MAJOR(dev) == 4 || MAJOR(dev) == 5) {
145         if (MAJOR(dev) == 5)
146             min = current->tty;
147         その他
148             min = MINOR(dev);
149         if (min < 0)
150             1を返す。
// プライマリ疑似端末デバイスは、1つのプロセスでのみ独占的に使用することができます。そのため
// サブデバイス番号がプライマリ疑似端末を示し、オープンファイルのi-nodeの参照がある場合
// countが1より大きい場合は、デバイスが他のプロセスによって使用されたことを示します。 so
// キャラクターデバイスはもう開くことができないので、-1が返されます。それ以外の場合は
// ポインタ tty は、tty テーブルの対応する構造項目を指します。
// ファイル操作のオープンフラグにO_NOCTTYフラグが含まれておらず、かつ、プロセスが
// はグループリーダーであり、制御端末を持たず、tty構造のセッションフィールド
// は0（その端末がまだどのプロセスグループの制御端末でもないことを示す）。
// とすると、このプロセス用のmin端末装置を制御端末として設定することができます。
// そのため、プロセスのタスク構造に端末デバイス番号フィールドttyの値を設定する
// をminに設定し、tty構造のセッション番号とグループ番号を
151 // それぞれ、プロセス(O_SEA_PTY_MAJ番号に並び)と端末番号に等しいように
152             1を返す。
153             tty = TTY_TABLE(min)です。
154             if (!(flag & O_NOCTTY) &&
155                 current->leader &&
156                 current->tty<0 &&
157                 tty->session==0) {
158                 current->tty = min;
159                 tty->session= current->session;
160                 tty->pgrp = current->pgrp;
161             }
// オープンファイル操作フラグにO_NONBLOCKフラグが含まれている場合は、関連する
// キャラクター端末機器の設定。読み取り操作を満足させるためには、最小の数
読み出す文字数の//が0で、タイムアウトのタイミング値が0に設定されており、端末機器の
//が非正規のモードに設定されています。ノンブロッキングモードは非正規モードでのみ動作します
。
// このモードでは、VMINとVTIMEの両方が0に設定されている場合、何文字であっても
162 //補助キューで、flagは何文字読み取るか、{すぐに戻ります。
163             TTY_TABLE(min)-> termios.c_cc[VMIN] =0;
164             TTY_TABLE(min)-> termios.c_cc[VTIME] =0;
165             TTY_TABLE(min)-> termios.c_lflag &=
~ICANON;
166         }
167     }
168 } 0を返す。
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_IWRXG
// （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
イルでは、これらの

```

```

// 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。を作成するオープンコールは、
読み取り専用の
また、 // ファイルは、読み取りと書き込みが可能なファイルハンドルを返します。呼び出し操作が
成功すると
// ファイルハンドル（ファイルディスクリプター）が返され、そうでない場合はエラーコードが返さ
れます。参照
173 // ファイルは sys/stat.h の struct file です。
174 int sys_open(const char *filename, int flag, int mode)
175 { 。 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
// 自由項目を見つけます。空いている項目のインデックス番号fdがハンドル値となります。がある
場合は
177 // 空き ハンドルがない場合は return -EINVAL; (無効なパラメータ) が返されます。
178     for(fd=0 ; fd< NR_OPEN ; fd++)
179         if (! current->filp[fd]) // 無効のアイテムが
180             // ブレークします。
181         if (fd>=NR_OPEN)
182 // そして、現在の close_on_exec ファイルハンドルのビットマップを設定し、対応する
// bit. close_on_exec は、プロセスのすべてのファイルハンドルに対するビットマップフラグです
// 。各ビットが表すのは
// 開いているファイル記述子で、そのファイルハンドルを閉じる必要があります。
// システムコールの execve() が呼び出されます。プログラムがfork()を使って子プロセスを作成す
ると
// 関数を呼び出すと、通常は子プロセスのexecve()関数を呼び出して、別の新しい
// のプログラムを作成し、子プロセスで新しいプログラムの実行を開始します。のビットが消えて
いたら
ファイルハンドルの // close_on_exec が設定されている場合、対応するファイルハンドルは、以下
のときに閉じられます。
// execve()が実行された場合、そうでなければファイルハンドルは常にオープン状態になります。
// ファイルがオープンされると、子プロセスではデフォルトでファイルハンドルもオープンされるの
で
183     current->close_on_exec &= ~(1<<fd);
184 // 対応するビットをここで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
185 // 探します for (i=0 ; i< NR_FILE ; i++ f++)
186 // (参照カウント0のファイルを開いているファイルのために、fにそのファイルを指し示すようにし
// ます。
187 // テーブルの配列を指定して検索を開始します。また、行のポインター割り当て「0+file_table」は
188 // 184は「file_table」と「file_table[0]」に相当しますが、こちらの方がわかりやすいかもしれません
// この時点で、プロセスのファイルハンドルfdのファイル構造ポインタに
// 検索されたファイル構造を、ファイル参照カウントを1つ増やしてから
// 関数 open_namei() で開く操作を行います。戻り値が0より小さい場合は
// エラーを示しているので、先ほど適用したファイル構造を解除し、エラーコード
// が返されます。ファイルオープン操作が成功した場合、inodeは次のようなi-nodeポインタになり
// ます。
189 // 開かれ( inode->filp[fd]=f )->f_count++;
190     if ((i=open_namei(filename, flag, mode, &inode))<0) {
191         current->filp[fd]=NULLとなります。
192         f->f_count=0;
193         i. を返します。
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
。については

```

```

// ファイルの種類によっては、特別な処理が必要になります。文字デバイスが
// ファイルが開かれたら、check_char_dev() 関数を呼び出して、現在の
// プロセスはファイルを開くことができます。許可されていれば（この関数は0を返す）、
// check_char_dev()で
// コントロールターミナルは、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。
// 文字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、ファイルアイテムをリリースすることしかできません。
196 // 上記で要求されたinodeをmode)、エラーコードを返します。
197 /* tty はやや特殊 (ttyxx major=4, ttymajor=5) i_zone[0], flag) {
198     iput(inode);
199     current->filp[fd]=NULL;
200     f->f_count=0;
201     return -EAGAIN; // リソースは一時的に利用できません。
202 }
203 // ブロックデバイスファイルが開かれている場合、ディスクが交換されたかどうかを確認します。置き換えられている場合は、すべての
204 /* ブロックデバイスの場合も同様に、floppy_change を無効にする必要があります。
205 ます */ if (S_ISBLK(inode->i_mode))
206     check_disk_change(inode->i_zone[0]);
207 // ここで、開いたファイルのファイル構造を初期化します。ファイル構造のプロパティを設定する
208 // とフラグを設定し、ハンドルの参照カウントを1にして、i-nodeフィールドを
209 // 開いているファイル、ファイルの読み書きポインタを0にして、最後にファイルハンドルを返します。
210 f->f_mode = inode->i_mode
211 // となります。
212 f->f_flags = flag;
213 f->f_count = 1;
214 f->f_inode = inode;
215 f->f_pos = 0;
216 } // を返します (fd) 。
217
218 // //// Create file system call.
219 // パラメータのpathnameはパス名、modeは上記のsys_open()と同じです。
220 // 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。
221 int sys_create(const char * pathname, int mode)
222 {
223     return sys_open(pathname, O_CREAT | O_TRUNC, mode);
224
225 // ファイルシステムコールを閉じる。
226 // パラメータ fd はファイルハンドルです。
227 // 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
228 .
229 int sys_close(unsigned int fd)
230 {
231     struct file * filp;
232
233     // 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
234     // 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
235     // パラメータ）が返されます。そして、ファイルハンドルの対応するビットをリセットします。
236     // プロセスの // close_on_exec ビットマップ。さらに、ファイルのファイル構造ポインタが
237     // ハンドルがNULLの場合は、エラーコードが返されます。
238     if (fd >= NR_OPEN)
239         return -EINVAL;
240     current->close_on_exec &= ~ (1<<fd);

```

```

226      if (!(filp = current->filp[fd]))
227          return -EINVAL;
// ここで、ファイルハンドルのファイル構造ポインタをNULLに設定します。もし、ハンドルの参照が
// ファイルが閉じられる前に、ファイル構造のカウントがすでに0になっていた場合、カーネルエラ
// ーが発生する
// となり、マシンが停止します。そうでなければ、ファイル構造体の参照カウントがデクリメントさ
// れる
// この時点で、まだ0でなければ、他のプロセスが
// ファイルなので、0(成功)を返します。現在、参照カウントが0であれば、そのファイルには
228 // 処理の参照を行い、ファイル構造がフリーになりました。その後、ファイルのi-nodeを解放して
229 // 0を返す。
230     current->filp[fd] = NULL;
231     if (filp->f_count == 0)
232         panic ("Close: file count is 0");
233     if (--filp->f_count)
234         を返します(0)。
235 }     iput(filp->
236     f_inode); return
237     (0),

```

15. exec.c

1. 機能

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)の関数番号のCハンドラです。

NR_execve()と、exec()関数群のカーネル実装関数です。他の5つの関連するexec関数は、一般的にライブラリ関数で実装され、最終的にはこの関数が呼び出されます。

Linuxカーネルバージョン0.12では、a.out形式の実行ファイルのみをサポートしています。このフォーマットは、シンプルでわかりやすく、入門的な学習に適しています。a.out形式の実行ファイルのヘッダには、以下のようなexecデータ構造があります。実行ファイルの基本的なフォーマットと内容が記述されています。a.out形式の詳細な説明は、第14章のinclude/a.out.hヘッダファイルを参照してください。

```

7 // 符号なし 長いa_magic構造です。/* アクセスにはマクロ N_MAGIC 等を使用すること。
8 符号なし {a_text; /* テキストの長さをバイト単位で表示します。
9   符号なし a_data; /* データの長さをバイト単位で表示します。
10  符号なし a_bss; /* ファイルの初期化されていないデータ領域の長さをバイト単
11    単位で示しています。
12    符号なし a_syms; /* ファイル内のシンボル・テーブル・データの長さ（バイト単位）
13  } 符号なし a_entry; /* 開始アドレス */ /* スタート
14  } 符号なし a_trsize; /* テキストの再配置情報の長さ（バイト）*2
15  } 符号なし a_drsiz; /* データの再配置情報の長さをバイト単位で示します。

```

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。

- コマンドラインパラメータや環境パラメータ空間ページの初期化操作を行う
 - 空間開始ポインタの初期設定、空間ページポインタ配列の(NULL)への初期化、実行ファイル名に応じたファイルのi-nodeの取得、パラメータ数と環境変数数の計算、ファイルタイプと実行許可の確認。
- 実行ファイルの先頭にあるヘッダーデータ構造に従って、その中で情報が処理される - 実行されたファイルのi-nodeに従って、ファイルのヘッダーが読み込まれる；シェルスクリプトであれば(最初の行が#!)始まる場合は、シェルプログラム名とそのパラメータを解析し、実行ファイルをパラメータとしてシェルプログラムを実行し、ファイルのマジックナンバーとセグメントの長さによって実行可能かどうかをチェックします。
- 現在の呼び出しプロセス上で新しいファイルを実行する前の初期化操作 -- 新しい実行ファイルのi-nodeをポイントし、信号処理ハンドルをリセットし、ローカルディスクリプターのベースを設定する
 - ヘッダー情報に応じたアドレスとセグメント長の設定、パラメータと環境パラメータのページポインタの設定、プロセスの各フィールドの内容の変更。
- スタック上の元のcall execve()プログラムのリターンアドレスを新しい実行プログラムの実行アドレスに置き換え、新しくロードされた実行プログラムを実行します。

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close on execフラグに関係するので、linux/fs/fcntl.cファイルの記述を参照してください。プロセス中のオープンファイル記述子は、クローズオン実行フラグを持つ。プロセス制御構造では、このフラグは unsigned long integer close_on_exec で表され、その各ビットが各ファイル記述子のフラグを表しています。close_on_execの対応するビットにファイルディスクリプターがセットされていれば、execve()が実行されたときにディスクリプターはクローズされ、そうでなければディスクリプターは常にオープンされます。ファイル制御関数fcntl()を使ってこのフラグを特別に設定しない限り、カーネルのデフォルトの動作では、execve()の実行後もディスクリプターはオープンのままです。

以下に、コマンドラインパラメーターと環境パラメーターの意味を説明します。ユーザーがコマンドプロンプトでコマンドを入力すると、実行指定されたプログラムは入力されたコマンドを受け入れます。

コマンドラインからのライン引数を使用します。例えば、ユーザーが次のようなファイル名リストコマンドを入力した場合。

```
ls -l /home/john/
```

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。3つのパラメータ ls、-l、および

また、/bin/ls実行ファイルのロード時にコマンドラインで指定した/home/john/は、新しいプロセスに継承されます。C言語をサポートしている環境では、プログラムのメイン関数main()を呼び出す際に、2つの引数を取ります。

```
int main(int argc, char *argv[])
```

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であるため、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。

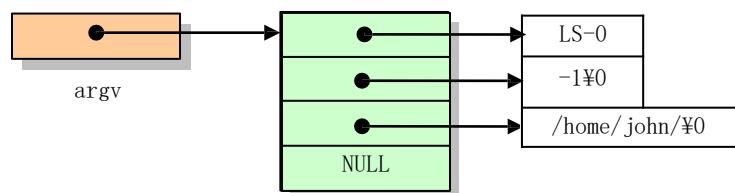


図 12-30 コマンドライン引数ポインタ配列 argv[].

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一覧が表示されます。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。

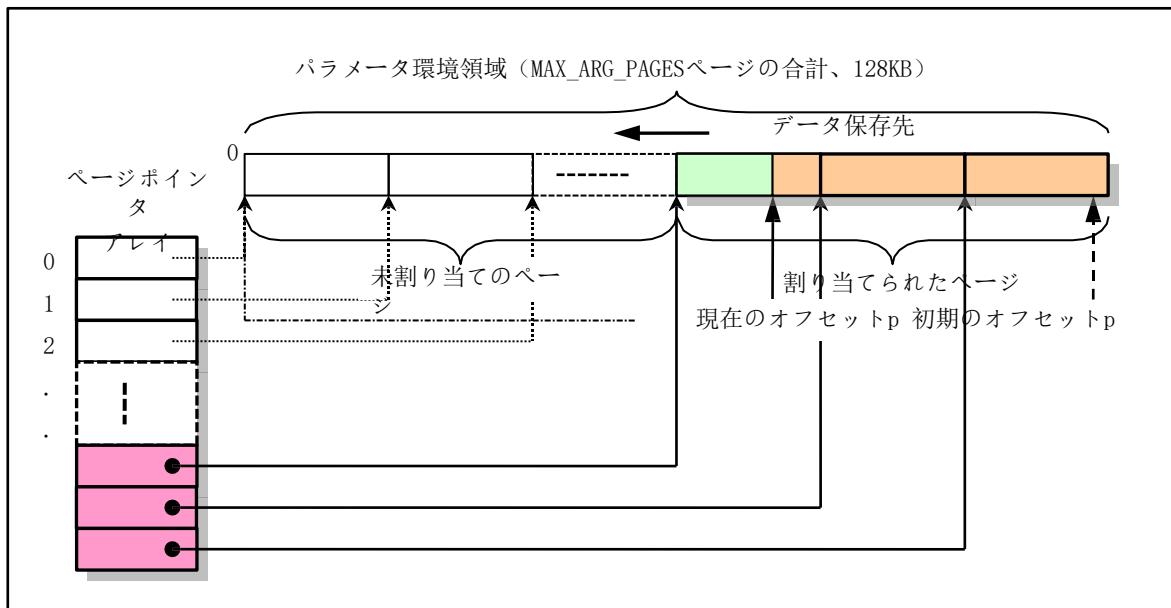


図12-31 パラメーターと環境変数の文字列スペース

`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`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。

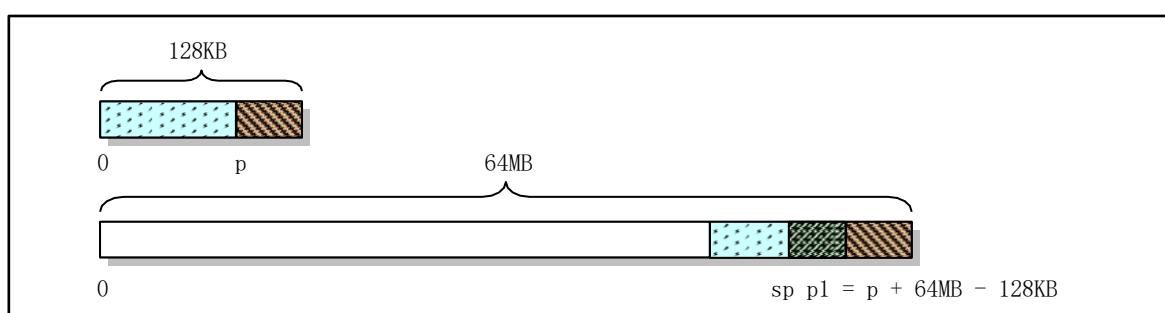


図12-32 pを初期スタックポインタに変換する方法

`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、ポインタをページアラインし、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。

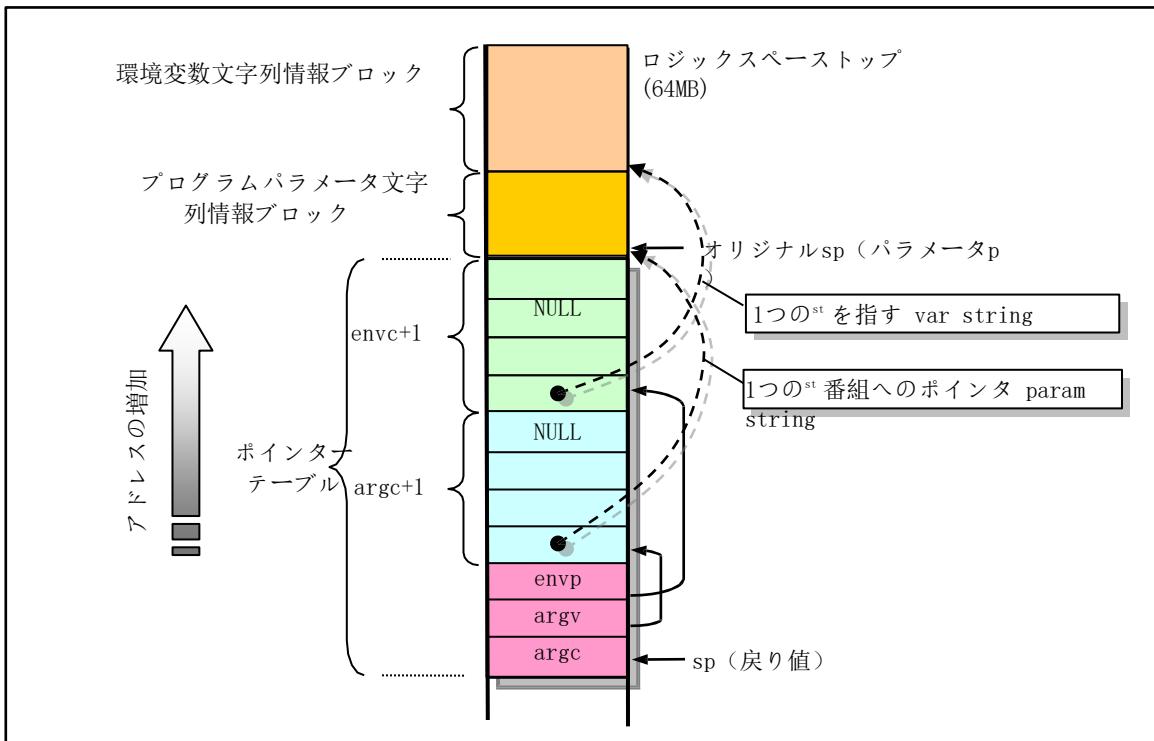


図 12-33 新プログラムスタックのポインタテーブルの模式図

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`に置き換えていきます。その後、このシステムコールのリターン命令によって、最終的には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ようになります。

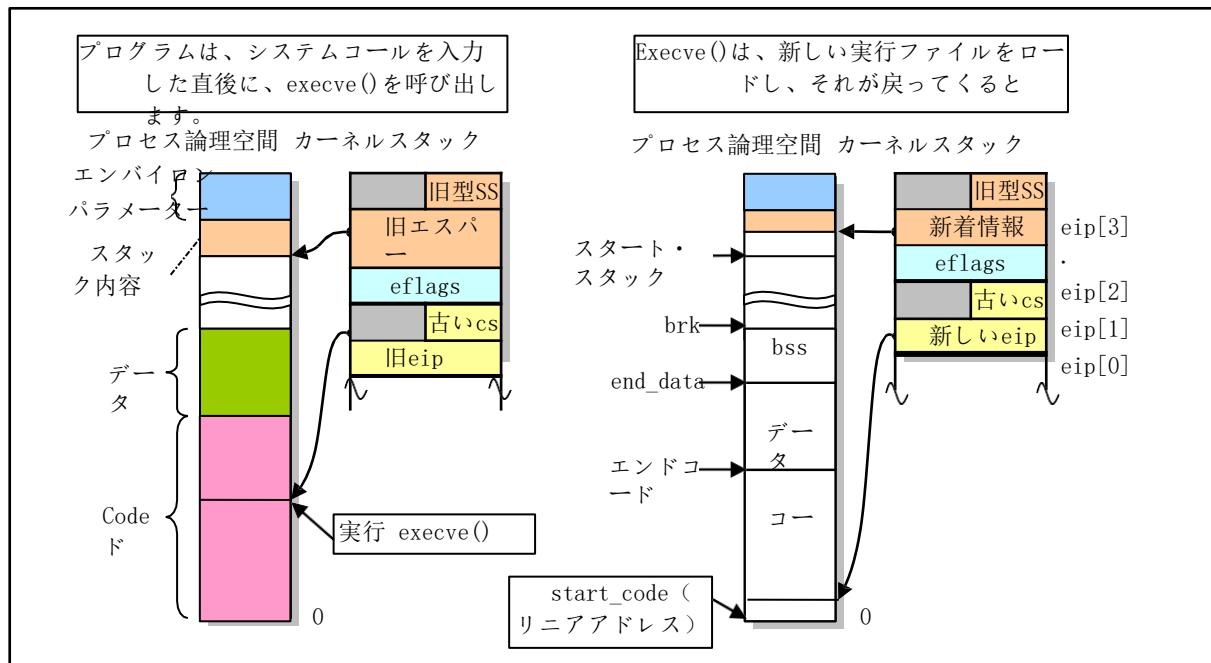


図 12-34 実行ファイルの読み込み時のスタック内のespおよびeipの変化

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 コードアノテーション

プログラム 12-14 linux/fs/exec.c

```

1 /*
2 * linux/fs/exec.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * #!-checking implemented by tytso.
9 */
10
11 /*
12 * デマンドローディング実装 01.12.91 - 何も読む必要はありませんが
13 * ヘッダをメモリに格納します。実行ファイルのinodeを
14 * "current->executable"で、ページフォールトが実際の読み込みを行います。クリーンです。
15 *
16 * もう一度、linuxが変更に耐えられたことを誇りに思うことができます：それは
17 * デマンドローディングを完全に実装するのに2時間もかかりませんでした。
18 */
19
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナルの定義

```

```

// 操作関数のプロトタイプ。
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を含む stat{}。
// そして、定数です。
// <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/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/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義され
20 #include <signal.h>
21 // include <linux/kernel.h> タスクの操作。
22 #include <string.h>
23 #include <sys/stat.h>
24 #include <a.out.h>
25
26 #include <linux/fs.h>
27 #include <linux/sched.h>
28 #include <linux/kernel.h> (日本語)
29 #include <linux/mm.h>
30 #include <asm/segment.h>
31
32 extern int sys_exit(int exit_code);           // kernel/exit.c, line 365.
33 extern int sys_close(int fd);                 // fs/open.c, line 219.
34
35 /*
36 * MAX_ARG_PAGES は、引数に割り当てられるページ数を定義します。
37 * と新しいプログラムのためのエンベロープです。32個あれば十分です、これで
38 * 最大128kBのenv+arg !
39 */
40 #define MAX_ARG_PAGES 32
41
42 ///////////////////////////////////////////////////////////////////
43 // ライブラリファイルを使用します。
44 // パラメータ: library - ライブラリファイルの名前。
45 // プロセス用のライブラリファイルを選択し、現在のライブラリのi-nodeフィールドを置き換える
46 // ここで指定されたライブラリファイルのi-nodeを持つプロセスのファイルです。もし、指定された
47 // ライブラリ
48 // パラメータの // が空の場合、プロセスの現在のライブラリファイルが解放されます。
49 // 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。
50 int sys_uselib(const char * library)
51 {
52     struct m_inode * inode;
53     unsigned long base;
54
55     // 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
56     // 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
57     // task_size (64mb)。そのため、プロセスの論理アドレス空間のサイズが

```

```

// をTASK_SIZEに設定した場合は、エラーコード（無効なパラメータ）が返され、そうでない場合は
// ライブラリファイルのinode
47 // が取れます。ライブラリファイル名ポインタが空の場合は、inodeをNULLに設定します。
48     return -EINVAL;
49     if (library) {
50         if (!(inode=namei(library)) /* ライブラリのinodeを取得 */。
51             return -ENOENT
52     } else
53         inode = NULL
54 /* ファイルタイプ(ハッダなど)をチェックすべきだが、していない*/
55 // 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-
56 // nodeをプリセットする。
57 // 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
58 // 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
59 // 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
60 // に
61 // 実行ファイルの場合は、実際のライブラリコードがメモリにロードされてから実行されます。
62 // を使用する(つまりlibrary); ライブラリファイルのコードは、プロセス空間の最後に配置されている
63 // サイズのcurrent->library = NULL;
64 // は4MBの倍数でgetpagesize()を参照してください。
65     base += LIBRARY_OFFSET;                                // linux/sched.h, line 26.
66     free_page_tables(base, LIBRARY_SIZE);
67     current->library = inode;
68     0を返す。
69 }
70 */
71 /* create_tables() は、新しいユーザのenv-stringとarg-stringを解析します。
72 * メモリからポインターテーブルを作成して、そのポインターテーブルを
73 * 「スタック」上のアドレス、新しいスタックポインタの値を返す。
74 */
75 // 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ーを生成します。
76 // 前述の図12-31から図12-33を参照してください。
77 // 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。
78 // argc - パラメータの数、envc - 環境変数の数。
79 // この関数は、スタックポインタを返します。
80 static unsigned long * create_tables(char * p, int argc, int envc)
81 {
82     符号付きロング
83     *argv, *envp; 符号付きロング
84     グ * sp;
85     // 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
86     .
87     // 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
88     // spダウン(ロードアドレス方向)で、環境変数のポインタが占める空間を空ける
89     // 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
90     // されている
91     // 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
92     // する
93     // 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
94     // 使用する
95     // でNULL値を保持します。この時点では 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。
96     // そして、環境パラメータブロックポインタenvpとコマンドラインパラメータをプッシュします。
97     // コマンドラインパラメータの数と同様に、ブロックポインタをスタックに格納します。
98     // sp = (unsigned long *) (0xfffffffffc & (unsigned long) p);
99     sp -= envc+1; // つまり、sp = sp - (envc+1) です。

```

```

76     envp = sp; sp
77     -= argc+1;
78     argv = sp;
79     put_fs_long((unsigned long)envp,--           // ユーザースペースに
80     sp);put_fs_long((unsigned long)argv,--           置く。
81     sp);put_fs_long((unsigned long)argc,--
// そしてp)コマンドラインのポインターと環境変数のポインターを
// while (argc-->0) {
82         put_fs_long((unsigned long) p,argv++);
83         while (get_fs_byte(p++))/* nothing */ ; // p は次の param 文字列を指します
84         .
85     }
86     put_fs_long(0,argv);
87     while (envc-->0) {
88         put_fs_long((unsigned long) p,envp++);
89         while (get_fs_byte(p++))/* nothing */ ; // p は次の param 文字列を指します
90         .
91     }
92     put_fs_long(0, argv)//現在構築されている新しいスタックポインタを返します。
93 }
94 /*
95 * count()は、引数/エンベロープの数をカウントする
96 */
97 /**
98 * ///////////////////////////////////////////////////////////////////
99 * パラメータの数をカウントします。
100 * // 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。
101 * // 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。
102 static int count(char ** argv)
103 {
104     int i=0;
105     char ** tmp;
106
107     if (tmp = argv)
108         while (get_fs_long((unsigned long *) (tmp++))
109             i++;
110
111     i. を返し
112     ます。
113
114 /*
115 * 'copy_string()' は、ユーザーからの引数/封筒の文字列をコピーします。
116 * メモリからカーネルメモリ内のページを解放します。これらは、すぐに使えるフ
117 * オーマットで
118 * 新しいユーザーメモリーのトップに直接入れられるようにしました。
119 */
120
121 * TYT, 11/24/91による修正で、以下を指定するfrom_kmem引数が追加されました。
122 * 文字列と文字列配列が、ユーザーセグメントのものか、カーネルセグメントのも
123 * のかを判断します。
124 */
125 * ** 0 ユーザースペ ユーザースペ
126 * from_kmem argv *argv **。 一ス
127 *   1 カーネルスペ ユーザースペ
128 *     一ス 一ス
129 * ** ここでは、fsセグメントレスメモリでダブルバームを行います。
130 *

```

```

124 * セグメントレジスタのロードにはコストがかかるため、なるべく
125 * set_fs() は、絶対に必要な場合を除きます。
126 */
    //// パラメータ文字列をプロセスのパラメータ空間と環境空間にコピーします。
    // パラメータ: argc - 追加される引数の数、 argv - 引数ポインタの配列。
    // page - 環境空間ページポインタへの引数の配列; p - オフセットポインタ
    // 引数テーブルのスペースに、コピーされた文字列のヘッダを常に指し示す。
    // from_kmem - 文字列のソースフラグです。
    // do_execve()関数では、p は引数の最後のロングワードを指すように初期化されます。
    // テーブル(128kB)のスペースに、引数の文字列が逆にスタック操作でコピーされていく
    // モードになります。そのため、コピー情報の増加に伴い、pポインタは徐々に減少していきます。
    // で、常にパラメータ文字列の先頭を指します。文字列ソースフラグ from_kmem は
    // execve()にスクリプトファイルを実行する機能を持たせるためにTYT(Tytso)が追加した新しいパラ
    // メータである。
    // execve()にスクリプトファイルを実行する機能がない場合、すべてのパラメータ文字列は
    // ユーザーデータスペースの
    // 戻り値: 引数/環境空間の現在のヘッドポインタ、またはエラーの場合は0。
128 static unsigned __attribute__((__aligned__(4))) __attribute__((__packed__)) __attribute__((__alloc_size__(2)))
129 {           from_kmem)
130     char *tmp, *pag; int
131     len, offset = 0;
132     符号付きロングのold_fs,
133     new_fs;
    // まず、カレントセグメントレジスタDS（カーネルデータセグメントを指している）とFS（ユーザー
    // データセグメント）の値を取得し、それぞれ変数new_fs、old_fsに格納します。
    // 文字列とそのポインタがカーネル空間にある場合、fsをカーネル空間を指すように設定する
134     if (!p)
135         return 0;          /* 防弾少年団 */ // オフセットポインタの検証
136     new_fs = get_ds();
137     old_fs = get_fs();
138     if (from_kmem==2)      // 文字列と文字列ポインタはすべてカーネル空間にあ
139         set_fs(new_fs)    ります。
    // 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。
140 // カーネルデータ各の文字列ポインタtmpを取得した直後にFSを復元します。
141 // そうでない場合は(文字列ポインタがユーザ空間からtmpが直接取り空間にあ)場合、FSはカーネ
    // ル空間を指します。
142         set_fs(new_fs) で
    // す。
143 if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
144     panic("argc is wrong");
145 // その後、ユーザif (from_kmem == 1)文字列からif (from_kmem == 1)文字列が取り出され// カーネル内の文字列ポインタの場合
146 // 計算され、その後、tmpは文字列の終わりを指す。もし、文字列の長さが
    // この時点でパラメータと環境の空間に残っている自由長は、空間の
    // //では不十分です。その後、FSセグメントレジスタの値が（変更されていれば）復元され、0が返され
    // ます。
147 // len=0; /* ゼロパディングを忘れずに */ (注) 1.
148         do {

```

```

149             len++です。
150         } while (get\_fs\_byte(tmp++));
151         if (p->len < 0) {...} /* このようなことがあってはならない
152             - 128kB */ (注)
153             set\_fs(old_fs)です。
154             0を返す。
155 // 次に、文字列をcharずつ逆にコピーして、パラメータと環境の最後に置く
// のスペースです。文字列の文字をループさせる過程で、まずは
// パラメータと環境の対応する位置にメモリページがあるかどうか
// のスペースを確保します。存在しない場合は、まずメモリのページを申請します。オフセット」は
// 、その時の
// ページ内の現在のポインタのオフセット値。オフセット変数'offset'は初期化されているので
// この関数の先頭で // を0にすると、次のように'(offset-1 < 0)'のチェックが確実に行われます
156   ページ範囲内の現在のpポインタに'offset'が再設定されるように、// 真。while (len) {
157     --p; --tmp; --len;
158     if (--offset < 0)
159       {. if (from_kmem==2) // 文字列がカーネル空間にある場
160           set\_fs(PAGE\_SIZE; 合。
161 // 文字列空間のページポインタの配列項目 (page[p/PAGE_SIZE]) // がまだの場合、それはスに戻ります
162 // pポインタが配置されている空間メモリページがまだ存在しない場合は
163 // 空いているメモリページを探し、そのページポインタを配列に記入します。また、それと同時に
164 // ページポインタ pag が新しいページを指すようにします。空きページがない場合は 0 を返します
165   .
166   if (! (pag = (char *) page[p/PAGE\_SIZE\]\]\) &&
167       !\(pag = \(char \*\) page\[p/PAGE\\_SIZE>\\] =
168         \\(unsigned long \\*\\) get\\\_free\\\_page\\(\\)\\)\\)
169     if \\(from\\_kmem==2\\) // 文字列がカーネル空間にある
170       set\\\_fs\\(new\\_fs\\) 場合。
171       です。 // FSはカーネル空間を指しています。
172     .
173     if \\(from\\_kmem==2\\)
174       set\\\_fs\\(old\\_fs\\)
175       です。
176   }
177 // 最後に、文字列と文字列配列がカーネル空間にある場合は、元の値であるFS
178 // セグメント・レジスタが復元され、パラメータでコピーされたパラメータのヘッダ・オフセットが
179 // と環境変数が返されます。
180   if \\(from\\_kmem==2\\)
181     set\\\_fs\\(old\\_fs\\)
182     です。
183   }
184 // 最後に、文字列と文字列配列がカーネル空間にある場合は、元の値であるFS
185 // ローカルディスクリプターのセグメントベースアドレスと長さの制限を変更する
186 // テーブルLDTで、データセグメントの最後にパラメータと環境スペースのページを配置します。
187 // Parameters: text\\_size - a\\_textフィールドで指定されたコードセグメントの長さの上限。
188 // page - パラメータと環境変数のページポインタの配列。
189 // データセグメントの制限値\\(64MB\\)を返します。
190 static unsigned long change\\\_ldt\\(unsigned long text\\_size, unsigned long \\* page\\)
191 {
192   unsigned long code\\_limit, data\\_limit, code\\_base, data\\_base;

```

```

180     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
// 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
182 読み込み中の // は元のプログラムと同じなので、繰り返す必要はありません。ということです。
183 // セグメントのベースアドレスを設定する186, 188行目の2つのステートメントは冗長です。
184 // と省略されるbaseができるbase(current->ldt[1]);           // include/linux/sched.h, line 277
185     data_base=code_limit;
186     set_base(current->ldt[1], code_base);
187     set_limit(current->ldt[1], code_limit);
188     set_base(current->ldt[2], data_base);
189     set_limit(current->ldt[2], data_limit)
190 /* fsがNEWデータセグメントを指していることを確認します。
// ローカルテーブルデータセグメント記述子(0x17)のセレクタをFSレジスタに入れる。
// つまり、デフォルトでは、FSはタスクデータセグメントを指しています。そして、データが格納さ
れているページを
// データの最後にパラメータと環境空間(MAX_ARG_PAGESページ、128kBまで)で
// セグメント。その方法は、AAの最初から、1ページずつ逆に入れていく。その方法は
// です。プロセス空間のライブラリ・コードの位置の先頭から、逆算して
// セグメントを1ページずつ表示します。ライブラリファイルのコードは、プロセス空間の端を占め
ており
191 // は4MBの倍数でpush関数(0x17)で物理ページ()は、物理ページを
192 // プロセスとbase各空間stamp/memory LIBRARY STを参照してください。
193     for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
194         data_base -= PAGE_SIZE;
195         if (page[i])                                // 存在していれば、そのペ
196             put_dirty_page(page[i], data_base);      //ジを置く。
197     }
198     return data_limit; // データセグメントの長さの上限(64MB)を返す。
199 }
200
201 /*
202 * 'do_execve()' は新しいプログラムを実行します。
203 *
204 * 注: データエリアの最上部に4MBの空きを設けて、ロード可能な
205 * ライブラリーです。
206 */
// 他のプログラムを読み込んで実行します。execve()システムコールで呼び出されます。
// これは、システムコール割り込み(int 0x80)で呼び出される関数で、関数番号はNR_execveです
。
// 関数の引数は、次のようにしてスタックに徐々に押し込まれる値です。
// 関数が呼び出されるまで、system-callが実行される(kernel/system_call.s, line 217)。
// これらの値(system_call.sファイル内)は以下の通りです。
// (1) 89~91行目でスタックにプッシュされたedx, ecx, ebxのレジスタ値は
それぞれ **envp, **argv, *filename への // // // // (2) 関数の戻り値(tmp)。
// 99行目でsys_call_tableのsys_execve関数が呼ばれたときに、スタックにプッシュされます。
// (3) 216行目に、プッシュされたシステム割り込みを呼び出すプログラムコードポインタeip
この関数 do_execve() を呼び出す前に、 // スタックに入れておきます。

```

```

// Parameters: eip - システム割り込みを呼び出すためのプログラムコードポインタ。
// tmp - _sys_execveを呼び出したときのシステム割り込みのリターンアドレス、使われていません
//
// filename - 実行可能ファイル名へのポインタです。
// argv - コマンドライン引数ポインタの配列へのポインタ。
// envp - 環境変数のポインタの配列へのポインタです。
// 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。
207 int do_execve(unsigned long * eip, long tmp, char * filename)
208 char ** argv, char ** envp) 209 {
    // 以下のローカル変数の一部の意味は以下の通りです。
    // 213行目 - 引数と環境文字列(A&E)の空間ページポインタの配列です。
    // 217行目 - シェルスクリプトファイルを実行するかどうかを制御するフラグです。
    // 218行目 - 引数&環境空間の最後尾を指すのに使われる整数 struct m_inode * inode;
210     struct buffer_head * bh;
211     struct exec_ex;
212
213                                     // A&E空間のページポインタの配列。
214     int e_uid, e_gid;
215     int retval;                                // 有効なユーザーとグループのID
216     int sh_bang = 0;                            // です。
217                                         // スクリプトファイルを実行するかどうか
218     unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4; // を制御する空欄の端を指します。
219
    // 正式に実行ファイルの環境を整える前に、いくつかの準備をしておきましょう。
    // 作業を行います。カーネルは128KB(32ページ)のスペースを用意して、コマンドラインの引数や
    // 実行ファイルの環境文字列変数。前の行では、pを初期状態である
    // 128KBスペースの最後のロングワードに配置されます。初期化パラメータと
    // pは、128KB空間内の現在の位置を示すために使用されます。
    // さらに、パラメータeip[1]は、オリジナルのユーザープログラムコードのセグメントレジスタCS値
    // このシステムコールを呼び出す // セグメントセレクタはもちろんコードセグメントでなければなりません。
    // 現在のタスクのセレクタ(0x000f)。これが値でない場合、CSはセレクタにしかなりません。
    // カーネルコードセグメントの // 0x0008を使用しています。しかし、これは絶対に許されないことで
    // 、カーネル
    // コードは常駐しており、交換することはできません。そのため、以下のような値になっています。
    // eip[1]で正常かどうかを確認します。そして、128KBの引数と環境を初期化します。
    // 文字列空間、すべてのバイトをクリアし、実行ファイルのi-nodeを取得します。次に、以下に従つ
220    // 関数のパラメータを計算し、コマンドラインパラメータargcの数と
221    // 環境文字列の(envpからMAX_ARG_PAGESまで) // 実行ファイルは通常のページでなければなりません。 if
222    // (0xffff & eip[1]) != 0x000f)                    // リア */
223    // panic("execve called from supervisor mode") // 実行ファイルのinodeを取
224    if                                                 得 */
225        (! (inode=namei(filename))               // コマンドライン引数の数です。
226            || me) return -ENOENT;                  // 環境変数の数です。
227
228    argc = count(argv);                         /* 通常のファイルである必
229    restart_entryp: count(envp);                 要があります。
230    if (! S_ISREG(inode->i_mode))             // 正規のファイルでない場合、376行目にジ
231        { retval = -EACCES;                      ャンプします。
232            goto exec_error2;
233        }
234
    // そして、現在のプロセスが、指定された実行ファイルを実行する権利を持っているかどうかをチェックします。
    // つまり、実行ファイルのi-nodeにあるモードに応じて、そのプロセスが

```

```

// 実行する許可を得ています。まず、「set-user_id」フラグと
// "set-group-id" フラグがモードに設定されます。これらの2つのフラグは、主に、一般的な
// ユーザーが passwd などの特権ユーザー（スーパーユーザー root など）のプログラムを実行でき
// るようにします。
// パスワードを変更するものです。set-user-id フラグが設定されている場合は、その後の
// 実行プロセスには実行ファイルのユーザーIDが設定され、そうでない場合は
// 現在のプロセスの euid。実行ファイルのset-group-idが設定されている場合、有効なグループ
// 実行ファイルのグループIDに実行プロセスのID(egid)が設定され、それ以外は
// 現在のプロセスのegidに設定されます。ここでは、決定した2つの値を一時的に
234 // 変数e_uidとe_gidに格納されます。
235     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
// 処理では、属性が3ビット右にシフトされ、最下位の3ビットが
// 実行ファイルグループのアクセス許可フラグ。それ以外の場合は、最下位3ビットの
// この時の//モードは、他のユーザーが実行ファイルにアクセスするための権限です。
// 次に、現在のプロセス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
237 // プロセスユーザがスーパーユーザでない場合は、現在のプロセスが
238 // 実行ファイルを実行する権限を持つ。そして、実行不可能なエラーコードを設定し、次のようにジ
239 ャンプします。
240 // exec_error2で終了処理を行う。 if
241     (current->euid == inode->i_uid)
242     && (!((i & 1) &&
243         i >>= 6;
244     else!((inode->mode&111)&gid&SUSER())){
245         retval = -ENOEXEC;
246         goto exec_error2;
247 // ここでコードが実行できる場合、現在のプロセスに実行権限があることを意味する
// 指定された実行ファイルを表示します。ですから、ここからは実行ファイルからデータを抽出する
// 必要があります。
// ファイルのヘッダを解析し、その情報に基づいて実行環境を設定したり、ラン
// 別のシェルプログラムでこのスクリプトファイルを実行します。まず、実行の最初のブロックである
// ファイルがバッファブロックに読み込まれ、バッファブロックのデータがex構造体にコピーされま
// す。
// 実行ファイルの最初の2バイトが文字'#!' であれば、それはスクリプトファイルです。
// スクリプトファイルを実行したい場合は、そのためのインターペリタ（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ペリタです。実行方法は、インターペリタを
// 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）を入力して
// これらのパラメータとスクリプトファイル（実行ファイル）名を、インターペリタの
// コマンドライン引数スペース。
// その前に、もちろん、オリジナルのコマンドラインパラメーターと環境を
// 関数で指定された文字列を128KBのスペースに格納し、コマンドラインのパラメータを
// ここで確立された//は、彼らの前に置かれます（逆になっているので）。最後に、このように
// カーネルがスクリプトファイルのインタプリタを実行します。そして、このようなパラメータを設
// 定した後
// スクリプトのファイル名として // を指定すると、インターペリタのi-nodeが取り出され、229行目にジ
// ャンプします。
// インタペリタを実行するために実行されたコード229行目にジャンプする必要があるので

```

```

// 次のスクリプト処理コードの実行を禁止するフラグ sh_bang を設定する。
// スクリプトファイルを確認して処理した後に、再び // を表示します。以下のコードでは、このフラグ
// は、実行ファイルのコマンドライン引数を設定したことを示すためにも使われます。
246 // といった設置を繰り返す(必要はありませぬ->i_zone[0])) {
   し
247         retval = -EACCES;
248         goto exec_error2;
249     }
250     元 = *((struct exec *) bh->b_data); /* read exec-header */;
251     も ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
   し
252     /*
253      * このセクションでは、#! の解釈を行います。
254      * ちょっと複雑ですが、うまくいくといいですね -TYT
255      */
256
257     char buf[128], *cp, *interp, *i_name, *i_arg;
258     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。
// スクリプトのインタープリタ名(例えば/bin/sh)を含むバッファbufに
// おそらくインターパリタのいくつかのパラメータ。次に buf の内容を処理する: replace
// 最初の改行をNULLにして、スペースタブを削除します。
260     strncpy(buf, bh->b_data+2, 127);
261     brelse(bh); // バッファブロックを解放します。
262     iput(inode); // スクリプトファイルのi-nodeを戻します。
263     buf[127] = '\0';
264     if (cp = strchr(buf, '\n')) {
265         *cp = '\0'; // スペース、タブを削除します。
266         for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
267     }
268     if (!cp || *cp == '\n') { // その行に何もなければ、error!
269         retval = -ENOEXEC; /* インタープリタ名が見つかりませんでした */。
270     // この時点で、スクリプトの名前で始まるコントンツ(文字列)の行を取得します。
271     // インタープリターです。以下にその行を分析します。まず、最初の文字列を取得しますが、これは
272     // インタープリタ名を指定し、その時点で i_name がその名前を指します。の後に文字がある場合は
273     // インタープリタ名ではなく、引数の文字列にして、i_arg がその文字列を指すようにします。
274     interp = i_name = cp;
275     i_arg = 0;
276     for ( ; *cp && (*cp != '\n') && (*cp != '\t'); cp++) {
277         if (*cp == '/')
278             i_name = cp+1です。
279     }
280     if (*cp) {
281         *cp++ = '\0'; // インタープリタ名の最後にNULLを追加します。
282         i_arg = cp; // i_arg は引数を指します。
283     }
284     /*
* OK, we've parsed out インタープリタ名と
* (オプション)引数。

```

```

285      */
286 // 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
287 // 必要があります。
288 // の名前をパラメータとしてインターパ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
289 // の前に必要なのは
290 // 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。
291 //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
292 //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
293 //す。"bash -> arg1 arg2 example.sh -arg1 -arg2"
294 // 内容が、"sh/bash/bashを設定した後でなければ、最初にbash環境变量を置いた後に
295 // 関数のpでは新しい引数を提供されれば、引数を空間に入れる環境文字列の数と
296 // 引数はそれぞれ envc と argc-1 です。コピーされない元の引数の一つ
297 //は元の実行ファイル名で、ここではスクリプトファイルの名前になっていて、次のようにになります
298 // 。
299 // は、以下のように処理されます。
300 // 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
301 // 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
302 // 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。
303 //とpはプログラムの引数文字列の最初を指します。さらに、最後のパラメータである
304 copy_strings()のif(!!is_bang)の文字列がユーザ空間にあることを示しています。
305
306     p = copy_strings(envc, envp, page, p, 0);
307     p = copy_strings(--argc, argv+1, page, p, 0);
308 }
309 /*
310 * argv[0]に(1)インターパリタの名前をスプライスしま
311 * す。
312 * (2) (オプション)シェルスタック内の引数
313 * ル名
314 * これは逆の順序で行われます。
315 * ユーザー環境や引数が保存されます。
316 */
317
318 // 次に、スクリプトのファイル名、インターパリタのパラメータ、インターパリタのファイル名を逆
319 // コピーする
320 // 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
321 // 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
322 // 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
323 // カーネルにあります。
324 // スペースが必要なので、このコピー文字列関数の最後のパラメータ（文字列ソースフラグ）は
325 // 文字列がカーネル空間にある場合copy&string()の最後のパラメータには、// 1を設定します。
326 // 以下の301行目と304行目を示すように、// 2に設定します。
327
328     if (i_arg) { // インターパリタの複数の引数をコピーします。
329         p = copy_strings(1, &i_arg, page, p, 2)となります。
330         argc++です。
331     }
332
333     p = copy_strings(1, &i_name, page, p, 2);
334     argc++です。
335     if (!p) {
336         retval = -ENOMEM;
337         goto exec_error1;
338     }
339 /*
340 * OK、ではインターパリターのinodeでプロセスを再起動します。
341 */
342

```

```

// 最後にインターフリタのi-nodeポインタを取得し、229行目にジャンプして実行します。
// インターフリターです。インターフリタのi-nodeを取得するためには、namei()を使う必要があります。
// 関数ですが、関数で使用するパラメータ（ファイル名）はユーザーデータから取得します。
// の空間、つまり、セグメントレジスタFSが指す空間からです。そのため
// namei()関数では、FSが一時的にカーネルのデータ空間を指すようにしておく必要があります。
// この関数は、カーネル空間からインターフリターの名前を取得し、デフォルトの設定を復元する
// と実行可能にするファイル — スクリプトファイルのインタ
// namei()が戻った後のFSなので、その後、restart_interp(229行目)にジャンプして、新しい
313         old_fs = get_fs();
314         set_fs(get_ds)()である。
315         if (!(inode=namei(interp))) {           /* 実行ファイルのinodeを取
316             得 */
317             set_fs(old_fs)です。
318             retval = -ENOENT;
319             goto exec_error1;
320         }
321         set_fs(old_fs)です。
322         goto restart_interp;
323     }
324 // この時点では、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
325 // 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
326 // します
327 // 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
328 // 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
329 // 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
330 // 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
331 // 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
332 // (ZMAGIC)、またはコードとデータの再配置部分が0にならない、または（コード部+データ
333 // セグメント部+セクションが50MB超え実行不可能な場合）または実行ファイルサイズが（コードセクション
334 // ブレルス (bh) です。
335         if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||)
336             ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
337             inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex) ) {
338             .
339             retval = -ENOEXEC;
340             goto exec_error2;
341         }
342 // 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
343 // ページ(1024バイト)の//を超えると、実行できません。デマンドページング方式では
344 // 実行ファイルの内容がページ単位で読み込まれること、コードやデータが
345 // 実行可能なファイルイメージは、ページ境界で開始する必要があります。
346         も N_TXTOFF(ex) != BLOCK_SIZE ) {
347             .
348             .
349             printk ("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename)と
350             なります。
351             retval = -ENOEXEC;
352         }
353 // 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
354 // 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
355 // がセットされていれば、スクリプト・インターフリターが実行されることを意味し、環境変数
356 page
357 // 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
358 // 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
359 // 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
360 // 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。
361 // プログラムの文字列です。実際には、pは128KBのパラメータと環境でのオフセット値

```

```

// スペース、つまり p=0 の場合は、環境変数とパラメータスペースのページが
// すでに をフルに活用しています。
335     もし (!sh_bang) {
336         p = copy_strings(envc, envp, page, p, 0);
337         p = copy_strings(argc, argv, page, p, 0) となり
338             ます。
339         if (!p) {
340             retval = -ENOMEM;
341             goto exec_error2;
342     }
343 /* OK, This is the point of no return */
344 /* 実行しても current->library は変更されないことに注意してください。
   // 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ピターテーブルのディスクリピターの内容 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。
   // ディスクリピターが再設定され、取得したe_uidとe_gidを使って関連フィールドが設定される
   // をプロセスタスクに入れます。最後に、これを実行するプログラムのリターンアドレスeip[]を
   // システムコールは、実行ファイルのコードの先頭にポイントされます。このようにすると
   // システムコールが終了して戻ると、新しい実行ファイルのコードを実行します。
   // 新しい実行ファイルのコードとデータはまだメモリにロードされていませんが、注意してください
   。
   ファイル、パラメータ、環境ブロックから // get_free_page() を使用した
   // copy_strings() でデータを格納する物理メモリページを取得し、 put_page() を使って
   // change_ldt() でプロセスロジック空間の最後にデータを格納します。また、イン
   // create_tables() では、引数や環境を格納する際にもページフォルト例外が発生します。
   メモリマネージャが物理的なメモリページもマッピングするように、ユーザースタック上の // ポイン
   ターブル
   // ユーザーのスタックスペースに
   //
345 // 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
346 // 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
347 // ルをリセット
348 // current->executable = inode;
349 // 元のプロセスの処理ハンドルですが、SIG_IGNのためにリセットする必要はありません。
350 // ハンドルになります。その後、close_on_exec ビットマップフラグに従って、指定されたオープン
351 // ファイルを閉じて
352 // current-> sigaction[i].sa_mask = 0;
353 // フラグをリセットします。
354 // current-> sigaction[i].sa_flags = 0;
355 // if (current-> sigaction[i].sa_handler != SIG_IGN)
356 //     current-> sigaction[i].sa_handler = NULL;
357     }
358     for (i=0 ; i< NR_OPEN ; i++)
359         if ((current->close_on_exec>>i)&1)
360             svs_close(i);
361     current->close_on_exec = 0;

```

```

// その後、ベースアドレスと現在のプロセスで指定された長さの制限に従って
// 物理的なメモリページと、ページテーブルに対応して指定されたページテーブル自体が
// オリジナルプログラムのコードとデータセグメントへの // を公開します。この瞬間、新しい
// 実行ファイルが主記憶領域のどのページも占有していないので、ページフォールトのアボート
// は、プロセッサが新しい実行ファイルコードを実際に実行するときに発生します。メモリ管理は
// その後、プログラムはページフォールト処理を行い、メモリページの申請と関連ページの設定を行
// います。
// 新しい実行ファイルのためのテーブルエントリーを読み、関連する実行ファイルページを
// メモリを使用しています。また、「最後のタスクがコプロセッサを使用した」が現在のプロセスを
359 指している場合は
360 // free_page_tables(get_base(current)->ldt[1]), get_limit(0x0f));
361     free_page_tables(get_base(current)->ldt[2]), get_limit(0x17));
362     if (last_task_used_math == current)
363         last_task_used_math = NULL;
364     current->used_math = 0;
// そして、LDTのディスクリピターベースアドレスとセグメント長を以下のように変更します。
// 新しい実行ヘッダー構造体のコード長フィールドa_textを、128KBの
データセグメントの最後に、 // 引数と環境スペースのページがあります。を実行した後
// 次の文では、pをデータの先頭からのオフセットに変更しています。
// のセグメントではなく、引数と環境空間のデータの先頭を指しています。
// つまり、pはスタックポインタの値に変換されています。次に、内部関数
// スタックに環境変数とパラメータ変数のポインタテーブルを作成する create_tables()
364 // p += change_ldt(ex.a_text, page)...プログラムのmain()の空間をパラメータにし
365     て、スタックポインタを返す。
366     p -= LIBRARY_SIZE + MAX_ARG_PAGES*PAGE_SIZE;
            p = (unsigned long) create_tables((char *)p, argc, envc);
// そして、プロセスフィールドの値を新しい実行ファイルの情報になるように修正します。
// つまり、プロセススタスク構造コードテールフィールドend_codeは、コードセグメント
// 実行ファイルの長さa_text; データテールフィールドend_dataはコードセグメントに等しい
// 実行ファイルの長さにデータセグメントの長さ (a_data + a_text) を加えたもの、およびプロセ
スの
// ヒープのエンドフィールド brk = a_text + a_data + a_bss. の終了位置を示すのにbrkが使われま
す。
// プロセスの現在のデータセグメント（未初期化部分を含む）、およびカーネルの
// プロセスにメモリを割り当てる際に、割り当ての開始位置を指定します。
367 // そして、プロセスのスタックスタートフィールドをスタックポインタがあるページに設定して
368 // プロセスの有効なユーザIDと有効なグループIDを再設定 current->brk =
369     ex.a_bss +.
370     (current->end_data = ex.a_data +)
371     (current->end_code = ex.a_text));
372     current->start_stack = p & 0xffffffff000;
            current->suid = current->euid = e_uid;
            current->sgid = current->egid = e_gid;
// 最終的には、システムコール割り込みを最初に呼び出したプログラムポインタが置き換えられる
// 新しい実行ファイルのエンタリーポイントを指すように、スタック上のコードポインターと一緒に
// スタックポインタが新しい実行ファイルのスタックポインタに置き換えられます。それ以降は
// return命令は、スタック上のデータをポップアップし、CPUが
373 // 新しい実行アドレスを取る；最初にシステムを呼び出しありがつくランタイムに戻りません。
374 // 割り込み戻す p; :-*/* */
375     0を返す。 /* スタックポインタ */
376 exec_error2です。
377 iput(inode); 378 // i-nodeを戻します。
exec_error1:

```

```

379     for (i=0 ; i< MAX_ARG_PAGES ; i++)
380         free_page(page[i]);           // エラーが発生した場合、メモリページを解放します。
381     return(retval)になります。      // エラーコードを返す。
382 }
383

```

16. stat.c

1. 機能

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。

```

// ファイルのステータス構造。すべてのフィールドは、ファイルのi-node構造から利用できます。
6 struct stat {
7     dev_t    st_dev;        // ファイルが格納されているデバイス番号
8     ino_t   st_ino;        // ファイルのi-node番号。
9     umode_t st_mode;       // ファイルタイプとモード。
10    link_t  st_nlink;      // ファイルへのリンクの数。
11    uid_t   st_uid;        // ファイルのユーザー識別番号。
12    gid_t   st_gid;        // ファイルのグループ番号。
13    dev_t   st_rdev;       // デバイス番号（ファイルがcharまたはblockデバイスファイルの場合）。
14    off_t   st_size;       // ファイルが通常のファイルの場合は、 // ファイルサイズ（バイト）を指定します。
15    time_t  st_atime;      // ファイルが最終的にアクセスされた時間。
16    time_t  st_mtime;      // ファイルが最終的に変更された時間。
17    st_ctime;            // ラストアクセスタイム
18 };
```

// i-node last changed time.

12.16.2 コードアノテーション

プログラム 12-15 linux/fs/stat.c

```

1 /*
2 * linux/fs/stat.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 // <errno.h> エラー番号のヘッダーファイルです。システムの様々なエラー番号を含みます。
8 // <sys/stat.h> ファイル状態のヘッダーファイルです。ファイルやファイルシステムの状態を表す構造
9 // 体を含む stat{}。
10 // そして、定数です。
11 // <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する (file,
12 buffer_head,
13 // m_inode など) を使用しています。
14 // <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ

```

```

// の初期タスク0と、記述子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
// 数のマクロ文が含まれています。
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプ
// ロトタイプ定義が含まれています。
// カーネルの使用する機能
// <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が定義され
// ています。
// セグメントレジスタの操作。 7
#include <errno.h>
8 #include <sys/stat.h>
9
10 #include <linux/fs.h>
11 #include <linux/sched.h>
12 #include <linux/kernel.h> (日本語)
13 #include <asm/segment.h>
14
    //// i-nodeを使ってファイルのステータス情報を取得します。
    // パラメータのinodeはファイルのi-node、statbufはstatファイルの状態構造
    // 取得した状態の情報を格納するために使用される、ユーザーデータ空間内のポインタです。
15 static void __user stat(struct m inode * inode, struct stat * statbuf)
16 {
17     int i;
18
19     // まず、データを格納するのに十分なメモリ領域を確認（または確保）し、一時的に
    // 対応するファイルのi-nodeの情報をtmpに送る。最後に、これらのステータス情報
    // verify_area(statbuf, sizeof (struct stat));
20
21     tmp.st_dev = inode->i_dev;           // ファイルが格納されているデバイス番号
22     tmp.st_ino = inode->i_num;          // ファイルのi-node番号。
23     tmp.st_mode = inode->i_mode;         // ファイルタイプとモード。
24     tmp.st_nlink = inode->i_nlinks;      // ファイルへのリンクの数。
25     tmp.st_uid = inode->i_uid;          // ユーザーID。
26     tmp.st_gid = inode->i_gid;          // グループID。
27     tmp.st_rdev = inode->i_zone[0];       // デバイス番号（ファイルがchar/blockファイルの
                                            // 場合）。
28     tmp.st_size = inode->i_size;         // ファイルサイズ（バイト）。
29     tmp.st_atime = inode->i_atime;        // ラストアクセスタイム
30     tmp.st_mtime = inode->i_mtime;        // 最終更新時刻。
31     tmp.st_ctime = inode->i_ctime;        // i-node last changed time.
32     for (i=0; i<sizeof(tmp) ; i++) // i-node last changed time.
33         put_fs_byte(((char *) &tmp)[i], i + (char *) statbuf)
34 }
35
    //// ファイル・ステータス・システム・コール機能。
    // 与えられたファイル名に応じて、関連するファイルのステータス情報を取得する。
    // パラメータ statbuf は、状態情報を格納するためのユーザバッファポインタです。
    // 成功した場合は0を、エラーが発生した場合はエラーコードを返します。
36 int sys_stat(char * filename, struct stat * statbuf)
37 {
38     struct m inode * inode; 39
39
        // まず、ファイル名から対応するi-nodeを見つけ、ファイルの状態をコピーします。
        // i-nodeの情報をユーザバッファに入れ、最後にi-nodeを戻す。 if
40         (! (inode=namei(filename)))
41             return -ENOENT

```

```

42     cp_stat(inode, statbuf);
43     iput(inode);
44     0を返す。
45 }
46
//// シンボリックリンクファイルの状態をシステムコールする機能です。
// 与えられたファイル名に応じて、関連するファイルのステータス情報を取得します。もし、シンボリックな
ファイルパス名に // リンクファイル名が含まれている場合、シンボルファイル自体のステータス情報は
//が取られ、リンクが踏まれていない。
// パラメータ statbuf は、ファイルのステータス情報を格納するためのユーザバッファポインタで
す。
// 成功した場合は0を、エラーが発生した場合はエラーコードを返します。
47 int sys_lstat(char * filename, struct stat * statbuf)
48 {
49     struct m_inode * inode; 50
// まず、ファイル名に応じて対応するi-nodeを探します。それがシンボリックリンクの場合
51 // ファイルが(既に)存在しない場合は inode が取得できません。その後この i-node を取得する情報がコピ
52 // 一されます。      return -ENOENT;
53 // をユーザバッファに入れます。i-nodeを戻します。
54     iput(inode); 55
// 0を返す。
56 }
57
//// ファイルハンドルを使って、ファイルの状態を取得します。
// 与えられたファイルハンドルに基づいて、ファイルのステータス情報を取得します。
// パラメータfdは指定されたファイルのハンドル（記述子）、statbufは
// ステータス情報を保持するユーザバッファポインタ。
// 成功した場合は0を、エラーが発生した場合はエラーコードを返します。
58 int sys_fstat(unsigned int fd, struct stat * statbuf)
59 {... 60
    struct file * f; 61
    m_inode * inode; 62
// まず、ファイルハンドルに対応するファイル構造を取得して、i-nodeの
// ファイルを表示します。その後、i-nodeのファイル・ステータス情報がユーザー・バッファにコピ
一されます。
// ファイルハンドルの値が、以下の方法で開くことのできるファイルの最大数よりも大きい場合は
// プログラムが NR_OPEN であるか、ハンドルのファイル構造ポインタが NULL であるか、i-node フ
63 ィールドが
64 // if (fd >= NR_OPEN || !(f=<current->filp[fd]) || !(inode=f->f_inode))
65 //           return -EBADF;
66     cp_stat(inode, statbuf);
67 } 68
// Read symbolic link file system-call.
// この関数は、シンボリックリンクファイルの内容（つまり、パス名の文字列である
シンボリックリンクで指示されたファイルの//）をユーザーバッファに配置します。もしバッファ
が
// が小さすぎる場合は、シンボリックリンクの内容が切り捨てられます。
// パラメータ: path はシンボリックリンクファイルのパス名、buf はユーザーバッファ、bufsiz
は
// バッファの長さ。
// 戻り値: 成功した場合はバッファ内の文字数、失敗した場合はエラーコード。

```

```

69 int sys_readlink(const char * path, char * buf, int bufsiz)
70 {
71     struct m_inode * inode;
72     struct buffer_head * bh;
73     int i;
74     チャーC
75
// まず、関数のパラメータの有効性を確認・検証し、調整します。ユーザーが
// バッファサイズ bufsi は 1 から 1023 の間でなければなりません。次に、シンボリックリンクフ
// ァイルのi-nodeを取得します。
76 // としもレフ(bufsizのゼロ)内容の最初のブロックを読みます。そして、i-nodeを戻します。
77     return -EBADF;
78    もし (bufsiz > 1023)
79         bufsiz = 1023です。
80     verify_area(buf, bufsiz)です。
81     if (!(inode = lnamei(path)))
82         return -ENOENT;
83     if (inode->i_zone[0])
84         bh = bread(inode->i_dev, inode->i_zone[0])です。
85     その他
86     bh = NULL。
87     input(inode)です。
// ファイルデータの内容が正常に読み込まれた場合、最も多いbufsiz文字がコピーされる
// ファイルをユーザバッファにコピーする際、NULL文字はコピーされません。最後に、バッファを解
放します。
88 // ブロックして、コピーされたバイト数を返す if
89     (!bh)
90         0を返す。
91     i = 0;
92     while (i<bufsiz && (c = bh->b_data[i])) {
93         i++;
94         put_fs_byte(c, buf++);
95     }
96     brelse(bh);
97 }    return i;
98

```

17. fcntl.c

1. 機能

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、ファイルテーブルのエントリを

元のハンドルがコピーまたは複製されます。例えば、プロセスが他のファイルを開いていないときに、dup()またはdup2()関数を使用し、新しいハンドルを3に指定した場合、関数実行後のファイルハンドルは図12-35のようになります。

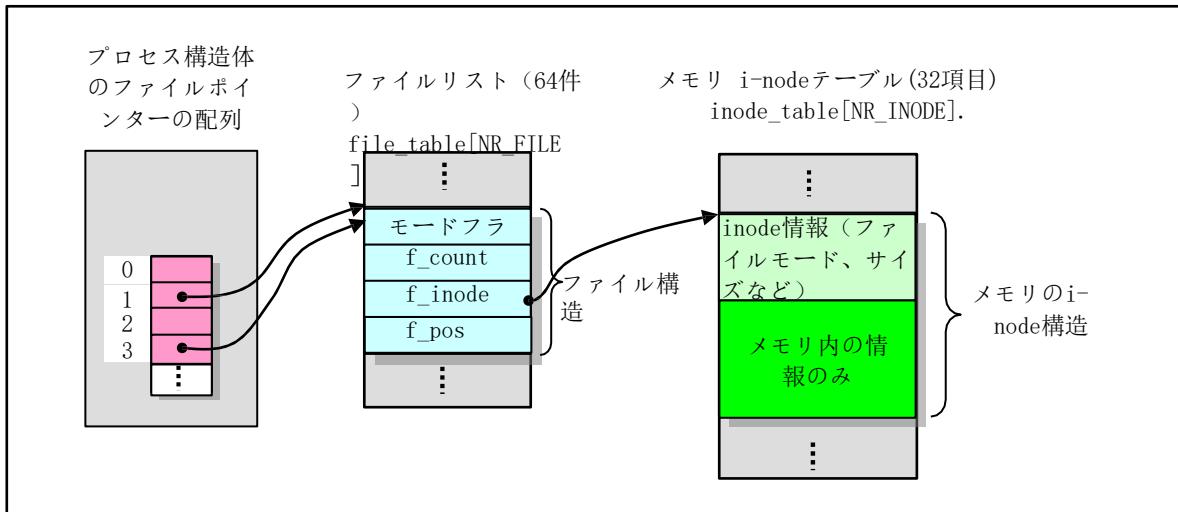


図12-35 dup(1), dup2(1,3)関数実行後のファイル構造

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。この関数は、パラメータに制御コマンドcmdを持つ関数で、4つの機能を統合しています。

- `cmd = F_DUPFD`で、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`。この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は、関数の第3引数に新しいビット

フラグの値です。

- `cmd = F_GETFL` または `F_SETFL` です。これらの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される。これらのフラグは、`RONLY`、`O_WRONLY`、`O_RDWR`、`O_APPEND`、`O_NONBLOCK`です。 詳細は、`include/fcntl.h`ファイルを参照のこと。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る。
- `cmd = F_GETLK, F_SETLK, F_SETLKW`のいずれかです。これらのコマンドは、ファイルロックフラグの読み取りや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 コードアノテーション

プログラム 12-16 linux/fs/fcntl.c

1 /*

```

2 * linux/fs/fcntl.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <sys/types.h> 型のヘッダファイル。基本的なシステムデータ型が定義されています。
// <linux/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を定義します。
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネルの使用する機能
// <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が定義されています。
// セグメントトレジスタの操作。
// <fcntl.h> ファイル制御のヘッダーファイルです。演算制御定数記号の定義は
// ファイルとそのディスクリプターに使用されます。
7 #ifndef _LINUX_FCNTL_H_
8 #include <errno.h>
9 #include <sched.h>
10 #include <linux/kernel.h> (日本語)
11 #include <asm/segment.h>
12
13 #include <fcntl.h>
14 #include <sys/stat.h>
15                                     // ファイルシステムコールを閉じる。
16 extern int sys_close(int fd);           (fs/open.c, 219)
17 //// ファイルハンドル（ファイル記述子）を複製します。
// パラメータfdは複製するファイルハンドル、argは最小数を指定します。
// 新しいファイルハンドルの
// 新しいファイルハンドルまたはエラーコードを返します。
18 static int dupfd(unsigned int fd, unsigned int arg)
19 {
    // この関数はまず、関数パラメータの有効性をチェックします。もし、ファイルハンドルの値が
    // がプログラムのオープンファイルの最大数NR_OPENよりも大きい場合、またはファイル構造が
    // ハンドルの // が存在しない場合は、エラーコードが返されます；指定された新しいハンドル値が
    // argが開いているファイルの最大数よりも大きい場合は、エラーコードも返されます。備考
    // ファイルハンドルが、実際にはファイル構造ポインタの配列項目のインデックス番号であることを
20 示しますif (fd >= NR_OPEN || ! current-
21             >filp[fd]) return -EBADF;
22     if (arg >= NR_OPEN)
23         return -EINVAL;
    // 次に、プロセスのファイル構造ポインタ配列の中から、インデックス番号を持つエントリを探します。
    // argと同じかそれ以上で、まだ使用されていない。見つかった新しいハンドルが
24 // オープンファイル（argの最大数NR_OPEN（つまり、空きアイテムがない））の場合は、エラーコードが返
25 されます。      if (current->filp[arg])
                    arg++です。
26
27     その他
28         ブレークします。

```

```

29     if (arg >= NR_OPEN)
30         return -EMFILE;
// そうでなければ、見つかったアイドルアイテム（ハンドル）について、ハンドルの対応するビット
// がリセットされる
// をフラグ close_on_exec で指定しています。つまり、exec() クラスの関数を実行すると、作成さ
// れたハンドルは
dup()を使用した // は閉じられません。そして、ファイル構造体のポインタは、ポインタ
31 // を元のハンドルfdに戻し、ファイル参照数を1つ増やします。最後に
32 // 新しいファイルハンドルargが返されます。
33     current->close_on_exec &= ~(1<<arg);
34 }     (current->filp[arg] = current->filp[fd])->f_count++;
35     return arg;
//// ファイルハンドルsystem-call2を複製します。
// 与えられたファイルハンドルoldfdを複製すると、新しいファイルハンドルの値はnewfdと同じにな
// ります。もしnewfd
// がすでに開いている場合は、先に閉じてください。
// パラメータ: oldfd -- 元のファイルハンドル、newfd -- 新しいファイルハンドル。
// 新しいファイルハンドルを返します。
36 int sys_dup2(unsigned int oldfd, unsigned int newfd)
37 {。    sys_close(newfd);           // すでに開いている場合は、最初にニューファ
38     return dupfd(oldfd, newfd);   // ンドを閉じました。
40 }                           // 複製して新しいハンドルを返します。
41
//// ファイルハンドルのsystem-callを複製する。
// 与えられたファイルハンドルoldfdを複製し、新しいハンドルの値は現在の最小値になります。
// 未使用のハンドル値です。
// Parameters: fildes -- 複製されるファイルハンドル。
// 新しいファイルハンドルを返します。
42 int sys_dup(unsigned int fildes)
43 {。
44     return dupfd(fildes, 0);
45 }
46
//// ファイル・コントロール・システム・コールです。
// パラメータfdはファイルハンドル、cmdは制御コマンド（include/fcntl.h, lines
// 23-30）；argは、コマンドによって意味が異なります。重複ハンドルコマンドの場合
// F_DUPFD, arg は新しいファイルハンドルに取ることのできる最小の値で、セットファイルの場合
// は
// 操作とアクセスフラグのコマンド F_SETFL, arg が新しいファイル操作とアクセスモードになりま
// す。
// ファイルロックコマンド F_GETLK, F_SETLK, F_SETLKW では、arg はロックへのポインタ
// 構造体になっています。ただし、このカーネルでは、ファイルロック機能は実装されていない。
// 返します。エラーが発生した場合、すべての操作は-1を返します。成功した場合、F_DUPFDは、新
// しい
// F_GETFD はファイルハンドルの close_on_exec フラグを返し、F_SETFL はファイルハンドルの
// ファイル操作とアクセスのフラグ。
47 int sys_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)
48 /* 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。
49 struct file *filp; 50
// ファイルハンドルの値が最大数よりも大きい場合は
// プロセスでファイルを開く NR_OPEN、またはハンドルのファイル構造ポインタが NULL の場合。
// エラーコードが返されます。
51     if (fd >= NR_OPEN || !(filp = current->filp[fd]))
52         return -EBADFE;

```

```

53     スイッチ (cmd) {
54         case F_DUPFD: // ファイルハンドルを複製する。
55             return dupfd(fd, arg);
56         case F_GETFD: // ファイルのclose_on_execフラグビットを取得します。
57             return (current->close_on_exec->fd)&1;
58         case F_SETFD: // close_on_exec フラグを設定 / リセットする argが1の場合に設
59             定する。
60             if (arg&1)
61                 current->close_on_exec |= (1<<fd);
62             その他
63                 current->close_on_exec &= ~(1<<fd);
64             0を返す。
65         case F_GETFL: // ファイルのステータスフラグとアクセスモードを取得します。
66             return filp->f_flags;
67         case F_SETFL: // ステータスとアクセスモード (append, non-block) を設定します
68             .
69             filp->f_flags &= ~(O_APPEND | O_NONBLOCK);
70             filp->f_flags |= arg & (O_APPEND | O_NONBLOCK);
71             0を返す。
72         case F_GETLK: case F_SETLK: case F_SETLKW: // 実装されていません。
73             1を返す。
74             のデフォルトです。
75             1を返す。
76     }
77 }
```

18. ioctl.c

1. 機能

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ーフェース制御関数を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。

2. コードアノテーション

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。
852
```

```

// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造
体を含む stat{}。
// そして、定数です。
// <linux/fs.h> の一部で、この構造体はファイルシステムの操作関数を定義する。
// のマクロ文が含まれています。
7 #include <string.h>
8 #include <errno.h>
9 #include <sys/stat.h>
10
11 #include <linux/sched.h>
12
13 extern int tty_ioctl(int dev, int cmd, int arg); // chr_drv/tty_ioctl.c, line 133.
14 extern int pipe_ioctl(struct inode*pino, int cmd, int arg); // fs/pipe.c, line 118.
15
16 // 入出力制御(ioctl)機能のポインタタイプを定義する。
17 typedef int (*ioctl_ptr)(int dev, int cmd, int arg);
18
19 // システム内のデバイスタイプの数を取得します。
20 #define NRDEVS ((sizeof (ioctl_table))/(sizeof (ioctl_ptr)))
21
22 // Ioctl操作関数ポインタテーブル。
23 static ioctl_ptr ioctl_table[]={{}};
24
25 // ルートノード
26 // デバイスドライバ
27 // フィルタリング
28 // パイプ
29 // フォルダ
30 // ファイル
31
32 // // 入出力制御システムコールです。
33 // この関数はまず、パラメータで与えられたファイルディスクリプタが有効かどうかを判断します。
34 // そして、i-nodeのファイルモードに応じてファイルタイプをチェックし、関連する
35 // ファイルの種類に応じたioの処理機能。
36 // 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。
37 // 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。
38 int sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
39 {
40     struct file* filp;
41     int dev, mode;
42
43     // 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
44     // 開放可能なファイルの // の数が少ないか、対応するハンドルのファイル構造ポインタが NULL の場
45     // 合は
46     // のエラーコードが返されます。ファイル構造がパイプノードに対応している場合には
47     // プロセスが権利を持っているかどうかに従って、パイプIOを実行するかどうかを判断するために
48     // を使ってパイプを操作します。実行の許可があればPipe_ioctl()が呼ばれ、なければ
49     // 無効なファイルのエラーコードが返されます。
50     if (fd >= NR_OPEN || !(filp = current->filp[fd]))
51         return -EBADF;
52
53     if (filp->f_inode->i_pipe)
54         return (filp->f_mode&1)? pipe_ioctl(filp->f_inode, cmd, arg):-EBADF;

```

```

// その他のタイプのファイルの場合は、対応するファイルのモードを取り、タイプを決定する
ファイルの // を適宜変更する。ファイルがキャラクタデバイスファイルでもブロックデバイスファ
イルでもない場合は
// ファイルの場合、エラーコードが返されます。キャラクタまたはブロックデバイスファイルの場合
は、デバイス番号
40 //は、ファイルのi-nodeから取得します。の数よりも大きい場合は、デバイス番号が
41 // mode=filp->f_inode->i_mode. システム内のデバイスには
42     // エラー番号が返されます。
43     if (!$iSCHR(mode) & & _zone[0];           // デバイスタイプファイルのデバイス
44         if (MAJOR(sdesBLK(mode))<-           No.
45             EINVAL; -ENODEV;
46     // そして、IO制御テーブルioctl_tableにしたがって、ioctl関数ポインタの
47     // 対応するデバイスが見つかり、その関数が呼び出されます。もし、デバイスが
48     // if (!ioctl_table[MAJOR(dev)])
49         return -ENOTTY;
50     ioctl_table[MAJOR(dev)](dev, cmd, arg)を返す。
51 }
52

```

19. select.c

1. 機能

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、select.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ること。

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。

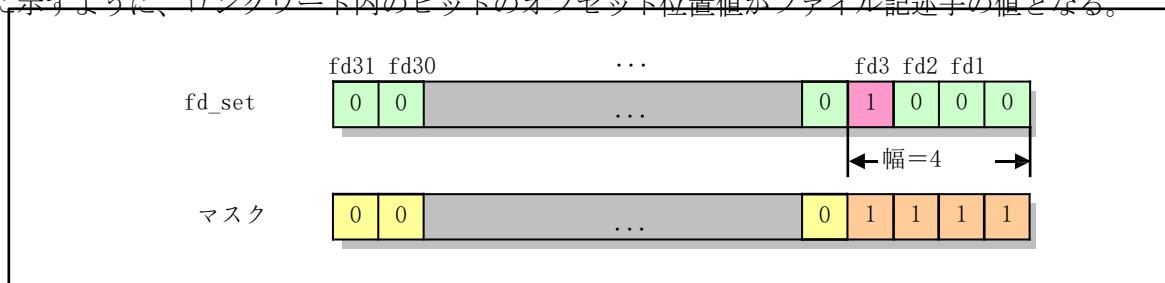


図12-36 ファイルディスクリプターセットは1ビットにつき1つのディスクリプターを表す

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い4つのマクロが用意されている。これらのマクロは、ディスクリプターセットのクリア、ディスクリプター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)          // ディスクリプター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設定します。
#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。
```

記述子セレクタ変数を宣言した後、(*(fdsetp) & fd)ではなくFD_ISSET(fd, fdsetp)をタリアイテムの次に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はセット/リセットする必要がある。

FD_ISSETは、select()が戻ってきたときに、ディスクリプターセットの指定されたビットがまだセットされているかどうかをテストするために使用される。select()が戻ってきたとき、3つのディスクリプターセットの中でまだセットされているビットは、対応するファイルディスクリプターの準備ができていることを示している（読み取り、書き込み、または例外）。なお、これらのマクロでは、第2引数に記述子セットのポインタを用いる必要がある。

select()関数の最後のパラメータであるtimeoutは、任意のディスクリプターの準備が整うまで、プロセスがselect()を待つ最大時間を指定するために使用されます。型の構造体へのポインタである。

timeval (include/sys/time.hファイルで定義されています) を以下のように使用します。

```
構造体 timeval {
    ロング tv_sec;           /* 秒 */
    ロング tv_usec;          /* マイクロ秒 */
};                           /* マイクロ秒 */
```

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は、待機処理が中断され、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、待ち時間がないことを意味する。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、次のように返すことができます。

をすぐに実行します。2つのタイムフィールド値のうち、少なくとも1つが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。待機時間中にディスクリプタの準備ができた場合は、直接戻り、このとき2つの時間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の準備ができていない場合、select()は0を返す。また、待ち時間中にシグナルによって中断されることがあり、-1を返す。

一般に、select()が-1を返す場合はエラーを示し、select()が0を返す場合は、指定された条件で準備ができているディスクリプタがないことを示し、select()が正の値を返す場合は、ディスクリプタセットの中でアクセスの準備ができているファイルディスクリプタの数を示している。このとき、3つのディスクリプタセットのうち、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がレディディスクリプタとなる。

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。

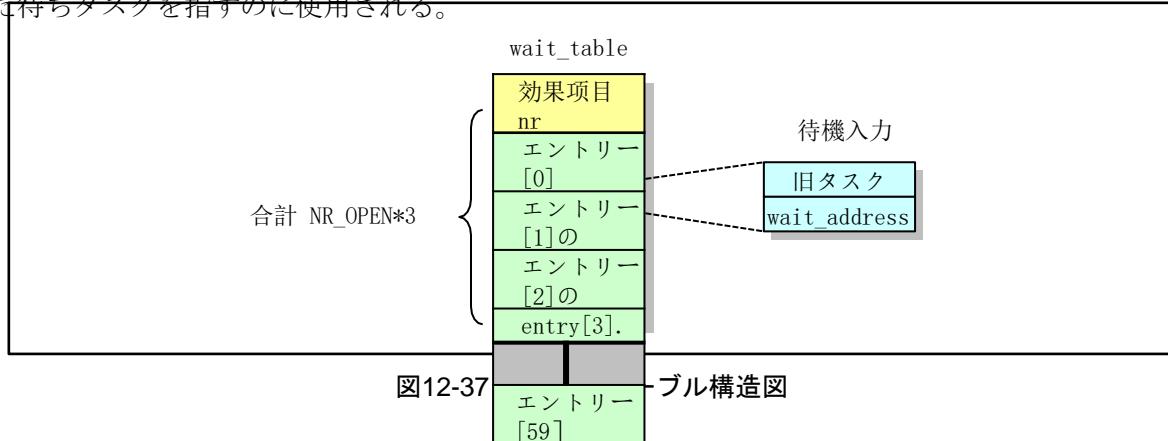
```
1. #define LIBRARY
2. #include <sys/time.h>
3. #include <unistd.h>
4
5 int select(int nd, fd_set * in, fd_set * out, fd_set * ex, struct timeval * tv)
6 {   ロングレステで
7
8     register long foobx asm ("bx") = (long) &nd;
9     // ます戻り値の変数resを定義しレジスタ変数foobxを定義する
10    // volatile("int $0x80")asm("bx");
11    // を最初のパラメータへのポインタとして使用します。その後、システムコールのインラインアセン
12    // ブリコードを使用してセット"r"(foobx);
13    // 最後に、シフトするレジスタの開数番号を選択し、そりでない場合は
14    // グローバルなエラー番号の変数 errno が設定され、-1 が返される if ( res
15    >= 0)
16        return (int) res;
17    errno = - res;
18    1を返す。
```

16 }

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、select()操作の主な作業はdo_select()関数で行われます。do_select()は、まずファイル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、その後、関数該当するディスクリプターセットの check_XX() を実行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内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完了していれば、この関数は直ちに戻り、そうでなければ、プロセスはadd_wait()関数を起動して、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。



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るないとき、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。既に存在している場合は、待機テーブルには追加されず、直接返されます（つまり、異なる待機キューに1つの待機アイテムだけが挿入されます）、そうでない場合は、待機テーブルアイテムのwait_addressフィールドが待機キューのヘッドポインタを指し、old_taskフィールドが

を、キューのヘッドポインタが元々指していたタスクに変更します。次に、待ち行列の先頭ポインタが現在のタスクを指すようにします。最後に、waitテーブルの有効なアイテムカウント値nrを1つインクリメントします。

例えば、リードバッファキューが空で、端末ttyの文字入力を待っているディスクリプタに対して、対応する端末のリードバッファキュー「セカンダリ」には、バッファキュー内の読み取り可能な文字を待つタスク待ち行列ヘッダポインタproc_listが与えられている（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いかった場合、old_task=NULLとなります。そして、proc_listに現在のタスクを指させる。この処理を図12-38に示します。図中、(a)は元々のタスクがadd_wait()関数を呼び出す前にキューの先頭ポインタを待っている状態、(b)はadd_wait()を実行した後にエントリを待っている形を示しています。なお、図ではwaitテーブルのwait_entryエントリが1つだけ表示されています。

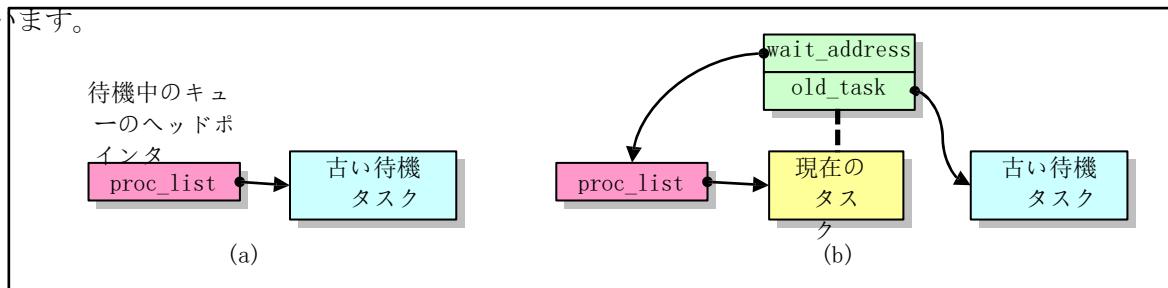


図12-38 ウェイトテーブルに1つのウェイトアイテムを追加する

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。

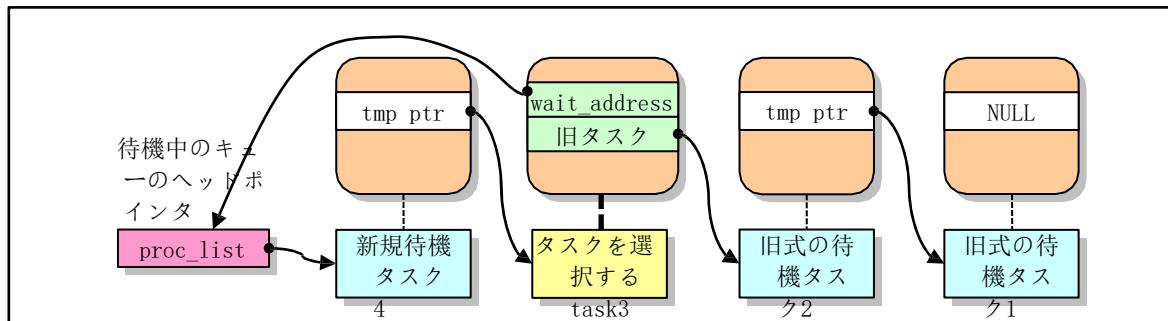


図12-39 待機中の新しいタスクが待機キューに挿入される

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。

待機中のリソースが利用可能になると、例えば、ttyのリードバッファキューの二次バッファに文字が入力された場合、待機中のヘッドポインタが指すタスクが起こされます。のです。

タスクは、そのtmpポインタが指すタスクをウェイクアップします。select()を実行しているタスクが起こされると、すぐにfree_wait()関数を実行します（コードの204行目参照）。タスクがウェイクアップし、待ち行列の先頭ポインタがこのタスクを指していれば (*wait_address == current) 、 free_wait() 関数は、old_taskが指している後続のタスクを直ちにウェイクアップさせます。free_wait()の機能は、sleep_on()がタスクを起こすコードと全く同じであることがわかります。select関数を実行しているタスクが覚醒し、他のプロセスがsleep_on()関数を呼び出して待ち行列にスリープした場合、待ち行列の先頭ポインタが現在のプロセスを指していない(*wait_address != current)ので、まずこれらのタスクを覚醒させる必要があります。操作方法は、待ち行列の先頭が指しているタスクをレディ状態にする（state

= 0）に設定し、自身を非中断可能な待機状態にします。つまり、タスク自身は、これらの後続のキューイングされたタスクが実行を開始するのを待ってから、自分自身を目覚めさせるのです。そして、スケジューラーを再実行します。

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）するため、select()が他の多くのOSの実装ではこのような処理が行われていないため、select()実行時にタイムアウト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します。同様に、ループ内でタイムアウトを初期化せず、select()関数を複数回使用するプログラムを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 コードアノテーション

プログラム 12-18 linux/fs/select.c

```

1 /*
2 * このファイルには、セレクトの取り扱いに関する手順が記載されています。
3 *
4 * Mathius Lattner氏のminixをベースにLinux用に作成されています。
5 * Peter MacDonaldによるパッチ。Linusによる大規模な編集。
6 */
7

// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する (file,
buffer_head,
// m_inode など) を使用しています。
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネル初期化用子引機能の記述子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
// <linux/tty.h> tty.h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。
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義され
// ています。
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// セグメントレジスタの操作。
// <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。
// <sys/stat.h> ファイル状態のヘッダーファイルです。ファイルやファイルシステムの状態を表す構造
// 体を含む stat{}。
// そして、定数です。
// <sys/types.h> 型のヘッダーファイル。基本的なシステムデータ型が定義されています。
// <string.h> 文字列のヘッダーファイルです。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ーファイルです。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 timeval構造体が定義されています。
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナル
// の定義
// 操作関数のプロトタイプ。

```

```

8 #include <linux/fs.h>
9 #include <linux/kernel.h> (日本語)
10 #include <linux/tty.h>
11 #include <linux/sched.h>
12
13 #include <asm/segment.h>
14 #include <asm/system.h>
15
16 #include <sys/stat.h>
17 #include <sys/types.h>.
18 #include <string.h>
19 #include <const.h>
20 #include <errno.h>
21 #include <sys/time.h>
22 #include <signal.h>
23
24 /*
25 * Ok, Peterは複雑だけどわかりやすいmultiple_wait()関数を作りました。
26 * いくつかのショートカットを使って、これを書き直しました。このコードは簡単では
27 * ないかもしれません。
28 * に従いますが、人種的な条件がなく、実用的であることが望ましいです。もし、あなたが
29 * 私がここでやっていることを理解していただければ、linuxのsleep/wakeupがどのように
30 * 機能しているかを理解していただけると思います。
31 * メカニズムが働きます。
32 *
33 * add_wait()とfree_wait()という2つの非常にシンプルな手続きがすべての作業を行います。我々は
34 * セレクト中は割り込みを無効にしなければなりませんが、それは実際には
35 * このような損失は、スリープ状態では自動的に割り込みが解除されるため、このような
36 * 状況ではありません。
37 * タスクです
38 struct task_struct * old_task;
39 */
40 // 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。
41 // wait_entry;
42
43 typedef struct {
44     int nr;
45     wait_entry エントリー
46 } select[NR_OPEN*3] です。
47
48 /**
49 // ウエ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。
50 // 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
51 // *wait_address は、ディスクリプターに関連付けられたウェイトキューのヘッダーポインタです。
52 // 例えば、以下のようになります。
53 // ttyのリードバッファキューの待ち行列の先頭ポインタはproc_listです。パラメータpは
54 // do_select()で定義された wait テーブル構造体へのポインタ。
55 static void add_wait(struct task_struct ** wait_address, select_table * p)
56 {
57     // 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
58     // ます。その後、イン
59     // ウエイトテーブルでは、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
60     // 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
61     // パイプのファイルディスクリプターのための//。例えば、パイプが読み込みを待っている場合、その
62     // パイプには

```

```

//すぐに書けるように
51     if (!wait_address)
52         を返すことができます。
53     for (i = 0 ; i < p->nr ; i++)
54         if (p->entry[i].wait_address == wait_address)
55             return;
// そして、ディスクリプターの待ち行列のヘッダポインタをwait_tableに格納して
// wait table entry の old_task フィールドは、wait queue header で指定されたタスクを指します。
// ポインタ（ない場合はNULL）を指定して、待ち行列のヘッダに現在のタスクを指定させます。最後
56 に
57     p->entry[p->nr].wait_address = wait_address // ウェイトテ
58     ーブルのアイテムカウント値 nr を 1 つずつ増やしま
59     す。
60 }     p->entry[p->nr].old_task = * wait_address;
61     *wait_address = current;
//// ウエイトテーブルの各ウェイトキューを解放します。
// パラメータ p は wait table 構造体へのポインタです。この関数（204, 207行目）は
// do_select()関数で現在のプロセスが起こされたときに呼び出されて
// 各待ち行列にある他のタスクを待ち行列にアップします。とほとんど同じです。
// kernel/sched.cにあるsleep_on()関数の後半部分、その説明を参照。
62 static void free_wait(select_table * p)
63 {...}
64     int i;
65     struct task_struct ** tpp;
66
// ウェイトテーブルのエントリーのウェイトキューのヘッダー（トータルnrエントリー）が示すのは
// 後から追加された他の待機タスクがある場合（例えば、他のプロセスがsleep_on()
// 関数を使用して待機キューをスリープさせた場合）は、キューのヘッダーが現在の
// の処理を行うためには、まずこれらのタスクを起こす必要があります。操作方法は、タスクに
// キューへッダが指す // をレディ状態 (state = 0) にし、自分自身を
// 割り込み禁止の待機状態、つまり、これらの後続のキューイングされたタスクを待って
// 目覚めたので、このタスクを起こすために再度実行します。そこで、スケジューラー
67     を再実行します。 for (i = 0; i < p->nr ; i++) {
68         tpp = p->entry[i].wait_address;
69         while (*tpp && *tpp != current)
70             {...}
71             (*tpp)-> state = 0;
72             current-> state =
73             TASK_UNINTERRUPTIBLE; schedule();
74     }
// ここで実行は、現在のエントリーのウェイトキューのヘッダーフィールドwait_addressを示す
待機テーブルの // は、現在のタスクを指します。これがNULLの場合は、現在のタスクが
// スケジュールコードに問題があり、警告メッセージが表示されています。その後、待ち時間を
// キューのヘッダは、先にキューに入ったタスクへのポインタです（76行目）。もし、ヘッドポイン
タが
// この時点でNULLではなくタスクを指していれば、キューにはまだタスクが存在していることになり
74 ます。
75 // (*tppは空ではない)なので、タスクはレディ状態に設定され、ウェイクアップします。最後に、タ
76 スクに
77 // ウェイトリストのアイテムカウントフィールドnrを0にして、ウェイトテーブルを
78     クリアする。 if (!*tpp)
79         printk("free_wait: NULL");
        if (*tpp = p->entry[i].old_task)
            (**tpp). state = 0;
    }
p->nr = 0となります。

```

```

80 }
81
82 ///////////////////////////////////////////////////////////////////
83 // i-node に応じた tty を取得します。
84 // ファイルがi-nodeに従った文字端末デバイスファイルであるかどうかを確認する。もしそうであれば
85 // tty構造体のポインタを返し、そうでなければNULLを返します。
86 static struct tty_struct * get_tty(struct m_inode * inode)
87 {
88     int メジャー、マイナー;
89
90     // 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
91     // 5（コントロール端子）または4ではない。
92     if (! S_ISCHR(inode-
93         >i_mode)) return
94         NULL;
95
96     if ((major = MAJOR(inode->i_zone[0])) != 5 && major != 4)
97         return NULL;
98
99     // メジャーデバイス番号が5の場合、プロセスのttyフィールドがその端末デバイスとなる
100    // 番号、そうでない場合はキャラクターデバイス番号のマイナーデバイス番号と同じになります。
101    // 端末のデバイス番号が0より小さい場合は、プロセスが制御されていないことを示す
102    // 端末を使用していないのでNULLを返し、そうでない場合は
103    // 対応するtty構造のポインタ if (major
104        == 5)
105        minor = current->tty;
106
107    // その他
108    minor = MINOR(inode->i_zone[0]);
109
110    if (minor < 0)
111        return NULL;
112
113    return TTY_TABLE(minor);
114
115 /*
116 * check_XX関数は、ファイルをチェックアウトします。それは、次のいずれかであることがわかっています。
117 * パイプ、キャラクターデバイス、FIFO (FIFOは実装されていません。)
118 */
119
120 ///////////////////////////////////////////////////////////////////
121 // リードインレディを確認する。
122 // ファイル読み取り操作の準備ができているかどうか、つまり、端末の読み取りバッファのキューが
123 // secondary に読み込むべき文字があるかどうか、パイプファイルが空でないかどうか。
124 // 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。
125 // 103 static int check_in(select_table * wait, struct m_inode *
126 inode) ...続きを読む
127
128 struct tty_struct * tty; 106
129
130
131 // 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
132 // 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。
133 // 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
134 // 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
135 // 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、入っていないければ（パイプが空）。
136 // 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
137 // 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
138 // パイプが空の場合パイプのi-nodeのi_zone[0]とi_zone[1]フィールドには、現在の
139 // パイプの先頭と末尾のポインター if
140     (tty = get_tty(inode))
141         if (! EMPTY(tty->secondary))§62

```

```

109          を返しま
110      その    す。
111      他      add_wait(&tty->secondary->proc_list, wait);
112      else if (inode->i_pipe)
113          if (!
114              PIPE_EMPTY(*inod
115              e)) return 1;
116          その他
117              add_wait(&inode->i_wait, wait);
118 }      0を返す。
119

120 ///////////////////////////////////////////////////////////////////
121 // チェックライトアウトの準備ができました。
122 // ファイルの書き込み操作の準備ができているかどうか、つまり、空きのある場所があるかどうかを
123 // 確認する
124 // 端末の書き込みバッファキューライフタイムwrite_qに入っているか、パイプファイルがいっぱいになっていない
125 // いか。
126 // 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。
127 static int check_out(select_table * wait, struct m_inode *
128 inode) 121 {...}
129
130 struct tty_struct * tty; 123
131
132 // 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
133 // ます。
134 // (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
135 // ェック write_q
136 // 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
137 // 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
138 // パイプラインに文字を書き込むための空き領域がある。ある場合は1を返し、ない場合は
139 // そうでない場合(パイプラインが満杯の場合)、現在のタスクはパイプラインの待ち行列に追加さ
140 // れる
141 // i-nodeで0を返します。
142     if (tty = get_tty(inode))
143         if (! FULL(tty-
144             >write_q))
145             return 1;
146
147     その他
148         add_wait(&tty->write_q->proc_list, wait);
149
150     else if (inode->i_pipe)
151         if (!
152             PIPE_FULL(*inod
153             e)) return 1;
154
155 ///////////////////////////////////////////////////////////////////
156 // 異常な状態を確認する。
157 // ファイルが異常な状態であるかと、それが確認しますwait)。ミナルデバイスファイルの場合、カーネ
158 // ルは常に0を返す。
159 // パイプファイルの場合、2つのパイプディスクリプターのうち、どちらか一方が閉じられていれば1
160 // を返します。
161 // 今回は、そうでなければ、現在のタスクをi-nodeの待ち行列に追加し、0を返します。
162 // 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。
163 // 例外条件が発生した場合は1を、そうでない場合は0を返します。
164 static int check_ex(select_table * wait, struct m_inode * inode)
165 {...}
166
167     if (tty = get_tty(inode))
168         if (! FULL(tty-
169             >write_q))
170             0を返す。

```

```

144         その
145             他      return 0;
146         else if (inode->i_pipe)
147             if (inode->i_count < 2)
148                 を返します。
149             その他
150                 add_wait(&inode->i_wait, wait)
151             です。
152 }       0を返す。
153

     //// select()の内部機能です。
     // do_select()は、カーネルがselect()システムコールを実行するための実際のハンドラです。これは
     // この関数は、まずディスクリプターセット内の各ディスクリプターの有効性をチェックし、次に
     // 関数check_XX()で各ディスクリプターセットのディスクリプターをチェックし、その数をカウント
     // します。
     // ディスクリプタセットで現在準備ができているディスクリプタのいずれかのディスクリプターが準
     // 備できている場合は
     // この関数はすぐに戻りますが、そうでない場合はプロセスがスリープ状態になり、プロセスの
155 // は、タイムアライプが切れるか、待ち行列のプロセスが終了しても実行し続けます。
156 記述子があるところの//を覚醒させる。
157 int do_select(fd_set in, fd_set out, fd_set ex,      // レディディスクリプターの数
158               select_table wait_table;
159               int i;
160               fd_setマスク。
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が
     // 記述子セットをマスクで取得します。その後、現在のプロセスをループして
     // 各ディスクリプターが有効で、ディスクリプターセットに含まれているかどうか。ループ内では、
     // 各
     // ディスクリプターが判定されると、マスクが1ビット右にシフトします。したがって、それに基づ
     // いて
     // マスクの最下位ビットで、対応するディスクリプターの有無を判断します。
     // は、ユーザーが与えたディスクリプターセットにあります。有効な記述子は、パイプファイル記述
162 子でなければならない。
163 // キャラクタデバイスのファイル記述子またはFIFO記述子のいずれかであり、残りのタイプは
164 // 無効な記述子と上でEBADFを返す mask = in | // ディスクリプタセットにない
165     out | ex;          を続けています。
166     for (i = 0; (!current->filp[i] && mask >>= 1) // ファイルが開かれ
167         return -EBADF;    ていません。
168     if (! current->filp[i] - // ファイルのi-nodeは
169         >f_inode) return - nullです。
170     if (current->filp[i]->f_inode->i_pipe) // 有効: パイプファイ
171         ル. Continue;
172     if (S_ISCHR(current->filp[i]->f_inode->i_mode)) // 有効: char devファイル。
173         を続けています。
174     if (S_ISFIFO(current->filp[i]->f_inode->i_mode)) // 有効: FIFOファイルで
175         す。
176         を続けています。
177     return -EBADF; // 残りはすべて無効です。
178 // 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。この時点では「マスク」は、現在ディスクリプターのマスクとして使用されて
     いる
     // 処理されています。ループ内の3つの関数check_in()・check_out()・check_ex()

```

```

// は、ディスクリプターがリードイン、ライトアウトの準備ができているかどうかを判断するために
使用されます。
//の状態になります。ディスクリプタが動作可能な状態であれば、該当するビットが設定されます。
//ディスクリプターがセットされ、レディディスクリプターのカウント数がインクリメントされる
// 1です。183行目の「mask += mask」という記述は、「mask << 1」と同じです。
178回繰り返す wait_table.nr = 0;
180     *inp = *outp = *exp = 0;
181     count = 0;
182     mask = 1となります。
183     for (i = 0 ; i < NR_OPEN ; i++, mask += mask) {...}
// この時点でチェックしたディスクリプターが、読み取り操作ディスクリプターセットの中にあり、
かつ
// ディスクリプタが読み取り操作の準備ができている場合、ディスクリプタセットの対応するビット
184 が
185 // 1に設定され、レディディスクリプターのカウントが1つ増加する if (mask
186     & in)           *inp |= mask;           // セットになっているビッ
187             if (check_in(&wait_table, current->filp[i]->f_inode)) {...}
188             count++です。                   // 用意された番号。
189 // この時点でチェックしたディスクリプターが、書き込み操作用ディスクリプターセットに入って
いて
// ディスクリプタが書き込み操作の準備ができている場合、ディスクリプタセットの対応するビット
190 が
191 // 1に設定され、レディディスクリプターの数が1つ増える if (mask & out)
192         if (check_out(&wait_table, current->filp[i]->f_inode)) {...}
193             *outp |= mask;
194             count++;
195             }
196             // この時点でチェックしたディスクリプターが異常ディスクリプターセットの中にあり、かつ、デ
197             ィスクリプター
198             // が異常な状態である場合には、EXディスクリプターセットの対応するビットを1にして
199             // 異常ディスクリプターの数を1つ増やす if (mask & ex)
200             if (check_ex(&wait_table, current->filp[i]->f_inode)) {...}
201             count++です。
202             }
203             // 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
204             // (count==0)で、プロセスがノンブロッキング・シグナルを受信しておらず、かつ、待機中の
205             // この時点でディスクリプターを設定するか、待ち時間が経過していない場合は、現在のプロセスを
206             // の状態を割り込み可能なスリープ状態にして、スケジューラーを実行して他のタスクを実行します
207             。
208             // カーネルがこのタスクを再びスケジューリングする際には、free_wait()を呼び出してタスクを目
覚めさせます。
209             // 関連する待ち行列のタスクの前後に、リピートラベルにジャンプします（行
210             // 178）、気になるディスクリプターがあるかどうかを再検出する。 if
211             (!(current->signal & ~current->blocked) &&
212             (wait_table.nr || current->timeout) && ! count) {
213                 current-> state =
214                     TASK_INTERRUPTIBLE; schedule();
215                 free_wait(&wait_table); // タスクが起こされ、ここに戻される goto repeat;
216             }
217             // この時点でcountが0になっていない場合、あるいはシグナルを受信した場合、あるいは待機時間
218             // が
219             // が立ち上がり、待つべきディスクリプターがない場合、free_wait()を呼び出して
220             // free_wait(&wait_table);

```

```

208     countを返す。
209 }
210
211 /*
212 * 入力を変更すると、-ERESTARTSYSを返せないことに注意してください。
213 * パラメーター。悲しいですが、そういうことです。で調整することができました。
214 * ライブライ機能...
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。
// によって与えられたバッファポインタに従って、ユーザデータ空間からカーネル空間に // を転送し
// ます。
// パラメータを設定し、待機時間を設定した後、do_select()を呼び出します。を返した後。
// 結果はユーザースペースにコピーバックされます。
// パラメータバッファは、ユーザ領域にあるselect()関数の最初のパラメータを指します。
// 返り値が0より小さい場合は、実行中にエラーが発生し、返り値が
// が0の場合、指定された待機時間内に操作可能なディスクリプターがないことを意味します。
// 時間。戻り値が0より大きい場合は、準備の整ったディスクリプターの数を示す。
216 int sys_select( unsigned long *buffer )
217 {
218 /* select(nd, in, out, ex, tv)システムコールを実行します。*/
// 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。
219 // ポインタの引数で渡される引数。
220     fd_set res_in, in = 0, *inp;           // fdセットの読み込み。
221     fd_set res_out, out = 0, *outp;        // fdセットを書き出す。
222     fd_set res_ex, ex = 0, *exp;          // 異常な fd が設定されています。
223     fd_set のマスクを使用しています。    // 記述子の値の範囲 (nd) のマスクコードで
// 個体 timeval *timeout                // 待ち時間構造体のポインタ。
224
225 // 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。
// 变数と、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变数です。の最大ディスクリプタ値+1に基づいて、3つの
// 記述子のセット（つまり、最初のパラメータndの値）は、すべてのマスクに設定されます。
// ユーザープログラムが关心を持つ記述子。例えば、nd = 4の場合、mask = 0b00001111となります
226
227 // (合計32ビット)。
228     mask = ~((~0) << get_fs_long(buffer++));
229     inp = (fd_set *) get_fs_long(buffer++);
230     outp = (fd_set *) get_fs_long(buffer++);
231     exp = (fd_set *) get_fs_long(buffer++);
232     if (inp) { // struct timevalを読み込む。
233         in = mask & get_fs_long(inp);
234     }
235     if (outp) { // 書き出しセット
236         out = mask & get_fs_long(outp);
237     }
238     if (exp) { // 異常なセット
239         ex = mask & get_fs_long(exp);
240     }
241
242 // 次に、待機（スリープ）時間の値「timeout」をtime構造体から取り出します。まずは
// 「タイムアウト」は最大（無限）の値に初期化され、「タイムアウト」の時間値を

```

```

// 時間構造がユーザー空間から取得され、現在のティック値のジフ?
// システムが変換されて追加され、最後に時間を刻む値「タイムアウト」が
// が待つ必要があることを取得します。この値を使って、現在のプロセスが必要とする遅延時間を設
定します。
// を待ちます。また、241行目のtv_usecフィールドはマイクロ秒の値です。これを次の式で割ると
// 1000000は対応する秒数を与え、それにシステムのティックを掛けています。
239 tv_usecをjiffiesの値に変換する、 // per second HZ。
240     if (tvp) {
241         timeout = get_fs_long((unsigned long *)&tvp->tv_usec)/(1000000/HZ) です。
242         timeout += get_fs_long((unsigned long *)&tvp->tv_sec)* HZ です。
243         timeout += jiffies;
244     }
245     current->timeout = timeout; // 現在のプロセスが必要とする遅延時間を設定します。
// select()の主な作業は、関数do_select()で行われます。
// 関数は、処理結果をユーザーデータ領域にコピーし、それを
// ユーザーになります。競合状態を避けるためには、do_select()を呼び出す前に割り込みを無効に
する必要があります。
// とし、関数が戻った後に有効にします。
// do_select()が戻った後、プロセスの待ち時間フィールドのタイムアウトがまだ大きい場合
現在のシステムのタイミングの目盛り値よりも // ジフティに、ディスクリプターの準備ができてい
ることを示します。
// タイムアウトの前に。そこでここでは、タイムアウトまでの残り時間値を記録して
// であれば、この値をユーザーに返します。プロセスの待機遅延フィールドのタイムアウトが
246 // すでに現在のシステムを無効にするよりも小さいか等しい場合は、do_select() が
247 // タイムアウトを返すれたのと、残り時間の値がゼロな 場合は;
248     if (current->timeout > jiffies)
249         timeout = current->timeout - jiffies;
250     その他
251     timeout = 0;
252     sti(); // intを有効にする。
// 次にプロセスのタイムアウトフィールドをクリアします。返されたレディディスクリプターの数が
do_select()による // が0より小さい場合は、実行エラーを意味するので、エラーフラグを返す。
// その後、処理されたディスクリプターセットの内容と、遅延時間構造の内容を書き戻す
// をユーザーのバッファスペースに配置します。また、時間構造のコンテンツを書き込む際には
// tickの時間単位で表される残りの遅延時間を秒とマイクロ秒に変換する
// 値を表示します。
253     current->timeout = 0;
254     if (i < 0)
255         return i;
256     if (inp) {
257         verify_area(inp, 4);
258         put_fs_long(res_in, inp); // 読み込み可能なディスクリプター
259     }    セット。
260     if (outp) {
261         verify_area(outp, 4);
262         put_fs_long(res_out, outp); // 書き込み可能なディスクリ
263     }    プターセット。
264     if (exp) {
265         verify_area(exp, 4);
266         put_fs_long(res_ex, exp); // 異常な記述子のセット。
267     }
268     if (tvp) {
269         verify_area(tvp, sizeof(*tvp));

```

```

270         put_fs_long(timeout/HZ, (unsigned long *) &tvp->tv_sec); // 秒数。
271         timeout %= HZ;
272         timeout *= (1000000/HZ) となります。
273         put_fs_long(timeout, (unsigned long *) &tvp->tv_usec); // マイクロ秒。
274     }
275     // この時点では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
276     // 中断されたエラー番号が返される。それ以外の場合は、レディディスクリプターの数
277     // を返します。
278     if (!i && (current->signal & ~current->blocked))
279     return -EINTR;
280 } i. を返します。
281

```

12.20 まとめ

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、カーネル内他のプログラムがバッファ・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ーフェースの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機構とDemand Loading機構の原理と実装についても説明します。