




6 ブーティングシステム

本章では、リスト6-

1に示すように、`boot/`ディレクトリ内の3つのアセンブリ言語ファイルについて説明します。前章で述べたように、これら3つのファイルはすべてアセンブリ言語プログラムですが、2つの異なる構文形式を使用しています。`bootsect.S`と`setup.S`はインテルのアセンブリ言語構文を使用するリアルモードの16ビットコードプログラムで、8086アセンブリコンパイラとリンカ`as86`と`ld86`が必要です。しかし、`head.s`はAT&Tのアセンブリ構文フォーマットを使用し、プロテクトモードで実行されるため、GNUの`as(gas)`アセンブラでコンパイルする必要があります。

当時、リーナス・トーバルズが2つのアセンブラを使っていた主な理由は、インテルx86プロセッサの場合、1991年のGNUアセンブラはi386以降の32ビットCPUコード命令しかサポートしていなかったからです。リアルモードで動作する16ビットコードのプログラムを生成することはサポートされていません。1994年までは、GNU `as`アセンブラは16ビットコードをコンパイルするための`.code16`命令をサポートするようになっていました(GNUアセンブラのマニュアルの「80386関連機能」の「16ビットコードの記述」を参照してください。)。カーネル 2.4.X 以降、`bootsect.S` と `setup.S` のプログラムは、統一的に GNU `as` を使って書かれるようになりました。

リスト 6-1 linux/boot/ ディレクトリ

ファイル名				サイズ	最終更新時刻 (GMT)	説明
	bootsect.S	7574	バイト		1992-01-14 15:45:22	
	head.s	5938	バイト		1992-01-11 04:50:17	
	setup.S	12144	バイト		1992-01-11 18:10:18	

これらのコードを読むには、8086のアセンブリ言語の知識に加えて、Intel 80X86マイクロプロセッサを搭載したいくつかのPCアーキテクチャと、32ビットプロテクトモードでのプログラミングの基本原則を理解している必要があります。ですから、ソースコードを読み始める前に、前の章の基本的な理解をしておく必要があります。コードを読む際には、具体的に遭遇した問題について詳しく説明します。

6.1 主な機能

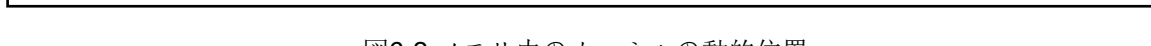
まず、Linux OSの起動部分の主な実行プロセスを説明します。PCの電源を入れると、80x86のCPUは自動的に実働モードに入ります。プログラムコードは、アドレス `0xFFFF0` から自動的に実行されます。このアドレスは、通常、ROM-BIOS内のアドレスです。PCのBIOSは、システムのハードウェア検出と診断動作を行い、物理アドレス0で割り込みベクターの初期化を開始します。その後、起動デバイスの第1セクタ（ディスクブートセクタ、512バイト）をメモリの絶対アドレス`0x7C00`に

読み込み、この場所にジャンプして起動処理を開始します。ブートデバイスは通常、フロッピードライブやハードディスクです。ここでの説明は簡単ですが、カーネルの初期化作業の始まりを理解するには十分です。

Linuxの最初の部分は8086アセンブリ言語（boot/bootsect.S）で書かれており、ブートデバイスの第1セクタに格納されています。BIOSによってメモリの絶対アドレス0x7C00(31KB)にロードされます。実行されると、メモリの絶対アドレス0x90000（576KB）に自身を移動し、2KBのバイトコードを読み出す



2は、Linuxシステムの起動時に、これらのプログラムやモジュールのメモリ上の動的な位置を明示したものである。図中の縦棒は、ある瞬間のメモリ上の各プログラムのイメージロケーションマップを表している。システムのロード中は「Loading...」というメッセージが表示されます。その後、同じくリアルモードのアセンブリ言語プログラムであるboot/setup.Sのコードに制御が移ります。



239

ブート部分では、ホストの特定の機能やVGAカードの種類を識別し、必要に応じて、コンソールの表示モードを選択するように指示します。その後、システム全体をアドレス0x10000から0x0000に移動し、プロテクトモードに入り、システムの残りの部分（0x0000の部分）にジャンプします。この時点で、すべての32ビットモードのブート設定が完了しました。IDT、GDT、LDTがロードされ、プロセッサとコプロセッサも確認され、ページングも設定されています。最後に、init/main.cのmain()コードが呼び出されます。上記の動作のソースコードはboot/head.sにありますが、これはおそらくカーネル全体の中で最も厄介なコードです。なお、これまでのステップで何か問題が発生した場合、コンピュータはデッドロックします。OSが完全に動作する前に処理することはできません。

なぜブートセクタはシステムモジュールを物理アドレス0x0000の先頭に直接ロードせず、セットアッププログラムで再度移動させるのか、という疑問があるかもしれません。これは、後続のセットアップコードが、マシンの構成に関するいくつかのパラメータ（ディスプレイカードのモード、ハードディスクのパラメータテーブルなど）を取得するために、ROM BIOSが提供する割り込み呼び出し機能も使用する必要があるからです。しかし、BIOSの初期化時には、サイズ0x400バイト（1KB）の割り込みベクターテーブルが物理メモリの先頭に配置されています。この位置にシステムモジュールを直接配置すると、BIOSの割り込みベクターテーブルが上書きされてしまいます。そのため、ブートローダは、BIOS割り込みコールを使用した後、システムモジュールをこの領域に移動させる必要があります。

また、上記のカーネルモジュールをメモリ上にのみロードするだけでは、Linuxシステムを動作させることはできません。完全に動作するLinuxシステムとして、基本的なファイルシステムのサポートであるルートファイルシステムも必要となります。Linuxの

0.12のカーネルは、MINIX 1.0のファイルシステムのみをサポートしています。ルートファイルシステムは通常、別のフロッピーディスクやハードディスクのパーティションに存在します。ルートファイルシステムがどこに保存されているかをカーネルに知らせるために、bootsect.Sプログラムの44行目にルートファイルシステムがあるデフォルトのブロックデバイス番号ROOT_DEVが与えられています。ブロックデバイス番号の意味については、プログラム中のコメントを参照してください。ブートセクターの509,510(0x1fc--0x1fd)バイトに指定されたデバイス番号は、カーネルの初期化時に使用されます。スワップデバイス番号SWAP_DEVはbootsect.Sの45行目に与えられており、仮想記憶装置のスワップスペースとして使われる外部デバイス番号を示しています。

6.2 bootsect.S

6.2.1 機能説明

bootsect.Sコードは、ディスクの第1セクター（ブートセクター、0トラック（シリンダー）、0ヘッド、第1セクター）に存在するディスクブートブロックプログラムです。PCの電源を入れ、ROM BIOSがセルフテストを行うと、ROM BIOSはブートセクタコードbootsectをメモリアドレス0x7C00にロードして実行します。bootsectコードの実行中に、自分自身をメモリのアブソリュートアドレス0x90000の先頭に移動させ、実行を続けます。このプログラムの主な機能は、まず、ディスクの第2セクタから始まる4セクタのセットアップモジュール（setup.sからコンパイル）をbootsect（0x90200）の直後のメモリにロードします。次に、BIOS割り込み0x13を使用して、現在の起動ディスクのパラ

メータをディスクパラメータテーブルに取り込み、「Loading system...」という文字列を画面に表示します。そして、ディスク上のセットアップモジュールの後ろにあるシステムモジュールを、メモリ 0x10000の先頭にロードします。その後、ルートファイルシステムのデバイス番号を決定する。指定されていない場合は、起動ディスクの1トラックあたりのセクタ数の保存状況により、ディスクの種類(1.44MはAディスクか?)を判別し、デバイス番号をroot_dev(起動ブロックの508番の位置)に格納します。最後にセットアッププログラムの先頭(0x90200)にロングジャンプしてセットアップを実行します。ディスク上では、ブートブロック、セットアップモジュール、システムモジュールの位置とサイズを図6-3に示します。

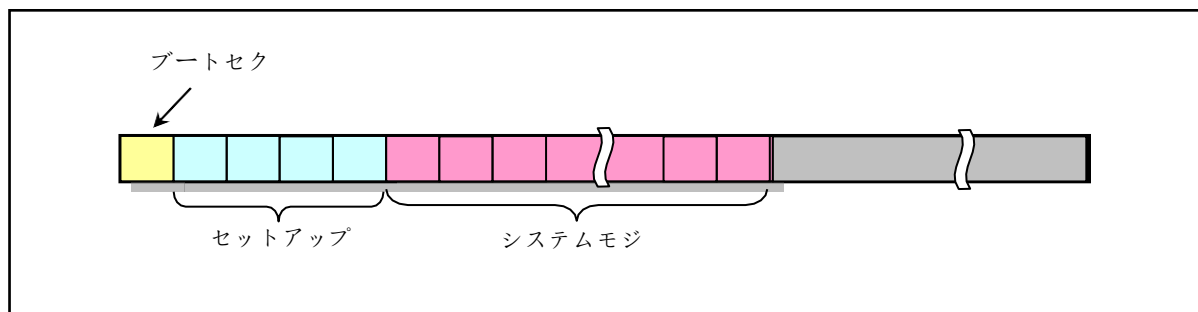


図 6-3 1.44MB のディスクに収められた Linux 0.12 カーネルの配布状況

図は、1.44MBのディスク上のLinux

0.12カーネルが占有するセクタの分布を示しています。1.44MBディスクのプラッタの片面には80のトラック（シリンダー）があり、各トラックには18のセクタがあり、合計2880のセクタがあります。最初のセクタにはブートプログラムコード、次の4セクタにはセットアップモジュール、そして次の260セクタには0.12カーネルのシステムモジュールが配置されています。まだ2610セクタ以上の未使用領域が残っています。これらの残りの未使用スペースには、基本的なルートファイルシステムを格納することができ、1枚のディスクを使用してシステムを実行できる統合ディスクを作成することができます。これについては、ブロックデバイスドライバの章で詳しく説明します。

また、このプログラムのファイル名は、他のガスアセンブル言語プログラムとは異なります。その接尾辞は、大文字の「.S」です。このような接尾辞を使うと、gasがGNUコンパイラの前処理機能を使えるようになり、アセンブリ言語プログラムに「#include」や「#if」などの文を入れることができますようになります。このプログラムでは、主に大文字のサフィックスを使用して、プログラム内の「#include」文で、linux/config.hヘッダーファイルで定義されている定数をインクルードしています。プログラムの6行目をご覧ください。

なお、ソースファイル中の行番号の付いた文章や記述はオリジナルであり、行番号のない記述は作者のコメントであることにも注意してください。

6.2.2 コードコメント

プログラム 6-1 linux/boot/bootsect.S

```
!ここでは、行頭の「!」や「;」がコメント文を表しています。1!
2 !SYS_SIZEは、ロードされるクリック数（16バイト）です。
3 !0x3000は0x30000バイト=196kBで、現在の4つのバージョンのLINUXに
    は十分すぎるほどです。
    !SYS_SIZEは、ロードするシステムモジュールのサイズです。サイズの単位はパラグラフです。
    !X86のメモリセグメンテーションの範囲で1段落16バイト。0x3000は0x30000バイト=196KB。
    !1KB=1024バイトとすると、192KBとなります。現在のカーネルにはこの容量で十分です
    のバージョンです。これが0x8000の場合、カーネルは最大でも512KBであることを意味します。の場合は
    ブートセクタとセットアップコードのメモリは0x90000に格納されていますが、その値は0x90000を超えてはなりま
    せん。
    !0x9000（584KBを示す）。5!
    !linux/config.hファイルでは、カーネルが使用するいくつかの定数シンボルと、デフォルトの
    ! Linus氏自身が使用しているハードディスクのパラメータブロックです。例えば、次のような定数があります。
    !”と定義されています。
    !DEF_SYSSIZE = 0x3000 - デフォルトのシステムモジュールのサイズ（段落単位）です。
```

!DEF_INITSEG = 0x9000 - 移動するデフォルトの目的地の位置です。

!DEF_SETUPSEG = 0x9020 - セットアップコードのデフォルトの場所です。

! DEF_SYSSEG= 0x1000 - ディスク からのシステムモジュールのデフォルトの場所です。

```

6 #include <linux/config.h>
7 SYSSIZE = DEF_SYSSIZE
8 !
9                                     ! bootsect.s(C) 1991
Linus Torvalds 10                       ! modified by Drew
Eckhardt
11 !
12 ! bootsect.s は bios-startup ルーチンによって 0x7c00 でロードされ、13 !
  iself を 0x90000 番地に移動してそこにジャンプします。
14 !
15 !その後、自分自身の直後(0x90200)に「setup」をロードし、0x10000でBIOS割
  り込みを使ってシステム16!
17 !
18 !注! 現在のシステムの長さは最大でも8*65536バイトです。これは将来的にも問
  題ないはずで、シンプルに考えたいと思います。この512キロバイトの
20 ! カーネルサイズは十分なはずで、特にminixのような21! バッファキャッ
  シュが含まれていないので。
22 !
23 !ローダーは可能な限りシンプルに作られており、継続的な
24 ! 読み取りエラーが発生すると、抜け出せないループに陥ります。手で再起動してください。それは
25 可能な限りセクタ全体を一度に取得することで、ロードがかなり速くなります。
26
  !ディレクティブ(疑似オペレータ)の .globl または .global を使用しています。
  後続の識別子は、外部またはグローバルなものであり、たとえ
  !"は使用されません。.text、.data、.bssは、現在のコードセクションを定義するために使用されます。
  それぞれ、データセクション、初期化されていないデータセクションとなっています。
  !複数のオブジェクトをリンクする場合、ld86は対応するセクションを結合(マージ)します。
  各オブジェクトモジュールの中で、カテゴリー別に「!」がついています。ここでは、3つのセクションすべてが
  同じアドレス範囲で定義されているので、プログラムは実際には分割されていません。
  !また、文字列の後にコロンが続く場合は、「begtext:」のようにラベルになります。
27 .globl begtext, begdata, begbss, endtext, enddata, endbss
28                                     .text! テキストセクシ
  ョン。
29 begtext。
30 .データ! データセクション
31 begdata:
32                                     .bss! 初期化されていないデー
  タセクション 33 begbss:
34 .テキスト
35
  !識別子やラベルの値を定義するには、等号「=」や記号「EQU」を使用します。
36 SETUPLEN                                     = 4! セットアップ・セクター の数
  セットアップコードで占有されるセクタです。
37 BOOTSEG                                     = 0x07c0: ブートセクタの元のアドレス
38 INITSEG = DEF_INITSEG: ブートをここに移動して
  邪魔にならないようにする
                                     !0x90000に移動 - システムで使用される場所を避けます。
39 SETUPSEG = DEF_SETUPSEG! ここ からセットアップが始まります。
  セットアップコードは0x90200から始まります。
40 SYSSEG= DEF_SYSSEG! 0x10000(65536) で
  システムをロード。41
  ード を停止するか
42
43 !ROOT_DEVとSWAP_DEVが "build "で書き込まれるようになりました。

```


!root fs のデバイス番号 ROOT_DEV と swap デバイス SWAP_DEV が build で書き込まれるようになりました。
!デバイス番号0x306は、ルートfsデバイスが第1パーティションであることを示しています。
2台目のハードディスクです。リーナスは、2番目のハードディスクにLinux 0.11システムをインストールしたので

!ROOT_DEVは0x306に設定されています。このカーネルをコンパイルする際に、デバイス番号を変更することができません。

ルートfsがある端末の場所に応じて、「!」をつけてください。例えば、以下のような場合
ルートファイルシステムが1つ目のハードディスクの1つ目のパーティションにある場合、この値はこのデバイス番号は、昔ながらのドライブ

カーネル0.95までのLinuxのネーミング方法です。ハードディスク・デバイスの具体的な値
番号は以下の通りです。

!デバイスnr = メジャーデバイスnr * 256 + マイナーデバイスnr, (またはdev_no = (メジャー<<8) + マイナー)

!(Major nr: 1-メモリ、2-ディスク、3-ドライブ、4-ttyx、5-tty、6-パラレルポート、7-無名パイプ)

!0x300 - /dev/hd0 - 1台目のハードドライブ全体を表します。

!0x301 - /dev/hd1 - 1番目のディスクの1番目のパーティションです。

!...

!0x304 - /dev/hd4 - 1枚目のディスクの4番目のパーティションです。

!0x305 - /dev/hd5 - 2番目のハードドライブ全体を表します。

!0x306 - /dev/hd6 - 2番目のディスクの1番目のパーティションです。

!...

!0x309 - /dev/hd9 - 2番目のディスクの4番目のパーティションです。

!

[44](#) ROOT_DEV = 0! ルート fs デバイスは、同じブートデバイスを使用します。

[45](#) SWAP_DEV = 0! スワップデバイスは、同じブートデバイスを使用します。

!ディレクティブは、リンカが指定された識別子やラベルを

!生成された実行ファイル(a.out)です。ここでは、プログラムの実行を開始します。

!49~58行目の機能は、現在のセグメントから自分自身(ブートセクタ)を移動させることです。

!ポジション0x07c0(31KB)から0x9000(576KB)まで、合計256ワード(512バイト)を、そして

!移動したコードのラベル go、つまりプログラムの次のステートメントにジャンプします。[47](#)

エントリー スタート!リンカに、ラベル start でプログラムが始まることを伝えます。[48](#) スタート。

[49](#) movax, #BOOTSEG!dsセグメントレジスタを0x7C0 に設定します。

[50](#) movds, ax

[51](#) movax, #INITSEG!esセグメントレジスタを0x9000 に設定します。

[52](#) move, ax

[53](#) movcx, #256!設定した移動回数=256ワード(512バイト)。

[54](#) subsi, si!ソースアドレス ds:si = 0x07C0:0x0000

[55](#) subdi, di!送信先アドレス es:di = 0x9000:0x0000

[56](#) rep! cx = 0 になるまで、カウントダウン。

[57](#) movw!cxワードをメモリ[si]から[di] に移動させる。

!ジャンプインターセグメント。INITSEG - ジャンプ先のセグメント、go - セグメント内のオフセット。

[58](#) jmpigo, INITSEG [59](#)

!下から順に、0x90000の位置に移動したコードでCPUが実行されます。

!このコードは、スタックレジスタssとspを含むいくつかのセグメントレジスタを設定します。

!スタックポインタspは、512バイトよりも遠くを指していればOKです。

!オフセット(つまりアドレス0x90200)を使用します。セットアッププログラムは、このようにして

!0x90200のアドレス(4セクタのサイズ)なので、spはより大きな位置を指す必要があります。

!(0x200 + 0x200 * 4 + スタックサイズ)よりも大きくなります。ここでは、spは0x9ff00 - 12に設定されています(パラメータテーブル

サイズ)、すなわち sp = 0xfef4 となります。自作のドライブパラメータリストは、この上に格納されます。

の位置にあることがわかります。実際には、BIOSがブートセクタを0x7c00にロードすると

!”と表示され、ブートローダに実行を委ねると、ss = 0x00, sp = 0xffffeとなります。

!また、65行目のpush命令の目的は、一時的に

!スタックにセグメントを保存し、次のように待って番号を決定します。

スタックをポップする前に、トラック・セクタの!をセグメント・レジスタfsと

!gs(109行目)となります。しかし、67行目と68行目の2つのステートメントは、位置を修正するのでスタックのしたがって、スタックを元の状態に戻さなければ、このデザインは間違っている。

スタックポップ操作を行う前に、元の位置に戻ってしまいます。つまり、ここにはバグがあるということです。
!一つ の修正点は、65行目を削除し、109行目を「mov ax, cs」に変更することです。

```

60 movax, cs!ds, es, ssを移動後のセグメント(0x9000)
   のように
   なり
   ます。
   に設定。

61 movab    dx, #0xfef4          ! 任意 値 >>512 - ディスクパームサイズ
   le

62
63 movab    ds, ax
   le
64 movab    es, ax
   le
65 プッシュ 軸                  !温度保存 セグメント(0x9000)で109行分。    (スリッ
   シュ                          パ!)

66
67 movab    ss, ax              !!!プットスタック at 0x9ff00 - 12.
   le                               ク
68 movab    sp, dx
   le

69 /*
70 *多くのBIOSのデフォルトのディスクパラメータテーブルでは
71 *最大セクタ数 を超えるマルチセクタリードを認識する
72 *デフォルトのディスクパラメータテーブルで指定されている。
73 *場合によっては      7セクタになることもあります。
74 *
75 *シングルセクタの読み出しは遅くて問題外な      ので。
76 *新しいパラメータテーブル      を作成して対応しなければなりません。
77 * (1枚目のディスク用)をRAMに格納しここでは、セクタの      最大値を
78 * HD 1.44      では、18個が限界です。
79 *
80 * 高くても痛く低いと痛い。
81 *
82 ds=es=ss=cs - INITSEG      ※セグメンテーションは以下の通りです。
83 *fs = 0, gs = パラメータテーブルセグメント
84 */
85 !BIOSで設定された割り込み0x1Eは、実際には割り込みではありません。対応する場所は
   割り込みベクターには、フロッピーディスクのパラメータテーブルのアドレスを指定します。
   ベクターはメモリ 0x1E * 4 = 0x78 にあります。このコードは、まず元のフロッピーディスクをコピーします。
   ! ディスクのパラメータテーブルをメモリ0x0000:0x0078から0x9000:0xfef4に変更して
   テーブルのオフセット4にあるトラックあたりの最大セクタ数を18にしました。テーブルのサイズは12バイトです。

86
87 push#0! セグメントレジスタfs=0      を設定。
88 ポップス
89 movbx, #0x78! fs:bxはパラメータテーブルのアドレス      です。
   !次の命令は、次のステートメントのオペランドが fs であることを示す。
   ! "というセグメントがあり、それは次のステートメントにのみ影響します。ここでは、テーブルの
   ! fs:bx は、元のアドレスとしてレジスタペア gs:si に配置され、レジスタペア
   ! es:di = 0x9000:0xfef4 が宛先アドレスとして使用されます。

90 セグfs
91 lgssi, (bx)! gs:siはソース92
   です。

93 movdi, dx! es:diが目的地! dx = 0xfef4 (61      行目で設定)。
94 movcx, #6! 12バイト      コピー
95 cld!方向フラグをクリアします。コピー 時にポインタを増加させる。96

```

```
97      rep!12バイトのテーブルを0x9000:0xfef4      にコピーします。  
98      セグGs  
99      movew  
100  
101      movdi, dx!Es:diは新しいテーブルを指し、その後テーブルを変更します。  
102      movb4(di),*18! パッチセクター数
```

```

103
104     セグ fs                      !割り込みベクター0x1Eに新しいテーブルを指定させる。
105     movable (bx),di
106     セグ fs
107     movable 2(bx),es
108
    65行目で保存されたセグメント値(0x9000)です。fs=gs=0x9000とすることで、元のセグメントに戻すことができます。
109     ポパックス
110     movfs, ax
111     movgs, ax 112
    !BIOS INT 0x13 関数 0 は、フロッピーディスクコントローラをリセットするために使用されます。
    コントローラでドライブヘッドを再調整（トラック0にシーク）。
113     キソラ、アァ！リセットFDC
114     xordl,dl! dl = ドライブ、ここでは最初のディスクドライブ          に設定されています。
115     int     0x13
116
117 !"セットアップ・セクター"をブートブロックの直後にロー
    ドします。118 !es'はすでに設定されていることに注意してくだ
    さい。
119
    !121～137行目の目的は、BIOS INT 0x13の機能2（ディスクセクタの読み取り）を使用することです。
    ディスクの第2セクタの先頭からセットアップモジュールを読み込んで
    !0x90200、合計4セクタです。読み出し時にエラーが発生した場合、その場所は
    ! エラーセクタが表示された後、ドライブをリセットして再試行すると、再試行せずに終了します。
    !INT 0x13（Read Sector）のパラメータは以下のように設定されています。
    ! ah = 0x02 - read disk          sector;al = nr of sectors to read;
    ch=シリンダーnrの下位8ビット、cl=セクターnr(bit0-5)シリンダーの上位2ビット(bit6-7)。
    ! dh = ヘッド                    ナンバー;dl = ドライブナンバー(ハードディスク の場合はビット7を設定)
    !"を返します。
    !エラーの場合、フラグCFがセットされ、ahにはエラーコードが格納されます。
    ! es:bx ->point to data buffer;
120 load_setup:
121     xordx, dx! ドライブ0、ヘッド0
122     movcx, #0x0002! セクター2、トラック0
123     movbx, #0x0200! アドレス=512, in INITSEG
124     movax, #0x0200+SETUPLEN! サービス2、セクタ数
125     int0x13! 読んでみて          ください。
126     jncok_load_setup! ok - continue 127
127     pushax! ダンプエラーコード
128     callprint_nl! 次の行          を表示します。
129     movbp, sp! ss:bpがchars(word)を指す。
130     callprint_hex! 16進数の値          を表示します。
131
132     ポパックス
133
134     xordl, dl! reset FDC !
135     xorah, ah
136     int0x13
137     jload_setup! j = jmp 138
138
139 ok_load_setup:
140
141 !ディスクドライブのパラメータの取得（セクタ数/トラック数など

```

! 次のコードでは、INT 0x13のファンクション8を使用して、ディスクドライブのパラメータを取得しています。
! 実際には、トラックごとのセクタ数を取得して、ロケーションセクタに格納しています。
! INT 0x13（ディスクドライブパラメーターの取得）のパラメーターは以下のように設定されています。
! ah = 0x08d1 = ドライブ番号(ハードディスクの場合はビット7を設定)
! 戻る。
! CFはエラーが発生した場合に設定され、ah=ステータスコードとなります。
! ah = 0, al = 0, bl = ドライブタイプ
! (AT/PS2), ch = 最大シリンダ番号の下位8ビット。
! cl = 最大セクタ番号(ビット5-0)、最大シリンダ番号の上位2ビット(ビット7-6)、
! dh = 最大ヘッド 番号、dl = ドライブ の数。
! es:di -> ドライブパラメータテーブル(フロッピーのみ)。

142

143

xordl, dl

144

movah, #0x08!AH=8はドライブパラメータの 取得

145

int0x13

146

xorh, ch

! 次の命令は、次のステートメントのオペランドがcsであることを示しています。
! セグメントになります。その次のステートメントにのみ影響を与えます。実際には、コードとデー
! タはすべて同じセグメントに設定されているので、セグメントレジスタcsとds, esの値は同じにな
! ります。したがって、この命令はここでは使用できません。

147

セグCS

! 次の文では、トラックごとのセクタ数を保存します。フロッピーディスク(dl=0)の場合、その最大
! トラック番号は256を超えることはなく、それを表現するにはchで十分なので、clのビット6-7は0
! でなければなりません。また、146行がch=0になっているので、この時点では、cxがトラックごと
! のセクタ数になります。

148

movsectors, cx

149

movax, #INITSEG

150

moves, ax!割り込みがesに変わったので、 ここで元

に戻します。

151

152

! 意味のないメッセージを表示する
! BIOS INT 0x10関数0x03と0x13を使用して、メッセージを表示します。"Loading" + cr + lf "の
! ように、キャリッジリターンとラインフィードの制御文字を含めた合計9文字を表示します。BIOS
! INT 0x10(リードカーソル位置)のパラメータは以下のように設定されています。
! ah = 0x03, カーソルの位置とサイズを読む;
! bh = ページ番号
! 戻る。
! ch = 走査開始ライン、cl = 走査終了ライン。
! dh = row(0x00は上); dl = column(0x00は左);
!
! BIOS INT 0x10(write string)のパラメータは、ah = 0x13, write
! stringとなっています。
! al = 書き込みモード。0x01はblの属性を使用し、カーソルは文字列の最後で停止します。
! bh = ページ番号、bl = 属性、dh, dl = 書き込み を開始する行、列
! cx = 文字列の文字数。
! es:bp -> 書き込む文字列。
! 戻ります。何もありません。

153

154

movah, #0x03! カーソルの位置 を読む

155

xorh, bh

156

int0x10! 位置: dh - row(0-24), dl - column(0-79)。

157

158

movcx, #9!合計9個のチャラチャラしたもの。

```

159     movable    bx, #0x0007    ! ページ 0, 属性 7 (通常)
160     movable    bp, #msg1      ! es:bpはメッセージを指しま
                                す。
161     movable    ax, #0x1301     文字列の書き込み、カーソルの移
                                動
162     int        0x10
163
164 ! OK、メッセージを書きました、次は
165 システムのロード (0x10000) を行います。
166     movabl     AX, #SYSSEG
167     e
168     movabl     es, ax          0x010000のセグメント
169     e
170     コール     read_it        システムモジュールのロード、esはパラメ
                                ータです。
171     コール     キルモーター    ドライブの状態を知るために、モーターを
                                停止させてください。
172     コール     print_nl

```

173 !その後、どの root-device を使用するかをチェックします。デバイス
が定義されていれば (!=0)、何もせず、与えられたデバイスを使用します。

175 !そうでなければ、BIOS が現在報告しているセクタ数に応じて、
/dev/PS0 (2,28) または /dev/at0 (2,8) のいずれかを使用します。

!

!上記2つのデバイスファイルの意味は以下の通りです。

!Linuxでは、フロッピードライブのメジャーナンバーが2 (43行目のコメント参照) の場合は

! マイナーデバイス番号 = タイプ*4 + nr, ここで nr はフロッピードライブ A, B, C または D の 0-3 です。

!"; typeはフロッピードライブのType (2--1.2MB、7--1.44MBなど)。

!7*4 + 0 = 28なので、/dev/PS0(2,28)は1.44MB デバイス番号0x021cのAドライブを指します。

!同様に、/dev/at0 (2,8)は、デバイス番号0x0208の1.2MB Aドライブを指します。

!root_devは、ブートセクタ508、509バイトの位置に定義されており、参照しています。

ルートファイルシステムのデバイス番号に !

!この値は、ルートfsが配置されているドライブに応じて変更する必要があります。

!例えば、ルートfsが1つ目のハードディスクの1つ目のパーティションにある場合には

の値は0x0301、つまり (0x01, 0x03) である必要があります。もし、ルートfsが2番目の1.44MBの
フロッピーディスクの場合は、0x021Dとなり、(0x1D, 0x02) となります。

!カーネルをコンパイルする際、Makefileで独自の値を指定することができます。カーネルの
画像ファイル作成プログラムのツール / ビルドでも、指定した値を使って
ルートファイルシステムのデバイス番号です。

```

177
178     セグCS
179     movax, root_dev
180     orax, ax!   root_devが定義されている (0ではない) ?
181     jneroot_defined

```

!以下の記述は、'sector' に保存されているトラックごとのセクタ数を

! 上の148行目でディスクの種類を判断します。セクタ数が15の場合、1.2MBのドライブを意味します。

セクタ数が18の場合、1.44MBのフロッピードライブということになります。起動可能なドライブであることから
!!!間違いなくAドライブです。

```

182     セグCS
183     movbx, sector
184                                     movax, #0x0208! /dev/ps0 - 1.2Mb
185     cmpbx, #15! sectors = 15 ?
186     jeroot_defined
187                                     movax, #0x021c! /dev/PS0 - 1.44Mb

```

[188](#) cmpbx, #18

[189](#) jeroor_defined

[190](#) undef_root: !そうでない場合は、無限ループ（死）となります。


```

191      jmp undef_root
192 root_defined:
193      セグCS
194      movroot_dev, ax! チェックしたデバイス番号を root_dev
        に保存します。195

```

196 ! その後（すべてがロードされた後）、198 !

ブートブロックの直後にロードされる 197 !

!セグメント間ジャンプ命令は、0x9020:0000にジャンプしてセットアップコードを実行します。

199

```

200      jmp i0, SETUPSEG! この時点でブートセクタのコードは終了!

```

!ここではいくつかのサブルーチンを紹介します。 read_it はディスク上のシステムモジュールを読み出すのに使
用します。

!Kill_moterはフロッピードライブのモーターを閉じるのに使われます。また、いくつかの画面表示があります。

! サブルーチン。

201

202 !このルーチンは、アドレス 0x10000 にシステムをロードし、64kB の
境界を越えないようにします。可能な限りトラック全体をロードして、で
きるだけ速くロードするようにしています。

205 !

206 ! in:es - 開始アドレスセグメント（通常は0x1000）

207 !

!以下のディレクティブ.wordは、2バイトのターゲットを定義しており、これは
C言語のプログラムで定義されている変数と、占有されているメモリ領域の量。

!値「1+SETUPLEN」は、最初に1つのブートセクタが読み込まれたことを示します。

セットアップコード（SETUPLEN=4）で占有されるセクタ数を加えたものです。

208 sread : .word 1+SETUPLEN! 現在のトラック のセクタリード 209

head:.word 0! 現在のヘッド

210 track

: .word 0! カレ

ントトラック 211

212 read_it:

!まず、入力セグメントを確認します。ディスクから読み込まれたデータが先頭に格納されている必要があります。
そうしないと無限ループに陥ってしまいます。

!レジスタbxは、現在のセグメントにデータを格納するための開始位置です。

!214行目のテスト命令は、2つのオペランドを持つビットごとの論理演算です。もし、ビット

両方のオペランドに対応する! が1の場合、結果の値は1、それ以外は0となります。

この操作のうち、フラグ（ゼロフラグZFなど）にのみ影響を与えます。例えば、AX=0x1000の場合。

テストの結果は (0x1000 & 0x0fff) = 0x0000となり、ZFフラグが設定されます。この時点で

次のインストラクションでは、条件が満たされていません。

213 mov ax, es

214 test ax, #0x0fff

215 die: jne die! esは64kBのバウンダリ でなければなりません。

216 xor bx, bx! bxはセグメント217 rp_read 内の

開始アドレスです。

!そして、すべてのデータが読み込まれたかどうかを確認します。現在読み込まれているセグメントが

! システムデータの終わりがあるセグメント(#ENDSEG)です。そうでない場合は、ジャンプして

データの読み込みを継続する場合は、下のラベル ok1_read をクリックしてください。それ以外の場合はリターン
します。

218 mov ax, es

219 cmp ax, #ENDSEG! まだ全部ロードしていないのか?

220 jb ok1_read

221 レット

222 ok1_readです。

!次に、現在のトラックが読み取る必要のあるセクタ数を計算して確認します。

その方法は以下の通りです。

!現在のトラックに読み込まれていないセクタの数や

セグメント内のデータ・バイトのオフセット位置を計算し、その結果をもとに、全体のデータ・バイト数を計算します。

未読のセクタがすべてであると、読み込んだバイト数がセグメント長64KBの制限を超えてしまいます。

が読み込まれます。それを超えた場合、今回読み込まなければならないセクタ数は

読み込み可能な最大バイト数（64KB-オフセット）に基づいて算出されます。

```

223     セグCS
224     mov     ax,s sectors!   トラック   ごとのセクタ数を取得します。
225     sub     ax,sread!      トラックが読み込ま   れたセクタ数を減算します。
226     mov     cx,ax!         cx = ax =   トラック   上の未読セクタの数です。
227     shl     cx,#9!         cx = cx * 512 +   現在のオフセット (bx).
228     add     cx,bx!=        操作           後に読み込まれたバイトの合計数。
229     jnc     ok2_read!      64KBを超えていなければ、ok2_read   に飛びます。
230     je ok2_read

```

!トラック上の未読セクタのデータを追加します。その結果が64KBを超える場合は

この時点で読める最大のバイト数は、(64KB-オフセット)となります。

!そして、読み取るべきセクタの数に変換します。ここで、0からある数値を引いたものが

! 64KBの補数です。

```

231     xor ax,ax
232     サブ ax,bx
233     shr ax,#9

```

234 ok2_read:

!指定された開始セクタ (c1) と番号に基づいて、es:bxまでのトラックのデータを読み出す

セクタの数(al)です。現在のトラックで読み込まれたセクタの数が

をカウントし、トラックの最大セクタ数と比較します。

!'sector'の場合、トラックにまだ未読のセクタがあるので、ok3_readにジャンプして続行します。

```

235     call    read_track!   トラック   上の指定されたセクタ数のデータを読み込みます。
236     mov     cx,ax!         cx =   今回   読み込んだセクタ数。
237     ax,sread!             に、読み込まれた   セクタの数を加えます。
238     セグCS
239     cmp     ax,s sectors!  トラック上に未読のセクタがある場合、ok3_read   にジャンプする
240     jne ok3_read

```

!トラックの現在のヘッドのすべてのセクターが読み込まれた場合、次のヘッドのデータは

トラックの先頭（ヘッド1）が読み込まれます。すでに行われている場合は、次のトラックに進みます。

```

241     mov ax,#1
242     sub     ax,head!       現在のヘッドナンバー   を確認する。
243     jne     ok4_read!      それがヘッド0であれば、ヘッド1   のセクタを取りに行く。
244     inc track

```

245 ok4_read:

```

246     mov     head,ax!!!     現在のヘッドNoをヘッド   に格納。
247     xor     ax,ax!         読んだ   セクタ数をクリア

```

する 248 ok3_read:

!現在のトラックに未読のセクタが残っている場合は、まずセクタ数を保存します。

!"を読んで、データが保存されている開始位置を調整します。よりも小さい場合は

64KBのバウンダリ値を取得した後、rp_read (217行目) にジャンプし、データの読み込みを続けます。

```

249     mov     sread,ax!     トラック   に読み込まれたセクタの数を保存します。
250     shl     cx,#9!         nr of sectors read * 512 bytes.
251     add     bx,cx!         データ   の開始位置を調整します。
252     jnc rp_read

```

!それ以外の場合は、64KBのデータが読み込まれたことを示します。この時点での調整は

現在のセグメントは、次のセグメントを読むための準備をします。

```

253     mov ax,es
254     add     ah,#0x10!      セグメントのベースアドレスが次の64KBを指すように調整します。

```

```

255      mov es, ax
256      xor                                bx, bx! オフセの値をクリアします。
257      jmp rp_read
258
!Read_track サブルーチン。指定された開始セクタのデータを読み込み、指定された数の
!!! es:bxの先頭までのトラック上のセクタです。BIOSディスクの説明を参照
119行目のint 0x13, ah=2の読み込み割り込み。
! al -      読まれるセクタ、es:bx - データバッファ。
259 read_trackです。
!最初にBIOS INT 0x10, function 0x0e (Write characters by telex)を呼び出すと、カーソルが移動します。
!"を1つ前の位置に表示します。ドット「...」を表示します。
260      PUSHA! すべてのレジスターを押してください
261      プーシャ
262      mov ax                                , #0xe2e! ローディング... メッセージ 2e = .
263      mov bx                                , #7! キャラクターの前景色属性。
264      int 0x10
265      ポパ
266
!その後、正式にトラック・セクタ・リード・オペレーションが行われます。
267      mov                                dx, track! 現在のトラック。
268      mov                                cx, sread! 現在のトラックですでに読み込まれたセクタ。
269      inc                                cx! cl = 読み始めのセクターNr.
270      mov                                ch, dl! ch = 現在の頭のNr.
271      mov                                dx, head!!! 現在のヘッドのnr      を取得。
272      mov                                dh, dl! dh = ヘッドのnr, dl = ドライブ (Aドライブ      は0)
273      と                                dx, #0x0100! 頭のnrは1以上で      はありません。
274      mov                                ah, #2! ah = 2, read
sectors. 275
276      push                                dx! Save for error dump
277      push cx
278      push bx
279      プッシ
ユアックス 280
281      int 0x13
282      jc                                bad_rt! エラーの場合、bad_rt      にジャンプします。
283      SP, #8を追加                                ! OKならステータス情報を破棄。
284      ポパ
285      レット
286
!ディスクの読み取りエラーです。最初にエラーメッセージが表示され、次にドライブのリセット操作が行われます
!(ディスク割り込み関数番号0)が実行された後、read_trackにジャンプして再試行します。
287 bad_rt: プッシュアックス                                エラーコードの保存
288      print_allを呼び出す                                !"ah "はエラー、"al "はリード
289
290
291      xor ah, ah
292      xor dl, dl
293      int 0x13
294
295
296      Add Sp, #10                                !エラー状態で保存された情報を破棄します。
297      ポパ
298      jmp read_track

```

```

299
300 /*
301  *print_allはデバッグ用です。
302  *すべてのレジスターをプリント前提として、これは
303 ルーチンから呼び出され、次のような スタックフレームを持つ。
304 *dx
305 *cx
306 *bx
307 *ax
308 *エラー
309 *ret <- sp
310 *
311 */
312
313 print_allです。
314     mov cx                , #5! エラーコード+4レジスタ
315     mov bp,              sp!現在のスタックポインタを保存
    する sp. 316
317 print_loopです。
318     push                cx! save count left
319     読みやすく         するためにprint_nl!
320     jae                 no_reg! 登録名が必要   かどうかを見る
321                        ! CF=0の場合は、レジスタを表示せずにジャンプしま
    す。スタックのレジスタ順に対応して、その名前を表示します。"AX : " などと表示します。
322     mov ax, #0xe05 + 0x41 - 1! ah = function 0x0e; al = char (0x05 + 0x41 -1)
323     サブアル、クラ
324     int 0x10
325
326     mov al                , #0x58!X
327     int 0x10
328
329     mov al                , #0x3a!:
330     int 0x10
331
    !bpが指すスタックの内容を表示します。もともとbpはリターンアドレスを指しています。
332 no_reg:
333     add bp                , #2! 次のレジスタ
334     print_hexを呼び出して、それ   を表示します。
335     ポップCIX
336     ループ print_loop
337     レット
338
    !BIOS INT 0x10を呼び出して、キャリッジリターンとラインフィードの制御文字を表示します。
339 print_nlです。
340     mov ax                , #0xe0d!CR
341     int 0x10
342     mov al                , #0xa!LF
343     int 0x10
344     レット
345
346 /*
347  *print_hex はデバッグ用で、単語   を表示します。
348  ss:bpをヘキサデシマルで表記してい   ます。

```

```

349 */
350 !BIOS INT 0x10を呼び出して表示 ss:bp が指す単語を 4 桁の 16 進数で表したものを。
351 print_hexです。
352         movcx, #4           !16進数4桁
353         movdx, (bp)         ワードをdxにロード
354 print_digitです。
355 !ハイバイト はディस्पの方が先なので、dxを4回転させて上位4ビットをdxの下位4ビットに移
        rol             レイdx、動させます。
        #4             下位4ビットが使用されるように回転
356     movable     あ、#0xe
357     movable     al, dl       マスクを外しているので、次のニブルがあるだけです。
358     そして     al, #0xf     !"と入力すると、下位4ビットのみが表示されます。
!0 "の追加     コード を使って、値をcharに変換します。alの値が0x39を超える場合は
ASCII         0x30
表示されている値が9番を超えているので、 'A'-'F' の加算値、 #0x30で表す必要があるとい
359     うこと     です! 0ベースの桁、'0' に変換します。
360         cmpal             , #0x39! オーバーフロー のチェック
361         jbegood_digit
362         addal, #0x41 - 0x30      - 0xa!'A' - '0' - 0xa
363
364 good_digit。
365         int0x10
366         loopprint_digit! cx--。cx>0の場合、次の値を表示しま
        す。
367     レット
368
369
370 /*
371 この手順では、フロッピーディスクドライブのモーターをオフにするので
372 私たちが既知の状態カーネルを入力すること、そして
373 後から心配する必要はありません。
374 */
!下記377行目の0x3f2という値は、フロッピーディスクコントローラのポートアドレスであり
デジタル出力レジスタ(DOR)ポートと呼ばれています。これは、8ビットのレジスタで
ビット7~4は、4台のフロッピードライブ(D-A)の起動・停止を制御します。ビット3~2
DMAや割り込み要求の有効化/無効化、フロッピーディスクのスタート/リセットに使用されます。
! ディスクコントローラFDCです。ビット1~ビット0は、選択されたフロッピードライブの
!"の操作を行います。378行目のalに設定された0の値は、Aドライブの選択、ターン
FDCをオフにして、DMAと割り込み要求を無効にし、モーターをオフにします。を参照してくださ
い。
フロッピーの詳細については、kernel/blk_drv/floppy.c プログラムの背後にある指示を参照して
ください。
! コントロールカードのプログラミング。
375 kill_motor。
376     push dx
377     mov dx, #0x3f2         ! フロッピーコントローラポートDOR。
378     xor al, al             !Aドライブ、FDCを閉じる、DMAとintを無効にする、モー
        ターを閉じる
379     アウトプ             ! ポートDXにアルを出力。
380     ポップDX
381     レット
382
383     ハクターにありませ

```

[389](#)

!アドレス506(0x1FA)から始まるので、root_devはブートセクタの508から始まる2バイトになります。

[390](#) .org 506

[391](#) swap_dev.

[392](#) .word SWAP_DEV

[393](#) root_dev:

[394](#) .word ROOT_DEV

!0xAA55は、ブートディスクにBIOSで使用する有効なブートセクタがあることを示すフラグです。

ブートセクタをロードするためのプログラムです。ブートセクタの最後の2バイトでなければなりません。

[395](#) boot_flag:

[396](#) .word 0xAA55

[397](#)

[398](#) .テキスト

ト

[399](#) endtext

[400](#) .データ

[401](#) enddata:

[402](#) .bss

[403](#) endbssで

す。

[404](#)

6.2.3 参考情報

bootsect.Sプログラムの説明については、ネット上に大量の文書があります。その中でも、Alessandro Rubini氏の論文「Tour of the Linux kernel source」は、カーネルのブートプロセスをより包括的に説明しています。このプログラムはCPUリアルモードで動作しますので、より理解しやすいと思います。もし、この時点でまだ読むのが難しいのであれば、まず80x86のアセンブリとそのハードウェアについて復習してから、本書を読み進めることをお勧めします。新しく開発されたLinuxカーネルについては、このプログラムはあまり変更されておらず、基本的に0.12バージョンのbootsect ファイルの外観を維持しています。

6.2.3.1 Linux 0.12 ハードディスクのデバイス番号

Linuxシステムでは、さまざまなデバイスは、デバイス番号（論理デバイス番号）によってアクセスされます。デバイス番号は、メジャーデバイス番号とマイナーデバイス番号で構成されています。メジャーデバイス番号はデバイスの種類を示し、マイナーデバイス番号は特定のデバイスオブジェクトを示す。メジャーデバイス番号とマイナーデバイス番号は1バイトで表され、各デバイス番号は2バイトである。ブートセクタプログラムの対象となるハードディスクはブロックデバイスで、メジャーデバイス番号は3です。

- 1 - メモリ。
- 2 - フロッピーディスク
- 3 - ハードディスクです。
- 4 - ttyx;
- 5 - tty;
- 6 - パラレルポート。
- 7 - 無名のパイプ

従来のハードディスクでは1～4のパーティションが存在するため、ハードディスクもマイナーデバイス番号を使ってパーティションを指定していました。ハードディスクの論理デバイス番号は以下のように構成されています。

デバイス番号＝メジャーデバイス番号 <<8 + マイナーデバイス番号。

両ハードディスクのすべての論理デバイス番号を表6-1に示します。0x0300と0x0305は、特定のパーティションに対応するものではなく、ハードディスク全体を表しています。また、Linuxカーネルバージョン0.95では、このような面倒な命名方法を採用していないため、現在と同じ命名方法を採用していることにも注意が必要です。

表6-1 ハードディスクの論理デバイス番号

デバイス nr	デバイスファイル	説明
0x0300	/dev/hd0	初代ハードドライブ全体を表す
0x0301	/dev/hd1	第1ディスクの第1パーティション
0x0302	/dev/hd2	第1ディスクの第2パーティション
0x0303	/dev/hd3	1枚目のディスクの3番目のパーティション
0x0304	/dev/hd4	第1ディスクの第4パーティション
0x0305	/dev/hd5	2枚目のハードドライブ全体を表す
0x0306	/dev/hd6	2枚目のディスクの最初のパーティション
0x0307	/dev/hd7	2つ目のディスクの2つ目のパーティション
0x0308	/dev/hd8	2番目のディスクの3番目のパーティション
0x0309	/dev/hd9	2枚目のディスクの4番目のパーティション

6.2.3.2 ハードディスクからの起動

bootsect プログラムは、フロッピーディスクから Linux システムを起動するためのデフォルトの方法とプロセスを提供します。ハードドライブからシステムを起動したい場合は、通常、Shoelace、LILO、Grub などの複数の OS ブートローダを使ってシステムを起動する必要があります。この時、bootsect.S が実行する必要のある操作はこれらのプログラムで完了し、bootsect プログラムは実行されません。なぜなら、ハードディスクから起動した場合、通常、カーネルイメージファイルは、ハードディスクのアクティブなパーティションのルートファイルシステムに格納されます。そのため、カーネルイメージファイルがファイルシステムのどこにあるのか、どのファイルシステムなのかを知る必要があります。つまり、ブートセクタプログラムがファイルシステムを認識してアクセスし、そこからカーネルイメージファイルを読み取る必要があります。

ハードディスクからの起動の基本的な流れは、システムの電源を入れた後、起動可能なハードディスクの第1セクター（MBR - Master Boot Record）がBIOSによってメモリ0x7c00に読み込まれ、実行が開始されます。プログラムは、まず自身をメモリ0x600に移動させ、MBRのパーティションテーブルで指定されたアクティブパーティションの第1セクター（ブートセクター）に従ってメモリ0x7c00にロードし、実行を開始します。

Linux 0.12系では、カーネルイメージファイルはルートファイルシステムから独立しています。この方法で直接ハードディスクからシステムを起動すると、ルートファイルシステムとカーネルイメージファイルが共存できないという問題が発生します。解決策は2つ考えられます。ひとつは、小容量のアクティブ・パーティションにカーネル・イメージ・ファイルを置き、それに対応するルート・ファイル・システムを別のパーティションに置く方法です。これによりハードディスクのプライマリパーティションが1つ増えますが、bootsect.S プログラムに最小限の変更を加えるだけで、ハードディスクからシステムを起動することができるはずです。もう一つの方法は、カーネルイメージファイルとルートファイルシステムを一つのパーティションにまとめる方法です。つまり、カーネルイメージ

ファイルをパーティションの最初のいくつかのセクタに配置し、ルートファイルシステムをその後の指定されたセクタから格納するのです。どちらの方法も、コードの修正が必要です。最後の章を参照して、**bochs**シミュレーション・ソフトウェアを使って実験をしてみてください。

6.3 setup.S

6.3.1 機能説明

setup.S はオペレーティングシステムのローダです。主な機能は、ROMのBIOS割り込みを利用してマシンの設定データを読み取り、0x90000の先頭（bootsectプログラムがある場所をカバー）に保存することです。取得したパラメータと保存されたメモリの位置を表6-2に示します。これらのパラメータは、カーネル内の関連プログラムで使用されます。例えば、キャラクタデバイスドライバセットの console.c プログラムや tty_io.c プログラムなどです。

表6-2 セットアッププログラムが読み込んで保存するパラメータ

アドレス	サイズ(バイト)	名前	説明
0x90000	2	カーソルの位置	Colum (0x00-left), Row (0x00-topmost)
0x90002	2	拡張メモリ	1MBアドレスから始まる拡張メモリーのサイズ（単位：KB）
0x90004	2	表示ページ	現在の表示ページ
0x90006	1	表示モード	
0x90007	1	シャルコロン	
0x90008	2	シャルロー？	
0x9000A	1	ディスプレイメモリー	0x00-64k, 0x01-128k, 0x02-192k, 0x03=256k
0x9000B	1	表示状態	0x00-Color, I/O=0x3dX; 0x01-Mono, I/O=0x3bX
0x9000C	2	プロパティパラス	ディスプレイアダプタのプロパティパラメータ
0x9000E	1	スクリーンの列	現在のディスプレイの行数を表示します。
0x9000F	1	スクリーンコラム	現在のディスプレイの列を表示します。
...			
0x90080	16	Hd Paras Table	1枚目のハードディスクのパラメータテーブル
0x90090	16	Hd Paras Table	2つ目のHdパラメータテーブル（ない場合は0）。
0x901FC	2	ルートデビীরの	ルートファイルシステムのデバイス番号(bootsec.sで設定)

続いてセットアッププログラムにより、システムモジュールを0x10000-0x8ffffから絶対アドレス0x00000に移動させる（当時、カーネルのシステムモジュール系の長さはこの値を超えないと考えられていた：512KB）。次に、割り込みディスクリプターテーブルレジスタ（IDTR）とグローバルディスクリプターテーブルレジスタ（GDTR）をロードし、A20アドレスラインをオンにして、2つの割り込み制御チップ8259Aを再構成し、ハードウェア割り込み番号を0x20～0x2fに再構成します。最後に、CPUの制御レジスタCR0（マシン・ステータス・ワードとも呼ばれる）を設定し、32ビットのプロテクトモードに入り、システムモジュールの最前面にあるhead.sのプログラムにジャンプして実行を続けます。

head.sを32ビットのプロテクトモードで動作させるために、プログラム内で割り込みディスクリプターテーブル（IDT）とグローバルディスクリプターテーブル（GDT）を一時的に設定し、GDTには現在のカーネルコードとデータセグメントのディスクリプターを設定しています。以下のhead.sのプログラムでは、これらのディスクリプターテーブルもカーネルの必要性に応じて再設定されます。

まず、セグメントディスクリプターのフォーマット、ディスクリプターテーブルの構造、セグメントセレクターのフォーマットについて説明します。Linuxカーネルで使用されているコードセグメントとデータセグメントの記述子のフォーマットを図6-4に示す。各フィールドの詳しい意味については、第4章の説明を参照してください。

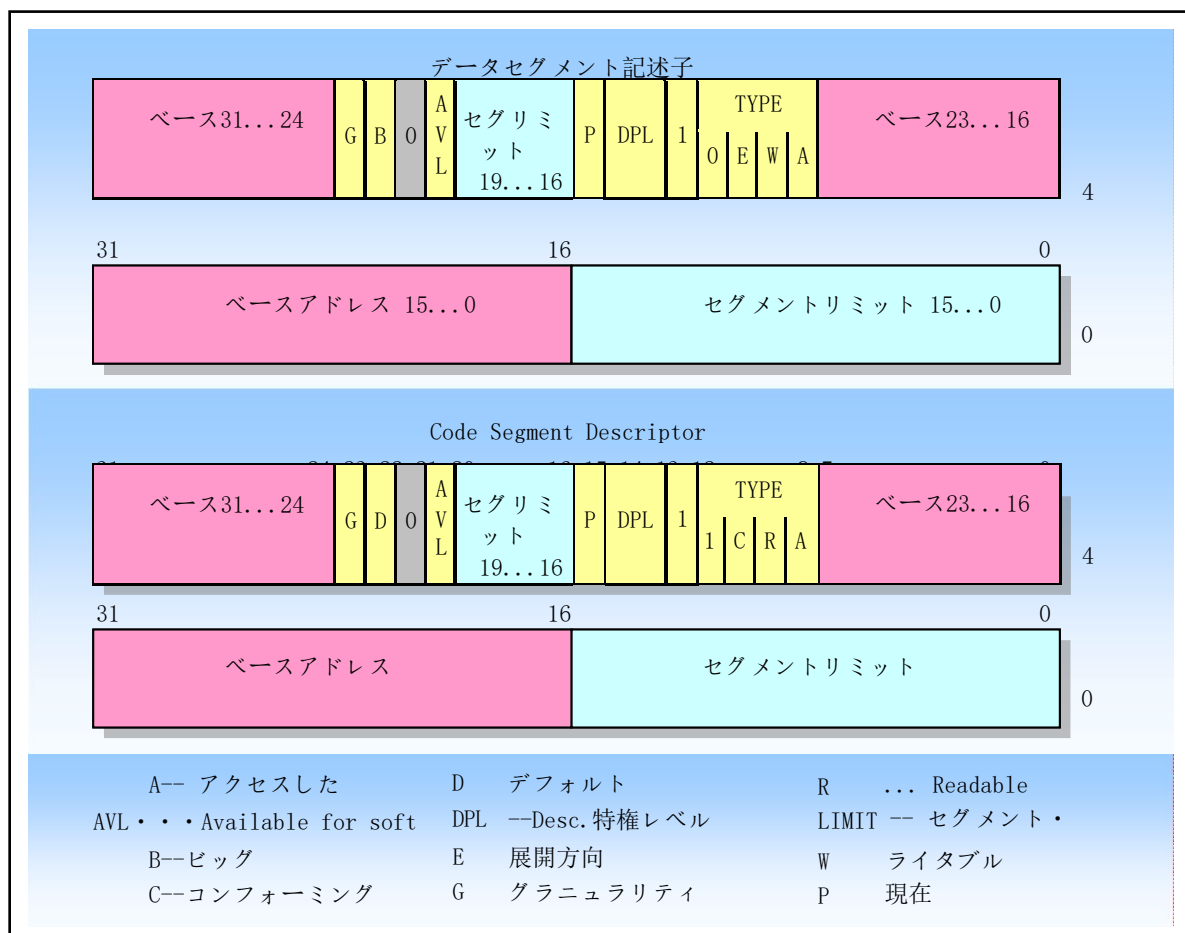


図6-4 コードおよびデータセグメントの記述子フォーマット

セグメントディスクリプターは、ディスクリプターテーブルに格納されています。ディスクリプターテーブルは、実際には、メモリ内のディスクリプターアイテムの配列です。ディスクリプターテーブルには2種類あります。グローバル・ディスクリプター・テーブル（GDT）とローカル・ディスクリプター・テーブル（LDT）です。プロセッサは、GDTRおよびLDTRレジスタを使用して、GDTテーブルおよび現在のLDTテーブルの位置を特定します。この2つのレジスタは、ディスクリプターテーブルのベースアドレスと、テーブルの長さをリニアアドレスで保持します。GDTRレジスタへのアクセスにはLGDT命令とSGDT命令が、LDTRレジスタへのアクセスにはLLDT命令とSLDT命令が使用されます。LGDTは、メモリ上の6バイトのオペランドを使用してGDTRレジスタをロードします。最初の2バイトはディスクリプターテーブルの長さを表し、最後の4バイトはディスクリプターテーブルのベースアドレスです。ただし、LLDT命令がLDTRレジスタにアクセスする際に使用するオペランドは、グローバルディスクリプターテーブルGDTのディスクリプター項目のセクタを表す2バイトのオペランドであることに注意してください。セクタに対応するGDTテーブルのディスクリプター項目は、ローカルディスクリプターテーブルに対応する必要があります。

例えば、setup.Sプログラムで設定されたGDT記述子の項目（567～578行目参照）。コード・セグメント記述子の値は、0x00C09A00000007FF（つまり、0x07FF、0x0000、0x9A00、0x00C0）です。コードセグメントのリミットレングスが8MB（ $(0x7FF + 1) * 4KB$ 、リミットレングスの値は0からカウントされるため1が加算される）、リニアアドレス空間におけるセグメントのベースアドレスが0、セ

グメントタイプの値0x9Aは、セグメントがメモリ上に存在すること、セグメントの特権レベルが0、セグメントタイプが読み取り可能な実行可能コードセグメント、セグメントコードが32ビット、セグメントの粒度が4KBであることを示す。データセグメント記述子の値は0x00C09200000007FF（例：0x07FF、0x0000、0x9200、0x00C0）で、これはデータセグメントの限界長が8MBであることを意味し、リニアアドレスのセグメントのベースアドレスは0x00C09200000007FFです。

また、セグメントタイプは読み取り/書き込み可能なデータセグメント、セグメントコードは32ビット、セグメントグラニュラリティは4KBとなっています。

ここでは、セレクトラについての説明をします。セレクトラ部は、セグメントディスクリプタを指定するためのもので、ディスクリプタテーブルを指定し、ディスクリプタアイテムの1つにインデックスを付けることで行う。図6-5にセレクトラのフォーマットを示す。



図6-5 セグメントセレクトラのフォーマット

このインデックスは、指定されたディスクリプターテーブルの8192 (2^{13}) 個のディスクリプターのうちの1つを選択するために使用されます。プロセッサはインデックスを8倍し、記述子テーブルのベースアドレスを加えて、テーブルで指定されたセグメント記述子にアクセスします。テーブル・インジケータ(TI)は、セレクトラが参照するディスクリプター・テーブルを指定するために使用されます。値が0の場合はGDTテーブルが指定されていることを示し、値が1の場合は現在のLDTテーブルが指定されていることを示す。RPL(Requestor's Privilege Level)は、メカニズムを保護するために使用されます。

GDTテーブルの最初の項目（インデックス値0）は使用されないで、インデックス値が0でテーブル・インジケータ値が0のセレクトラ（つまりGDTの最初の項目を指すセレクトラ）は、ヌル・セレクトラとして使用できます。セグメント・レジスタ（CSやSSは不可）がヌル・セレクトラをロードしても、プロセッサは例外を発生させません。しかし、セグメント・レジスタがメモリ・アクセスに使用された場合は例外が発生します。この機能は、未使用のセグメント・レジスタを初期化して、予期せぬ参照があった場合に特定の例外を発生させるアプリケーションに有効です。

プロテクトモードに入る前に、まず、グローバルディスクリプターテーブルGDTなど、使用するセグメントディスクリプターテーブルを設定する必要があります。そして、命令LGDTを用いて、ディスクリプターテーブルのベースアドレスをCPUに通知する（GDTテーブルのベースアドレスはGDTRレジスタに格納されている）。そして、マシン・ステータス・ワードの保護モード・フラグをセットして、32ビットの保護モードに入ります。

また、setup.Sの215～566行目は、マシンで使用されているディスプレイカードの種類を特定するために使用されます。システムがVGAディスプレイカードを使用している場合は、ディスプレイカードが25行×80列以上の拡張表示モード（またはディスプレイモード）をサポートしているかどうかを確認する。いわゆるディスプレイモードとは、ROM BIOSがINT 0x10の設定画面表示情報の関数0（ah=0x00）に割り込みをかける方法を指します。alレジスタの入力パラメータ値は、設定したい表示モードまたは表示モード番号です。通常、IBM PCの発売当初に設定できるいくつかの表示モードを標準表示モードと呼び、後から追加されたいくつかのモードを拡張表示モードと呼びます。例えば、ATIのディスプレイカードは、標準ディスプレイモードに加えて、拡張ディスプレイモード0x23と0x33にも対応している。つまり、132列×25行と132列×44行の2つのディスプレイモードを使って、

画面に情報を表示することもできるのだ。VGAやSVGAが存在するだけで、これらの拡張ディスプレイモードはディスプレイカードのBIOSでサポートされています。既知のタイプのディスプレイカードが識別された場合、プログラムはユーザーに解像度を選択する機会を提供します。

この部分は、多数のディスプレイカードそれぞれに固有のポート情報を含むため、このフラグメントプログラムは複雑なものとなっている。幸いなことに、この部分はカーネルの動作にはほとんど関係しません。

の原則がありますので、読み飛ばしても構いません。もし、このコードを徹底的に理解したいのであれば、Richard F. Ferraro氏の著書「Programmer's Guide to the EGA, VGA, and Super VGA Cards」を参照するか、オンラインでダウンロードできる古典的なVGAプログラミング教材:「VGADOC4」を参照してください。この部分はMats Andersson (d88-man@nada.kth.se)がプログラムしたのですが、Linusは誰がd88-manなのか忘れてしまいました :-)。

6.3.2 コードコメント

プログラム 6-2 linux/boot/setup.S

```

1 !
2                               !setup.s(C) 1991 Linus
Torvalds 3 !
4 ! setup.sは、BIOSからシステムデータを取得し、システムメモリの適切な
場所配置する役割を担っています5 !
6 ! セットアップ.sとシステムの両方がブートブロックによってロ
ードされました。7 !
8 ! このコードは、バイオスにメモリ/ディスク/その他のパラメータを尋
ね、9 !"安全な"場所に置きます。0x90000-0x901FF、つまり
10 ブートブロックを使用していました。その後、プロテクトモードに
11 バッファブロックの場合は、領域が上書きされる前に、システムがそ
こから読み取ることができます 12 !
13 !
14
15 !注意! これらはbootsect.sと同じでなければなりません。
! config.hの定義です。DEF_INITSEG = 0x9000; DEF_SYSSEG = 0x1000; DEF_SETUPSEG = 0x9020
16 #include <linux/config.h>
17
18 INITSEG = DEF_INITSEG! ブートをここに移動して
邪魔に ならないようにする 19 SYSSEG= DEF_SYSSEG! 0x10000(65536)
でシステムがロードされる。20 SETUPSEG = DEF_SETUPSEG! これは現在のセグメ
ントです。
21
22 .globl begtext, begdata, begbss, endtext, enddata,
endbss 23 .text
24 begtext。
25 .データ
26 begdata:
27 .bss
28 begbss。
29 .テキスト
30
31 エントリースタート
32 を開始しました。
33
34 ! 読み込みがうまくいったので、現在のカーソル位置を取得し、後世のために保存
します 35 !
36
37 movax, #INITSEG! dsをINITSEGに設定(0x9000)
38 movds, ax 39
40 !メモリサイズの取得(拡張メモリ、kB
!BIOS INT 0x15 関数 0x88 を使用して拡張メモリサイズを取得し、0x90002 に格納します。

```


!戻る。

! ax = 0x100000(1MB)からの拡張メモリ(KB) エラーCFが設定されている場合、ax = エラーコード。

```

41
42      movah, #0x88
43      int0x15
44      mov[2], ax! 0x90002に

```

格納 [45](#)

[46](#) ! EGA/VGAといくつかの設定パラメータのチェック

! BIOS INT 0x10 function 0x12 (video subsystem configuration)を使用して、EGAの設定情報を取得します。
 ! ah = 0x12; bl = 0x10 - ビデオコンフィギュレーション情報を返す。
 ! 戻る。
 ! bh = 映像状態 (0x00 - カラー, I/O ポート = 0x3dX; 0x01 - モノ, I/Oポート = 0x3bX).
 ! bl = 搭載メモリ (0x00 - 64k, 0x01 - 128k, 0x02 - 192k, 0x03 = 256k)。
 ! cx = アダプタの機能と設定 (このリストの後にあるINT 0x10の説明を参照)

```

47
48      movah, #0x12
49      movbl, #0x10
50      int0x10
51      mov[8], ax! 0x90008 = ??
52      mov[10], bx! 0x9000A = インストールされたメモリ、0x9000B = 表示状態。
53      mov[12], cx! 0x9000C = アダプタの機能と設定。

```

! 画面の行と列を検出します。アダプターがVGAカードの場合、ユーザーには
 ! "で行を選択し、0x9000Eに保存します。

```

54      movable    AX, #0x5019      ! のデフォルトの行と列を設定し ax(ah = 80, al = 25)で
                                     ます。
55      cmp        BL, #0x10      ! blが0x10の場合、それはaではな VGAカード、ジャンプ。
                                     いことを意味します。
56      ジー      ノブガ
57      コール    chsvga          ! カードのメーカーと種類を知る 列の変更(215行目)
58 novga: mov     [14], ax        ! 画面の行と列の値 (0x9000E, 0x9000F)を保存する。

```

! BIOS INT 0x10関数0x03を使用してカーソル位置を取得し、0x90000 (2バイト) に保存します。

! ah = 0x03 カーソル位置の取得 bh = ページ番号

! 戻る。

ch = 走査開始ライン、cl = 走査終了ライン。

! dh = row(0x00は上); dl = colum(0x00は左);

```

59      movah, #0x03! カーソルの位置を読む
60      xorbh, bh
61      int0x10! 既知の場所に保存し、con_initがフェッチします。
62      mov[0], dx!!! 0x90000からです 63
64 ! ビデオカードのデータを取得します。

```

! BIOS INT 0x10, function 0x0fを使用して、現在のディスプレイモードとステータスを取得します。

! ah = 0x0f - 現在のビデオモードと状態を取得する

! 戻る。

! ah = 画面のコラム数、al = 表示モード、bh = 現在の表示ページ。

! 0x90004(word) - 現在のページを格納する; 0x90006 - 表示モード; 0x90007 - 画面の列

```

65
66      movah, #0x0f
67      int0x10
68      mov[4], bx! bh = 表示ページ
69      mov[6], ax! al = ビデオモード、ah = ウィンドウ幅 70
71 ! hd0データの取得

```

! 最初のハードディスクのパラメータテーブルのアドレスは、実際には、ベクター値である

! 割り込み0x41!そして、2つ目のハードディスクのパラメータテーブルは、1つ目のパラメータテーブルの隣にあります。があります。

! 割り込み0x46のベクター値も、第2ハードディスクのパラメータテーブルを指しています。

! テーブルサイズは16バイトです。以下の部分では、BIOSの2つのパラメータテーブルを最初のテーブルは0x90080に、2番目のテーブルは0x90090に格納されています。

! ハードディスクのパラメータテーブルの説明は、6.3.3項の表6-4を参照してください。

[72](#)

!75行目は、メモリからロングポインタの値を読み込み、dsとsiに配置します。

!レジスターです。ここでは、メモリ4 * 0x41 (=0x104)に格納されている4バイトが読み込まれます。これらの4このバイトは、ハードディスクのパラメータテーブルの開始アドレスです。

[73](#)

```
movax, #0x0000
```

[74](#)

```
movds, ax
```

[75](#)

```
ldssi, [4*0x41]!INT 0x41ベクター、hd0パラテーブルのadrを取得 -> ds:si
```

[76](#)

```
movax, #INITSEG
```

[77](#)

```
move, ax
```

[78](#)

```
movdi, #0x0080!レプリケーションの宛先。0x9000:0x0080 -> es:di
```

[79](#)

```
movcx, #0x10! 合計16バイト の移動。
```

[80](#)

```
repetition
```

[81](#)

```
movsb
```

[82](#)

[83](#) !hd1データの取

得 [84](#)

[85](#)

```
movax, #0x0000
```

[86](#)

```
movds, ax
```

[87](#)

```
ldssi, [4*0x46]!INT 0x46 ベクトル値 -> ds:si
```

[88](#)

```
movax, #INITSEG
```

[89](#)

```
move, ax
```

[90](#)

```
movdi, #0x0090!0x9000:0x0090 -> es:di
```

[91](#)

```
movcx, #0x10
```

[92](#)

```
repetition
```

[93](#)

```
movsb
```

[94](#)

[95](#) !hd1があることを確認してください。)

!マシンに2つ目のハードドライブがあるかどうかを確認します。ない場合は、2台目のテーブルをクリアしてください。

!ROM BIOSのINT 0x13関数0x15を使って、ディスクタイプを取得します。

! ah = 0x15 - ディスクタイプを取得します。

! dl = ドライブ番号 (0x8Xはハードドライブ、0x80はドライブ0、0x81はドライブ1)

!戻る。

! ah = タイプコード (00 - ドライブなし、01 - フロッピー、変化検知なし。

!02 - フロッピー (またはその他のリムーバブル)、変更検知、03 - ハードディスク)。

! cx:dx - 512バイトのセクタの数。

!CFはエラー時に設定され、ah = ステータスとなります。

[96](#)

[97](#)

```
movax, #0x01500
```

[98](#)

```
movdl, #0x81
```

[99](#)

```
int0x13
```

[100](#)

```
jcnodisk1
```

[101](#)

```
cmpah, #3! ハードドライブ? (type == 3)?
```

[102](#)

```
jeisdisk1 103
```

no_disk1:

[104](#)

```
movax, #INITSEG! 2台目のハードドライブがないので、2台目のパラメータテーブルをクリーンにします。
```

[105](#)

```
move, ax
```

```
106      movdi, #0x0090  
107      movcx, #0x10
```

```

108             movax, #0x00
109             repetition
110             ストスブ
111 is_disk1です。
112
113 ここで、保護モードに移行したいと思います。
114
115             cli! 割り込みは禁止です。
116
117 まず、システムを本来の場所に移動させます。
!以下のプログラムの目的は、システムモジュール全体を0x00000の
!の位置、つまりメモリブロック（512KB）（0x10000～0x8ffff）をメモリの下端に移動させること
!です。
118
119             movax, #0x0000
120             cld!'direction'=0, movsが前に進む
121 do_moveです。
122             moves, ax! デスティネーションセグメント! es:di (0x0:0x0
            initially
            )
123             addax, #0x1000
124             cmpax, #0x9000!最後のセグ（0x8000セグからの64KB）のコードが
            移動した?
125             jzend_move! はい、ジャンプ。
126             movds, ax! ソースセグメント! ds:si (0x1000:0x0初期値)
127             サブディ、ディ
128             サブシ、シ
129             movcx, #0x8000! 0x8000ワード（64Kバイト）を動
            かす
130             repetition
131             movsw
132             jmpdo_move
133
134 そして、セグメント記述子を読み込みます。
!ここから先は32ビットのプロテクトモードでの動作となります。第4章では
!の情報を提供しています。プロテクトモードを実行する前に、まずは
!使用するセグメントディスクリプターテーブル。ここでは、グローバルディスクリプターテーブル
!を設定する必要があります。
!GDTと割り込みディスクリプターテーブルIDTです。
!
!LIDTは、割り込みディスクリプターテーブルレジスタをロードするための命令です。その
!オペランド（idt_48）は6バイトです。最初の2バイト（0-1バイト）はディスクリプターのサイズ
!です。
!最後の4バイト（2-5バイト）は、ディスクリプターテーブルの32ビットリニアベースです。
!次の580--486行を参照してください。IDTテーブルの各8バイトのエントリは、コードを示す
!割り込みが発生したときに呼び出される必要のある情報です。と多少似ています。
!割り込みベクターが、より多くの情報を含んでいます。
!
!LGDT命令は、グローバルディスクリプターテーブルレジスタのロードに、同じ
!LIDT命令と同様のオペランド形式です。GDTの各ディスクリプターアイテム（8バイト）には、以下の
!内容が記述されています。
!プロテクトモードのデータセグメントとコードセグメント（ブロック）の情報を表示します。この
!セグメントの最大値（16ビット）、リニアベースアドレス（32ビット）、特権
!レベル、インメモリー・フラグ、リード・ライト・パーミッション、その他のフラグを設定しま
!す。567--578行目をご覧ください。
135
136 end_moveです。

```

137		<code>movax, #SETUPSEG!</code> そうだ、最初はこれを忘れていた。)
138		<code>movds, ax!</code> dsはこのコードセグメントを指している (セットアップ
	ブ	
139		<code>lidtidt_48! 0,0</code> でidtを読み込む。
140		<code>lgdtgdt_48! gdtに適切な</code> ものを読み込ませる
141		

[142](#)! それは簡単だった、我々はA20を有効にする。

!1MB以上の物理メモリにアクセスして使用するためには、まず、その物理メモリを有効にする必要があります。
!A20アドレスライン。このプログラムの後にあるA20ラインの説明をご覧ください。に関しては
実際にA20のラインを使えるようにするためには、A20に入ってからテストも必要です。
保護モード（1MB以上のメモリにアクセスできるようになってから）になっています。この作業を行うことで
!!! head.Sプログラム（32〜36行）。

[143](#)

[144](#)

callempty_8042!8042のステータスレジスタをテストし、入力バッファが空 になるの
を待ちます。

!書き込みコマンドは、入力バッファが空の場合にのみ実行できます。

[145](#)

movl, #0xD1! コマンドライト !0xD1のコマンドコードを8042 のP2に書き込みます。

[146](#)

アウト#0x64, al! P2のビット1は、A20ライン のストローブに使用されます。

[147](#)

callempty_8042!cmdが受け入れられるかどうかを確認するために、バッファが空にな
るのを待ちます。

[148](#)

movl, #0xDF! A20 オン! A20ライン のパラメータ。

[149](#)

out#0x60, al! ポート0x60 に書き込みます。

[150](#)

callempty_8042! 入力バッファが空であれば、A20ラインが有効になります。

[151](#)

[152](#)!!! まあ、うまくいったと思います。今度は、割り込みを再プログラムしなければ
なりません :- ([153](#) ! 私たちは、インテルが予約したハードウェア割り込みの直後に、
割り込みを置きました。

[154](#)!!! int 0x20-0x2F. これで何も混乱することはないでしょう。悲しいことに、
IBMは初代PCでこれを台無しにしてしまい、その後も修正することができませんで
した。バイオスは 0x08-0x0f に割り込みを入れますが、これは内部ハードウェア
割り込みにも使われます。 [158](#) ! 8259のプログラムをやり直さなければならないの
ですが、それは楽しいことではありません。

[159](#)

!PCには2つのプログラマブル・インタラプト・コントローラ・チップ8259Aが使用されています。プログラミングの
ために

8259Aの方法については、この番組の後の紹介を参照してください。2つの言葉

!(0x00eb)は、162行目で定義された2つの相対ジャンプ命令で、直接

マシンコードで表現され、ディレイとして機能します。

!

!0xebは、相対的なオフセット値を持つダイレクトニアジャンプ命令のオペコードです。

!1バイトです。CPUは、この相対的なオフセットを加えて、新しい実効アドレスを作成します。

!EIPレジスタです。実行にかかるCPUクロックサイクル数は7〜10です。0x00eb

!"はジャンプオフセットが0の命令で、次の命令が実行されます。

を直接実行します。この2つの命令で、合計14〜20CPUクロックサイクルの遅延時間が発生します。

!as86では対応する命令のニーモニックがないため、Linuxは

一部のアセンブリファイルでは、この命令を機械語で直接表現しています。さらに

NOP命令1つあたりのクロックサイクル数は3なので、6〜7個のNOP命令が

同じ遅延効果を得るためには

!

!8259Aチップのマスターポートは0x20-0x21、スレーブポートは0xA0-0xA1です。出力は

! 値0x11は、ICW1コマンドである初期化コマンドの開始を示す。

エッジトリガ、複数の8259カスケードを示し、ICW4コマンドを必要とする! ワード
最後に送られるべき言葉。

[160](#)

movl, #0x11! 初期化シーケンス

[161](#)

out#0x20, al! 8259A-1 に送る。

[162](#)

.word 0x00eb, 0x00eb! jmp \$+2, jmp \$+2!'\$'は現在のアドレス です。

[163](#)

アウト#0xA0, al! と8259A-2 への

[164](#)

.word 0x00eb, 0x00eb

!Linuxシステムのハードウェア割り込み番号は、0x20から始まるように設定されています。

[165](#)

movl, #0x20! ハードウェアintの開始(0x20)

166	out#0x21, a1! ICW2のコマンドをマスターチップ	に送信。
167	.word0x00eb, 0x00eb	
168	movl, #0x28! ハードウェアint's 2 (0x28)	の開始。


```

169      アウト  #0xA1, a1          スレーブチップにICW2のコマンドを入力します。
170      .word   0x00eb, 0x00eb
171      movabl  アル, #0x04        !8259-1はマスター
172      e
173      アウト  #0x21, a1          !ICW3のcmdは、ピンIR2をスレーブチップのピンINTにチ
174      .word   0x00eb, 0x00eb      ェーンします。
175      movabl  アル, #0x02        !8259-2はスレーブ
176      e
177      アウト  #0xA1, a1          !ICW3のコマンドで、ピンINTをマスターチップのピン
178      .word   0x00eb, 0x00eb      IR2にチェーンします。
179      e
180      アウト  #0xA1, a1          !8259-2はスレーブ
181      .word   0x00eb, 0x00eb
182      e
183      アウト  #0xA1, a1          !ICW3のコマンドで、ピンINTをマスターチップのピン
184      .word   0x00eb, 0x00eb      IR2にチェーンします。
185      e
186      アウト  #0xA1, a1          !ICW3のコマンドで、ピンINTをマスターチップのピン
187      .word   0x00eb, 0x00eb      IR2にチェーンします。
188      e
189      アウト  #0xA1, a1          !ICW3のコマンドで、ピンINTをマスターチップのピン
190      .word   0x00eb, 0x00eb      IR2にチェーンします。
191      e
192      アウト  #0xA1, a1          !ICW3のコマンドで、ピンINTをマスターチップのピン
193      .word   0x00eb, 0x00eb      IR2にチェーンします。
194      e
195      アウト  #0xA1, a1          !ICW3のコマンドで、ピンINTをマスターチップのピン
196      .word   0x00eb, 0x00eb      IR2にチェーンします。
197      e
198      アウト  #0xA1, a1          !ICW3のコマンドで、ピンINTをマスターチップのピン
199      .word   0x00eb, 0x00eb      IR2にチェーンします。
200      e

```

!8086モード。これは通常のEOI、アンバッファードモードを意味し、リセットするには命令を送る必要があります
!初期化が終わり、チップの準備が整いました。

```

177      mov al, #0x01!両方とも      8086モード
178      out #0x21, al!ICW4のcmd(8086モード      )です。
179      .word 0x00eb, 0x00eb
180      out #0xA1, al! ICW4をスレーブチップ      に送る。
181      .word 0x00eb, 0x00eb
182      mov al, #0xFF! 今は      すべての割り込みをマスクオフする
183      アウト #0x21, al
184      .word 0x00eb, 0x00eb
185      アウト #0xA1, al
186

```

それは確かに楽しいものではありませんでした:-)。うまくいくといいですね。そして、188!

189 !BIOS ルーチンは多くの不要なデータを必要としており、それは 190 !
「興味深い」ものではありません。これが本物のプログラマーのやり方です。

191 !

192 !さて、いよいよ実際にプロテクトモードに移行します。出来るだけシンプルにするために、レジスタのセットアップなどは行わず、gnuでコンパイルされた32ビットプログラムに任せます19 !19 ! 32 ビットのプロテクトモードで、絶対アドレス 0x00000 にジャンプするだけです。

196

!以下では、32ビットのプロテクトモードを設定して実行に入ります。まず、マシンの状態をロードします。

! ワード (LMSW、別名コントロールレジスタCR0) のビット0がセットされていると、CPUが保護されたモードに切り替わり、特権レベル0、つまり現在の特権レベルで実行されます。

!CPL=0. セグメントレジスタは、実アドレスの場合と同じリニアアドレスを指します。

! モード (リニアアドレスは、リアルアドレスモードの物理メモリアドレスと同じ)。

!このビットをセットした後、次の命令はセグメント間ジャンプ命令でなければなりません。

CPUの現在の命令キューをフラッシュする!

!

!CPUがメモリから命令を読み込んでデコードしてから実行するので

!“という命令があります。そのため、プリフェッチされた命令のうち、リアルモードに属するものはプロテクトモードに入ってから「!’は無効になります。セグメント間ジャンプ

!!!命令は、CPUの現在の命令キューをフラッシュする、つまり、これらの

! 無効な情報です。また、インテル社のマニュアルでは、以下のように推奨されています。

80386以上のCPUでは、“mov cr0, ax ”という命令で

の保護モードです。lmsw命令は、以前の286CPUとの互換性のためだけのものです。

```

197      movabl  AX, #0x0001        プロテクトモード (PE) ビット
198      e
199      LMSW    軸                  !これだ! 」と思いました。
200      jmp i    0, 8              セグメント8(cs)のオフセット0への
201                                  ジャンプ

```

!システムモジュールを0x00000の先頭に移動させているので、オフセットアドレスの値8はプロテクトモードではすでにセグメントセクターになっています。
これは、ディスクリプターテーブルとそのエントリー、および必要な特権を選択するために使用されます。

!レベルである。セグメントセクタ8(0b0000, 0000, 0000, 1000)は、「特権」を意味します。レベル0が要求され、GDTテーブルの2番目のディスクリプター項目が使用されます。このエントリ!はベースアドレスが0であることを示している(571行目参照)、ここでのジャンプ命令はシステムモジュールでコードを実行します。

[200](#)

[201](#) !このルーチンは、キーボード・コマンド・キューが空であることをチェックします。

[202](#) !タイムアウトは使用しません。これがハングアップした場合、[203](#) !

!

!書き込みコマンドは、入力バッファが空のとき(ステータスレジスタのビット1=0)にのみ実行できます。

[204](#) 空_8042。

[205](#) .word 0x00eb, 0x00eb

[206](#) イナル、#0x64!8042 ステータスポート

[207](#) TESTAL, #2! 入力バッファがいっぱい ですか?

[208](#) jnz empty_8042! はい〜ループ

[209](#) レット

[210](#)

!なお、以下の215~566行のコードは、多くのグラフィックカードのハードウェアを含んでいます。

!という情報を持っているので、より複雑になっています。しかし、このコードは

カーネルの場合は、最初にスキップすることができます。

[211](#) !ルーチンは、SVGAボードのタイプを認識しようとします(もしあれば)

[212](#) !

[213](#) !見つかった場合、選択された解像度はal, ah (rows, cols)で示されます。

!

!次のコードは、まず588-589行目でmsg1を表示し、次にキーボードをループしています。

コントローラの出力バッファは、ユーザがボタンを押すのを待っています。ユーザーがEnterを押すとキーを押すと、システムのSVGAモードをチェックし、行と列の最大値を返します。

!をALとAHに設定します。それ以外の場合は、デフォルトのAL=25行、AH=80列が設定され、返されます。

[214](#)

[215](#) chsvga: cld

[216](#) pushds! dsを保存すると、231行目(または490行目、492行目)にポップアップ表示されます。

[217](#) pushcs! ds = cs

[218](#) ポップス

[219](#) movax, #0xc000

[220](#) moves, ax! esは0xc000セグを指します。これはVGAカードのBIOS領域です。

[221](#) leasi, msg1! ds:siは、ヌルで終端したメッセージmsg1を指しています。

[222](#) call prtstr! がmsg1を表示し

!まず、ボタンが押されたときに生成されるスキャンコードが

! メイクコード。押したボタンを離れたときに発生するスキャンコードをブレイクコードと呼ぶ。

!次のコードは、キーボードコントローラの出力バッファを読み、スキャンコードやコマンドを取得します。

!受信したスキャンコードが0x82よりも小さければ、それはメイクコードです。

ブレイクコードの最小値が 0x82 未満の場合は、ボタンが押されていないことを示す。

!をリリースしました。スキャンコードが0xe0より大きい場合は、拡張スキャンを行っていることを示します。

!コードプレフィックスを受信します。ブレイクコードが0x9cの場合、ユーザーがボタンを押した/離れたことを示します。

!Enterキーを押します。プログラムは、システムにSVGAモードがあるかどうかをチェックするためにジャンプします。それ以外の場合は

! リターンの行と列は、デフォルトではAL=25行、AH=80列に設定されています。

[223](#) nokey: in al, #0x60! コントローラのバッファからスキャンコードを読み込んでください。

[224](#) cmp al, #0x82! 最小ブレイクコード0x82と比較してください。

[225](#) jbnkey! それ以下の場合は、まだキーがリリースされていません。

[226](#) CMPAL, #0xe0

[227](#)

janokey! 0xe0より大きい場合は、コード

の接頭辞です。

[228](#)

CMPAL, #0x9c

```

229         jesvga!!! ブレークコードが0x9cの場合、エンターキーが押される/離される。
230         movax, #0x5019! それ以外の場合は、al = 25, ah = 80         を設定します。
231     ポップス
232     レット

```

!以下は、ROMの指定された位置にある機能データ文字列に基づいています。
!VGAカードのBIOSやサポートされている機能から、ディスプレイカードのブランドを判断します。
がマシンにインストールされています。プログラムは合計10個のディスプレイカード拡張に対応しています。
!220行目では、プログラムがVGAのBIOSセグメント0xc000を指していることに注意してください。
カード（第2章参照）を使用しています。

!まず、ディスプレイカードがATIのアダプタであるかどうかを確認します。
!595行目のATIカードの機能データ文字列をds:siに指定し、es:siに指定します。
!VGA BIOSの指定された場所（オフセット0x31）に設置します。この機能文字列には、合計
!9文字（"761295520"）で、特徴的な文字列をループします。もし、同じであれば
このマシンに搭載されているVGAカードはATIブランドです。そこで、ds:siに行と列のモードを指定させます。
ディスプレイカードが設定できる値dscati (615行目)は、数字のモードをdiが指すようにします。
を設定することができ、さらに設定するためにラベルselmod (438行)にジャンプします。

```

233         svga:leasi, idati!ATIの「手掛かり」をチェック
234         movdi, #0x31! フィーチャーデータは0xc000:0x0031         にあります。
235         movcx, #0x09!9バイト
236     リピート
237     cmpsb!9バイトが同じであれば、ATIカードであることがわかります。
238     jnenovati

```

!さて、アダプターのブランドはわかりました。次は、オプションのATIディスプレイカードを指定してみましょう
! 行値テーブル（dscati）、diは拡張モード番号と拡張モード番号を指す
! リスト（moati）からselmod（438行）にジャンプして処理を続けます。

```

239     リーシ、ドスカティ
240     レディ、モアティ
241     leacx, selmod
242     jmpcx

```

!それでは、Aheadブランドのディスプレイカードであるかどうかをテストしてみましょう。
!まず、EGA/VGAパターンにアクセスするためのメインイネーブルレジスタインデックス0x0fを書き込む
! インデックスレジスタ0x3ceに、オープン拡張レジスタフラグ値0x20を書き込み、0x3cf
ポート（ここではメイン・イネーブル・レジスタに相当）に接続します。メインイネーブルレジスタ
この値は、0x3cfポートから読み込まれ、拡張レジスタフラグが有効かどうかをチェックします。
を設定することができます。それが可能であれば、それはAheadブランドのカードです。なお、ワードが出力される
と
! al→ポートn、ah→ポートn+1。

```

243 noati:         movax, #0x200f!アヘッドの「手がかり」         をチェック
244         movdx, #0x3ce! データポート0x0f→0x3ceポート
245         outdx, ax! セット拡張REGフラグ: 0x20→0x3cfポート
246         incdx! でしたら、フラグが設定されているか         どうかを確認します。
247     inal, dx
248         cmpal, #0x20! 0x20ならAhead Aアダプタが見つかった。
249         jeisahed! 0x20ならAhead Bアダプタです。
250         cmpal, #0x21! Aheadアダプタでない場合はジャンプ。
251     jmenoahed

```

!これで、アダプターのブランドがわかりました。では、オプションのAheadディスプレイカードを指差してみま
しょう。
行値テーブル（dscahead）、diは拡張モード番号と拡張モードを指します。
! 番号リスト（moahead）の後、selmod (438行)にジャンプして処理を続けます。

```

252 isahed:         leasi, dscahead

```

[253](#)

リーディ、モアヘッド

```

254         leacx, selmod
255         jmpcx

```

! それでは、Chips & Tech社製のグラフィックカードであるかどうかを確認してみましょう。

! VGAイネーブルレジスタのエントリーモードフラグ（ビット4）は、ポート0x3c3（0x94または0x46e8）を介して設定されます。

! ”と表示され、ポート0x104からディスプレイカードのチップセットの識別値が読み込まれます。もし

! idが0xA5の場合、Chips & Tech社製のディスプレイカードであることを意味します。

```

256 noahed:                                movdx, #0x3c3! Chips & Techをチェックする。'clouds'
257         inal, dx! ポート0x3c3からイネーブルレジを読み込み、セットアップフラグ（ビット4
        を追加。
258         口頭、#0x10
259         アウトドックス、アル
260         movdx, #0x104! b1              に格納されているグローバルIDポート0x104からチップIDを読み込む。
261         inal, dx
262         movbl, al
263         movdx, #0x3c3! ポート0x3c3          にセットアップフラグをリセット。
264         inal, dx
265         アンダル、#0xef
266         アウトドックス、アル
267         cmpbl, [idcandt]! idcandt( 596      行目)でb1とid(0xA5)を比較します。
268         jnenocant

```

! アダプタのブランドは Chips & Tech であることがわかりました。では、オプションのカードにポイントをつけてみましょう。

! 行値テーブル（dsccandt）、diは拡張モード番号と拡張モードを指します。

! 番号リスト(mocandt)の後、selmod(438行)にジャンプして処理を続けます。

```

269         リーシ、dsccandt
270         リーディ、モカント
271         leacx, selmod
272         jmpcx

```

! では、そのカードがCirrusのディスプレイカードかどうかを確認してみましょう。

! 検知方法は、CRTコントローラのインデックス番号0x1fの内容を

拡張機能を無効にしようとするために、Eagle ID! このレジスタは、Eagle ID

!!!のレジスタです。上位と下位のニブルの値が交換され、第6レジスタに書き込まれます。

ポート0x3c4のインデックスレジスターです。の拡張機能を無効にする必要があります。

! Cirrusのディスプレイカードです。禁止されていなければ、それはCirrusのディスプレイではないということです。カードを使用しています。ポート0x3d4でインデックスされた0x1fのイーグルレジスタから読み込まれた内容がメモリストार्टアドレス上位バイトのレジスタ内容のXOR演算後の値

イーグル値と0x0cのインデックス番号に対応した「!」が表示されます。したがって、読み込む前に

0x1fの内容を保存するには、メモリストार्टハイバイトレジスタの内容を保存する必要があります。

! ”と書いてクリアし、チェック後に元に戻します。さらに、エスケープされたEagleを書き込むと

! ID値を0x3c4ポートインデックスのNo. 6シーケンス/拡張レジスタに入力すると、再度有効になります。

! ”の拡張子です。

```

273 nocant:                                movdx, #0x3d4! シーラスの「手掛かり」          をチェック
274         moval, #0x0c! REGインデックス0x0cをポート0x3d4に書き込み、mem addrを取得し
275         outdx, al!
276         incdx! ポート0x3d5からb1          までのmem addrの上位バイトを読み込みます。
277         inal, dx
278         movbl, al
279         xoral, al
280         アウトドックス、アル
281         decdx! REGインデックス0x1fをポート0x3d4に書き込み、Eagle ID          を取得。
282         moval, #0x1f

```

283	アウトドックス、アル
284	インデックス


```

285          inal, dx! ポート0x3d5からEagle IDを取得し、bh          に格納する。
286          movbh, al! ニブルを入れ替えてclに格納。左シフトでch      に格納。
287          xorah, ah! そして6番をclに。

```

```

288          シュラル、#4

```

```

289          movcx, ax

```

```

290          moval, bh

```

```

291          シュラル、#4

```

```

292          addcx, ax

```

```

293          shlcx, #8

```

```

294          addcx, #6

```

!最後にcxの値をaxに格納する。この時、ahは後の「Eagle ID」の値になります。

!”の転調、そしてalはインデックス番号6で、シーケンス/エクステンションに対応しています。

!レジスタを使用します。0x3c4のポートインデキシング・シーケンス / エクステンション・レジスタにahを書き込むと

シーラス社のグラフィックカードが拡張機能を無効にする原因となります。

```

295          movax, cx

```

```

296          movdx, #0x3c4

```

```

297          アウトドックス、アックス

```

```

298          インデックス

```

!拡張機能が本当に無効であれば、読み込まれた値は0になるはずですが、そうでなければ

これは、シーラス社のディスプレイカードではないことを意味しています。

```

299          inal, dx

```

```

300          アンダル、アル

```

```

301          jznocirr

```

!ここまで実行すると、マシンに搭載されているカードがCirrusディスプレイである可能性があります。

カードを使用します。その後、bh（286行目）に保存されている「Eagle ID」の元の値を使用して、カードを再び有効にします。

!シーラスカードの拡張機能です。読み込んだ戻り値は1でなければなりません。そうでない場合は、やはりシーラスのディスプレイカードではありません。

```

302          moval, bh!

```

```

303          outdx, al!

```

```

304          inal, dx!

```

```

305          CMPAL, #0x01

```

```

306          jnenocirr

```

!さて、これでグラフィックカードがシーラスブランドであることがわかりました。そこでまず、rst3d4を呼び出します。

CRTコントローラの表示開始アドレス上位バイトレジスタを復元するサブルーチンです。

!”というコンテンツがあれば、siはブランドディスプレイカードのオプションの行値テーブルを指します。

!(dsccurrus)の場合、diは拡張モード番号を指し、拡張モード番号リスト

!(mocirrus)と入力した後、selmod(438行目)にジャンプして設定操作を続けます。

```

307          コールスター3D4

```

```

308          リーシ、dsccirrus

```

```

309          リーディ、モシラス

```

```

310          leacx, selmod

```

```

311          jmpcx

```

!本サブルーチンでは、表示開始アドレス上位バイトのレジスタ内容を

!CRTコントローラは、blに格納された値を使用しています（278行目）。

```

312 rst3d4:          movdx, #0x3d4

```

```

313          moval, bl

```

```

314          xorah, ah

```

```

315          shlax, #8

```

```

316          addax, #0x0c

```

```

317          outdx, ax! 注、ワード出力、al→0x3d4, ah→0x3d5。

```

```

318          レット

```

!ここで、Everexのグラフィックカードがシステムに入っているかどうかを確認します。その方法は、Everexの

割り込み0x10関数0x70(ax=0x7000, bx=0x0000)の拡張ビデオBIOS関数。

!Everesタイプのディスプレイカードの場合、割り込みコールはシミュレーションに戻る必要があります。

以下のようなリターン情報があります。

! al = 0x70 (TridentベースのEverexディスプレイカードである場合)

! cl = タイプ。00-mono、01-CGA、02-EGA、03-digital multi-freq、04-PS/2、05-IBM 8514、06-SVGA。

! ch = attr:Bit7-6 :00-256K, 01-512K, 10-1MB, 11-2MB; Bit4-Enable VGA protect; Bit0-6845Simu.

! dx = ボードモデル。ビット15-4: ボードタイプID、ビット3-0: ボード補正ID。

!0x2360-Ultragraphics II; 0x6200-Vision VGA; 0x6730-EVGA; 0x6780-Viewpoint。

! di = BCDコードで表現されたビデオBIOSのバージョン番号。

```

319 nocirr:          callrst3d4!Everexの「手掛かり」をチェック
320                  movax,#0x7000! int 0x10でax=0x7000, bx=0x0000。
321                  xorbx,bx
322                  int0x10
323                  cmpal,#0x70! alにはEverexカード          用の0x70を入れてください。
324                  jnenoevrax
325                  shrdx,#4! ボードの固定番号(bit3-0)          を無視します。
326                  cmpdx,#0x678! ボードタイプが0x678であれば、Tridentカード          です。
327                  jstori
328                  cmpdx,#0x236! ボードタイプが0x236の場合、Tridentカード          でもあります。
329                  jstori
!さて、これでこのカードがEverexブランドであることがわかりました。そこでまず、siにカードの
! オプションの行値テーブル(dsceverex)は、diが拡張モード番号を指しており
! 拡張モード番号リスト(moeverex)からselmod(438行目)にジャンプして設定を続けます。
330                  lea      si,dsceverex
331                  lea      edi,moeverex
332                  lea      cx,selmod
333                  jmp      cx
334                  lea      cx,ev2tri          !TridentタイプのEverexカード、ev2triへのジャンプ
335                  jmp      cx

```

!次に、ジェノバのディスプレイカードかどうかを確認します。その方法は、機能番号の文字列を確認することです

! (0x77, 0x00, 0x66, 0x99)をビデオBIOSに設定しています。なお、現時点では、esが

! VGAカードのROM BIOSが配置されているセグメント0xc000になります。

```

336 noevrx: lea      si,idgenoa          ! ジェノバの「手掛かり」をチェック
337         xor     ax,ax          ! ds:siはフィーチャーデータを指します。
338         seg     es
339         mov     al,[0x37].      ! 0x37でVGAカードからフィーチャーデータを
         abl     e               取得。
340         mov     di,ax          ! es:diは0x37を指します。
         abl     e
341         mov     cx,#0x04
         abl     e
342         dec     si
343         dec     edi
344 11:      inc     si          ! 4つのフィーチャーバイトを比較します。
345         inc     edi
346         mov     al,(si)
         abl     e

```

```
347      セ  es
      グ
348      そ      al, (di)
      し
      て
349      cmp      アル、(シ)
350      ルーペ  ll
351      cmp      cx, #0x00
352      jne      ノーゲン
```

!さて、このカードがジェノバのカードであることがわかりました。そこで、siにそのカードの
! オプションの行値テーブル(dscgenoa)は、diが拡張モード番号を指しており
.拡張モードリスト(mogenoa)から、selmod(438行目)にジャンプして設定を続けます。

```

353     LEASI, DSCGENOVA
354     レディ、モゲノア
355     leacx, selmod
356     jmpcx

```

!パラダイス・ディスプレイ・カードであるかどうかを確認してください。機能を比較する場合も同様です
ディスプレイカードのBIOSにある「!」の文字列（「VGA=」）。

```

357 nogen:                                leasi, idparadise!パラダイスの「手掛かり」           をチェック
358                                movdi, #0x7d! es:diは0xc000:0x007d           を指しています。
359                                movcx, #0x04! 4バイトあるはずです: "VGA="
360                                リピート
361                                cmpsb
362                                jnenopara

```

!さて、このカードがParadiseカードであることはわかりました。そこで、siにカードのオプションである
!行の値のテーブル（dscparadise）では、diは拡張モード番号と拡張
!モードリスト(moparadise)からselmod(438行目)にジャンプして設定を続けます。

```

363     leasi, dscparadise
364     リーディ、モパラダイス
365     leacx, selmod
366     jmpcx

```

!ここで、Trident(TVGA)カードかどうかを確認します。TVGAディスプレイカード拡張のビット3--0

!モードコントロールレジスタ1(0x3c4ポートインデックスの0x0e)は、64Kのメモリページ値です。この
フィールドの値には特性があり、書き込み時にはまず0x02とXORする必要があります。

値を読み出す際には、XOR演算は必要ありません。つまり、値の

XOR前の値は、書き込み後に読み込んだ値と同じである必要があります。以下のコードでは

この機能を使って、Tridentのディスプレイカードであるかど

```

うかをチェックします。367                                nopara:
movdx, #0x3c4!Tridentの「手掛かり」           をチェック
368                                moval, #0x0e! 出力インデックス0x0e(モードctrl reg 1)からポート0x3c4           へ。
369                                outdx, al! ポート0x3c5から元の値を読み取り、al           に格納する。
370                                インデックス
371                                inal, dx
372                                xchgah, al

```

!そして、このレジスタに0x00を書き込み、その値を読み取ります ->al. 0x00を書き込むことは

元の値 "0x02 "または "0x02 "の後に書かれている値に対して、Tridentカードであれば、「!

!この後に読み込まれる値は、0x02になるはずです。スワップ後は、a = 元の値の

!モードコントロールレジスタ1、ah = 最後に読んだ値。

```

373     moval, #0x00
374     アウトドックス、アル
375     inal, dx
376     xchgal, ah
377     movbl, al!本編ではこれがなかったのです。
378     andbl, #0x02! 必要なのですが、           私のカードではうまくいきました。
379     jzsetb2! は三連符これがないと、画面は
380     andal, #0xfd!!!ボケた。
381     jmpclrb2!
382 setb2:                                oral, #0x02
!383 clrb2:                                outdx, al
384     andah, #0x0f! ページ番号フィールド(ビット3-0)を取得(375           行目)
385     cmpah, #0x02! 0x02と同じなら、それはTridentカード           だ。
386     jnenotrid

```

!さて、このカードがTridentカードであることはわかりました。そこで、siにカードのオプションである

! 行値テーブル (dsctrident)、diは拡張モード番号を指し、拡張
! モードリスト (motrident) から selmod (438行目) にジャンプして設定を続けます。387

```

ev2tri: leasi, dsctrident
388     リーディ、モトリデント
389     leacx, selmod
390     jmpcx

```

! ここで、Tsengカード (ET4000AXまたはET4000/W32) であるかどうかを確認してください。その方法は、リード 0x3cd のポートに対応するセグメントセレクトレジスタへの! および書き込み操作。が行われます。
レジスタの上位4ビット (ビット7~4) は、対象となる64KBのセグメント番号 (バンク番号) です。
下位4ビット (ビット3~0) が書き込み用のセグメント番号です。もし
指定されたセグメントセレクトレジスタの値が0x55 (読みと
! 6番目の64KBセグメントを書き込む)、Tsengディスプレイカードの場合は、値を
!!! レジスタはまだ0x55のはずです。

```

391 notrid:      movdx, #0x3cd! ツェンの「手掛かり」      をチェック
392             inal, dx! こんなにシンプルでいいの? :-)
393             movbl, al! 0x3cdからbl      にオリジナルのセグセクターデータを読み込む。
394             movax, #0x55! 値0x55で書き込んで、再度、ああ      に読み込んでください。
395             アウトドックス、アル
396             inal, dx
397             movah, al
398             movax, bl! 元のデータ      に戻す。
399             アウトドックス、アル
400             cmpah, #0x55! 読み取り値と書き込み値が同じなら、それはTsengカードです。
401             jnenotsen

```

! よし、このカードがTsengカードであることはわかった。そこで、siにカードのオプションを指定させます。
! 行値テーブル (dsctseng)、diは拡張モード番号を指し、拡張
! モードリスト (motseng) から、selmod (438行目) にジャンプして設定を続けます。

```

402     リーシ、dsctseng
403     リーディ、モツェン
404     leacx, selmod
405     jmpcx

```

! Video7のディスプレイカードかどうかを確認します。ポート0x3c2は混合出力レジスタ書き込みポート
! と0x3ccは混合出力レジスタのリードポートです。このレジスタのビット0は、モノ/カラーの
フラグです。これが0の場合はモノラル、それ以外はカラーを意味します。かどうかを判断する方法は
! Video7カードを使用する場合は、CRT制御拡張識別レジスタ (インデックス
! "の数字は0x1f)) このレジスタの値は、実際には、XOR演算の結果である
メモリアドレス上位バイトレジスタ (インデックス番号0x0c) の「! 」と値「0xea」が一致しています。したがっ
て
今回は、メモリスタートアドレスの上位バイトレジスタに特定の値を書き込むだけです。
! "と表示し、識別レジスタから識別値を読み出して確認します。

```

406 notsen:      movdx, #0x3cc! ビデオ7の「手掛かり」      をチェック
407             inal, dx
408             movdx, #0x3b4! dxをモノラルコントロールインデックスレジスタポート0x3b4      に設定。
409             アンドル、#0x01! ミックス出力REGのビット0が0(モノ)の場合は、そのまま      ジャンプ
             します。
410             jzeven7! それ以外の場合は、dxをカラーコントロールインデックスREGポート0x3d4
             に設定します。
411             movdx, #0x3d4
412 even7:      movax, #0x0c! インデックスを0x0cに設定して、memアドレスの上位バイトのREG
             にします。
413             アウトドックス、アル
414             インデックス

```

[415](#) inal, dx! vmemアドレスの上位バイトregを読み、bl に保存します。
[416](#) movbl, al
[417](#) movl, #0x55! その後、上位バイトのREGに0x55を書き込み、それを読み出す。

```

418      アウ      dx, al
419      ト
420      で      al, dx
421      ! 次に、インデックス番号が0x1fのVideo7ディスプレイカード識別レジスタを選択します。
422      ! CRTインデックスレジスタポート0x3b4または0x3d4を介して。このレジスタの内容は
423      ! 実際には、メモリストートアドレスと値0xeaのXORの結果です。
424      decdx
425      movbh, al
426      decdx! select addr high byte regで元の値
427      movah, #0x1f
428      アウトドックス、アル
429      インデックス
430      movah, bh
431      アウトドックス、アル

```

保存します。

decdx! select addr high byte regで元の値に

movah, #0x0c

アウトドックス、アル

インデックス

movah, bl

アウトドックス、アル

! そして、「Video7ディスプレイカード識別登録値が

! XOR演算後のメモリ開始アドレス上位バイトと0xeaの結果値」とあります。

! したがって、0x55と0xeaのXOR演算の結果は、テストの

! 識別レジスタの値を設定します。Video7カードではない場合、デフォルトの表示を設定する

! 行と列の値（492行）。それ以外はVideo7カードです。そこで、siが指すように

! ディスプレイカード列値テーブル(dscvideo7)では、diが拡張された数を指します。

! モードとモード番号のリスト（movideo7）です。

```

432      movah, #0x55
433      xorah, #0xea
434      CMPAL, BH
435      jnenovid7
436      LEASI, DSC_VIDEO7
437      リーディ、ムービデオ7

```

!上記のディスプレイカードとVideo7のディスプレイの検査と分析を通して
このカードを見ると、検査プロセスは通常3つの基本的なものに分かれていることがわかります。
この手順を説明します。1つ目は、必要なレジスタの元の値を読み取って保存することです。
テストのために、特定のテスト値を書き込みと読み出しの操作に使用して
最終的には元のレジスタ値に戻し、チェック結果を判断します。

!

!以下は、上記のコードに基づいて、ディスプレイの種類を判断したものです。

カードと、それに関連する拡張モード情報（以下に示す行と列の値のリスト）を含みます。

! by si; di は拡張モードの数とモード番号のリストを指します）、プロンプティング

ユーザーが利用可能なディスプレイモードを選択し、それに応じてディスプレイモードに設定することができます

!で設定されている画面の行と列の値を返します。

!のシステム（ah=列、al=行）。例えば、システムがATIのグラフィックカードの場合。

画面には次のようなメッセージが表示されます。

!モードです。COLSxROWS:

! 0.132 x 25

! 1.132 x 44

!対応する数字を押して、モードを選択します。

!

!以下のコードは、ヌル文字で終端された文字列 "Mode. COLSxROWS: "を画面に表示します。COLSxROWS: "を画面に表示します。

```

438 selmod:      プッシュシー
439      leasi, msg2

```


[440](#)

callpritsr

```

441      xorcx, cx
442      movcl, (di)! clはチェックしたカード          の拡張モードです。
443      ポプシ
444      プシ
445      pushcx
!そして、現在のディスプレイカードで選択可能な拡張モードの行と列は
!"と表示され、ユーザーが選択できるようになっています。
446      tbl:popbx! bx = トータル・エクステンド・モード・ナンバー。
447      プッシュボックス
448      movax, bl
449      subax, cl
450      calldprnt! 値を10進法で表示します
451      は、ドットと4つのスペースで構成されています。
452      lodsw! si で指定された行と列を ax にロードし、si++ と          します。
453      xchgal, ah!!! スワップ、al=列。
454      calldprnt! カラム番号          を表示します。
455      xchgah, al! al = rows.
456      プシックス
457      movax, #0x78! "x "          を表示します。
458      コールプリント1
459      popax! al=行番号
460      calldprnt! 行番号          を表示します。
461      カルドクル! cr, lf
462      looptbl! 次の行の列を表示し、モード番号を1つ          減らします。
!そして、"Choose mode by pressing corresponding number" というプロンプト文字列を表示します。
463      popcx! cl = トータル・エクステンド・モード・ナンバー。
464      カルドク
465      leasi, msg3!"対応する数字を押してモードを選ぶ"
466      callpritrstr

```

!続いて、キーボードポートからユーザーボタンのスキャンコードを読み取り、その行とユーザーが選択したコラムモード番号をスキャンコードに応じて決定するステップとROM BIOSのINT 0x10関数0x00を使って、対応する表示モードを設定します。

!468行目の「モード番号+0x80」は、数字キー-1のブレイクスキャンコードです。

!"が押されています。0~9の数字キーの場合、そのブレイクコードは

!0 - 0x8B, 1 - 0x82, 2 - 0x83, 3 - 0x84, 4 - 0x85。

!5 - 0x86, 6 - 0x87, 7 - 0x88, 8 - 0x89, 9 - 0x8A

!したがって、リードブレイクコードが0x82より小さい場合は、数字のスキャンコードが0x8Bの場合、ユーザーが0番のキーを押したことを意味します。

```

467      popsi! pop up original row & column table pointer.
468      addcl, #0x80! cl + 0x80 = "数字キー -1 "のブレイクコード 469 nonum:
      inal, #0x60! 早い者勝ちだな・・・。
470      cmpal, #0x82! 0x82より小さい場合は無視してください。
471      jbnonum
472      cmpal, #0x8b! スキャンコード=0x8b? ナンバーキー0です。
473      jezero
474      cmpal, cl! モードの数よりも大きい?
475      janonum! 数字キーが押され          ていない。
476      jmpnozero

```

!次に、ブレイクスキャンコードを対応するデジタルキー値に変換して

! モード番号とモード番号リストから該当するモード番号を選択する

!"の値を使用します。その後、ROM BIOS割り込みINT 0x10のファンクション0を呼び出して

画面をモード番号で指定したモードに切り替えます。最後に、モード番号を使ってディスプレイカードの行と列のテーブルから、対応する行と列を返します。
!"の価値を軸にしています。

```

477 zero:subal,#0x0a! al = 0x8b - 0x0a = 0x81
478 nozero: subal,#0x80! ユーザーが 選択したモードを取得するために0x80を
    引きます。
479 デカール! 0 からのカウント
480 xorah,ah! ディスプレイモード の設定
481 アドディ,アックス
482 incdi! diはモード番号を指します(最初 はスキップ)。
483 ブシックス
484 mov al,(di)! モード番号→al, intを呼び出してモード を設定。
485 int0x10
486 ポパックス
487 shlax,#1! mode nr x 2: 行と列のテーブル へのポインタ。
488 addsi,ax
489 lodsw! axにrowとcolumnを取得(ah = column, al = row)。
490 popds! 216行目で保存したdsを復元します。
491 レット

```

!もし、上記のようなグラフィックカードがない場合は、デフォルトの80×25を使用する必要があります。
! 標準的な行と列の値です。

```

492 novid7 ポッ ds !ここでは をサポートするコー スタンダ 80x50, 80x30
    です。 ブ ドです。 ード
493 movab AX,#0x5019
494 le
495 レッ
496 ト

```

496 !次の列に「タブ」するルーチン497
ドット(.)と4つのスペースを表示します。

```

498 spcing: mov al,#0x2e! a dot '.
499 コールプリント1
500 mov al,#0x20
501 コールプリント1
502 mov al,#0x20
503 コールプリント1
504 mov al,#0x20
505 コールプリント1
506 mov al,#0x20
507 コールプリント1
508 レット
509

```

510 !DS:SI 511でasciiz-stringを印刷するルーチン

```

512 prtstr:lodsb
513 アンダル、アル
514 jzfin
515 callprnt1! al にcharを表示する
516 jmppritrstr
517 fin:ret
518

```

519 !画面に10進数の値を表示するルーチンで、表示する値はal(0～255)で指定します。

[521](#)[522](#)

dprnt: pushax

```

523          pushcx
524          movah, #0x00
525          movcl, #0x0a
526          イディバる
527          CMPAL, #0x09
528          jbelt100
529          calldprnt
530          jmpskip10
531 lt100:    アダル, #0x30
532          コールプリント1
533 skip10:   moval, ah
534          アダル, #0x30
535          コールプリント1
536          popcx
537          ポパックス
538          レット
539

```

540 !上のルーチンの一部で、これは単にアスキー文字を印刷するだけです。

!本サブルーチンは、割り込み0x10関数0x0Eを使用して、画面に文字を書き込む

をテレックスで送信します。カーソルは自動的に次の位置に移動します。行が書かれている場合

!カーソルは次の行の先頭に移動します。画面の最終行に

が書き込まれると、画面全体が1行分スクロールします。0x07(BEL)、0x08(BS)の文字が表示されます。

!0x0A(LF)、0x0D(CR)はコマンドとして表示されません。

!を入力します。AL...文字、BH...ページ番号、BL...前景色（グラフィックモードの場合）。

```

541
542 prnt1:    ブシャックス
543          pushcx
544          movbh, #0x00! ページ番号
545          movcx, #0x01
546          movah, #0x0e
547          int0x10
548          popcx
549          ポパックス
550          レット
551

```

552 !<CR> + <LF> を印字する

```

553
554 docr:pushax
555          pushcx
556          movbh, #0x00
557          movah, #0x0e
558          moval, #0x0a
559          movcx, #0x01
560          int0x10
561          moval, #0x0d
562          int0x10
563          popcx
564          ポパックス
565          レット
566

```

!ここからがグローバルディスクリプターテーブルGDTです。このテーブルは、複数の8バイト長のディスクリプターの項目。ここでは3つのディスクリプター項目が与えられている。最初の項目は役に立たない

!(568行)ですが、必ず存在します。2番目の項目は、システムコードセグメント記述子
!(570~573行目)、3つ目はシステムデータセグメントの記述子(575~578行目)です。

[567](#) gdtです。

[568](#) .word0,0,0,0! ダミー

[569](#)

!GDTのここでのオフセットは0x08です。これはたまたまカーネルコードセクタの値です。

[570](#) .word0x07FF!8Mb - limit=2047(0--2047, so 2048*4096=8Mb)

[571](#) .word0x0000! ベースアドレス=0

[572](#) .word0x9A00! コード読み取り/実行

[573](#) .word0x00C0!

granularity=4096, 386 [574](#)

!GDTのここでのオフセットは0x10です。これはたまたま、カーネルデータセクタの値です。

[575](#) .word0x07FF!8Mb - limit=2047 (2048*4096=8Mb)

[576](#) .word0x0000! ベースアドレス=0

[577](#) .word0x9200! データのリード/ライト

[578](#) .word0x00C0!

granularity=4096, 386 [579](#)

!以下は、割り込みをロードする命令 lidt が必要とする 6 バイトのオペランドです。

! ディスクリプタテーブルのレジスタです。最初の2バイトは、IDTテーブルの限界値であり
最後の4バイトは、リニアアドレス空間におけるidtテーブルの32ビットベースアドレスです。

!CPUは保護モードに入る前に、IDTテーブルを設定する必要があるので、ここでは
最初に長さ0の空のテーブルがセットされます。

[580](#) idt_48です。

[581](#) .word0! idt limit=0

[582](#) .word0,0! idt

base=0L [583](#)

!lgdt命令でグローバルのロードに必要な6バイトのオペランドです。

! ディスクリプタテーブルのレジスタです。最初の2バイトはgdtテーブルの限界長であり
最後の4バイトは、gdtテーブルのリニアベースアドレスです。グローバルテーブルのサイズは
を2KB(0x7ff)に設定しています。8バイトごとにセグメントディスクリプターアイテムが構成されているので、8
バイトごとに

テーブルには合計256のエントリーがあります。

!4バイトリニアのベースアドレスは0x0009<<16 + 0x0200 + gdtで、0x90200 + gdtとなります。

!(シンボル gdt は、このブロックのグローバルテーブルのオフセットアドレスです。205行目を参
照してください。) [584](#) gdt_48:

[585](#) .word0x800! gdt limit=2048, 256 GDTエントリー

[586](#) .word512+gdt,0x9!

gdtのベース=0x9xxxx [587](#)

[588](#) msg1:.ascii "利用可能なSVGAモードを確認するには<RETURN>を、続行 するには他のキー
を押してください。" [589](#) db0x0d, 0x0a, 0x0a, 0x00

[590](#) msg2:.ascii "Mode: COLSxROWS:"

[591](#) db0x0d, 0x0a, 0x0a, 0x00

[592](#) msg3:.ascii "対応する番号を押し てモードを選択" [593](#)

db0x0d, 0x0a, 0x00

[594](#)

!以下は、4枚のディスプレイカードのフィーチャーデータの文字列です。

[595](#) idati:.ascii "761295520"

[596](#) idcandt:.byte0xa5! idcandtは "Chip AND

TechのID "という意味です。 [597](#) idgenoa:.byte0x77, 0x00, 0x66, 0x99

[598](#)

idparadise:.as

cii "VGA=" [599](#)

!拡張モードの数と対応するモード番号を以下に示します。

様々なディスプレイカードで使用可能な「！」が付いています。各行の最初のバイトは、ディスプレイカードの数を表します。

割り込み 0x10 で使用可能なモード番号を示す。

!!!機能0 (AH=0) です。例えば、602行目から、ATIブランドのカードの場合、2つの拡張標準モードに加えて、以下のモードが使用できます。0x23と0x33です。

```

600                                     !メーカー: Numofmodes: モード
601
602 moatiです。      . バイト    0x02,    0x23, 0x33
603 moahead:        . バイト    0x05,    0x22, 0x23, 0x24, 0x2f, 0x34
604 mocandt:        . バイト    0x02,    0x60, 0x61
605 mocirrusで      . バイト    0x04,    0x1f, 0x20, 0x22, 0x31
   す。
606 moeverex:       . バイト    0x0a    0x03, 0x04, 0x07 0x08, 0x0a 0x0b, 0x16, 0x18, 0x21, 0x40
   で                                     で
   す。                                     す。
607 mogenoaです。   . バイト    0x0a    0x58, 0x5a, 0x60 0x61, 0x62, 0x63, 0x64, 0x72, 0x74, 0x78
   で                                     です。
   す。
608 moparadiseで   . バイト    0x02,    0x55, 0x54
   す。
609 motrident:      . バイト    0x07,    0x50, 0x51, 0x52 0x57, 0x58, 0x59, 0x5a
   です。
610 motseng.        . バイト    0x05,    0x26, 0x2a, 0x23, 0x24, 0x22
611 movideo7で     . バイト    0x06,    0x40, 0x43, 0x44 0x41, 0x42, 0x45
   す。                                     です。
612

```

!以下は、様々なブランドで使用可能なモードの列と行のリストです。

VGAカードの! 615行目では、列と行の値が

ATIカードの拡張モードは、132×25と132×44の2種類です。

```

613                                     !msb =      Colslsb = Rows:
614
615                                     dscati:.word0x8419, 0x842c
616                                     dscahead:.word0x842c, 0x8419, 0x841c, 0xa032, 0x5042
617                                     dsccandt:.word0x8419, 0x8432
618                                     dsccirrus:.word0x8419, 0x842c, 0x841e, 0x6425
619 dsceverex:       .word    0x5022, 0x503c, 0x642b, 0x644b, 0x8419, 0x842c, 0x501e, 0x641b,
   0xa040,
   0x841e
620 dscgenoa:        .word    0x5020, 0x642a, 0x8419, 0x841d, 0x8420, 0x842c, 0x843c, 0x503c,
   0x5042,
   0x644b
621 dscparadiseで   .word    0x8419, 0x842b
   す。
622 dsctrident:     .word    0x501eで 0x502b, 0x503c, 0x8419 0x841e, 0x842b, 0x843c
   す。      です。
623 dsctseng.        .word    0x503cで 0x6428, 0x8419, 0x841c, 0x842c
   す。
624 dscvideo7で     .word    0x502bで 0x503c, 0x643c, 0x8419 0x842c, 0x841c
   す。      です。
625
626 .テキスト
627 endtext
628 .データ
629 enddata:
630 .bss
631 endbssです。

```

6.3.3 参考情報

このプログラムは、マシンの基本的なパラメータを取得したり、ブートプロセスのメッセージをユーザに表示するために、BIOSの割り込みサービスを複数回呼び出し、ハードウェアポートへのアクセス操作を伴うようになります。以下では、使用されているいくつかのBIOS割り込みサービスについて簡単に説明し、A20アドレスラインの問題の原因を説明します。最後に、80X86 CPUの32ビット保護モード動作の問題についても触れました。

6.3.3.1 現在のメモリーイメージ

setup.sプログラムの実行後、システムモジュールは物理メモリの先頭であるアドレス0x00000に移動し、0x90000の位置から、図6-6に示すように、カーネルが使用するいくつかの基本的なシステムパラメータが格納されます。

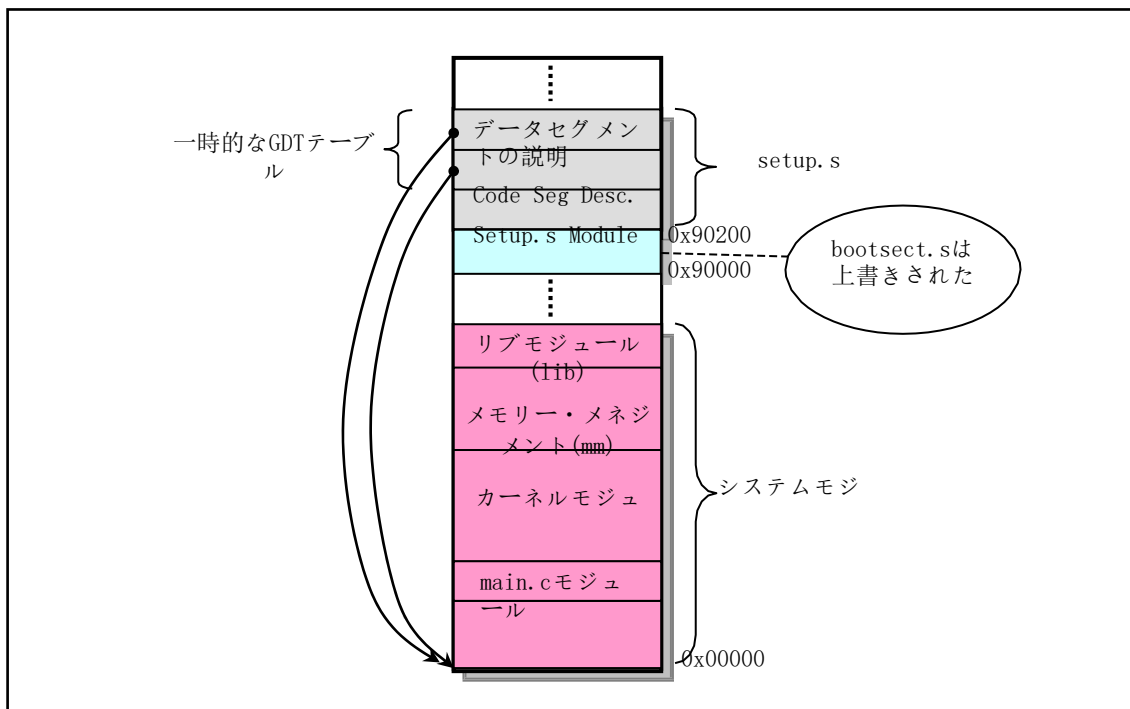


図6-6 setup.s終了後のメモリーマップの図

この時点で、一時的なグローバルテーブルGDTには3つのディスクリプターがあります。最初のものはNULLで使用されておらず、他の2つはコードセグメントとデータセグメントの記述子です。これらはすべて、システムモジュールの先頭、つまり物理メモリアドレス0x00000を指している。したがって、最後の命令「`jmp` 0,8'」（193行目）が実行されると、`head.s`のプログラムの先頭にジャンプして実行が継続されることになります。この命令の'8'はセグメントセレクターの値で、使用するディスクリプターの項目を指定するためのものです。これはGDT内のコードセグメントのディスクリプターです。'0'は記述子で指定されたコードセグメント内のオフセットです。

6.3.3.2 BIOS Video Interrupt 0x10

ここでは、上記プログラムで使用するROM BIOSのビデオ割り込みサービス機能について説明します。ディスプレイカード情報を取得する機能（その他の補助機能選択）については、表6-3を参照してください。その他の表示サービス機能については、プログラムコメントに記載しています。

表 6-3 表示カード情報の取得（機能：ah = 0x12, bl = 0x10

入力/出力	登録	説明
入力情報	ああ	ファンクション番号=0x12、ディスプレイカード情報を取得する。
	BL	サブファンクションNo.は0x10
	bh	ビデオの状態 0x00 - カラーモード（ビデオハードウェアI/Oポートのベースアドレスは0x3DX）、0x01 - モノラルモード（ビデオハードウェアI/Oポートのベースアドレスは0x3BX）、（ポートアドレスのX値は0～Fとする）。

リターン情報	BL	インストールされているビデオメモリのサイズ。 00 = 64k, 01 = 128k, 02 = 192k, 03 = 256k
	ch	フィーチャーコネクタ 一のビット情報です。ビット0～ 1Featureライン1～0、ステータス 2。 ビット2-3Feature line 1-0、Status 1。

		ビット4-7未使用（0 に設定）。
	cl	<p>ビデオスイッチの設定。</p> <p>ビット0～3は、スイッチ1～4に対応しています。</p> <p>ビット4-7は使用しません。オリジナルの EGA/VGA スwitch の設定です。</p> <p>0x00 MDA/HGC; 0x01-0x03 MDA/HGC。</p> <p>0x04 CGA 40x25; 0x05 CGA 80x25。</p> <p>0x06 EGA + 40x25; 0x07-0x09 EGA + 80x25。</p> <p>0x0A EGA + 80x25 Mono; 0x0B EGA + 80x25 Mono。</p>

6.3.3.3 ハードドライブ基本パラメータテーブル（INT 0x41

ROM BIOSの割り込みベクターテーブルでは、割り込みベクターの位置がINT 0x41（4 * 0x41 = 0x0000:0x0104）には、割込みプログラムのアドレスではなく、1枚目のハードディスクの基本パラメータテーブルのアドレスが格納されます。IBM PC完全互換機のBIOSの場合、ここに格納されているアドレスはF000h:E401hです。2台目のハードディスクの基本パラメータテーブルのアドレスは、INT 0x46の割り込みベクタ位置に格納されます。

表6-4 ハードディスクの基本パラメータ表

オフセット	サイズ	名前	説明
0x00	ワード	円筒	シリンダー数
0x02	バイト	ヘッド	ヘッドの数
0x03	ワード		書き込み電流を減らすためのスタートシリンダー（PC/XTのみ、他は0
0x05	ワード	wpcom	書き込み開始前の予備補正シリンダー数（4倍
0x07	バイト		最大ECCバーストサイズ(PC/XTのみ、他は0)
0x08	バイト	ctl	<p>コントロールバイト（ドライバのステップ選択）。</p> <p>ビット0 - 未使用 (0); ビット1 - 予約(0) (Close IRQ)</p> <p>ビット2 - リセットを許可、ビット3 - ヘッドの数が8より大きい場合に設定</p> <p>ビット4 - 未使用 (0)、ビット5 - シリンダー番号+1 に</p> <p>バッドマップがある場合に設定 ビット6 - ECCリトライの無効化、ビット7 - アクセスリトライの無効化</p>
0x09	バイト		標準のタイムアウト値（PC/XTのみ、他は0
0x0A	バイト		フォーマットのタイムアウト値（PC/XTのみ、他は0
0x0B	バイト		検出ドライブのタイムアウト値（PC/XTのみ、他は0
0x0C	ワード	lzone	ヘッドランディング（ストップ）シリンダー番号
0x0E	バイト	セクタ	トラックあたりのセクタ数
0x0F	バイト		Reserved.

6.3.3.4 A20アドレスライン問題

1981年8月、IBMの初代パーソナルコンピュータ「IBM PC」には、16ビットのCPU「インテル8088」が採用された。このCPUは、内部16ビット（外部8ビット）のデータバスと、20ビットのアドレスバス幅を持っている。そのため、このPCには20本のアド

レスライン（A0～A19）しかなく、CPUがアドレス指定できるのは1MBまでのメモリ範囲に限られている。普及機のメモリ容量が数十KBから数百KBしかなかった時代には、20本のアドレスラインで十分にアドレス指定ができた。アドレス可能な最高アドレスは0xffff:0xffffで、0x10ffefである。0x100000（1MB）を超えるメモリアドレスの場合、CPUはデフォルトで0x0ffefの位置に回り込みます。

IBMが1985年にPC/ATの新モデルを発表した際、CPUにはインテル80286が採用された。アドレスラインは24本

は、最大16MBのメモリをアドレス指定でき、8088と完全に互換性のあるリアルモードの動作を持っています。しかし、アドレス値が1MBを超えると、8088のCPUのようなアドレス回りを実装することができない。しかし、当時はこのアドレスラッピングの仕組みに対応したプログラムが存在していた。そこでIBMは、初代PCとの完全な互換性を実現するために、0x100000のアドレスビットを有効にするか無効にするかをスイッチで切り替える方法を考案した。当時のキーボードコントローラ8042には、ちょうど空きポートのピン（出力ポートP2、ピンP21）がありましたので、このピンをANDゲートとして使用し、このアドレスビットを制御しました。この信号をA20と呼びます。これが0であれば、ビット20以上がクリアされ、メモリアドレッシングの互換性が実現されたのです。その後のIBMの80X86ベースのマシンもこの機能を受け継いでいる。キーボードコントローラ8042チップの詳細については、`kernel/chr_drv/keyboard.S`プログラムの後の記述を参照してください。

互換性のため、マシンの起動時にはA20アドレスラインはデフォルトで無効になっているので、32ビットマシンのOSが適切な方法で有効にする必要があります。しかし、様々な互換機で使用されているチップセットが異なるため、これを行うのは非常に面倒です。そのため、通常は複数の制御方法の中から選択する必要があります。

A20信号線を制御する一般的な方法は、キーボードコントローラのポート値を設定することです。この典型的な制御方法は、`setup.s`プログラム（138～144行目）で使用されています。他の互換性のあるマイクロコンピュータでは、A20ラインの制御に他の方法を使用することができます。オペレーティングシステムの中には、リアルモードとプロテクトモードの動作を変換する標準的なプロセスの一部として、A20のイネーブルとディセーブルを使用するものがあります。キーボードのコントローラは非常に遅いので、キーボードのコントローラを使ってA20ラインを操作することはできません。このため、A20高速ドアオプション（Fast Gate A20）が導入された。これは、I/Oポート0x92を使用してA20信号ラインを処理するもので、低速のキーボードコントローラの操作が不要になります。キーボードコントローラのないシステムでは、0x92ポートのみを制御に使用することができます。ただし、このポートは他の互換性のあるマイコンのデバイス（ディスプレイチップなど）にも使用されている可能性があり、システムエラー動作になることがあります。別の方法としては、0xeeポートを読み込むことでA20信号線をオープンにし、ポートを書き込むことでA20信号線をディセーブルにすることができます。

6.3.3.5 8259Aインタラプトコントローラのプログラミング方法

第2章では、PC/AT互換機で採用されている割り込み機構とハードウェア割り込みサブシステムの基本的な仕組みを説明しました。ここでは、まず8259Aチップの動作原理を紹介し、その後、8259Aチップのプログラミング方法とLinuxカーネルの動作について詳しく説明します。

1. 8059Aチップの動作原理

前述したように、PC/ATシリーズ互換機では、図2-20に示すように、2つの8259Aプログラマブルコントローラ（PIC）チップをカスケード接続して、合計15個の割り込みベクターを管理しています。スレーブチップのINT端子からマスターチップのIR2端子に接続されています。マスターの8259Aのポートベースアドレスは0x20、スレーブチップは0xA0です。8259Aチップの論理ブロック図を図6-7に示します。

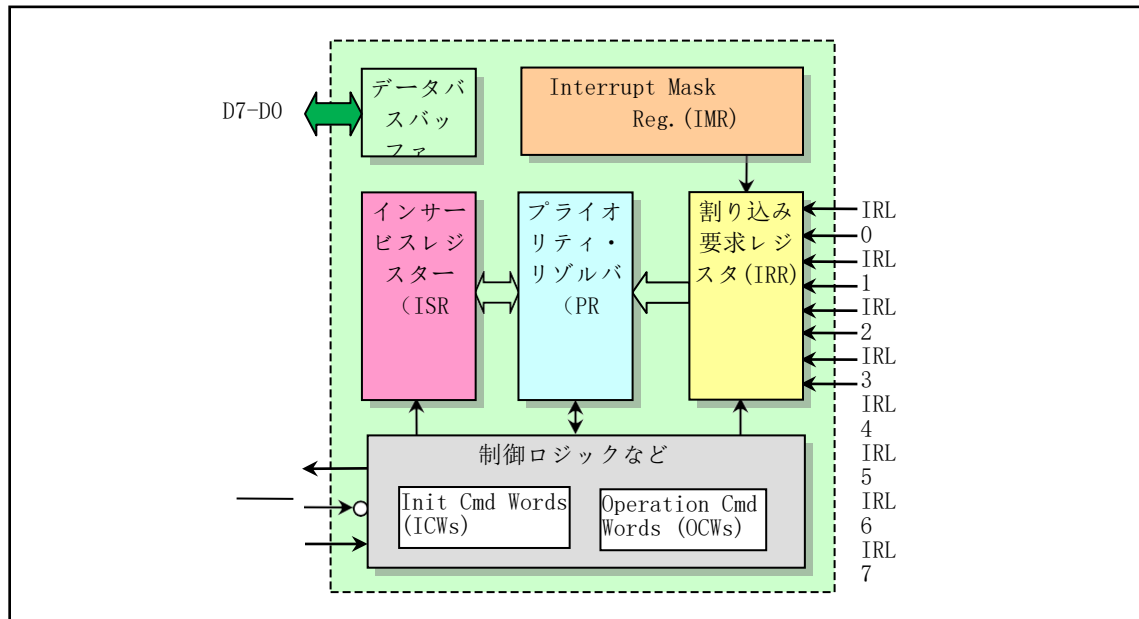


図6-7 プログラマブルインタラプトコントローラ 8259A チップダイアグラム

図では、割り込み要求レジスタ（IRR）を使用して、割り込み要求入力端子に要求されたすべてのサービス割り込みレベルを格納しています。レジスタの8ビット（D7-D0）は、ピンIR7-IR0に対応しています。割り込みマスクレジスタ（IMR）は、マスクされた割り込み要求ラインに対応するビットを格納するために使用されます。このレジスタの8ビットは、8つの割り込みレベルにも対応しています。どのビットが1にセットされるかで、どのレベルの割り込み要求がマスクされるかが決まります。つまり、IMRは、IRRの各要求ビットに対応する各ビットを処理します。優先度の高い入力ラインをマスクしても、優先度の低い割り込み要求ラインの入力には影響しません。プライオリティリゾルバ（PR）は、IRRに設定されているビットの優先順位を決定するために使用され、最も高い優先順位の割り込み要求はインサースビスレジスタ（ISR）にストローブされます。ISRは、サービスを受けている割り込み要求を保持します。コントロール・ロジック・ブロックに設定されたレジスタは、CPUが生成する2種類のコマンドを受け入れるために使用されます。8259Aが正常に動作するためには、まず初期化コマンド・ワード（ICW）レジスタの内容を設定する必要があります。その動作の中で、動作コマンドワード（OCW）レジスタを使って、いつでも8259Aの動作モードを設定・管理することができます。A0ラインは、操作のためのレジスタを選択するために使用されます。PC/ATマイコンシステムでは、A0ラインが0の場合、チップのポートアドレスは0x20（マスターチップ）と0xA0（スレーブチップ）となり、A0=1の場合、ポートは0x21と0xA1となります。

各デバイスからの割り込み要求ラインは、8259AのIR0-IR7割り込み要求端子に接続されています。これらの端子に1つ以上の割り込み要求信号が到着すると、割り込み要求レジスタIRRの対応するビットがセットされ、ラッチされます。このとき、割り込みマスクレジスタIMRの対応するビットがセットされていれば、対応する割り込み要求は優先パーサに送られません。マスクされていない割り込み要求が優先度リゾルバに送られた後、最優先の割り込み要求が選択されます。この時点で、8259AはCPUにINT信号を送り、CPUは現在の命令を実行した後、割り込み信号に応答するために、8259AにINTAを返します。応答信号を受信した後、8259Aは選択された最優先の割り込み要求をサービスレジスタISRにセーブし、すなわちISRの割り込み要求レベルに対応するビットをセットします。同時に、割り込み要求レジスタIRRの対応するビットがリセットされ、割り込み要求の処理が開始されること

を示します。

その後、CPUは2回目のINTAパルス信号を8259Aに送りますが、これは8259Aに割り込み番号を送るように知らせるためのものです。したがって、パルス信号の間、8259Aは割り込み番号を表す8ビットのデータをデータバスに送り、CPUが読み出せるようにします。

この時点で、CPUの割り込み期間が終了します。8259AがAEOI(Automatic End of Interrupt)モードを使用している場合、2つ目のINTAパルス終了時のサービスレジスタISR内のカレントサービス割り込みビットがリセットされます。そうでない場合、もし8259Aが非自動終了モードであれば、割り込みサービス・ルーチンの終了時に、プログラムはISRのビットをリセットするために8259AにEOI (End of Interrupt) コマンドを送る必要があります。割り込み要求が接続された2番目の8259Aチップから来る場合は、EOIコマンドを両方のチップに送る必要があります。その後、8259Aは次に優先度の高い割り込みをチェックし、上記の処理を繰り返します。以下では、初期化コマンドワードと動作コマンドワードのプログラミング方法を説明し、さらにそこでの動作方法を説明します。

2. 初期化コマンドワードプログラミング

プログラマブルコントローラ8259Aは、主に4つの動作モードを持っています。(1)フルネストモード、(2)回転優先モード、(3)特殊マスクモード、(4)プログラムポーリングモードです。8259Aをプログラミングすることで、現在の8259Aの動作モードを選択することができます。プログラミングは2つのフェーズに分かれています。1つ目は、8259Aが動作する前に、各8259Aチップの4つの初期化コマンドワード(ICW1 - ICW4)レジスタのプログラミングを書き込み、2つ目は、8259Aの8つの動作コマンドワード(OCW1 - OCW3)を、動作中にいつでもプログラミングすることです。初期化後は、いつでも動作コマンドワードの内容を8259Aに書き込むことができます。以下、8259Aの初期化コマンドワードのプログラミング動作について説明します。

初期化コマンド・ワードのプログラミング動作フローを図6-8に示します。この図からわかるように、ICW1とICW2の設定が必要です。ICW3は、システムに複数の8259Aチップが含まれ、接続されている場合のみ設定が必要です。これはICW1の設定に明記しておく必要があります。また、ICW4を設定する必要があるかどうか、ICW1に明記する必要があります。

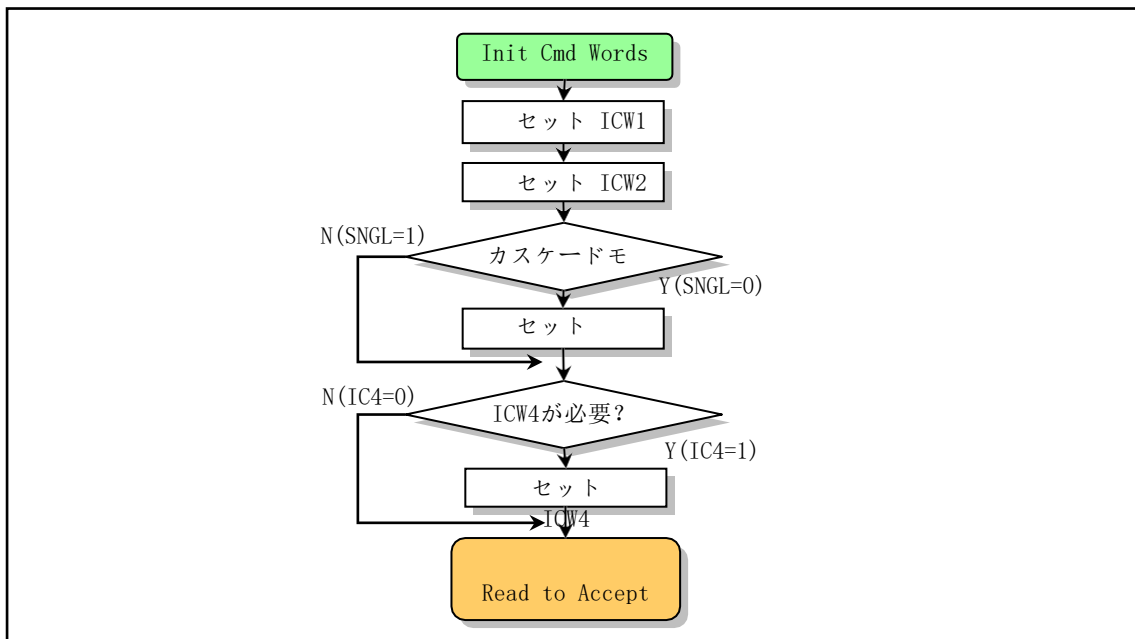


図 6-8 8259A 初期化シーケンス

(1) ICW1

送信されたバイト5ビット目(D4)=1、アドレスラインA0=0の時、ICW1がプログラムされていること

を示します。この時、PC/ATマイコンシステムのマルチチップカスケードの場合、8259Aメインチップのポートアドレスは0x20、スレーブチップのポートアドレスは0xA0となります。ICW1のフォーマットを表6-5に示す。

表6-5 割り込み初期化コマンドワード ICW1

ビット	名前	説明
D7	A7	A7-A5は、MCS80/85で割り込みサービス処理に使用されるページスタートアドレスを示します。これらはICW2ではA15-A8と組み合わせられます。これらは8086/88では使用されません。
D6	A6	
D5	A5	
D4	1	常に1
D3	LTIM	1 - レベルトリガの割り込みモード、0 - エッジトリガの割り込みモード。
D2	ADI	CALL命令のアドレス間隔に使用するMCS80/85です。8086/88では使用されていません。
D1	SNGL	1 - シングル8259A; 0 - カスケードモード。
D0	IC4	1 - ICW4が必要、0 - 必要ない。

Linux

0.12カーネルでは、ICW1は0x11に設定されています。これは、割り込み要求がエッジトリガされ、8259Aの複数のスライスがカスケード接続され、最後にICW4を送信する必要があることを示しています。

(2) ICW2 この初期化コマンドワードは、チップが送信する割り込み番号の上位5ビットを設定するために使用されます。ICW1が設定された後、A0=1の時にICW2が設定されたことを示す割り込み番号です。この時、PC/ATマイコンシステムのマルチチップカスケードの場合、8259Aメインチップのポートアドレスは0x21、スレーブチップのポートアドレスは0xA1となります。ICW2のフォーマットを表6-6に示す。

表6-6 インタラプト初期化コマンドワード ICW2

A0D7D6D5D432D1D0								
1	A15/T7	A14/T6	A13/T5	A12/T4	A11/T3	A10	A9	A8

MCS80/85システムでは、D7-D0ビットで示されるA15-A8とICW1で設定されるA7-A5で割り込みサービスプログラムのページアドレスを形成します。8086/88プロセッサを使用したシステムまたは互換システムでは、T7-T3が割り込み番号の上位5ビット、8259Aチップが自動的に設定する下位3ビットが8ビットの割り込み番号を形成します。8259Aが2回目の割り込み応答パルスINTAを受信すると、CPUが読めるようにデータバスに送信されます。

Linux 0.12システムでは、メインスライスのICW2が0x20に設定されており、メインチップの割り込み要求が0レベルであることを示しています-第7レベルの対応する割り込み番号範囲は0x20-0x27です。スレーブスライスのICW2は0x28に設定され、8レベルから15レベルのスレーブ割り込み要求に対応する割り込み番号の範囲が0x28-0x2fであることを示します。

(3) ICW3 複数の 8259A チップをカスケード接続した場合に、8ビットのスレーブレジスタをロードするためのコマンドワードです。ポート・アドレスは上記と同じです。ICW3のフォーマットを表6-7に示します。

表6-7 割り込み初期化コマンドワード ICW3

	A0	D7	D6	D5	D4	D3	D2	D1	D0
マスター	1	S7	S6	S5	S4	S3	S2	S1	S0
スレーブ	1	0	0	0	0	0	ID2	ID1	ID0

マスターチップのビットS7_S0は、カスケード接続されたスレーブに対応しています。どのビットが1であるかは、マスターの割り込み要求端子IRの信号がスレーブからのものであることを意味し、そうでない場合は、対応するIR端子がスレーブに接続されていないことを意味します。ID2_ID0のスレーブチップビットは、スレーブチップの識別番号、つまりマスターチップに接続されている割り込みレベルに対応しています。スレーブが自身のID2 - ID0と等しいカスケードライン（CAS2 - CAS0）の入力値を受信した場合、そのスレーブが選択されたことを意味します。このとき、スレーブは、現在選択されている割り込み要求の割り込み番号を、スレーブチップからデータバスに送信する必要があります。

Linux 0.12カーネルは、8259AメインチップのICW3を0x04、つまりS2=1に設定し、残りのビットを0にします。マスターチップのIR2端子がスレーブチップに接続されていることを表します。スレーブチップのICW3は0x02、つまり識別番号は2に設定されており、スレーブチップからのIR2端子がメインチップに接続されていることを表します。したがって、割り込みの優先順位は、レベル0が最も高く、次にチップのレベル8～15、最後にレベル3～7の順になります。

(4) ICW4 IC0ビット0（IC4）がセットされていると、ICW4が必要であることを示します。アドレスラインA0=1となります。ポートアドレスは上記と同じです。ICW4のフォーマットを表6-8に示します。

表 6-8 インタラプト初期化コマンドワード ICW4

ビット(s)	名前	説明
D7-5		常に0
D4	SFNM	1 - 特別なフルネストモード、0 - 特別なフルネストモードではない。
D3	BUF	1 - バッファモード、0 - アンバッファモード。
D2	M/S	1 - バッファードモード/スレーブ、0 - バッファードモード/マスター。
D1	AEOI	1 - オートエンドオブインタラプトモード、0 - ノーマルエンドオブインタラプトモード。
D0	μ PM	1 - 8086/88プロセッサシステム、0 - MCS80/85システム。

Linux

0.12コアが8259Aのマスターチップとスレーブチップに送るICW4コマンドワードの値は0x01です。8259Aチップが通常の完全ネスト型、アンバッファ型、非自動終了割り込みモードに設定されていることを示し、8086とその互換システムで使用されている。

3. 操作コマンド・ワード・プログラミング

初期化コマンドワードレジスタを8259Aに設定した後、チップはデバイスからの割り込み要求信号を受信できる状態になります。ただし、8259Aの動作中は、動作コマンドワードOCW1-OCW3を使用して、8259Aの動作状態を監視したり、初期化時に設定した8259Aの動作モードを変更することもできます。

(1) OCW1 割り込みマスクレジスタIMRのリード/ライトに使用するオペレーションコマンドワードです。アドレスラインA0は1にする必要があります。ポートアドレスの記述は上記と同様です。OCW1のフォーマットを表6-9に示す。

表6-9 インタラプト操作コマンドワード OCW1

A0D7D6D5D432D1D0

1	M7	M6	M5	M4	M3	M2	M1	M0
---	----	----	----	----	----	----	----	----

D7-D0ビットは8つの割り込み要求に対応しており、7レベル-0レベルのマスクビットM7-M0があります。M=1の場合は、対応する割り込み要求レベルがマスクされ、M=0の場合は、対応する割り込み要求レベルが許可されます。また、高優先度をマスクしても、他の低優先度の割り込み要求には影響しません。

Linux 0.12カーネルの初期化プロセスでは、コードが操作コマンドワードを使って

デバイスドライバの設定後に、該当する割り込み要求マスクビットを取得します。例えば、フロッピーディスク・ドライバの初期化の最後に、フロッピー・デバイスが割り込み要求を発行できるようにするために、ポート0x21を読み出して8259Aチップの現在のマスク・バイトを取得します。次に、AND ~0x40演算で、対応するフロッピーディスクコントローラに接続されている割り込み要求6のマスクビットをリセットします。最後に割り込みマスクレジスタに書き戻す。kernel/blk_drv/floppy.c プログラムの 461 行目を参照してください。

(2) OCW2は、EOIコマンドの送信や、割り込み優先の自動回転モードの設定に使用されます。ビットD4D3=00のとき、アドレスラインA0=0は、OCW2がプログラムされていることを示す。操作コマンドワードOCW2のフォーマットを表6-10に示す。

表6-10 インタラプト操作コマンドワード OCW2

ビット(s)	名前	説明
D7	R	プライオリティーローテーションの状態
D6	SL	優先順位設定フラグ。
D5	EOI	Non-Automatic End of Interruptフラグ。
D4-3		常に0
D2	L2	L2 -- L0 - 3ビットでレベル番号を構成し、割り込み要求レベルIRQ0--IRQ7 (またはIRQ8-IRQ15)に対応します。
D1	L1	
D0	L0	

ビットD7～D5の組み合わせの役割と意味を表6-11に示す。が付いているものは、L2--L0を設定することでISRをリセットするための優先順位を指定したり、現在の最下位の優先順位となる特殊な回転優先順位を選択したりすることができる。

表6-11 OCW2ビットD7--D5の組み合わせの意味

R(D7)	SL(D6)	EOI(D5)	説明	タイプ
0	0	1	非特定のEOIコマンド（完全入れ子モード）。	割り込みの終了
0	1	1	特定のEOIコマンド（完全に入れ子になっていない）。	
1	0	1	不特定多数のEOIコマンドで回転する。	自動回転
1	0	0	自動EOIモード（セット）で回転させます。	
0	0	0	自動EOIモード(Clean)で回転させる。	
1	1	1	特定のEOIコマンドで回転します。	特定の回転
1	1	0	Set priorityコマンド。	
0	1	0	操作はありません。	

Linux

0.12カーネルは、割り込み処理の終了前に、オペレーショナル・コマンド・ワードのみを使用して、終了割り込みEOIコマンドを8259Aに送信しています。使用されるOCW2の値は0x20で、フルネストモードでの非特殊な終了割り込みEOIコマンドを示しています。

(3) OCW3は、スペシャルマスクモードとリードレジスタのステータス（IRRとISR）の設定に使用されます。D4D3=01、アドレスラインA0=0の場合、OCW3がプログラムされている（リード／ライト）ことを意味します。ただし、この操作コマンドワードは、Linux 0.12カーネルでは使用されていない。OCW3のフォーマットを表6-12に示す。

表6-12 インタラプト操作コマンドワード OCW3

ビット	名前	説明
D7		常に0
D6	ESMM	特別なマスクモードで操作する。
D5	SMM	D6 -- D5: 11 - スペシャルマスクの設定; 10 - スペシャルマスクのリセット; 00,01 - 何もしない。
D4		常に0
D3		常に1
D2	P	1 - ポーリングコマンド、0 - ポーリングコマンドなし。
D1	RR	次のRDパルスでレジスタの状態を読み出すコマンド。
D0	RIS	D1 -- D0: 11 - Read In Service Reg. ISR; 10 - Read Interrupt Reg. IRR

4. 8259A動作モード説明

8259Aの初期化コマンド・ワードと動作コマンド・ワードのプログラミング・プロセスでは、いくつかの作業方法が言及されています。以下では、8259Aチップの動作をよりよく理解するために、いくつかの一般的な方法について詳しく説明します。

(1) フルネストモード

初期化後、操作コマンド・ワードで8259Aの動作を変更しない限り、自動的にこの完全ネストモードに入ります。このモードでは、割り込み要求の優先順位はレベル0からレベル7（レベル0が最も高い）の順になります。CPUが割り込みに応答すると、最も優先度の高い割り込み要求が決定され、その割り込み要求の割り込み番号がデータバスに置かれます。また、割り込みサービスレジスタISRの対応するビットがセットされ、そのセット状態は割り込みサービス手順から戻る前に割り込み終了EOIコマンドが送られるまで維持されます。また、ICW4で自動割り込み終了AEOIビットが設定されている場合、CPUが発行する2回目の割り込み応答パルスINTAの終了エッジでISRのビットがリセットされます。ビットがセットされているISR中は、同じ優先度の割り込み要求や低い優先度の割り込み要求はすべて一時的に無効になりますが、高い優先度の割り込み要求は応答して処理することができます。さらに、割り込みマスクレジスタIMRの対応するビットは、それぞれ8レベルの割り込み要求をマスクすることができますが、いずれか1つの割り込み要求をマスクしても、他の割り込み要求の動作には影響しません。最後に、コマンド・ワード・プログラミングの初期化後は、8259A端子のIR0が最も優先度が高く、IR7が最も優先度が低くなります。Linux 0.12カーネルコードは、本機の8259Aチップがこのモードに設定された状態で動作します。

(2) EOI（End of Interrupt）方式

上述したように、サービスレジスタISRで処理中の割り込み要求に対応するビットは、2つの方法でリセットすることができます。1つは、ICW4の自動割り込み終了ビットAEOIがセットされているときに、CPUが発行する2つ目の割り込み応答パルスINTAのエンドエッジによってリセットする方法です。この方法をAEOI（Automatic End of Interrupt）方式といいます。もう1つは、割り込みサービスプロセスから戻る前に、割り込みをリセットするための割り込み終了EOIコマンドを送信する方法です。この方法をEOI（End of Program Interrupt）方式といいます。カスケードシステムでは、スレーブ割り込みサービスルーチンは、スレーブチップ用とマスターチップ用の2つのEOIコマンドを送信する必要があります。

プログラムがEOIコマンドを発行するには、2つの方法があります。1つは特別なEOIコマンド、

もう1つは非特別なEOIコマンドと呼ばれています。スペシャルEOIコマンドは、非フルネストモードで使用され、EOIコマンドの特定のリセットのための割り込みレベルビットを指定することができます。つまり、特別なEOIコマンドをチップに送る際には、リセットISRで優先順位を指定する必要があります。特殊なEOIコマンドは、OCW2を用いて送信され、上位3ビットが011、下位3ビットが優先度の指定に使用されます。この特別なEOIコマンドは、現在のLinuxシステムで使用されています。の非特殊なEOIコマンドは、以下のようになります。

完全にネストされたモードでは、サービスレジスタISR内の現在の最高優先度ビットが自動的にリセットされます。なぜなら、完全入れ子モードでは、ISRの最高優先ビットは間違いなく最後のレスポンスとサービスの優先度だからです。また、OCW2を使用して送信されますが、最上位3ビットは001にする必要があります。この特別ではないEOIコマンドは、本書で取り上げているLinux 0.12のシステムで使用されています。

(3) 特殊フルネストモード

ICW4に設定されている特別なフルネスティングモード(D4=1)は、主に大規模なカスケードシステムで使用され、各スレーブチップにおける優先順位を保存する必要があります。この方法は、前述の通常のフルネスティングと同様ですが、以下の2つの例外があります。

A. スレーブチップからの割り込み要求を処理しているとき、スレーブチップはマスターチップの優先順位によって除外されません。そのため、チップから発行された他のより優先度の高い割り込み要求はマスターチップに認識され、マスターチップは直ちにCPUに割り込みを発行します。上記従来のフルネスティングモードでは、スレーブの割り込み要求を処理しているときには、スレーブチップはマスターチップによってマスクされる。そのため、スレーブチップから発行されたより優先度の高い割り込み要求は処理できません。

B. 割り込みサービスルーチンを終了する際、プログラムは現在の割り込みサービスがスレーブチップから発行された唯一の割り込み要求であるかどうかをチェックする必要があります。確認方法は、まずスレーブチップに非特殊割り込みEOIコマンドを発行し、その後スレーブチップのサービスレジスタISRの値を読み取ります。このとき、値が0になっているかどうかを確認します。0であれば、メインチップに非特殊なEOIコマンドを送ることができることを意味します。0でない場合は、メインチップにEOIコマンドを送る必要はありません。

(4) カスケードモード方式

8259Aはマスターチップと複数のスレーブチップに簡単に接続できます。8個のスレーブチップを使用した場合、最大64個の割り込み優先度を制御することができます。マスターチップは、カスケード接続された3つのラインを通じてスレーブを制御します。この3本のカスケードラインは、チップからのチップ選択信号に相当します。カスケードモードでは、スレーブチップの割り込み出力は、マスターチップの割り込み要求入力端子に接続されます。チップからの割り込み要求ラインが処理されて応答すると、マスターチップはスレーブチップを選択して、対応する割り込み番号をデータバスに配置します。

カスケード接続されたシステムでは、各8259Aチップは独立して初期化される必要があります、異なる方法で動作することができます。また、マスターチップとスレーブチップの初期化コマンドワードICW3は別々にプログラムされています。また、動作時には、メインチップ用とスレーブチップ用の2つの割り込み終了EOIコマンドを送信する必要があります。

(5) 自動回転優先モード

同じ優先度のデバイスを管理している場合、OCW2を使って8259Aチップを自動回転優先モードにすることができます。つまり、あるデバイスがサービスを受けた後、そのデバイスの優先順位は自動的に最下位になります。優先度は周期的に順番に変更されます。最も好ましくない状況は、割り込み要求が来たときに、サービスを受けるまでに7つのデバイスを待つ必要があることです。

(6) インタラプトマスクモード

割り込みマスクレジスタ(IMR)は、各割り込み要求のマスクを制御します。8259Aは2つのマ

スク方法に設定できます。一般的な通常のマスキングの場合、OCW1を使用してIMRを設定します。IMRビット(D7--D0)は、それぞれの割り込み要求ピンIR7～IR0に適用されます。割り込み要求をマスキングしても、他の優先度の高い割り込み要求には影響しません。この通常のマスキングモードでは、8259Aは応答・サービス中（EOIコマンド送信前）の割り込み要求に対して、すべての低優先度割り込み要求をマスクします。しかし、いくつかのアプリケーションでは、割り込みサービスプロセスがシステムの優先度を動的に変更する必要がある場合があります。この問題を解決するために、8259Aでは特別なマスキング方法が導入されました。OCW3を使って、まずこのモード（D6、D5ビット）を設定する必要があります。この特別なマスキングモードでは、OCW1で設定されたマスキング情報により、マスクされていないすべての優先順位の割り込みが割り込み中に応答されます。

(7) レジスタステータスの読み出し

8259Aには、CPUの状態を読み取るための3つのレジスタ（IMR、IRR、ISR）があります。IMRの現在のマスク情報は、OCW1を直接読み出すことで得られます。IRRまたはISRを読み出す前に、まずOCW3を使ってIRRまたはISRの読み出しコマンドを出力する必要があります。

6.4 head.s

6.4.1 機能説明

head.sプログラムは、オブジェクトファイルにコンパイルされた後、カーネル内の他のプログラムのターゲットファイルと一緒にシステムモジュールにリンクされ、システムモジュールの先頭に配置される。これがヘッドプログラムと呼ばれる所以である。システムモジュールは、ディスク上のセクタアップモジュールの後、ディスク上の第6セクタから始まるセクタに配置される。通常の場合、Linuxのシステムモジュールは

0.12カーネルのサイズは約120KBなので、ディスクの約240セクタを占めています。

これ以降、カーネルは完全にプロテクトモードで動作します。heads.sのアセンブリファイルは、これまでのアセンブリ構文とは異なります。AT&Tのアセンブリ言語フォーマットを使用しており、コンパイルとリンクにはGNUのgasとgldが必要です。そのため、コード中の代入の方向が左から右になっていることに注意してください。

このプログラムは、実際にはメモリの絶対アドレス0の先頭にあります。まず、各データセグメントレジスタをロードし、合計256項目の割り込みディスクリプターテーブルIDTを再構築し（boot/head.s, 78）、各エントリをエラー報告のみのダミー割り込みサブルーチンignore_intを指すようにします。このダミー割り込みベクターは、デフォルトの「割り込み無視」プロセスを指します（boot/head.s, 150）。割り込みが発生したときに、割り込みベクターが設定されていないと「Unknown interrupt」というメッセージが表示されます。ここで256項目すべてを設定すると、一般保護フォルト（例外13）の発生を防ぐことができます。そうしないと、IDTが256項目未満に設定されている場合、CPUは一般保護フォルト（例外13）を発生させる際に必要な割り込みで指定されたディスクリプターエントリが、設定されている最大ディスクリプターエントリよりも大きい場合。また、ハードウェアに問題があり、データバスにデバイスベクターが置かれていない場合、CPUは通常、データバスからベクターとしてオール1（0xff）を読み出すため、IDTテーブルの256番目の項目を読み出すことになります。そのため、ベクターが設定されていないと一般保護エラーも発生します。

システムで使用する必要のある一部の割り込みについては、カーネルは、その初期化（init/main.c）の過程で、これらの割り込みの割り込み記述子項目を設定し、対応する実際の割り込みハンドラ手続きを指し示します。通常、例外割り込みハンドラ（int0 -- int 31）はtraps.cの初期化関数（kernel/traps.c, 185行目）で再インストールされ、システムコール割り込みint 0x80はスケジューラの初期化関数（kernel/sched.c, 417行目）でインストールされます。

割り込みディスクリプターテーブルIDTの各ディスクリプター項目も8バイトを占めており、そのフォーマットは図6-9のようになっています。ここで、Pはセグメント・プレゼンス・フラグ、DPL

はディスクリプターの優先度です。

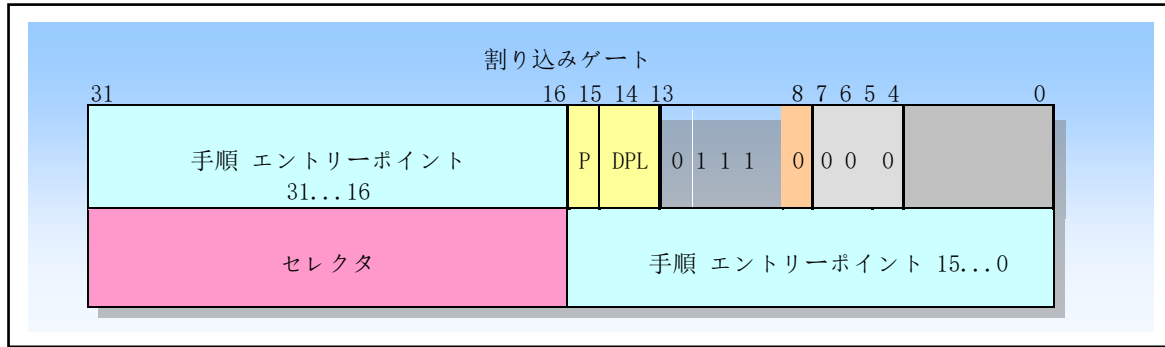


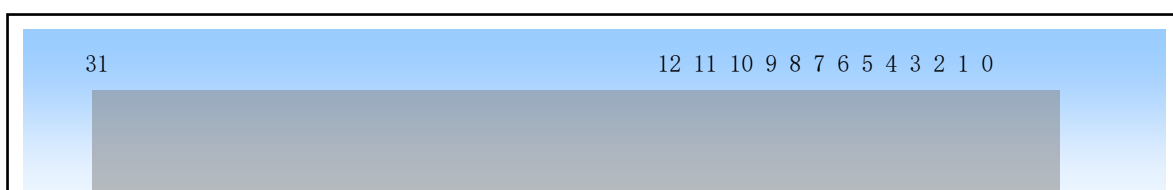
図6-9 IDTにおけるインタラプトゲート記述子のフォーマット

head.sプログラムでは、割り込みゲートディスクリプタのセグメントセクタフィールドが0x0008に設定されており、ダミーの割り込みサービスルーチンignore_intがカーネルコード内にあることを示しています。また、オフセットはhead.sプログラムのignore_int割り込みサービスハンドラのオフセットに設定されています。head.sプログラムはメモリアドレス0の先頭に移動しているため、割り込みサービスハンドラのオフセットもカーネルコード内のオフセットとなります。カーネルコードセグメントは常にメモリ内にあり、特権レベルは0（つまり、P=1、DPL=00）なので、図から割り込みゲートディスクリプタのバイト5とバイト4の値は0x8E00であることがわかります。

割り込みディスクリプターテーブルの設定後、プログラムはグローバルセグメントディスクリプターテーブルGDTを再構築します。実際には、新しく作成されたGDTテーブルのディスクリプターは、元のGDTテーブルのディスクリプターとあまり変わりません。制限値の違い（当初は8MB、現在は16MB）を除いて、その他の内容は全く同じです。そのため、setup.sでディスクリプターのセグメント制限を直接16MBに設定し、オリジナルのGDTテーブルをメモリ内の適切な位置に直接移動させることも可能です。つまり、ここで新たにGDTを再作成する主な理由は、GDTテーブルをカーネル空間の再利用可能な場所に置くためです。以前に設定したGDTテーブルは、メモリ上の0x902XXの位置にあります。この場所は、カーネルが初期化された後、メモリキャッシュの一部として使用されます。

そして、A20のアドレスラインがオンになっているかどうかを検出します。その方法は、メモリアドレス0から始まる内容と、アドレス1MBから始まる内容を比較することです。A20ラインが有効になっていない場合、CPUは1MB以上の物理メモリをアクセスする際に、アドレス（アドレスMOD 1MB）のコンテンツをサイクリックにアクセスします、つまり、アドレス0から始まる対応するバイトをアクセスするのと同じです。プログラムがオープンしていないことを検出した場合、無限ループに入ります。そうでなければ、プログラムはPCに数学コプロセッサチップ（80287、80387またはその互換チップ）が搭載されているかどうかをテストし続け、制御レジスタCR0に対応するフラグを設定します。

次にhead.sのコードは、メモリを管理するためのページング機構を設定し、ページディレクトリテーブルを物理アドレス0（このプログラムが置かれているメモリ領域でもあるので、実行が終了したコード領域は上書きされます）の先頭に配置します。その直後に、合計16MBのメモリをアドレス可能な4つのページテーブルを配置し、そのエントリを個別に設定する。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のフォーマットを図6-10に示す。ここで、Pはページ存在フラグ、R/Wはリード/ライトフラグ、U/Sはユーザー/スーパーユーザーフラグ、Aはページビットフラグ、Dはページ内容変更フラグ、左端の20ビットはページエントリに対応するメモリ内のページアドレスの上



位20ビットである。



図6-10 ページディレクトリとページテーブルのエントリ構造

ここでは、各エントリの属性フラグが0x07 (P=1, U/S=1, R/W=1) に設定されており、そのページが存在し、ユーザが読み書きできることを示している。カーネルページテーブルの属性をこのように設定する理由は、セグメンテーションとページング管理の両方に保護方法があるからである。ページディレクトリやページテーブルエントリに設定されている保護フラグ (U/S, R/W) は、セグメントディスクリプターにある特権レベル (PL) の保護と組み合わせる必要があります。しかし、セグメントディスクリプターのPLが大きな役割を果たします。CPUはまずセグメント保護をチェックし、次にページ保護をチェックします。現在の特権レベルCPL<3 (例えば0) の場合、CPUはスーパーバイザとして動作しています。この時点では、すべてのページにアクセスでき、自由に読み書きができます。CPL=3の場合、CPUはユーザー (利用者) として動作している。この時点では、ユーザー (U/S=1) に属するページのみがアクセス可能であり、読み取り可能・書き込み可能とマークされたページ (W/R=1) のみが書き込み可能です。このとき、スーパーユーザーに属するページ (U/S=0) は、書き込みも読み出しもできません。カーネルコードは少し特殊で、タスク0とタスク1のコードとデータが含まれているからです。そのため、ここでページプロパティを0x7に設定すると、2つのタスクはユーザーモードで実行できますが、任意にカーネルリソースにアクセスすることはできません。

最後に、head.sプログラムはreturn命令を使って、スタック上にあらかじめ置かれていた/init/main.cプログラムのエントリーアドレスをポップアウトさせ、実行権をmain()コードに移します。

6.4.2 コードコメント

プログラム 6-3 linux/boot/head.s

```

1 /*
2  * linux/boot/head.s
3  *
4  * (C) 1991 Linus Torvalds 5
5  */
6
7 /*
8  * head.s には、32 ビットの起動コードが含ま
9  * れています。9
10 注意!!! スタートアップは絶対アドレス0x00000000で行われます。
11 ページディレクトリが存在します。起動時のコードは次のように上書きされます。
12 * ページのディレクト
13 * リです。13 */
14 .テキスト
15 .globl _idt, _gdt, _pg_dir, _tmp_floppy_area
16 _pg_dir: # ページのディレクトリがここに 格納されます。

```

もう一度注意してください!!! これはすでに32ビットモードになっているので、0x10はディスクリプターのセクタ # になり、命令は対応するディスクリプターの内容をセグメント # レジスターにロードする。ここで、0x10の意味は、要求特権レベルRPLが0 (ビット0-1=0)、# グローバルディスクリプターテーブルGDTを選択 (ビット2=0)、テーブルの2番目の項目を選択 # (ビット3-15=2)。テーブル内のデータセグメント記述子の項目を指しているだけです (参照

setup.sの575～578行目に記述子の具体的な値があります)。)#
以下のコードは、setup.sでセクタ=0x10で構築されたカーネルデータセグメントに # ds, es, fs, gsをセットする（グローバルディスクリプタテーブルの項目3に対応）ことを意味する。そして、スタックをstack_startが指すuser_stackの配列領域に配置します。その後、新しい割り込み記述子テーブル(232行目)とグローバルセグメント記述子テーブルを # 使用します。このプログラムで後に定義される#(234--238行目)。新しいGDTテーブルの最初の内容は

は基本的に setup.s と同じで、セグメントの長さが 8MB から 16MB に変更されただけです。# Stack_start は kernel/sched.c の 82-87 行目で定義されています。これは終了点へのロングポインタです。

user_stack配列の#。23行目のコードでは、ここで使われるスタックを設定しており、現在は # システムスタック と呼んでいます。しかし、タスク0の実行に移ってから (init/main.cの137行)、タスク0とタスク1のユーザースタックとして # 使われるようになります。

```

17 startup_32:# 各データセグメントのレジスタ を設定します。
18     movl     $0x10,%eax# 直接のオペランドは'$'で始まるが、それ以外はアドレス である。
19     mov %ax,%ds
20     mov %ax,%es
21     mov %ax,%fs
22     mov %ax,%gs
23     lss     _stack_start,%esp# _stack_start -> ss:esp, システムスタック を設定し
      ます。
24     call     setup_idt# line 67-93
25     コール     セットアップ_gdt#ライン95--107
26     movl     $0x10,%eax# すべてのセグメント・レジスタ をリロードする
27     mov      %ax,%ds# は gdt を変更した後です。CSはすでに
28     mov      %ax,%es# 'setup_gdt' でリロードされました。
29     mov      %ax,%fs# GDTが変更されたため、すべてのセグメントをリロードする
      必要があります。
30     mov %ax,%gs

```

記述子のセグメント長が8MBから16MBに変更されているため (# setup.s 567-578行目および # 235-236行目を参照)、したがって、ロード
演算はすべてのセグメントレジスタに対して再度行う必要があります。さらに、# bochsシミュレーションソフトウェアを使ってコードを追跡することで、CSが再びロードされない場合、制限が
CSの見えない部分の # 長さは26行目まで実行しても8MBのままです。ここでCSをリロードすべきだと思われます。しかし、コードセグメント記述子には
がセグメントの長さを変更しても、その他の部分はまったく同じなので、8MBの制限長は
カーネルの初期化段階で問題を起こさないようにするためです。さらに、セグメント間ジャンプ命令は、カーネルの実行プロセス中に # CSを再ロードするので、これをロードしないと
ここでは、将来のカーネルエラーを引き起こすことはありません。
この問題に対応するため、現在のカーネルでは、25行目の後に # ロングジャンプ命令を追加しています: 'ljmp \$(KERNEL_CS), \$1f', 26行目にジャンプして、CSが
indeed reloaded.

```

31     lss _stack_start,%esp

```

32-36行目は、A20アドレスラインが有効であるかどうかをテストするために使用されます。その方法は、0x000000から始まるメモリアドレスに任意の値を書き込み、# 対応するアドレス0x100000 (1M) の値に同じ値が含まれているかどうかを確認するというものです。常に同じ値であれば、比較し続けることになり、つまり無限ループやクラッシュが発生します。これは # アドレスA20行がストローブされていないことを意味しており、カーネルは1MB以上の

#

memory.#

33行目の# '1:' は、ローカルシンボルからなるラベルです。この時、シンボル#はアクティブロケーションカウンターの現在の値を表しており、これを利用して

その命令のオペランドです。ローカルシンボルは、コンパイラやプログラマが # いくつかの名前を一時的に使用するために使用されます。再利用可能なローカルラベルは全部で10個あります。

プログラム全体を通して、#. これらのラベルは、'0', '1', ..., '9' という名前で参照されます。#

ローカルシンボルを定義するには、ラベルを'N:'という形式で記述する（ここでN # は数字を表す）。）この前に定義されたラベルを参照するためには、#「Nb」と表記する必要があります。次の定義のローカルラベルを参照するためには、#「Nf」と表記する必要があります。上記の「b」は「後方」を意味し、「f」は「前方」を意味します。いくつかの

アセンブリファイルの#ポイントでは、最大10個のラベルを前後に参照することができます。

```

32      xorl %eax,%eax
33 1:clock      %eax# A20が本当に有効  であることを確認する
34      movl    %eax,0x0000000# 35でなけれ
        ば永遠にループする      cmpl %eax,0x100000
36      je      1b# '1b'は後方のラベル1 を意
        味します。# '5f' は前方の 5 を
        意味する。

37 /*
38 * 注意! 486はビット16を設定して、スーパーバイザーのライトプロテクトをチェックしてください。
39 モードではありません。そうすると、"verify_area()"コールが不要になります。
40 * 486のユーザーは、NE (#5) ビットも設定したいと思うでしょう。
41 * int 16で計算エラーが発
    生します。42 */
    # 前のコメントで触れた486CPUのCR0コントロールレジスタの # ビット16は、書き込み禁止フラグ
    (WP)であり、スーパーユーザーレベルの
    # プログラムが一般ユーザの読み取り専用ページに書き込まないようにします。このフラグは主
    に # オペレーティングシステムが新しいプロセスを作成する際にコピーオンライト方式を # 実
    装するために使用されます。#
    # 以下のコード(43-65行目)は、数学コプロセッサチップが存在するかどうかを # 確認する
    ために使用されます。方法は、コントロールレジスタCR0を変更して、コプロセッサの実行
    コプロセッサが存在すると仮定して # 命令を実行します。何か問題が発生した場合、コプロセッサ
    # チップは存在しません。CR0のコプロセッサエミュレーションビットEM(ビット2)をセットし、 #
    コプロセッサ存在フラグMP(ビット1)をリセットする必要があります。

43      movl    %cr0,%eax# check math chip
44      andl    $0x80000001l,%eax#
Save PG, PE, ET 45 /* 486の場合はここで「orl
$0x10020,%eax」を使うと良いかもしれません */。
46      orl     $2,%eax# set MP
47      movl    %eax,%cr0
48      コールチェック_x87
49      jmp     after_page_tables# line 135 50

51 /*
52 * ETが正しいかどうか依存しています。287/387をチェックし
    ます。53 */

```

以下のfninitとfstswは、数学コプロセッサ(80287/80387)用の命令です。fninitはコプロセッサに初期化コマンドを発行し、コプロセッサを以前の操作の影響を受けない既知の状態に # し、コントロールワードをデフォルト値に設定し、ステータスワードとすべての浮動小数点スタックレジスタを # クリアします。この # 待機しない形式の fninit は、コプロセッサのすべての # 命令の実行を終了させます。

現在進行中の前の算術演算fstsw命令では

コプロセッサのステータスワードです。システムにコプロセッサがある場合は、fninit命令を実行した後、ステータス#ローバイトは0でなければなりません。

```

54 check_x87です。
55      fninit
56      fstsw %ax      # ステータスワードの取得 -> ax
57      cmpb $0,%al    # fninitの後、数学があればステータスワードは0になるは
        ずです。
58      Je 1f          /* コプロセッサなし: ビットを設定する必要がある。

```

[59](#)`movl %cr0,%eax`[60](#)`xorl $6,%eax`

MPをリセットし、EMをセットする */*。

```

61      movl %eax,%cr0
62      レット

```

.alignはアセンブリの指標です。その意味は、ストレージのバウンダリのアライメント # 調整を指します。ここで、「2」は、後続のコードやデータのオフセット位置が、# アドレス値の最後の2ビットが0になる位置(2²)に調整されることを示しており、# つまり、メモリアドレスが4バイト単位で整列されることを示している。(ただし、現在のGNU asは、2の累乗ではなく、アラインドされた値を # 直接書き込むようになっている)。これを使う目的はメモリアラインメントを実現するための # 指令は、32ビットCPUがメモリ上のコードやデータにアクセスする際の # 速度と効率を向上させるためのものです。

以下の2バイトは、80287命令 fsetpmのマシンコードです。この命令の役割は、80287をプロテクトモードに # 設定することです。80387はこの命令を必要とせず、nopとして # 扱われます。

```

63 .align 2
64 1:      .バイト      287 の fsetpm、387 では無視される */*.
        0xDB,0xE4
65      レット
66

```

```

68 * setup_idt
69 *

```

70 を指す256のエントリを持つidtを設定します。
71 ignore_int, 割り込みゲート。その後、ロードされます。
72 * idt.自分をインストールしたいと思うものすべて
73 idt-tableの*は、自分で行うことができます。インターラプト
74 他の場所で可能になるのは、私たちが相対的に
75 すべてが正常であることを確認してください。このルーティンが終わると
76 ページテーブルによって書き込まれます。77 */

割り込みディスクリプターテーブル(IDT)の各項目は8バイトで構成されていますが、その形式はGDTテーブルのそれとは # 異なり、ゲートディスクリプターと呼ばれます。0-1, 6-7 # バイトはオフセット、2-3バイトはセレクタ、4-5バイトはいくつかのフラグです。

このコードでは、まずEDXとEAXにデフォルトの割り込みディスクリプター値8バイトを # 設定し、idtテーブルの各項目にディスクリプターを配置します、合計256項目です。EAX # にはディスクリプターの下位4バイト、EDXには上位4バイトが格納されます。この間に続く初期化プロセスでは、カーネルは現在のデフォルト設定を、# 本当に便利な割り込みディスクリプターのエントリに置き換えます。

```

78 setup_idt:
79      lea      ignore_int,%edx# ignore_intの有効なadr -> edx
80      movl     $0x00080000,%eax# セレクタ0x0008をeax のハイワードに格納する。
81      movw     %dx,%ax/* セレクタ = 0x0008 = cs */。
        # オフセットの下位16ビットをeaxの下位ワードに格納する。
82      movw     $0x8E00,%dx/* 割り込みゲート - dpl=0, 現在 */
83      # edxはゲートディスクリプターの上位4バイトを含む。
84      lea      _idt,%edi# _idtはIDTテーブルのアドレス(オフセット) です。
85      mov      $256,%ecx
86 rp_sidt:
87      movl     %eax, (%edi)# ダミー記述子をIDTテーブルに格納する
88      movl     %edx, 4(%edi)# store eax to [edi+4].
89      addl     $8,%edi# edi プラス8 で次の項目を指します。
90      dec      %ecx
91      jne      rp_sidt

```

```

92         lidt                                idt_descr# 割り込み記述子テーブルレジスタ      をロードする。
93         レット
94
95 /*
96 * setup_gdt
97 *
98 * このルーチンは、新しい gdt をセットアップしてロードします。
99 * 現在、2つのエントリーのみが構築されており、同じ
100 init.sで構築されたものは、ルーチンの
101 * は全体で2行と非常に複雑なので、この
102 かなり長いコメントが必要ですね。)
103 このルーチンは、ページテーブルによって上書きされます。
104 */
105 setup_gdtです。
106         234～238行目のgt_descr#の内容をlgdtしています。
107         レット
108
109 /*
110 * カーネルのページテーブルを、ページディレクトリの直後に配置しました。
111 そのうち4つを使って、16MBの物理メモリを使っています。の人は
112 * 16MB以上の場合は、これを拡張する必要があります。
113 */

# 各ページテーブルのサイズは4KB（1ページメモリ）で、各ページテーブルのエントリは4 #
# バイト必要なので、1つのページテーブルには合計1024のエントリを格納することができます。
# ページテーブルのエントリが
# 4KBのアドレス空間を持つページテーブルは、4MBの物理メモリを扱うことができます。ページテ
# ーブルのエントリの # 形式は、項目の最初の0～11ビットにいくつかのフラグを格納します。
# メモリー内かどうか（Pビット0）、リード&ライトパーミッション（R/Wビット1）、ノーマルユ
# ーザーかスーパーユーザーか（U/S、ビット2）、モディファイされているかダーティか（Dビット
# 6）など、# ビット12-31はページフレームアドレスで、ページの物理的な開始アドレスを示すの
# に使われる。

114 .org 0x1000 # ここから最初のページテーブルが始まる ページディレクトリは0に格納される
115 pg0
116     で
117     す。
118
119
120 .org 0x2000
121 pg1
122     で
123     す。
124
125
126 .org 0x3000
127 pg2
128     で
129     す。
130
131 # 以下のコードやデータは、オフセット0x5000から始まります。
132 tmp_floppy_areaは、DMAができないときにフロッピードライバーが使用し
133 ます。
134 * バッファブロックに到達します。これをアラインメントする必要があります。
135 64kBのボーダーに
136 */
137 _tmp_floppy_area:

```


[133](#) `.fill` 1024, 1, 0# 1024バイトを0 で埋める。
[134](#)

以下のプッシュ操作は、init/main.cのmain()関数へのジャンプを # 準備するために使用されます。139行目の命令はリターンアドレス(ラベルL6)を # スタックにプッシュし、140行目はmain()関数コードのアドレスを # プッシュします。head.sが最終的に218行目のret命令を実行すると、main()のアドレスが # ポップアップされ、init/main.cプログラムに制御が移ります。第3章のC関数呼び出しメカニズムの説明を参照してください。

最初の3つのポップアップ0の値は、それぞれ主関数の引数envp、argvポインター、 # argcを表していますが、main()はこれらを使用しません。139行目のプッシュ操作
main()の呼び出しのリターンアドレスをシミュレートします。そのため、メインプログラムが本当に終了した場合、 # ラベルL6に戻って続きを行う、つまり無限ループを実行することになります。140行目 # main()のアドレスをスタックにプッシュしているので、'ret'命令が実行されるとページング処理の設定(setup_paging)を行った後に実行される # main()のアドレスがスタックからポップアウトされ、main()が実行される。

```

135 after_page_tables:
136     pushl                                $0# これらはmainのパラメータです :-)
137     pushl $0
138     pushl $0
139     pushl                                $L6# return address for main, if it decides it to ....
140     pushl                                $_main# '_main'は、main() の内部表現です。
141     jmp                                  setup_paging# 198 行目にジャンプします。
142 L6:
143     jmp                                  L6# メインはここで戻ってはいけませんが
144     # 念のため、どうなるかはわかっている。
145
146 /* これはデフォルトの割り込みハンドラです :-)*/
147 int_msg:
148     .asciz "Unknown interrupt\\r      "# 定義文字列 "未知中断(回车换行)".
149     .align 2# メモリ      上の4バイトとのアライメント。
150 ignore_int:
151     pushl %eax
152     pushl %ecx
153     pushl %edx
154     push                                %ds# ds, es, fs, gs は、スタック 上ではまだそれぞれ 2 ワードを占
        めています。
155     push %es
156     push %fs
157     movl                                $0x10,%eax# セレクタの設定 (ds, es, fs は gdt 内のデータ記述子
        を指す)
158     mov %ax,%ds
159     mov %ax,%es
160     mov %ax,%fs
        # printk()関数のパラメータポインタをスタックに置く。なお、'$'が付加されていない場合は
        int_msgの前にある#は、int_msgシンボルのロングワード('Unkn')がプッシュされることを意味しま
        す。
        # をスタックに置く。この関数は /kernel/printk.c にあります。
        printkのコンパイル済みモジュールでの # 表現。
161     pushl $int_msg
162     コール _printk
163     popl                                %eax# /kernel/printk.c
164     ポップ %fs
165     ポップ %es
166     ポップ %ds
167     popl %edx
168     popl %ecx
169     popl %eax

```

[170](#)

iret# iretはCFLAGSも 飛び出します。

```

171
172
173 /*
174 * Setup_paging
175 *
176 * このルーチンは、ページビットを設定することにより、ページングを設定します。
177 * in cr0. ページテーブルが設定され、IDマッピングが行われます。
178 * 最初の16MB。ペーザーは、不正なデータがないことを前提としています。
179 アドレスが生成されます（4MBのマシンで4MBを超える場合）。
180 *
181 * 注意！すべての物理的なメモリはアイデンティティであるべきですが
182 このルーチンでマッピングされるのは、カーネルページ関数のみです。
183 1Mb以上のアドレスを直接使用します。すべての「通常」機能
184 下位1MBのみを使用するか、ローカルデータ領域を使用します。
185 *は別の場所にマッピングされます - mmが記録します。
186 * その
187 *
188 * 16 Mb以上のメモリをお持ちの方は、大変です。私は
189 買っていないのになぜ買うのか :-) ソースはこちらです。変更
190 真面目な話、そんなに難しいことではないはず。ほとんどが
191 いくつかの定数などを変更しています。16Mbのままにしておいたのは、私のマシンが
192 これ以上の拡張はできません（でも、安かったからいいか）。
193 * どの定数を変更するかを示すために、次のようにしました。
194 16Mb "で検索してみてください）。
195 * won't guarantee that's all :-( )
196 */
# 上記オリジナルコメントの第2段落の意味は、マシン内の # 1MB以上のメモリ空間が主にメイン
メモリ領域に使用されているということです。このメインメモリ領域は、mmモジュールによって
管理され、ページマッピング操作が行われます。すべての
カーネル内の # その他の関数は、ここでいう一般的な（普通の）関数です。主記憶領域のページ
を利用するには、get_free_page()関数を使って # 取得する必要があります。主記憶領域のメモリ
ページは共有資源であるため、そこに
# リソースの競合を避けるために、統一された管理のためのプログラムである必
要があります。#
# 1ページのページディレクトリテーブルと4ページのページテーブルは、 # メモリの物理アド
レス0x0に格納されています。ページディレクトリテーブルはシステムの全プロセスに # 共通で、
ここでの4ページページテーブルはカーネル固有のものです。初期の16MBのリニア
# アドレス空間は、物理的なメモリ空間に1つずつマッピングされます。新しく作成された
# プロセスでは、システムはそのアプリケーションのページテーブルをメインメモリ # 領域に格
納します。さらには

197 .align                                2# メモリの境界を4バイトで揃える。
198 setup_paging:
199     movl                                $1024*5,%ecx/* 5ページ - pg_dir+4ページのテーブル */。
200     xorl %eax,%eax
201     xorl                                %edi,%edi/* pg_dir is at 0x000 */。
202     cld;rep;stosl# eax -> [es:edi], ediが4つ増えました

```

以下の記述は、ページディレクトリの項目を設定するものです。カーネルは合計4つのページテーブルを使用しているため、設定する項目は4つで # す。ページディレクトリの項目の構造は、ページテーブルの項目の構造と同じで、4バイトが # 1つの項目となります。上記113行目以下の記述を参照してください。

例えば、"\$pg0+7"は、0x00001007を意味し、ページディレクトリの最初の項目となります。

その後、最初のページテーブルのアドレス = 0x00001007 & 0xfffff000 = 0x1000; となります。
1ページ目のテーブルの属性フラグ = 0x00001007 & 0x00000fff = 0x07で、# ページが存在し、ユーザーが読み書きできることを示しています。

```
203      movl      $pg0+7, _pg_dir/* set present bit/user r/w */.
204      movl      $pg1+7, _pg_dir+4      /* ----- "" ----- */
205      movl      $pg2+7, _pg_dir+8      /* ----- "" ----- */
206      movl      $pg3+7, _pg_dir+12     /* ----- "" ----- */
```

以下の6行のコードは、4つのページテーブルのすべてのアイテムの内容を # 埋めており、アイテムの合計数は4 (ページテーブル) * 1024 (アイテム / ページテーブル) = 4096
アイテム (0~0xfff) は、4096 * 4Kb = 16Mbの物理メモリをマッピングすることができます。各アイテムの内容は、現在のアイテムがマッピングする物理メモリアドレス+ページフラグ (全7種)。# 記入方法は、現在のアイテムの最後の項目から逆順に記入していきます。
最後のページのテーブルです。各ページテーブルの最後の項目の位置は、1023*4=4092なので、
最終ページの最後の項目は、\$pg3 + 4092となります。

```
207      movl      $pg3+4092,%edi      # edi -> 最後のページの最後のアイテムを指す
208      movl      $0xffff007,%eax      16Mb - 4096 + 7 (r/w user, p) */*.
                                         最後の項目は、物理メモリ addr 0xffff000 + 7 にマップさ
209      std                                           れます。
210 1:      stosl      /* ページを逆に埋める - より効率的に :-)*/
211      subl      $0x1000,%eax      # アイテムが満たされるたびに、addrは0x1000だけ減少しま
                                         す。
212      jge      1b      # 0より小さい場合、すべてが満たされる。

# ここで、ページディレクトリベースレジスタcr3を設定します。これは、# ページディレクト
# リテーブルの物理アドレスを含んでいます。そして、ページングの使用を開始するように設定し
213      movl      $0,%cr3      # cr3 = 0 (物理アドレス)
                                         まず (cr0のPGフラグ、ビット31)。
214      xorl      %eax,%eax      # 0
                                         %eax,%eax/* pg_dir is at 0x0000 */.
215      movl      %cr3,%eax      # cr3
                                         %eax,%cr3/* cr3 - ページ・ディレクトリの開始 */.
216      orl      $0x80000000,%eax      # PGフラグの追加
217      movl      %eax,%cr0      # cr0
                                         %eax,%cr0/* ページング (PG) ビットの設定 */ (注)
218      ret      /* this also flushes prefetch-queue */.
```

ページングフラグを変更した後は、プリフェッチ命令のキューを更新するために # 分岐命令を使用する必要があります。ここで使われるのがret命令です。このret命令のもう一つの役割は、140行目の命令でプッシュされたmain()のアドレスを # ポップして、/init/main.cプログラムにジャンプして実行することです。このプログラムは本当にここで終わります。

```
219
220 .align      2# メモリの境界を4バイト      で揃える。
221 .word      0#は1ワード飛ばして、224行目が4バイト配列      になるように
      します。
```

以下は、LIDT命令がロードするために必要な6バイトのオペランドです。
割り込みディスクリプターテーブルのレジスタです。最初の2バイトはIDTテーブルの限界値、# 最後の4バイトはリニアアドレス空間におけるIDTテーブルの32ビットベースアドレスです。

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

グローバルディスクリプターテーブル # レジスタのLGDT命令で要求される6バイトのオペランドが以下にロードされます。最初の2バイトはGDTの限界値、最後の4バイトは

のバイトは、GDTのリニアベースアドレスです。ここでは、グローバルテーブルのサイズを2KBに設定しています。

バイト（つまり0x7ff）です。各ディスクリプターアイテムは8バイトなので、合計256のテーブルの中の # エントリです。シンボル `_gdt` は、プログラム内のグローバルテーブルのオフセット位置で、# 234行目を参照してください。

[227](#) `gdt_descr:`

[228](#) `.word`

256*8-1# は `gdt` も同様です（ただし、これは任意の

[229](#) `.long`

`_gdt#`マジックナンバーですが、私には効果

があります :) [230](#)

[231](#) `.align`

3# メモリ境界を8(2³)バイト 分揃える。

[232](#) `_idt:.fill`

256, 8, 0# `idt`は初期化さ

れていません [233](#)

Global descriptor table GDT. 最初の4つの項目は、空の項目（使用しない）、カーネル

#コードセグメント記述子、カーネルデータセグメント記述子、システムコールセグメント記述子。#

システムコールセグメント記述子は使われない。Linus氏は、コードセグメント記述子に

システムコールコードをこの独立したセグメントにのために252項目のスペースが後で確保されます。

新たに作成されたタスクのローカルディスクリプターテーブル（LDT）と # タスクステートセグメントTSSのディスクリプターを配置する。

(0-nul, 1-cs, 2-ds, 3-syscall, 4-TSS0, 5-LDT0, 6-TSS1, 7-LDT1, 8-TSS2 etc...)

[234](#) `_gdt: .quad 0x0000000000000000`

/* NULLディスクリプター */

[235](#) `.quad 0x00c09a00000000fff`

16Mb /*# 0x08, kernel code seg.

[236](#) `.quad 0x00c09200000000fff`

16Mb /*# 0x10, カーネル・データ・セグメント。

[237](#) `.quad 0x0000000000000000`

/* TEMPORARY - don't use */ のようになります。

[238](#) `.fill 252, 8, 0`

LDTやTSSなどのための/*スペース*。

6.4.3 参考情報

6.4.3.1 ヘッドのプログラム実行終了後のメモリーマップ

`head.s`プログラムの実行後、カーネルコードは、メモリページディレクトリとページテーブルの設定を正式に完了し、割り込みディスクリプターテーブルIDTとグローバルディスクリプターテーブルGDTを再作成しました。また、フロッピーディスクドライバ用に1KBバイトのバッファをオープンしたのもこのプログラムである。この時、メモリ上のシステムモジュールの詳細イメージは図6-11のようになります。

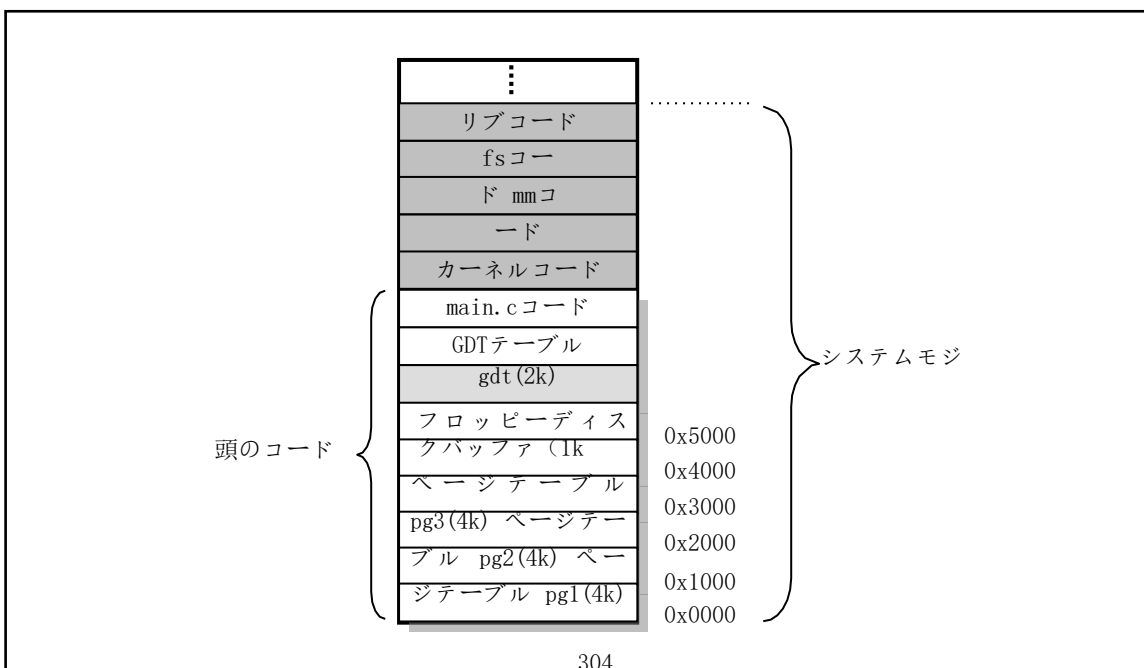


図6-11 メモリ内のシステムモジュールのマップ

6.4.3.2 インテル32ビット保護動作メカニズム

このプログラムを理解するには、インテル80X386の32ビットプロテクトモードの動作メカニズムを知ることが重要です。

8086のCPUと互換性を持たせるために、80X86のプロテクトモードはより複雑に設計されています。プロテクトモードの動作については、第4章を参照してください。ここでは、リアルモードとプロテクトモードを比較しながら、プロテクトモードについて簡単に紹介します。

CPUがリアルモードで動作しているとき、セグメントレジスタは、メモリセグメントのベースアドレスを置くために使用されます（例えば、0x9000）。メモリセグメントのサイズは64KBに固定されています。このセグメントでは、最大64KBのメモリをアドレス指定できます。ただし、プロテクトモードに入ると、セグメントレジスタには、メモリ上のセグメントベースアドレスではなく、ディスクリプターテーブルのセグメントに対応するディスクリプター項目のセレクターが格納されます。8バイトサイズのディスクリプターには、セグメントのリニアアドレスのベースアドレスとセグメントレンガスのほか、セグメントの特徴を表すその他のビットが含まれています。したがって、この時点でアドレス指定されているメモリ位置は、セグメントのベースアドレスに現在のオフセット値を加えたもので指定できる。もちろん、実際にアドレス指定される物理的なメモリアドレスは、メモリページング機構によって変換される必要がある。つまり、32ビットプロテクトモードでのメモリアドレスリングモードは、ディスクリプターテーブルのディスクリプターの使用とメモリページング管理によって決定されるという、もう一つの手順が必要なのである。

記述子テーブルは、目的に応じて3種類に分けられます。GDT（Global Descriptor Table）、IDT（Interrupt Descriptor Table）、LDT（Local Descriptor Table）です。CPUがプロテクトモードで動作している場合、GDTとIDTは同時に1つしか存在できず、テーブルのベースアドレスはそれぞれレジスタGDTRとIDTRで指定される。ローカル記述子テーブルの数は、GDTテーブル内の未使用項目の数や設計中の特定のシステムによって決定され、ゼロまたは最大8191とすることができます。ある時点で、現在のLDTテーブルのベースアドレスはLDTRレジスタの内容によって指定され、LDTRの内容はGDT内の記述子を使用してロードされる、つまり、LDTもGDT内の記述子によって指定される。

一般的に、カーネルは各タスク（プロセス）に対して1つのLDTを使用します。実行時には、プログラムはGDT内の記述子だけでなく、現在のタスクのLDT内の記述子も使用することができます。Linux 0.12カーネルの場合、同時に実行できるタスクは64個なので、GDTテーブルの中のLDTテーブルには最大64個の記述子エントリがあります。

割り込みディスクリプターテーブルIDTの構造は、LinuxカーネルのGDTテーブルのすぐ前にあるGDTと似ています。8バイトのディスクリプターが合計256個格納されています。ただし、各ディスクリプター項目のフォーマットはGDTとは異なり、対応する割り込みハンドラプロシージャのオフセット（0～1、6～7バイト）、セグメントのセレクタ（2～3バイト）、Someフラグ（4～5バイト）が格納されています。

図6-12は、Linuxカーネルで使用されているディスクリプターテーブルの模式図である。図では、各タスクがGDTの2つのディスクリプター項目を占有している。GDTテーブルのLDT0ディスクリプター項目は、最初のタスク（プロセス）のローカルディスクリプターテーブルのディスクリプターであり、TSS0は最初のタスクのタスクステートセグメント（TSS）のディスクリプターである。各LDTには3つの記述子があり、1つ目は使用されない記述子、2つ目はタスクコードセグメントの記述子、3つ目は

タスクデータセグメントとスタックセグメントの記述子です。DSセグメントレジスタが第1タスクのデータセグメントセレクタの場合、DS:ESIはタスクデータセグメントのあるデータを指します。

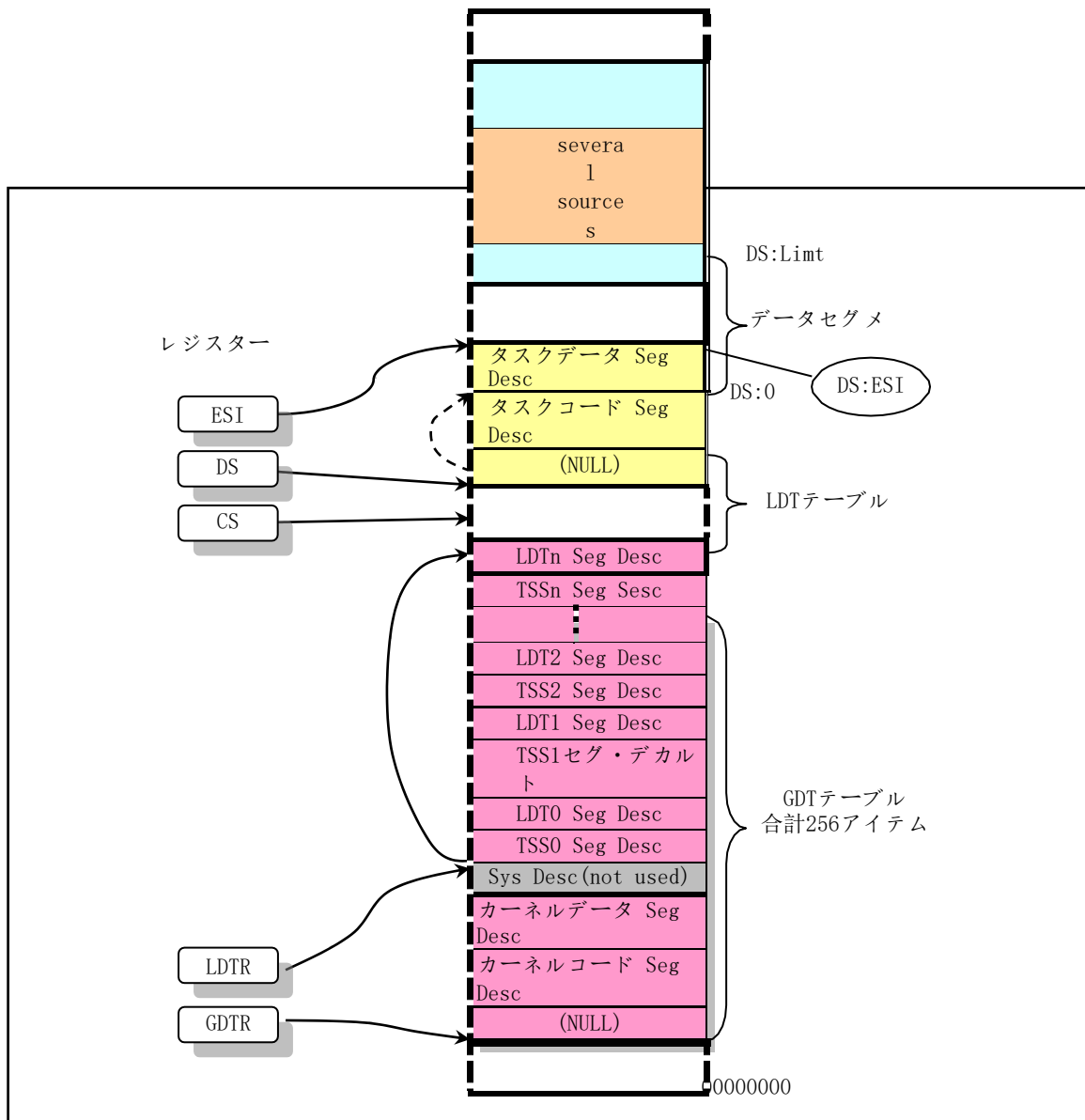


図6-12 Linuxカーネルが使用するディスクリプターテーブル

6.4.3.3 align ディレクティブ

`align`指令については、第3章でアセンブラを紹介したときにすでに説明しました。ここではそれを要約します。`.align`指令の使い方は、コンパイル時にロケーションカウンタ（命令カウンタのようなもの）を指定したメモリ境界に投入するようにコンパイラに指示することです。その目的は、CPUがメモリ上のコードやデータにアクセスする際の速度と効率を向上させることです。その完全なフォーマットは

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

1つ目のパラメータ値`val1`は必要なアラインメント、2つ目はパディングバイトで指定します。パディング値は省略することもでき、省略した場合は、コンパイラが0の値でパディングされます。オプションの第3パラメータ値`val3`は、パディングまたはスキップに使用できる最大数を示すために使用さ

れます。バウンダリ・アライメントがval3で指定された最大バイト数を超える場合、アライメントは一切行われません。2つ目のパラメータval2を省略しても、3つ目のパラメータval3を使用する必要がある場合は、カンマを2つ入れるだけでOKです。

ELFオブジェクト形式を採用しているプログラムでは、最初のパラメータval1にアラインメントが必要なバイト数を指定します。例えば、「`.align 8`」とは、位置カウンタが8の倍数の境界を指すように調整することを意味します。すでに8の倍数になっている場合は、コンパイラが変更する必要はありません。ただし、「`.align 8`」の場合は

ここでa.outオブジェクトフォーマットを使用するシステムでは、最初のパラメータval1は下位0ビットの数、つまり2の累乗 (2^{Val1}) となります。例えば、先のプログラムhead.sの「.align 3」は、位置カウンタを8の倍数の境界上に置く必要があることを意味しています。ここでも、すでに8の倍数の境界上にある場合には、この指示は何もしません。GNU as(gas)は、Gasが様々なアーキテクチャシステムに付属するアセンブラの動作を模倣するように形成されているため、これらの2つのターゲットフォーマットを異なって扱います。

6.5 まとめ

ブートローダ bootsect.S は、setup.s コードとシステムモジュールをメモリにロードし、自己と setup.s コードをそれぞれ物理メモリ 0x90000 と 0x90200 に移動させて、セットアッププログラムに実行を委ねます。システムモジュールのヘッダには、header.sのコードが含まれています。

セットアッププログラムの主な機能は、ROM BIOS割り込みプログラムを使ってマシンの基本的なパラメータを取得し、後のプログラムのために0x90000から始まるメモリブロックに保存することです。同時に、システムモジュールを物理アドレス0x00000の先頭に移動させます。そのため、システム内のヘッド.スコードは0x00000の先頭になります。その後、ディスクリプターテーブルベースアドレスをディスクリプターテーブルレジスタにロードし、32ビットプロテクトモードでの動作に備える。次に、割り込み制御のハードウェアを再構築する。最後に、マシンコントロールレジスタCR0をセットし、システムモジュールのhead.sコードにジャンプして、CPUを32ビットプロテクトモードで起動します。

Head.sプログラムの主な機能は、最初に割り込みディスクリプターテーブルの256項目のディスクリプターを初期化し、A20アドレスラインがすでに開いているかどうかをチェックし、システムに数学コプロセッサが含まれているかどうかをテストします。次にメモリページディレクトリテーブルを初期化し、メモリのページング管理に備える。最後に、システムモジュール内の初期化プログラムinit/main.cにジャンプして実行を継続する。

次章の主な内容は、init/main.cプログラムの機能を詳細に説明することです。

7初期化プログラム(init)

カーネルソースのinit/ディレクトリには、main.cファイルが1つだけあります。システムモジュールは、boot/head.sのコードを実行した後、main.cに実行権を渡します。このプログラムは長くはありませんが、カーネルの初期化のすべての作業を含んでいます。そのため、カーネルソースを読む際には、他の多くのプログラムの初期化部分を参照する必要があります。ここで呼ばれている関数をすべて理解できれば、この章を読み終えた時点で、Linuxカーネルの仕組みを大まかに理解することができます。

この章からは、大量のC言語プログラムに出会うことになりますので、読者はすでに一定のC言語の知識を持っている必要があります。C言語の参考書としては、Brian W. KernighanとDennis M. Ritchieの「C Programming Language」が最適でしょう。この本の第5章に出てくるポインタと配列の理解は、C言語を理解する上でのキーポイントと言えます。また、gccの拡張機能であるインライン関数やインラインアセンブリ文などは、カーネルのソースコードの随所で使用されていますので、GNU gccのマニュアルをリファレンスとして持つておく必要があります。

C言語のプログラムに注釈を付ける場合、プログラム中のオリジナルのコメントと区別するために、コメント文の開始記号として「//」を使用します。元のコメントの翻訳にも同じコメントマークを使用しています。プログラムに含まれるヘッダファイル(*.h)については、要約の意味のみを示しています。ヘッダファイルの具体的な詳細コメントは、コメントヘッダファイルの対応するセクションに記載されます。

7.1 main.c

7.1.1 機能説明

main.cプログラムでは、まず前のセットアップで得たマシンパラメータを使って、システムのルートファイルデバイス番号といくつかのメモリグローバル変数を設定します。これらのメモリ変数は、メインメモリ領域の開始アドレス、システムが持つメモリ量、キャッシュメモリの終了アドレスを示します。仮想ディスク（RAMDISK）も定義されている場合は、メインメモリ領域が適切に縮小されます。図7-1に、メモリ空間全体の模式図を示す。この図では、キャッシュ部分は、ディスプレイカードのビデオメモリやそのBIOSが占める部分も差し引く必要がある。高速キャッシュは、ディスクなどのブロックデバイスとの間でデータを一時的に保存するためのもので、1K（1024）バイトをブロック単位としている。主記憶領域は、ページング機構を介してメモリ管理モジュールmmが管理・割り当てを行い、4Kバイトをメモリページ単位としている。カーネルプログラムは、キャッシュ内のデータには自由にアクセスできるが、割り当てられたメモリページを使用するにはmmを通過する必要がある。

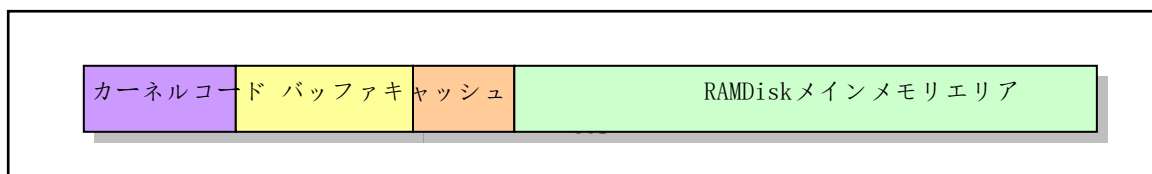


図7-1 システムにおけるメモリ機能の分割のイメージ図

その後、カーネルコードはハードウェアの初期化作業を様々な角度から行います。これには、トランプゲート、ブロックデバイス、キャラクタデバイス、`tty`などのほか、最初のタスク（タスク0）を手動で設定することも含まれます。すべての初期化作業が完了すると、カーネルは割り込み許可フラグを設定して割り込みをオンにし、タスク0に切り替えて実行します。この時点で、カーネルは基本的にすべての設定作業を終えたと言えます。その後、カーネルはタスク0を通じていくつかの初期タスクを作成し、シェルプログラムを実行し、コマンドラインプロンプトを表示することで、Linuxシステムは通常の動作状態になります。

なお、これらの初期化サブルーチンを読む際には、呼び出されているプログラムの内容をより深く理解することが大切です。どうしても理解できない場合は、そのまま置いておいて、次の初期化関数を見続けます。ある程度理解できたら、未読のところを続けて勉強する。

7.1.1.1 カーネル初期化フロー図

カーネル全体の初期化が終わると、カーネルは実行制御をユーザーモード（タスク0）に切り替えます。つまり、CPUは特権レベル0から特権レベル3に切り替わります。この時点では、メインプログラムはタスク

0. そして、システムはまず、プロセス作成関数`fork()`を呼び出し、`init()`を実行するための子プロセス（initプロセス）を作成する。システムの初期化プロセスの全体像を図7-2に示す。

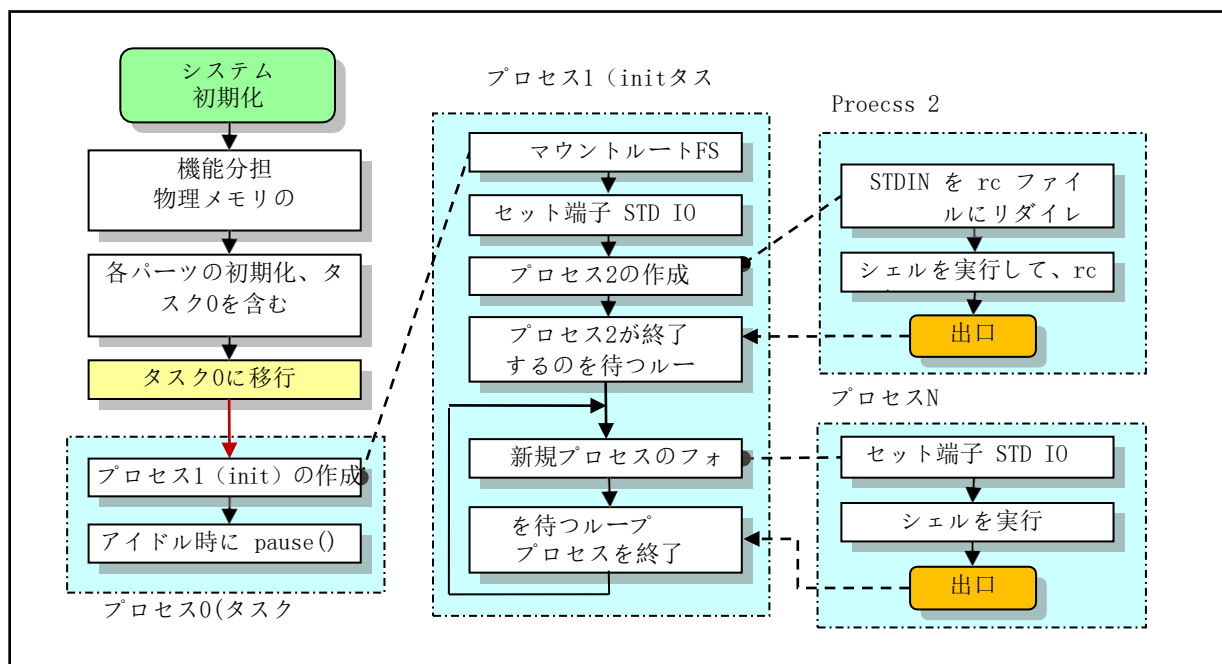


図7-2 カーネル初期化処理フロー図

図からわかるように、`main.c`プログラムは、まずシステムの物理メモリをどのように割り当てるかを決定し、次にカーネルの各部の初期化関数を呼び出して、メモリ管理、割り込み処理、ブロックデバイスとキャラクタデバイス、プロセス管理、ハードディスクとフロッピーディスクのハードウェアを初期化します。これらの操作が完了すると、システムの各部分はすでに動作可能な状態になっている。その後、プログラムは自分自身をタスク0（プロセス0）に「手動」で移動させ、`fork()`コールを使って初めてプロセス1（initプロセス）を作成し、そこで`init()`関数を呼び出します。この関数の中で

、プログラムはアプリケーション環境の初期化とシェルのログインプログラムの実行を続けます。元々のプロセス0は、システムがアイドル状態の時に実行されるようにスケジュールされているので、プロセス0は一般的にアイドルプロセスとも呼ばれています。この時点では、プロセス0は`pause()`システムコールを実行し、スケジューラ関数を呼び出すだけです。

`init()`の機能は、4つの部分に分けられる。(1)ルートファイルシステムのインストール、(2)システム情報の表示、(3)システム初期リソース設定ファイル`rc`の実行、(4)ユーザーログインシェルの実行。

1. ルートファイルシステムのインストール

このコードではまず`syscall setup()`を呼び出し、ハードディスクデバイスのパーティションテーブル情報を収集し、ルートファイルシステムをインストールします。ルートファイルシステムをインストールする前に、システムはまず最初に仮想ディスクを作成する必要があるかどうかを判断します。カーネルのコンパイル時に仮想ディスクのサイズが設定されており、カーネルの初期化処理時に仮想ディスクとして使用するためにメモリがオープンされている場合、カーネルはまず、ルートファイルシステムをメモリの仮想エクステンツにロードしようとしています。

2. システム情報の表示

次に、`init()`は、端末デバイス`tty0`をオープンし、そのファイル記述子またはハンドラを、標準入力`stdin`、標準出力`stdout`、エラー出力`stderr`の各デバイスにコピーします。カーネルは、これらのファイル記述子を使って、キャッシュ内のバッファブロックの総数や、主記憶領域の空きメモリの総バイト数などのシステム情報を端末に表示します。

3. リソース設定ファイルの実行

次に、`init()`は新しいプロセス(Process 2)を作成し、その中でいくつかの初期設定操作を行い、ユーザーとの対話環境を確立します。つまり、ユーザーがシェルのコマンドライン環境を使用する前に、カーネルは`/bin/sh`プログラムを使用して、設定ファイル`etc/rc`に設定されたコマンドを実行します。`rc`ファイルの役割は、DOS/OSオペレーティングシステムのルートディレクトリにある`AUTOEXEC.BAT`ファイルに似ています。このコードはまず、ファイルディスクリプター0を閉じて標準入力`stdin`を`etc/rc`ファイルに導き、すぐにファイル`/etc/rc`を開いてすべての標準入力データがファイルから読み込まれるようにします。その後、カーネルは`/etc/rc`ファイル内のコマンドの実行を実装するために、`/bin/sh`を非対話的に実行します。ファイル内のコマンドが実行されると、`/bin/sh`は直ちに終了するので、プロセス2は終了します。

4. ユーザーログインシェルの実行

`init()`関数の最後の部分は、新しいプロセスでユーザーの新しいセッションを作成し、ユーザーのログインシェル`/bin/sh`を実行するために使用されます。システムがプロセス2でプログラムを実行すると、親プロセス（`init`プロセス）はその終了を待ちます。プロセス2の終了により、親プロセスは無限ループに入ります。このループの中で、親プロセスは再び新しいプロセスを生成した後、プロセス内に新しいセッションを作成し、ログインのために再びプログラム`/bin/sh`を実行して、ユーザーとの対話シェル環境を作ります。その後、親プロセスは子プロセスを待ち続けます。ログインシェルは、前回の非対話型シェルと同じプログラム`/bin/sh`ですが、使用するコマンドライン引数（`argv[]`）が異なります。ログインシェルの0番目のコマンドライン引数の最初の文字は、マイナス記号「-」でなければなりません。この特別なフラグは、`/bin/sh`に通常の実行ではなく、`/bin/sh`をログインシェルとして実行することを伝えます。この時点から、ユーザーはLinuxのコマンドライン環境を通常通り使用できるようになり、親プロセスは待機状態に入ります。その後、ユーザーがコマンドラインで`exit`ま

たはlogoutコマンドを実行すると、現在のログインシェルの終了情報が表示された後、システムは再びログインシェルプロセスの生成処理を繰り返すという無限ループに陥ります。

タスク1で実行されるinit()関数の最後の2つの部分は、実際には独立環境初期化プログラムinitの関数であるべきである。これについては、プログラムリストの後の説明を参照してください。

7.1.1.2 初期ユーザースタックの操作

新しいプロセスを作成するプロセスは、親プロセスのコードを完全にコピーすることで実施されるため

とデータセグメントの間で、初めてfork()を使用して新しいプロセスinitを作成するとき、新しいプロセスのユーザプロセススタックにプロセス0の冗長な情報がないようにするために、最初の新しいプロセス（プロセス1）が作成されるまで、プロセス0はそのユーザモードスタックを使用してはいけません、つまり、タスク0は関数を呼び出す必要はありません。したがって、main.cプログラムがタスク0の実行に移った後は、タスク0のコードfork()を関数として呼び出すべきではありません。プログラムに実装されている方法は、以下のようにgccの関数インライン形式を使ってこのsyscallを実行します（プログラムの23行目を参照）。

インライン関数を宣言することで、gccはその関数のコードを呼び出したコードに統合することが出来ます。これにより、関数呼び出しのオーバーヘッドを節約できるため、コードの実行が速くなります。また、実引数のいずれかが定数である場合、コンパイル時にこれらの既知の値を使用することで、インライン関数のコードをすべて含めることなく、コードを単純化できる場合があります。第3章の関連する命令を参照してください。

[23](#) static inline [_syscall0](#)(int, [fork](#))

ここで、_syscall0()はunistd.h内のインラインマクロコードで、組み込みアセンブリの形式でLinuxのシステムコール割り込みint 0x80を呼び出します。

include/unistd.hファイルの150行目のマクロ定義に従って、このマクロを展開し、上記の行に置き換えます。この記述は、実際には以下のようにint fork()作成プロセスのシステムコールであることがわかります。

```
// unistd.hファイルでの _syscall0() の定義です。つまり、システムがマクロを呼び出すと
// パラメータのない関数: type name(void)。 150
#define \_syscall0(type,name) \ (^o^)
151 type name(void)
○152 ①152
153 long res; eldest
154 asm volatile ("int $0x80" : "=a" ( res ) : "0" ( NR_##name);
155 : "=a" ( res )
156 : "0" ( NR_##name);
157 if ( res >= 0 ) \\\ If return value >=0, the value is returned.
158 return (type) res; 。
159 errno = - res;
    ^^^ ) そうでない場合は、エラー番号を設定し
    て -1 を返す 160 return -1; ^^^ )
161 }
```

上記の定義によると、_syscall0(int, fork)を展開することができます。23行目を代入すると、以下のような記述になります。

```
static inline int fork(void)
{
    ロングレスって
    asm volatile ("int $0x80" : "=a" ( res ) : "0" ( NR_fork));
    if ( res >= 0 )
        return (int) res;
    errno = - res;
    1を返す。
```

```
}
```

gccは上記の "function "ボディにステートメントを挿入します。

をfork()文を呼び出すコードに直接書き込んでいるため、fork()を実行しても関数呼び出しにはなりません。また、マクロ名の文字列「syscall0」の最後の0はパラメータなし、1はパラメータ1を意味します。システムコールのパラメータが1つの場合は、マクロ「_syscall1()」を使用する必要があります。

上記のsyscall実行割り込み命令INTはスタックの使用を避けることができませんが、syscallはユーザスタックの代わりにタスクのカーネル状態スタックを使用し、各タスクは独立したカーネル状態スタックを持っているので、syscallはここで説明したユーザ状態スタックに影響を与えません。

また、新たにプロセスinit（プロセス1）を生成する過程で、システムはいくつかの特別な処理を行っています。実は、プロセス0とプロセスinitは、カーネル内の同じコードとデータのメモリページ（640KB）を使用していますが、実行されるコードは一か所ではないため、実際には、同じユーザースタック領域も同時に使用しています。親プロセス(プロセス0)のページディレクトリとページテーブルエントリを新しいプロセスinit用にコピーする際、640KBのページテーブルエントリのプロセス0の属性は変更されていません(読み書き可能なまま)が、対応するプロセス1の属性は読み取り専用に設定されています。このため、プロセス1が実行を開始すると、ユーザースタックへのアクセスでページ書き込み保護例外が発生し、カーネルのメモリマネージャ(mm)がメインメモリ領域にプロセス1用のメモリページを割り当て、タスク0のスタックの対応するページをこの新しいページにコピーします。この時点から、タスク1のユーザモードスタックは、独立したメモリページを持つようになります。つまり、タスク1からスタックにアクセスした後は、タスク0とタスク1のユーザースタックは互いに独立したものになります。したがって、コンフリクトを起こさないためには、タスク1がスタック操作を行う前に、タスク0がユーザースタック領域の使用を禁止することが必要です。

また、カーネルが各プロセスをスケジューリングする順番はランダムであるため、タスク1のためにタスク1が生成された後、タスク0が先に実行される可能性があります。そのため、タスク0がfork()操作を実行した後、タスク1よりも先にユーザースタックを使用してしまうことを避けるために、後続のpause()関数もインライン関数の形で実装する必要があります。

システム内のプロセス（initプロセスの子プロセスであるプロセス2など）がexecve()コールを実行した場合、プロセス2のコードとデータは主記憶領域に配置されるため、システムはコピーオンライト技術を利用して、その後いつでも他の新しいプロセスの生成と実行を処理することができます。

Linuxでは、シェルプログラムやネットワークサブシステムプログラムなど、多くのシステムアプリケーションを含め、すべてのタスクがユーザーモードで実行されます。カーネルソースコードのlib/ディレクトリにあるライブラリファイル（string.cプログラムを除く）は、新しく作成されたプロセスをサポートするための機能を提供するように設計されています。特権レベル0のカーネルコード自体は、これらのライブラリ関数を使用しません。

7.1.2 プログラムアノテーション

プログラム 7-1 linux/init/main.c

```

1 /*
2  *linux/init/main.c
3  *
4  *(C)      1991Linus Torvalds
5  */
6

```

// unistd.hは、標準的なシンボル定数と型のファイルです。様々なシンボル定数を定義しています。

//と型を定義し、様々な関数を宣言します。シンボル「LIBRARY」も定義されている場合。
// システムコール番号とインラインアセンブリコードsyscall0()も含まれています。

7 # defineLIBRARY

8 #include <unistd.h>

9 #include <time.h> // 時間型のヘッダーファイル。tm構造体、時間関数のプロト
タイプ。 10


```

11 /*
12 * インラインで必要です - カーネルスペースからフォークすると結果的に
13 execveが実行されるまで、NO COPY ON WRITE(!!!)になります。この
14 * は問題ありませんが、スタックがあります。これには
15 * main()はfork()以降、スタックを一切使用しません。したがって、どの関数も
16 * 呼び出し - つまり、フォークのインラインコードでもあります。
17 fork()'の終了時にスタックを使用します。
18 *
19 * 実際には pause と fork だけがインラインで必要です。
20 main()からはスタックを操作することはありませんが、次のように定義しています。
21 他にもいくつかあります。
22 */
// Linux は、カーネル空間でプロセスを作成する際に copy-on-write を使用しません。
// ユーザーモード(タスク0)に移行した後にインラインのfork()やpause()を行うことが保証されています。
// タスク0のユーザースタックが使用されていないこと。
// moveto_user_mode()を実行した後、main()プログラムはタスク0として実行されています。タスク0は
// 作成されるすべての子プロセスの親となります。子プロセスが作成されると
// プロセス(タスクまたはプロセス1、init)では、タスク1のコードもカーネル空間にあるので、無
// copy-on-write機能を使用します。この時点で、タスク0とタスク1は同じユーザ
// スタックスペースを一緒にしているので、実行時にスタック上での操作があると困る。
// をタスク0の環境で実行することで、スタックが混乱しないようにしています。再度 fork() を実行してから
// execve()関数を実行すると、ロードされたプログラムはもはやカーネル空間にはないので
// はコピーオンライト技術を使用することができます。5.3項の「Linuxカーネルのメモリ使用方法」を参照してく
// ださい。

// 以下の_syscall0()は、unistd.hで定義されたインラインマクロコードで、Linuxの
// システムコール割り込み0x80をエンベデッドアセンブリ形式で表示しています。この割り込みは、エントリーポ
// イント
// すべてのシスコールに対してこのステートメントは実際にはプロセスを作成するsyscall int fork().You
// を展開すれば、すぐに理解できるでしょう。syscall0の名前の最後の0は
// はパラメータを含まないことを、1はパラメータが1つあることを示します。include/unistd.hを参照してくださ
// い。
// 150~161行目
// syscall pause() : シグナルを受信するまでプロセスの実行を一時停止します。
// syscall setup (void * BIOS): linuxの初期化 (このプログラムでのみ呼び出されます)。
// syscall sync(): ファイルシステムを更新または同期させます。
23 static inline _syscall0(int, fork)
24 static inline _syscall0(int, pause)
25 static inline _syscall1(int, setup, void *, BIOS)
26 static inline _syscall0(int, sync)
27

// <linux/tty.h>では、tty_io (シリアル通信)のパラメータや定数を定義しています。
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// 初期のタスク0の//と、いくつかの組み込みアセンブリ関数のマクロ文の
// ディスクリプタのパラメータ設定と取得。
// <linux/head.h> ヘッドファイルです。セグメントディスクリプターの簡単な構造が定義されています。
// また、いくつかのセレクト定数もあります。
// <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。
// ディスクリプター/割り込みゲートなどが定義 されています。
// <asm/io.h> io のヘッダーファイルです。の io ポートを操作する関数を定義します。
// マクロの組み込みアセンブラ の形式。
// <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。
// <stdarg.h> 標準パラメータファイル。変数パラメータのリストを以下の形式で定義します。

```

//のマクロです主に、1つの型 (va_list) と3つのマクロ (va_start, va_arg
vsprintf, vprintf, vfprintf関数 では、//and va_end)を使用します。

```

// <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
// と様々な機能が宣言されていますLIBRARYが定義されて いれば、それに加えて
// システムコール番号とインラインアセンブリの_syscall0()。
// <fcntl.h> ファイル制御のヘッダファイルです。演算制御定数の定義
// ファイルとそのディスクリプターに使用されるシンボル。
// <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。
// <linux/fs.h> fsのヘッダファイル。ファイルテーブル構造の定義 (file, buffer_head, m_inode, etc.)
// <string.h> 文字列のヘッダファイルです。主に文字列操作のための組み込み関数を定義しています。
28 #include <linux/tty.h> // テレ
タイプ端末です。29 #include <linux/sched.h> // ス
ケジューラ。
30 #include <linux/head.h> //
カーネルヘッド。31 #include <asm/system.h> //
システムマシン。32 #include <asm/io.h> // ポー
トio。
33
34 #include <stddef.h> // 標準的な
定義。35 #include <stdarg.h> // 標準的な
引数。36 #include <unistd.h>
37 #include <fcntl.h> // フ
ァイル制御。38 #include <sys/types.h> //
日付の種類。39
40 #include <linux/fs.h> // ファイルシステムです。
42 #include <string.h> // 文字列
の操作。43
44 static char printbuf [1024]; // カーネルが情報を 表示するための
キャッシュ。45
46 extern char *strcpy (); // 別の場所で 定義された外部関数。
47 extern int vsprintf (); // 文字列に整形して出力 (vsprintf.c, 92). 48
extern void init (void); // 関数プロトタイプ、init (168).
49 extern void blk\_dev\_init (void); // ブロックデバイスの初期化
(blk_drv/ll_rw_blk.c, 210) 50 extern void chr\_dev\_init (void); // チャーデバイスの初
期化(chr_drv/tty_io.c, 402)
51 extern void hd\_init (void); // ハードディスクの初期化
(blk_drv/hd.c, 378) 52 extern void floppy\_init (void); // フロッピーの初期化
(blk_drv/floppy.c, 469)
53 extern void mem\_init(long start, long end ); // メモリの初期化 (mm/memory.c, 443)
54 extern long rd\_init(long mem_start, int length); // Ramdiskのinit (blk_drv/ramdisk.c, 52)
55 extern long kernel\_mktime(struct tm * tm ); // Calcのブートタイム
(kernel/mktime.c, 41) 56
// カーネル固有のsprintf()関数です。この関数を使って、フォーマットされた
// メッセージを指定されたバッファstrに出力します。パラメータ'fmt'は
// 出力が使用されるフォーマットは、標準的なC言語の
// この
// この関数は、vsprintf() を使用して、フォーマットされた文字列を str バッファに格納します。
179行目のprintf()関数。
57 static int sprintf(char * str, const char *fmt, ...)
58 {...
59 va\_list args;
60 int i;
61
62 va\_start(args, fmt)です。
63 i = vsprintf(str, fmt, args);
64 va\_end(args)です。

```

本を参照してくださ
この

```
65         return i;  
66     }。  
67  
68 /*
```

[illegible]

```
98         time.tm_min = CMOS_READ(2);    // 現在の分。  
99         time.tm_hour = CMOS_READ(4);    // カレントアワー
```

```

100         time.tm_mday = CMOS_READ(7);    // 一ヶ月に一度の日。
101         time.tm_mon = CMOS_READ(8);      // 現在の月（1～12）。
102         time.tm_year = CMOS_READ(9);     // 今年の
103     } while (time.tm_sec != CMOS_READ(0));
104     BCD_TO_BIN(time.tm_sec)です。        // BCDをバイナリ値に変換します。
105     BCD_TO_BIN(time.tm_min)です。
106     BCD_TO_BIN(time.tm_hour)です。
107     BCD_TO_BIN(time.tm_mday)です。
108     BCD_TO_BIN(time.tm_mon)です。
109     BCD_TO_BIN(time.tm_year)となります。
110     time.tm_mon--。                      // tm_monの月の範囲は0～11です。
111     startup time = kernel_mktime(& time)で // 起動時間の計算 (kernel/mktime.c, 41)
        す。
112 }
113
// main.cソースファイルからのみアクセス可能ないくつかのスタティック変数を定義します。
114 static long memory_end = 0; // マシンの物理メモリサイズ（バイト）
115 static long buffer memory_end = 0; // バッファの終了アドレス。
116 static long main_memory_start = 0; // メインメモリの開始位置  です。
117 static char term [32]; // 端末の設定（環境パラメータ）。118
// /etc/rcファイルを操作する際に使用するコマンドラインおよび環境パラメータです。
119 static char * argv_rc[] = { "/bin/sh", NULL }; // パラメータ文字列配列
120 static char * envp_rc[] = { "HOME=/", NULL, NULL }; // 環境文字列配列 121
// ログインシェルの実行に使用されるコマンドライン引数と環境パラメータです。
// 122行目のargv[0]の文字"-"は、シェルスクリプトshに渡されるフラグです。により
// このフラグを確認すると、shプログラムはログインシェルとして実行されます。その実行は
// シェルプロンプトでshを実行するのとは違います。
122 static char * argv[] = { "-/bin/sh", NULL }; // 上記
// の通りです。
123 static char * envp[] = { "HOME=/usr/root", NULL, NULL };
124
125 struct drive_info { char dummy[32]; } drive_info ; // ハードディスクのパラメー
// ターテーブル。126
// Main()は、カーネルの初期化のためのメインプログラムです。初期化の後
// 完了すると、タスク0で実行されます（アイドルタスクはアイドルタスク）。
127 void main(void) /* これは本当にvoidで、ここではエラーになりませ
// せん。*/
128 {
// スタートアップ・ルーチンは、以下のことを前提とし
// ています。
129 /*
130 割り込みは無効のままです。必要な設定を行ってから
131 それを可能にする
132 */
// まず、ルートファイルシステムのデバイス番号とスワップファイルのデバイス番号を保存します。その後、セッ
// ト
// コンソール画面の行と列の番号環境変数TERMに応じて
// setup.sプログラムで得られた情報を使用します。そして、それを使って環境を設定する
// etc/rcファイルとシェルスクリプトプログラムがinitプロセスで使用する // 変数です。次にコピー
// メモリ0x90080のハードディスクのパラメータテーブルを
// ROOT_DEV は linux/fs.h ファイルの 206 行目で extern int として宣言されており、SWAP_DEV は
// は、linux/mm.hファイルでも同じ宣言をしています。ここでのmm.hファイルは、明示的には
// linux/sched.hファイルにすでに含まれているため、プログラムの前に記載されています。
// 先に含まれています。
133     ROOT_DEV = ORIG_ROOT_DEV ; // ROOT_DEV は fs/super.c の 29 で定義されています。
134     SWAP_DEV = ORIG_SWAP_DEV ; // SWAP_DEV は mm/swap.c の 36 で 定義されていま

```

す。

135 sprintf(term, "TERM=con%dx%d", CON_COLS, CON_ROWS)。


```

136     envp[1] = term;
137     envp_rc[1] = term;
138     drive_info = DRIVE_INFO           ; // 0x90080       のhdパラメータテーブルをコピーする

// 物理的なメモリに合わせて、キャッシュやメインメモリ領域の位置を設定する
// マシンの容量です。
// キャッシュエンドアドレス -> buffer_memory_end; マシンのメモリ容量 -> memory_end;
// メインメモリのスタートアドレス -> main_memory_start.
139     memory_end = (1<<20) + (EXT MEM K<<10); // mem size = 1MB + extended mem (bytes)
140     memory_end &= 0xfffff000; // 4KB (1ページ)       未満を無視する
141     if (memory_end > 16*1024*1024) // メモリサイズ > 16MB の場
合、16MB   として計算 142         memory_end = 16*1024*1024 となります。
143     if (memory_end > 12*1024*1024) // メモリサイズが12MB以上の場合、バッファ
エンド=4MB   を設定
144         buffer_memory_end = 4*1024*1024;
145     else if (memory_end > 6*1024*1024) // サイズが 6Mb 以上の場合、バッファの端を
2Mb         に設定。
146         buffer_memory_end = 2*1024*1024;
147     その他
148         buffer_memory_end = 1*1024*1024; // それ以外の場合は buffer end = 1Mb を設定します。
149     main_memory_start = buffer_memory_end;

シンボルRAMDISKがMakefileで定義されている場合、仮想ディスクが初期化されます。この
とき
//点になると、メインメモリ領域が減少します。kernel/blk_drv/ramdisk.c を参照してください。
150 #ifdef RAMDISK
151     main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
152 #endif
// 以下は、カーネルのあらゆる面での初期化です。の方が良いでしょう。
読むときは、それぞれのinit関数に従う。
153     mem_init(main_memory_start, memory_end ); // メインのメモリ領域 (mm/memory.c, 443)
154     trap_init ( ); // トラップゲート (hwベクター) init (kernel/traps.c, 181)
155     blk_dev_init ( ); // ブロックデバイスの初期化 (blk_drv/l1_rw_blk.c, 210)
156     chr_dev_init ( ); // char dev init (chr_drv/tty_io.c, 402)
157     tty_init ( ); // ttyの初期化 (chr_drv/tty_io.c, 105)
158     time_init ( ); // 起動時間の設定 (92 行目)
159     sched_init ( ); // スケジューラの初期化 (kernel/sched.c, 385)
160     buffer_init(buffer_memory_end); // バッファの初期化 (fs/buffer.c, 348)
161     hd_init ( ); // ハードディスクの初期化 (blk_drv/hd.c, 378)
162     floppy_init ( ); // floppy init (blk_drv/floppy.c, 469)
163     sti ( ); // すべてのinitが完了したので、割り込み を有効にします。

// スタック内のデータを操作してタスク0を開始し、割り込みリターンを使用する
// の命令を行います。そしてすぐにタスク0に新しいタスク1 (initプロセスと呼ばれる) を派生させる
// そして、新しいタスクで init() を実行します。作成される子プロセスに対して、fork() は
// 0を返し、元のプロセス (親プロセス) に対しては、プロセス
// 子プロセスのpid数。
164     move_to_user_mode ( ); // ヘッドファイルinclude/asm/system.h を参照してください。
165     if (! fork(0)) { /* これがうまくいくことを期待しています */
166         init ( ); // タスク1 (initプロセス) でinit()を実行します。
167     }

// 以下のコードはタスク0で実行されます。
168 /*

```

[169](#) *注意他のタスクの 場合、'pause()'は、私たちが
[170](#) タスク0は唯一の例外です ('schedule()'参照)。

```

171 タスク0はすべてのアイドルタイム（他のタスクがないとき）に起動されるので、*。
172 を実行することができます。）タスク0の'pause()'は、単に他の
173 タスクは実行できるが、できなければここに戻る。
174 */
// pause() システムコールは、タスク0を実行する前に、割り込み可能な待ち状態に変換します。
// スケジューラ機能になります。しかし、スケジューラは、タスク0がある限り、タスク0に切り返します。
// の状態に依存せず、システム内の他のタスクが実行できないを見つけます。
// タスク 0. (kernel/sched.c, 144)を参照してください。
175     for(;;)
176         asm ("int $0x80::: \"a\" __NR_pause): \"ax\"          "); // syscall pause()
177 }
178
// printf()関数は、フォーマット情報を生成し、それを標準の
// 出力デバイスstdout(1)で表示します。パラメータ'fmt'は、表示に使用するフォーマットを指定します。
// 出力については、標準的なC言語の本を参照してください。このプログラムでは、vsprintf()を使って
// フォーマットされた文字列を printbuf バッファに格納し、 write() で内容を出力します。
// を標準出力に出力します。vsprintf()関数の実装については、kernel/vsprintf.cを参照してください。
179 static int printf(const char *fmt, ...)
180 {...
181     va_list args;
182     int i;
183
184     va_start(args, fmt)です。
185     write(1, printbuf, i=vsprintf(printbuf, fmt, args));
186     va_end(args)です。
187     return i;
188 }。
189
// init()関数は、新しく作成されたプロセス1(またはinitプロセス)で実行されます。この関数は、まず
// 最初に実行されるプログラム（シェル）の環境を初期化し、次に
// プログラムをログインシェルとして実行します。
190 void init(void)
191 {
192     int pid,i;
193
194     // setup()は、パーティションを含むハードディスクのパラメータを読み込むシスコールです。
195     // テーブル情報と仮想ディスクのロード（存在する場合）、ルートファイルのインストール
196     // システムデバイスです。この関数は25行目のマクロで定義されており、対応する
197     // 関数はsys_setup()です。その実装については kernel/blk_drv/hd.c, line 74 を参照してください。
198     setup((void *) & drive_info)。
199
200     // 端末のコンソールデバイス「/dev/tty1」を読み書き可能な状態で開く。これは
201     // システムが初めてファイルを開いたときの、結果としてのファイルハンドル番号（ファイルデスクリプタ
202     // ファイルハンドル（0）は、デフォルトのコンソール標準入力に対応しています。
203     // UNIX系OSのデバイスstdin。ここにコピーされ、個別に開かれる
204     // 標準出力ハンドルstdout(1)と標準エラーを生成するために、読み取りと書き込みで//。
205     // 出力ハンドルstderr（2）。関数の前にある「(void)」という接頭辞は、強制的に
206     // 値を返さないようにする関数です。
207     (void) open("/dev/tty1", O_RDWR, 0)となります。
208     (void) dup(0); // ファイルハンドル0を複製してstdout(1) を生成します。
209     (void) dup(0); // ファイルハンドル0を複製してstderr(2) を生成します。

```

```

// バッファブロック（1ブロック1024バイト）と総バイト数を表示するとともに
// メインメモリエリアのフリーバイト
208     printf("%d buffers = %d bytes buffer space\\r", NR_BUFFERS,
209             nr_buffers*block_size)である。
200     printf("Free mem: %d bytes_n\\r", memory_end-main memory start)

// 次に、子プロセスを作成し（タスク2）、その中で/etc/rcファイルのコマンドを実行します。
// 子プロセスを作成します。作成される子プロセスでは fork() は 0 を返し、子プロセスでは
// 元のプロセスは、子プロセスのプロセス番号pidを返します。つまり、行
// 202～206は子プロセスで実行されるコードです。子プロセスのコード
// 最初に標準入力(stdin)を/etc/rcファイルにリダイレクトしてから、/bin/shを実行します。
// このプログラムでは、標準入力からrcファイルのコマンドを読み込んでいます。
// 解釈された方法で実行されます。使用されるパラメータや環境変数は
// shでは、それぞれargv_rcとenvp_rcの配列で与えられます。
// ハンドル0を閉じてすぐに/etc/rcファイルを開くと、標準入力のリダイレクトされる
// stdinで/etc/rcファイルにアクセスします。これにより、/etc/rcファイルの内容を読むことができます。
// コンソールの読み取り操作でshは非対話的に実行されるので、次のように終了します。
// rcファイルを実行した直後に、プロセス2が終了します。の説明については
// execve()については、fs/exec.cプログラムの207行目を参照してください。関数_exit()実行時のエラーコード
// が終了します。1-操作は許可されていません。2-ファイルまたはディレクトリが存在しません。
201     if (!(pid=fork())){
202         close(0)です。
203         if (open("/etc/rc", O_RDONLY, 0))
204             _exit(1); // オープンに失敗しました (lib/_exit.c, 10)
205         execve("/bin/sh", argv_rc, envp_rc ); // プログラムの実行 /bin/sh
206         _exit(2)です。
207     }

// 以下は、プロセス1が実行するステートメントです。 wait() は、子プロセスが
// のプロセス識別子(pid)を返す必要があります。
// 子プロセスです。の終了を待つ親プロセスとして次のコードが動作します。
// 子プロセス.&i は戻り値のステータス情報を格納します。wait()の戻り値が
// が子プロセスIDと一致しない場合は、引き続き待機します。
208     if (pid>0)
209         while (pid != wait(&i))
210             /* nothing */;

// ここでコードが実行されれば、先ほど作成した子プロセスの実行が終了したことになります rc
// ファイルが存在しない)ので、子プロセスは自動的に停止または終了します。
// 以下のループでは、ログインとコンソールシェルプログラムを実行するために、再び子プロセスが作成されます
// 新しい子プロセスは、まずそれまで残っていたすべてのハンドルを閉じて、新しい
// のセッションを行います。そして、/dev/tty0をstdinとして開き直し、stdoutとstderrにコピーして、次のよう
// に実行します。
// /bin/sh プログラムを再び実行します。しかし、今回は引数と環境変数が
// シェルは別のセットです（上記122--123行目参照）。この時点で、親プロセスの
// (プロセス1)が再びwait()を実行します。子プロセスが再び停止した場合は、エラーメッセージ
// と表示され、その後もコードは試行錯誤を繰り返し、「大きな」無限ループを形成します。
211     while (1) {
212         もし ((pid=fork())<0){
213             printf("Fork failed in initr\\n").
214             を続けています。
215         }
216         if (!pid) { //新しいプロセス
217             close(0); close(1); close(2);

```

```

218         setsid                                (); // 新しいセッション      を作成します。
219         (void) open ("/dev/tty1", O\_RDWR, 0)となります。
220         (void) dup (0)です。
221         (void) dup (0)です。
222         \_exit(execve ("/bin/sh", argv, envp))です。
223     }
224     while (1)
225         if (pid == wait(&i))
226             ブレークします。
227     printf ("%d %d", pid, i)
228     sync                                (); // バッファをフラッシュします。
229 }
230 \_exit(0                                ); /* 注意! \_exit, not exit() */
// \_exit() も exit() も、関数を正常に終了させるために使われます。しかし、\_exit() は
// 直接的には sys\_exit syscall であり、exit() は通常、通常のライブラリの関数です。
// 各終了コードの実行、クローズなど、いくつかのクリーンアップ処理を行います。
// すべての標準的な I/O などを行い、その後、sys\_exit を呼び出します。
231 }
232

```

7.1.3 参考情報

7.1.3.1 CMOS

パソコンのCMOSメモリーは、電池で駆動する64バイトまたは128バイトのメモリーブロックで、通常はRTC（リアルタイムクロック）チップの一部として搭載されています。マシンによっては、より大きなCMOSメモリ容量を持つものもある。64バイトのCMOSは、もともとIBM PC-XTに搭載されていたもので、時計や日付の情報をBCD形式で格納していた。この情報は14バイトしか使われていないので、残りのバイトはシステムの設定データを格納するのに使うことができる。

CMOSのアドレス空間はベースメモリのアドレス空間の外にあるため、実行可能なコードは含まれていません。アクセスするには、I/Oポートを経由する必要があります。0x70がアドレスポート、0x71がデータポートです。指定されたオフセットのバイトを読み出すためには、まずOUT命令でバイトのオフセット値をポート0x70に送り、次にIN命令でデータポート0x71からバイトを読み出す必要があります。同様に、書き込みを行うためには、まず0x70番ポートにバイトのオフセット値を送り、次に0x71番ポートにデータを書き込む必要があります。

main.cプログラムの86行目のステートメントでは、0x80とのバイトアドレスのOR演算を行う必要はありません。当時のCMOSメモリの容量が128バイトを超えていないため、0x80とのOR演算には影響がありません。カーネル1.0以降では、この演算は削除されています(v1.0のkernel driver/block/hd.cの42行目からのコードを参照)。表7-1は、CMOSメモリの情報を簡潔にまとめたものです。

表7-1 CMOS 64バイト情報プロファイル

アドレス	説明	アドレス	説明
0x00	現在の秒数（リアルクロック）	0x11	予約
0x01	アラーム秒数	0x12	ハードドライブタイプ
0x02	現在の分（リアルクロック）	0x13	予約
0x03	アラーム秒数	0x14	デバイスバイト

0x04	現在の時間（実時間	0x15	基本メモリ（下位バイト
0x05	アラーム時間	0x16	基本メモリ（上位バイト
0x06	現在の曜日（リアルクロック	0x17	拡張メモリ（下位バイト

0x07	ある月のある日の日付（実時計	0x18	拡張メモリ（上位バイト
0x08	現在の月（リアルクロック	0x19-0x2d	予約
0x09	現在の年号（リアルタイムクロック	0x2e	チェックサム（下位バイト
0x0a	RTCステータスレジスタA	0x2f	チェックサム（上位バイト
0x0b	RTCステータスレジスタB	0x30	1MB以上の拡張メモリ（下位バイト
0x0c	RTCステータスレジスタC	0x31	1MB以上の拡張メモリ（上位バイト
0x0d	RTCステータスレジスタD	0x32	現在の世紀
0x0e	POST診断ステータスバイト	0x33	情報フラグ
0x0f	シャットダウンステータスバイト	0x34-0x3f	予約
0x10	フロッピーディスクドライブの種類		

7.1.3.2 新しいプロセスをフォークする

`Fork()`はシステムコール関数で、現在のプロセスをコピーして、元のプロセス（親プロセスと呼ぶ）とほぼ同じ内容の新しいエントリをプロセステーブルに作成し、同じコードを実行する。しかし、新しいプロセス（子プロセス）は、独自のデータ空間と環境パラメータを持っています。新しいプロセスを作成する主な目的は、プログラムを同時に実行したり、クラスタ関数`exec()`を使用して新しいプロセスで他の異なるプログラムを実行したりすることです。

フォークの戻り位置では、親プロセスは実行を再開し、子プロセスは実行を開始するだけです。親プロセスでは、`fork()`の呼び出しによって子プロセスのプロセスIDが返され、子プロセスでは、`fork()`によって0の値が返されます。このようにして、この時点ではまだ同じプログラムで実行されていますが、フォークを開始し、それぞれが独自のコードを実行しています。`fork()`の呼び出しに失敗した場合は、0よりも小さい値が返されます。図7-3に示すように。

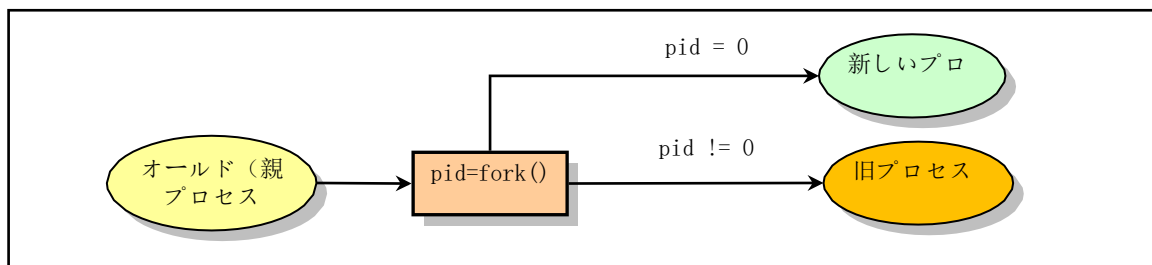


図7-3 `fork()`を呼び出して新しいプロセスを作る

`init()`は、`fork()`の戻り値を利用して、異なるコード部分を区別して実行します。`main.c`プログラムの201行目と216行目は、子プロセスが実行を開始するコードブロック（`execve()`システムコールを使用して他のプログラムを実行し、そこで`sh`が実行される）であり、208行目と224行目は、親プロセスが実行するコードブロックである。

`exit()`は、プログラムの実行が終了したときや、終了させる必要があるときに呼び出すことができます。この関数は、プロセスを終了させ、そのプロセスが占有していたカーネルリソースを解放します。親プロセスは、`wait()`コールを使用して子プロセスの終了を表示または待機し、終了したプロセスの終了ステータス情報を取得することができます。

7.1.3.3 セッションのコンセプト

先に述べたように、プログラムとは実行可能なファイルのことであり、プロセスとは実行中のプ

プログラムのインスタンスのことである。カーネルでは、各プロセスは、プロセス識別番号pid（Process ID）と呼ばれる0よりも大きな異なる正の整数で識別されます。プロセスは、fork()によって1つまたは複数の子プロセスを作成することができます。

で、これらのプロセスはプロセスグループを形成することができます。例えば、シェルのコマンドラインで入力したパイプコマンドの場合。

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

cat、grepなどのコマンドは、それぞれプロセスグループに属しています。

プロセスグループとは、1つまたは複数のプロセスを集めたものです。プロセスと同様に、各プロセスグループには固有のプロセスグループ識別番号gid（グループID）があり、これも正の整数です。各プロセスグループには、グループリーダーと呼ばれるプロセスがあります。グループリーダーのプロセスは、pidがプロセスグループ番号gidと等しいプロセスです。プロセスグループの概念には多くの用途がありますが、最も一般的なものは、端末上のフォアグラウンド実行プログラムに終了信号を発行し（通常はCtrl-Cキーを押して）、プロセスグループ全体のプロセスを終了させることです。例えば、上記のパイプコマンドに終了信号を発行すると、3つのコマンドが同時に実行を終了します。

セッションとは、1つまたは複数のプロセスグループの集合体である。通常、ユーザーがログインした後に実行されるプログラムはすべてセッションに属しており、ログインシェルがセッションリーダーであり、それが使用する端末がセッションの制御端末となります。そのため、セッションの最初のプロセスは、一般的に制御プロセスとも呼ばれています。ログアウトすると、セッションに属するすべてのプロセスが終了しますが、これがセッションという概念の主な用途のひとつです。setsid()関数は、新しいセッションを作成するために使用され、通常、環境イニシャライザによって呼び出されます。プロセス、プロセスグループ、セッション期間の関係を図7-4に示します。

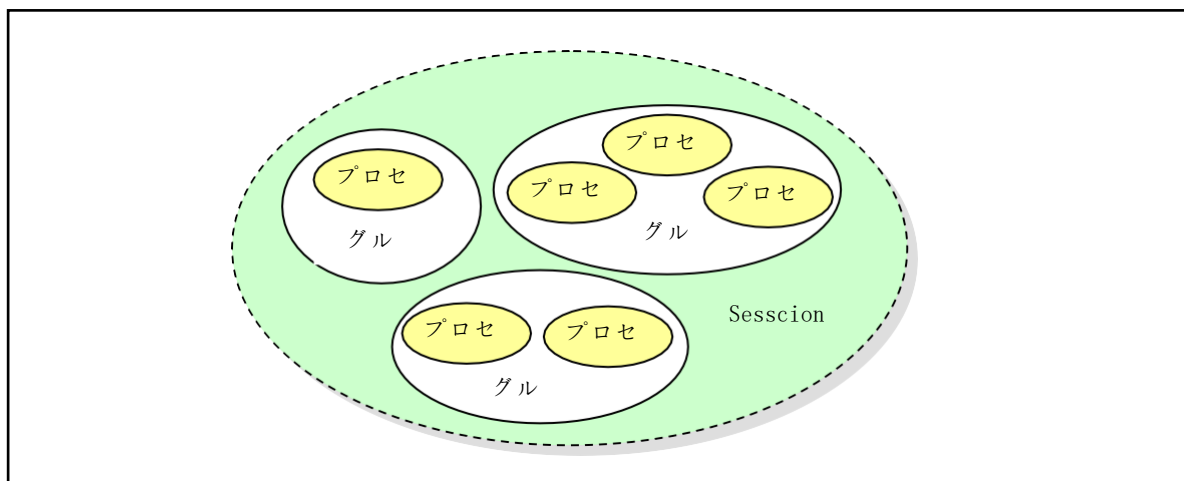


図7-4 プロセス、プロセスグループ、セッションの関係

セッション内のいくつかのプロセスグループは、フォアグラウンドプロセスグループと1つまたは複数のバックグラウンドプロセスグループに分けられます。端末は、セッションの制御端末としてのみ機能します。フォアグラウンドプロセスグループとは、セッション内で制御端末を持つプロセスグループのことで、セッション内の他のプロセスグループはバックグラウンドプロセスグループになります。制御端末は、/dev/ttyデバイスファイルに対応しているので、プロセスが制御端末にアクセスする必要がある場合は、/dev/ttyファイルを直接読み書きすることができます。

7.2 環境の初期化

カーネルを初期化した後、一般的なシステムの動作環境を実現するためには、特定の設定に応じてさらに環境の初期化を行う必要があります。上記の205行目と222行目では、`init()`関数がコマンドインタプリタ（シェル）の`/bin/sh`を直接実行していますが、実際に利用できるシステムではそうはいきません。ログイン機能や複数の人が同時にシステムを利用する機能を持たせるために、通常のシステムでは、システム環境の初期化プログラムである`init.c`をここやそれに類する場所で実行し、`/etc/ディレクトリ`内の設定ファイルの設定情報に基づいてプログラムを実行することになります。システムでサポートされている各端末機器は、子プロセスを作成し、その子プロセスの中で端末初期化設定プログラム`agetty`（総称して`getty`と呼ぶ）を実行する。ゲッティは、ユーザーのログインプロンプトメッセージ「`login:`」を端末に表示する。ユーザーがユーザー名を入力すると、`getty`はログインプログラムに置き換えられる。ユーザの入力したパスワードの正しさを確認した後、ログインプログラムは最後にシェルプログラムを呼び出し、シェルの対話インタフェースに入る。両者の実行関係を図7-5に示す。

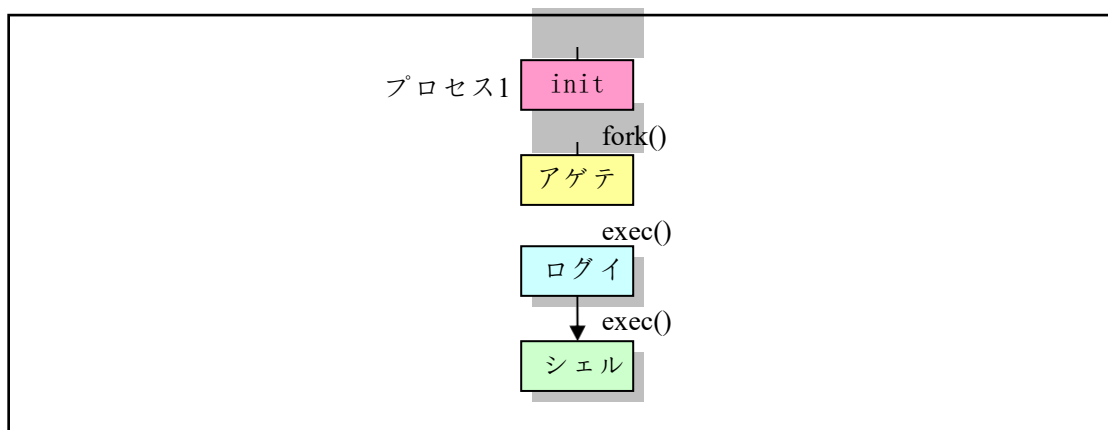


図7-5 環境初期化の手順

これらのプログラム（`init`、`getty`、`login`、`shell`）はカーネルの一部ではありませんが、これらの役割を基本的に理解することで、カーネルが提供する多くの機能を理解することができます。

`init`プロセスの主な仕事は、`/etc/rc`ファイルに設定されたコマンドを実行し、`/etc/inittab`ファイルの情報に従って、`fork()`を使ったログインが許可されている端末機器ごとに子プロセスを作成し、新しく作成された各子プロセスで`agetty(getty)`プログラムを実行することです。この時点で、`init`プロセスは`wait()`を呼び出し、子プロセスの終了を待ちます。その子プロセスの一つが終了して終了するたびに、`wait()`が返す`pid`によって、対応する端末のどのサブプロセスが終了したかがわかるので、対応する端末装置のために新しいサブプロセスを作成し、その子プロセスで`agetty`プログラムを再実行する。このようにして、許可された各端末装置には、常に対応するプロセスが処理を待っています。

通常の運用では、`init`は、ユーザがログインできるようにするためと、オーファン（孤児）プロセスを回収するために、`agetty`が動作していると判断します。孤児プロセスとは、親プロセスが終了したプロセスのことです。Linuxではすべてのプロセスが1つのプロセスツリーに属していなければならないので、オーファンプロセスを集めなければなりません。システムがシャットダウンされると、

initは他のすべてのプロセスを終了させ、すべてのファイルシステムをアンマウントし、プロセッサを停止させるなど、設定されている作業を行います。

gettyプログラムの主な仕事は、端末の種類、プロパティ、速度、ラインディシプリンを設定することです。ttyポートを開いて初期化し、プロンプトを表示し、ユーザーがユーザー名を入力するのを待ちます。このプログラムは、スーパーユーザーのみが実行できます。通常、`/etc/issue`テキストファイルが存在する場合、gettyはまずテキストメッセージを表示し、次にログインプロンプト情報を表示し（例：`plinux login:`）、ユーザーが入力したログイン名を読み取ってから、ログインプログラムを実行します。

ログインプログラムは、主にログインユーザにパスワードの入力を求めるために使用される。ユーザが入力したユーザ名に応じて、パスワード・ファイル`passwd`から対応するユーザのログイン項目を取得し、`getpass()`を呼び出して「password:」のプロンプト・メッセージを表示し、ユーザが入力したパスワードを読み取り、入力されたパスワードに暗号化アルゴリズムを使用して、パスワード・ファイルのユーザ・エントリの`pw_passwd`フィールドと比較する。ユーザーが入力したパスワードが無効な場合、ログインプログラムは、ログイン処理に失敗したことを示すエラーコード1で終了する。この時、親プロセス（プロセス`init`）の`wait()`が終了プロセスの`pid`を返すので、`init`は記録された情報に従って再度子プロセスを作成し、子プロセス内の端末装置に対して再度アゲハプログラムを実行する。これが上記の処理を繰り返す。

ユーザが入力したパスワードが正しければ、`login`は、現在の作業ディレクトリをパスワードファイルで指定されたユーザの開始作業ディレクトリに変更し、端末装置へのアクセス権をユーザ`read/write`、グループ`write`に変更し、プロセスのグループIDを設定する。そして、得られた情報をもとに、起動ディレクトリ(`HOME=`)、使用するシェルスプログラム(`SHELL=`)、ユーザー名(`USER=`、`LOGNAME=`)、システム・エグゼクティブのデフォルトパス・シーケンス(`PATH=`)などの環境変数情報を初期化します。次に、`/etc/motd`ファイルのテキストメッセージ（message-of-the-day）が表示され、ユーザーがメール情報を持っているかどうかチェックされます。最後に、ログインプログラムは、ログインしたユーザーのユーザーIDに変更し、パスワードファイルのユーザー項目で指定されたシェルスプログラム（`bash`や`csh`など）を実行します。

パスワードファイル `/etc/passwd` のシェル名に使用するシェルが指定されていない場合は、デフォルトの `/bin/sh` プログラムが使用されます。ユーザーのホームディレクトリが指定されていない場合は、デフォルトのルートディレクトリ `/` が使用されます。ログインプログラムの実行オプションや特別なアクセス制限については、Linuxのオンラインマニュアルページ(`man 8 login`)を参照してください。

シェルスプログラムとは、ユーザーがシステムにログインしてインタラクティブな操作を行う際に実行される複雑なコマンドラインインタプリタである。ユーザーがコンピューターと対話する場所である。シェルはユーザーが入力した情報を受け取り、コマンドを実行する。ユーザーはターミナル上のシェルと直接対話することもできるし、シェルスクリプトファイルを使ってシェルに入力することもできる。

ログイン時にシェルの実行を開始すると、パラメータ`argv[0]`の最初の文字が`'i'`となり、ログインシェルとして実行されることを示します。この時点で、シェルスプログラムは、その文字に基づいて、ログイン処理に対応するいくつかの操作を行う。ログインシェルは、まず、コマンドを`/etc/profile`ファイルと`.profile`ファイル（存在する場合）を読み込んで実行します。シェルに入るときに環境変数`ENV`が設定されている場合、またはシェルにログインするときに`.profile`ファイルに環境変数が設定されている場合、シェルは次にファイルからコマンドを読み込んで実行します。したがって、

ユーザはログイン時に実行するコマンドを`.profile`ファイルに記述し、シェルを実行するたびに実行するコマンドをENV変数で指定されたファイルに記述する必要があります。環境変数ENVを設定する方法は、ホームディレクトリの`.profile`ファイルに次のような記述をすることである。

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

シェルを実行する際に、指定されたいくつかのオプションに加えて、コマンドライン引数が

が指定されている場合、シェルは最初の引数をスクリプトファイル名として扱い、コマンドを実行します。残りの引数は、シェルのパラメータ（\$1、\$2など）として扱われます。それ以外の場合、シェルスクリプトプログラムはその標準入力からコマンドを読み込みます。

シェルスクリプトプログラムを実行するには多くのオプションがありますが、Linuxシステムのshのオンラインマニュアルページの説明を参照してください。

7.3 概要

本章のコード解析により、カーネル0.12では、ルートファイルシステムがMINIXであり、ファイル/etc/rc、/bin/sh、/dev/*、およびいくつかのディレクトリ（/etc/、/dev/、/bin/、/home/、/home/root/）を作成することで、Linuxシステムを動かすためのシンプルなルートファイルシステムを形成することができます。

以降の章を読む際には、main.cプログラムを本線として使用することができますので、章順に読む必要はありません。なお、メモリページングの仕組みを理解していない方は、まず第10章のメモリ管理の内容を読むことをお勧めします。

次の章の内容をスムーズに理解するためにも、この機会に32ビットの保護モード動作の仕組みを確認していただければと思います。プロテクションモードについては、付録の関連内容を詳しく読むか、インテル80x86の書籍を参照してください。

ここまで章立て順に來れば、Linuxカーネルの初期化プロセスを大まかに理解しているはずです。しかし、次のような疑問もあるでしょう。"一連のプロセスを生成した後、システムはどのようにしてこれらのプロセスを時分割で実行するのか、あるいはどのようにして実行するようにスケジューリングするのか。それが「車輪」の回り方なのか？"。答えは複雑ではありません。カーネルは、schedule()とタイマクロック割り込みハンドラ_timer_interruptを実行することで実行されます。カーネルは10ミリ秒ごとにクロック割り込みを設定し、割り込み処理中にdo_timer()関数を呼び出して全プロセスの現在の実行状況を確認することで、次のプロセスの状態を決定します。各プロセスの状態に応じて、スケジューラは各プロセスを順次実行するようにスケジューリングします。

一時的にリソースが不足してしばらく待つ必要がある場合は、sleep_on()を介して間接的にschedule()を呼び出し、自発的にCPU権を他のプロセスに渡します。システムが次にどのプロセスを実行するかについては、すべてのプロセスの現在の状態と優先度に応じてschedule()が完全に決定します。常に実行可能な状態にあるプロセスの場合、クロック割り込みプロセスが実行中のタイムスライスを使い切ったと判断すると、do_timer()でプロセススイッチ操作が行われ、そのプロセスのCPU使用権が不本意ながら奪われ、カーネルは他のプロセスを実行するようにスケジューリングします。

スケジューリング関数schedule()とクロック割り込みハンドラ手続きは、次章の重要なトピックの一つです。

8カーネルコード（カーネル

linux/kernel/ディレクトリには、リスト8-1に示すように、10個のCファイルと2個のアセンブリ言語ファイル、およびMakefileが含まれています。3つのサブディレクトリのコードについては、後の章で解説します。本章では、主にこの12個のコードファイルについて説明します。まず、すべてのプログラムの基本的な機能について一般的な紹介を行い、これらのカーネルコードで実装されている機能や呼び出し関係を大まかに理解できるようにしてから、各コードファイルについて詳細なコメントを行う。

リスト 8-1 linux/kernel/

		ファイル名		サイズ	最終更新時刻 (GMT)	説明
	blk_drv/				1992-01-16 14:39:00	
	christenin				1992-01-16 14:37:00	
	gdrv/					
	数学/				1992-01-16 14:37:00	
	Makefile	4034	バイト		1992-01-12 19:49:12	
	asm.s	2422	バイト		1991-12-18 16:40:03	
	exit.c	10554	バイト		1992-01-13 21:28:02	
	fork.c	3951	バイト		1992-01-13 21:52:19	
	mktime.c	1461	バイト		1991-10-02 14:16:29	
	panic.c	448	バイト		1991-10-17 14:22:02	
	printk.c	537	バイト		1992-01-10 23:13:59	
	sched.c	9296	バイト		1992-01-12 15:30:13	
	signal.c	5265	バイト		1992-01-10 00:30:25	
	sys.c	12003	バイト		1992-01-11 00:15:19	
	sys_call.s	5704	バイト		1992-01-06 21:10:59	
	traps.c	5090	バイト		1991-12-18 19:14:43	
	vsprintf.c	4800	バイト		1991-10-02 14:16:29	

8.1 主な機能

このディレクトリ内のコードファイルは、図8-1のように、ハードウェア（例外）割り込みハンドラファイル、システムコールサービスハンドラファイル、プロセススケジューリングなどの一般機能ファイルの3つに分けられます。ここでは、この分類に基づいて実装されている機能をより詳しく説明します。

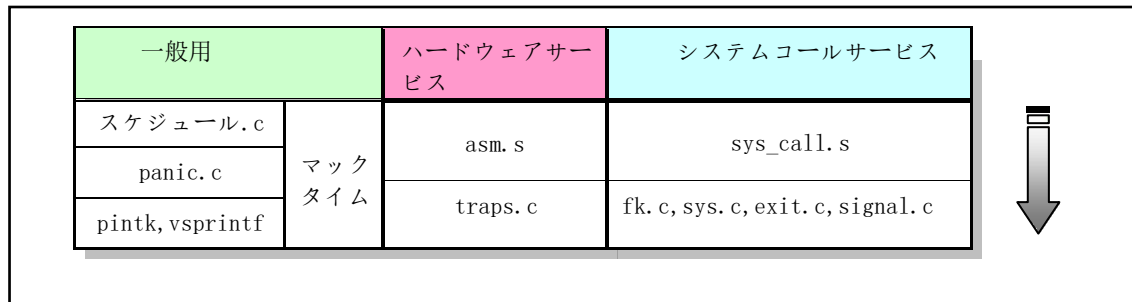


図8-1 各ファイルの呼び出しの階層関係

8.1.1 割り込み処理

割り込み処理に関連するファイルには、`asm.s`と`traps.c`の2つがあります。`asm.s`は、ほとんどのハードウェア例外による割り込みサービスのアセンブリ処理部分を実装するために使用され、`traps.c`は`asm.s`の割り込み処理で呼び出されるC関数部分を実装しています。このC関数部分は、割り込み処理のボトムハーフと呼ばれることもあります。このほか、いくつかのハードウェア割り込みハンドラが`sys_call.s`および`mm/page.s`ファイルに実装されています。PCに搭載されている8259Aプログラマブル割り込み制御チップの接続と機能については、図5-21を参照してください。

Linuxシステムでは、割り込みサービス機能はカーネルが提供しているため、割り込みハンドラはプロセスのカーネル状態スタックを使用します。ユーザプログラム（プロセス）が割り込みハンドラに制御を移す前に、CPUはまず、少なくとも12バイト（EFLAGS、CS、EIP）を割り込みハンドラのスタック（プロセスのカーネル状態スタック）にプッシュします。図8-2(a)参照。この状況は、ファークール（セグメント間呼び出し）と同様です。CPUはコード・セグメント・セレクタとリターン・アドレスのオフセットをスタックにプッシュします。また、80X86 CPUは、割り込みコードのスタックではなく、宛先コード（割り込みハンドラコード）のスタックに情報をプッシュする点も、セグメント間呼び出しと似ています。また、ユーザーレベルからカーネルレベルになるなど優先度が変わった場合、CPUは元のコードのスタックセグメントSSやスタックポインタESPも割り込みハンドラのスタックにプッシュします。しかし、カーネルが初期化された後、カーネルコードはプロセスのカーネルステートスタックを使用して実行されるので、ここでいう転送先コードのスタックとはプロセスのカーネルステートスタックを指し、割り込みコードのスタックはもちろんプロセスのユーザステートスタックです。ですから、割り込みが発生すると、割り込みハンドラはプロセスのカーネル状態スタックを使用します。また、CPUは常にEFLAGSの内容をスタックにプッシュしています。優先度の変化に伴うスタックの内容の模式図を図8(c)、(d)に示します。

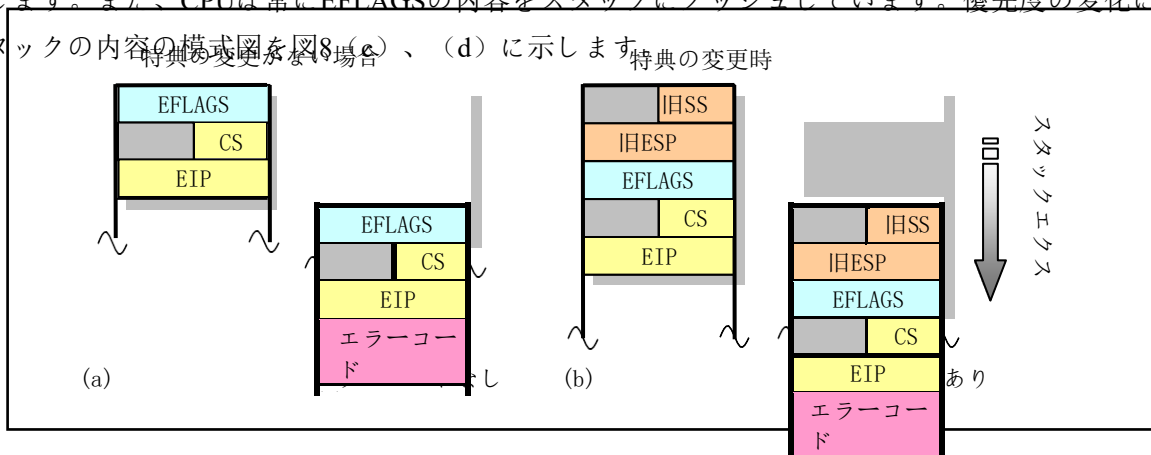


図8-2 割り込み発生時のスタックの内容

asm.sのコードファイルは、主にインテル社の予約割り込みINT0--INT16の処理を扱っており、残りの予約割り込み（INT17--INT31）はインテル社が将来の拡張のために予約しています。PICチップのIRQピンに対応する16個の割り込み（INT32--INT47）の処理は、クロック、キーボード、フロッピーディスク、演算コプロセッサ、ハードディスクなどの各種ハードウェアの初期化ルーチンで設定されます。Linuxのシステムコール割り込みINT128(INT0x80)のハンドラはkernel/sys_call.sに記述されています。各割り込みの具体的な定義は、コードファイルのコメントの後にある参考情報のセクションに記述されています。

一部の例外によって割り込みが発生した場合、CPUは内部でエラーコードを生成してスタックにプッシュしますが（図8-2(b)の例外割り込みINT8やINT10～INT14など）、それ以外の割り込みではこのエラーコードを持たない（例えば、ゼロ除算エラーやバウンダリチェックエラーなど）ため、asm.sはエラーコードを持っているかどうかで割り込みを2種類に分けています。しかし、処理内容はエラーコードがない場合と同じです。ハードウェア例外による割込みの処理を図8-3に示します。

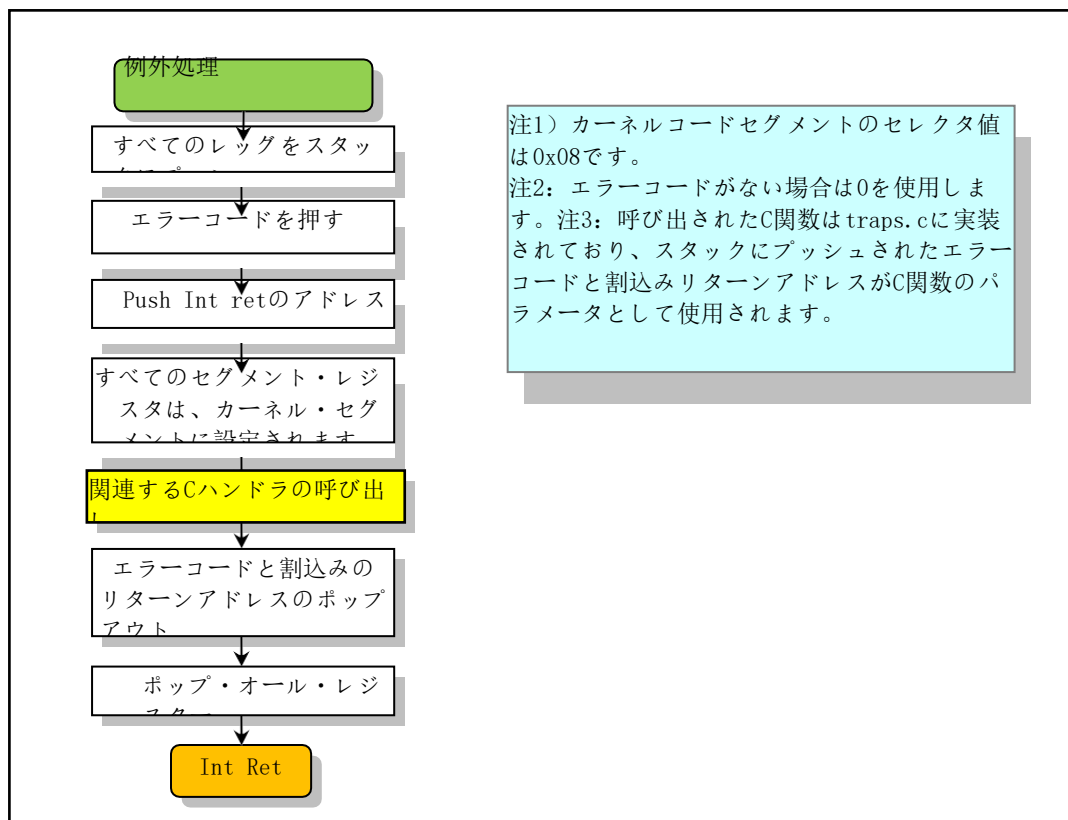


図 8-3 ハードウェア例外（フォールト，トラップ）処理フロー

8.1.2 システムコールの処理

Linuxでは、アプリケーションがシステムリソースを使用する際に、割り込みINT0x80を使用する必要があり、関数呼び出し番号をレジスタeaxに入れる必要があります。また、割り込みハンドラにパラメータを渡す必要がある場合は、レジスタebx、ecx、edxにパラメータを格納しま

す。したがって、割り込み呼び出しはシステムコール (syscall) と呼ばれます。システムコールを実装するための関連ファイルには、`sys_call.s`, `fork.c`, `signal.c`, `sys.c`があります。

とexit.cファイルがあります。

sys_call.sは、ハードウェア割り込み処理のasm.sプログラムのよう動作します。さらに、このプログラムは、クロック割り込み、ハードディスクやフロッピーディスクの割り込みを処理します。fork.cとsignal.cの割り込みサービス関数は、traps.cプログラムと同様に、システム割り込み呼び出しのためのCハンドラーを提供します。fork.cは、プロセスを作成するための2つのCハンドラー、find_empty_process()とcopy_process()を提供します。signal.cは、システムコールの割り込み処理の際に使用されるプロセス信号の処理に関する関数do_signal()を提供する。また、4つのシステムコールの実装も含まれています。

sys.cおよびexit.cプログラムは、他のいくつかのsys_xxx()システムコール関数を実装しています。これらのsys_xxx()関数は、対応するシステムコールに対して呼び出されるハンドラーです。関数の中には、sys_execve()のようにアセンブリ言語で実装されているものもあれば、C言語で実装されているものもあります（例えば、signal.cにある4つのsyscalls関数など）。

これらの割り込み関連の関数の名前の簡単な命名規則に基づいて、これを理解することができます。通常、'do_'で始まるC言語の関数は、システムコールに共通の関数か、システムコールに固有の関数であり、また、'sys_'で始まる関数は、通常、指定されたシステムコールの特別なハンドラです。例えば、do_signal()は基本的に全てのシステムコールが実行しなければならない関数ですが、sys_pause()やsys_execve()はシステムコールに特有のCプロセッサ関数です。

8.1.3 その他の汎用プログラム

これらのプログラムには、schedule.c、mktime.c、panic.c、printk.c、vsprintf.cなどがあります。

Schedule.cには、最も頻繁に使用される関数 schedule()、sleep_on()、wakeup()が含まれています。これはカーネルのコアスケジューラであり、プロセスの実行を切り替えたり、プロセスの実行状態を変更するために使用されます。また、システムクロックの割り込みやフロッピードライブのタイミングなどの機能も含まれています。mktime.cには、init/main.cの中で一度だけ呼び出される、カーネルが使用する時刻関数mktime()が含まれています。panic.cには、カーネルにエラーが発生したときにエラーメッセージを表示し、カーネルを停止させるpanic()関数が含まれています。printk.cとvsprintf.cは、カーネル固有の表示関数printk()と文字列形式の出力関数vsprintf()を実装したカーネルサポートプログラムです。

8.2 asm.s

8.2.1 機能説明

asm.sアセンブリファイルには、CPUで検出されるほとんどの例外に対するハンドラ手続きの基本コードと、数学コプロセッサ（FPU）の例外処理コードが含まれています。このプログラムはtraps.cと密接な関係にあり、割り込みハンドラ内でtraps.c内の対応するC関数プログラムを呼び出し、エラー箇所とエラーコードを表示した後、割り込みを終了するのが主な処理方法です。

このコードを読む際には、図8-4の現在のタスクのカーネルスタック変化図を参照すると便利です（各行は4バイトを表します）。エラーコードのない割り込み処理の場合、図8-4(a)を参照してスタックポインタの位置の変化を確認します。スタックポインタespは、対応する割り込みサービスルーチ

ンの実行を開始する前の割込みリターンアドレス（図中esp0）を参照しています。do_divide_error()やその他のC関数のアドレスがスタックにプッシュされると、ポインタ位置はesp1になります。この時点で、プログラムはswap命令を使用して関数のアドレスをeaxレジスタに入れ、元のeaxの値はスタックに保存されます。プログラムがいくつかのレジスタをスタックに格納した後、スタックポインタの位置はesp2になります。do_divide_error()の正式な呼び出しの前に、プログラムは元の関数のアドレスをプッシュします。

割り込みハンドラの実行開始時にeipをスタックに乗せ（つまりesp3の位置に置き）、プラス8でesp2の位置に戻してから全レジスタをポップアウトします。

CPUがエラーコードを発生させる割り込みの場合、スタックポインタの位置の変化は図8-4（b）を参照してください。割り込みサービスルーチンが実行される直前のスタックポインタは、図中のesp0を指しています。呼び出されるdo_double_fault()などのC関数のアドレスがスタックにプッシュされた後、スタックポインタの位置はesp1になります。このとき、プログラムは2つの交換命令を用いてeaxレジスタとebxレジスタの値をesp0とesp1の位置に保存し、エラーコードをeaxレジスタに交換し、関数アドレスはebxレジスタにスワップされます。その後の処理は、前述の図8-4(a)と同様です。

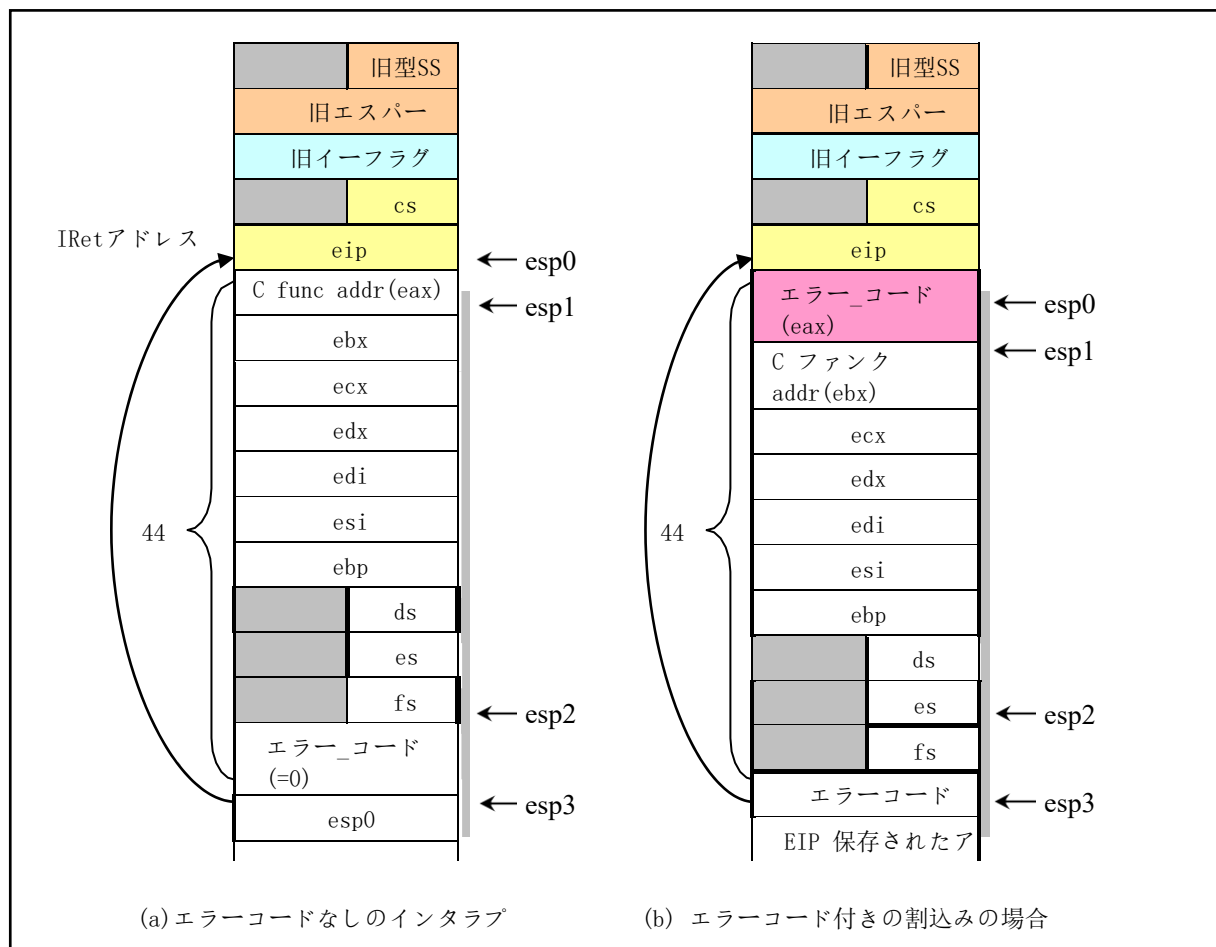


図8-4 エラー処理時のカーネルスタック変化の模式図

do_divide_error()を正式に呼び出す前にエラーコードとesp0をスタックに置くのは、エラーコードとesp0をパラメータとして使用するためです。traps.cでは、関数のシグネチャは次のようになっています。

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

そのため、このC関数でエラーの位置とエラーコードを出力することができます。ファイル内の残りの例外の処理は、基本的にここで説明した処理と同様です。

8.2.2 コードアノテーション

ヨン

プログラム 8-1 linux/kernel/asm.s

```

1  /*
2  * linux/kernel/asm.s
3  *
4  * (C) 1991 Linus Torvalds 5
5  */
6
7  /*
8  * asm.sには、ほとんどのハードウェア・フォールトの低レベルコードが含まれています。
9  * page_exceptionはmmで処理されるので、ここではありません。この
10 * ファイルは、TS-bit による fpu-exception も（できれば）処理します。
11 * fpuが適切に保存/復元されている必要があります。これはテストされ
    ていません。12 */
    TS - Task Switched、CR0のビット3。
13
14 # このファイルでは主にインテル予約割り込みINT0～INT16の処理を行います。# 以下は、実際に
    traps.cで行われている、いくつかのグローバル関数の宣言です。
15 .globl _divide_error, _debug, _nmi, _int3, _overflow, _bounds, _invalid_op
16 .globl _double_fault, _coprocessor_segment_overnrun
17 .globl _invalid_TSS, _segment_not_present, _stack_segment
18 .globl _general_protection, _coprocessor_error, _irq13, _reserved
19 .globl _alignment_check
20
21 # 以下の手順では、エラーコードのない例外を扱います。
22 # Int0 --
    ゼロで割ったときのエラーを処理します。タイプ：
    エラー;エラーコード。エラーコード：なし。
23
24 DIVまたはIDIV命令を実行する際、除数が0であれば、CPUは
25 # この例外が発生します。この例外は、EAX(またはAX, AL)が正当な除算の結果を保持していない場
    合にも # 発生します。21行目の「_do_divide_error」というラベルは、実際にはC関数
    do_divide_error()から # コンパイルされたオブジェクト内の対応する名前です。
    この関数はtraps.cの101行目にあります。
26
27 _divide_error:
28     pushl                                $_do_divide_error# 最初にC関数のアド
    レス                                no_error_code:# エラーコード処理のエ
    ントリ                                はありません。
29     xchgl                                %eax, (%esp)# EAX で交換されたスタックでの_do_divide_error
30     pushl %ebx
31     pushl %ecx
32     pushl %edx
33     pushl %edi
34     pushl %esi
35     pushl %ebp
36     push %ds
    スタック内で4バイトを占有していた。
37     push %es
38     push %fs
39     pushl $0
    # "error                                code "# 0 as error code
40     lea 44(%esp), %edx
    # espからedxで有効なaddrを取得し、ポイントする
41     pushl %edx
    #中断された元のコードのIP(esp0にて)
42     movl $0x10, %edx
    # ds, es, fsをカーネルデータのセグメンテーションセレ
    クタに初期化します。

```

```
37      mov %dx,%ds  
38      mov %dx,%es
```

```

39      mov %dx,%fs
# 以下の文中の'*'は、レジスタ値をアドレスで示しています。この
# ステートメントは、オペランドで指定されたアドレスのルーチンへの呼び出しを表します。# これは間接呼び出しです。この文の意味は、Cルーチン呼び出すことです。
do_divide_error()のように # 指定されます。これは、Cルーチンの2つのパラメータ(33行目と35行目
でスタックにプッシュされた)を破棄し、 # スタックポインタespがレジスタfsの位置を指すように
するために使用されるポップ命令を2回実行したと # 同じです。
40      do_divide_error(long esp, long error_code)のように*%eax# を呼び出します。
41      addl $8,%esp
42      ポップ %fs
43      ポップ %es
44      ポップ %ds
45      popl %ebp
46      popl %esi
47      popl %edi
48      popl %edx
49      popl %ecx
50      popl %ebx
51      popl                                %eax# recover original EAX
52      アイレット
53
# Int1 -- デバッグ割り込みのエントリーポイント。タイプです。Error/Trap
(Fault/Trap); エラーコードなし。# この例外は EFLAGS の TF フラグが設定
されているときに発生します。CPUは、この
ハードウェアブレークポイントが検出されたとき、命令トレーストラップやタスクスワップ # プ
トラップがオンになったとき、またはデバッグレジスタのアクセスが無効なときに # 例外が発生
します(エラー)。
54 _debug:
55      pushl                                $_do_int3    # _do_debug# push Cルーチンのアドレス
56      jmp                                no_error_code# 22      行目です。
57
# Int2 - Non Maskable Interrupt エントリーポイント。タイプです。Trap; エラーコードなし。
# ハードウェア割り込みの中で唯一、固定の割り込みベクターが与えられています。NMI信号を受信
するたびに、CPUは内部的に割り込みベクター2を生成し、標準的な割り込みアクトリッジサイクル
を実行するため、時間を節約することができます。NMIは通常、次のような場合に使用されます。
極めて重要なハードウェアイベントで使用されます。CPUがNMI信号を受信して
#が割り込みハンドラの実行を開始すると、それ以降のハードウェア割り込みはすべて無視されます。
58 _nmi:
59      pushl $_do_nmi
60      jmp no_error_code
61
# Int3 - ブレークポイント命令のエントリーポイントです。タイプです。Trap; エラーコードはありません。
# int 3命令による割り込みは、ハードウェア割り込みとは独立しています。# この命令は通常、デ
バグによってコードに挿入され、処理が
#方法は_debugと同じです。
62 _int3:
63      pushl $_do_int3
64      jmp no_error_code
65
# Int4 -- オーバーフローエラー割り込みのエントリーポイント。タイプです。Trap; エラーコードなし。
# この割り込みは、EFLAGSにOFフラグが設定されているときに、CPUがINTO命令を # 実行すること
で発生します。通常、コンパイラは算術演算のオーバーフローを追跡するために使用します。
66 _オーバーフロー。

```


[67](#)`pushl $do_overflow`

69

オペランドが有効範囲外の場合に発生する割り込みです。この割り込み#はBOUND命令が失敗したときに発生します。BOUND命令は3つのオペランドを持ちます。最初の1つが他の2つの間にない場合、# 5の例外が発生します。

```
71      pushl  $_do_bounds
```

73

Int6 -- 無効なオペコード割込みエントリ ポイントです。タイプです。
エラー; エラーコード なし。# CPUのアクチュエータ
が無効なオペコードを検出したことによる割り込みです。

```
75      pushl $do invalid op
```

77

Int9 -- コプロセッサのセグメントオーバーランのエントリポイントです。タイプです。放棄; エラーコードなし。# この例外は、基本的にコプロセッサのエラー保護と同じです。なぜなら、以下の場合
浮動小数点演算命令のオペランドが大きすぎると、データセグメントを超えた浮動小数点値をロード # またはセーブする機会があります。

```
79      pushl $ do_coprocessor_segment_overrun
```

81

82 reserved:

```
83      pushl $do_reserved
```

```
84      jmp no error code
```

85

```
# Int45 -- (0x20 + 13) Linuxカーネルが設定する数学コプロセッサのハードウェア割り込み。
# コプロセッサが演算を行うと、IRQ13の割り込み信号を発行して、# 演算が完了したことをCPU
に知らせます。80387が演算を行っているときは、# CPUはその演算が完了するのを待ちます。以
下の89行目では、0xF0がコプロセッサの
# ラッチのクリアに使用されるポートです。このポートに書き込むことで、この割り込みは、# CPU
のBUSY継続信号を除去し、80387プロセッサ拡張要求ピン # PEREQを再起動させます。この動作は主
に、80387の命令を継続して実行する前に、CPUがこの割り込みに応答することを# 確認するための
ものです。
```

```
87      pushl %eax
```

```
88      xorb %al,%al
```

```
89      outb %al, $0xF0
```

```
90      movb  $0x20,%a1
```

[91](#) outb %al, \$0x20# は 8259 のマスターチップ に EOI (End of Interrupt) を送信した。

```
92      imp      lf# delay a
```

```
while.93      1: jmp 1f
```

94 1:outb %al,\$0xA0# は 8259 のスレーブチップ^o に EOI を送った。

95 popl %eax

```
96      jmp      _coprocessor_error#
```

code in system call.s 97

以下の割り込みが呼ばれた場合、CPUはリターンアドレスを割り込ませた後に # エラーコードを

スタックにプッシュするので、リターン時にもエラーコードを # ポップアップする必要があります（図5.3(b)参照）。

Int8 -- 二重故障。タイプです。放棄；エラーコードがあります。

通常、CPUが例外ハンドラを起動して新たな例外を検出した場合、2つの # 例外は連続して処理することができる。しかし、CPUが2つの例外を連続して処理できない状況がいくつかあり、このときに二重障害例外が発生します。

```

98 _double_fault:
99     pushl                                $_do_double_fault# スタック      にプッ
シュされた C ルーチンの addr。100 error_code:
101     xchgl                                %eax,4(%esp)# エラーコード <-> %eax, 元の eax は保存されたスタック
      にあります。
102     xchgl                                %ebx,(%esp)# &function <-> %ebx, 元の ebx はスタック に格納されてい
      ます。
103     pushl %ecx
104     pushl %edx
105     pushl %edi
106     pushl %esi
107     pushl %ebp
108     push %ds
109     プッシュ %es
110     push %fs
111     pushl                                %eax# エラーコード
112     レア                                44(%esp),%eax# オフセット
113     pushl %eax
114     movl                                $0x10,%eax# カーネル・データ・セグメント・セレクトア      を設定します。
115     mov %ax,%ds
116     mov %ax,%es
117     mov %ax,%fs
118     call                                *%ebx# Cルーチン への間接的な呼び出し
119     addl                                $8,%esp# 使用したパラメータを破棄します。
120     ポップ %fs
121     ポップ %es
122     ポップ %ds
123     popl %ebp
124     popl %esi
125     popl %edi
126     popl %edx
127     popl %ecx
128     popl %ebx
129     popl %eax
130     アイレット
131

```

Int10 -- 無効なタスクステータスセグメント(TSS)です。 タイプです。エラー；エラーコードがあります。# CPUがプロセスに切り替えようとしたところ、そのプロセスのTSSが無効であることを示しています。

TSSのどの部分で例外が発生したかにもよりますが、TSSの長さが104 # バイトを超えると、現在のタスクでこの例外が発生し、ハンドオーバーが終了してしまします。# 他の問題により、切り替え後の新しいタスクでこの例外が発生することがあります。

```

132 _invalid_TSS:
133     pushl $_do_invalid_TSS
134     jmp error_code
135

```

Int11 -- このセグメントは存在しない。タイプです。エラー；エラーコードがあります。

参照されているセグメントはメモリ内にありません。セグメントディスクリプターのフラグはセグメントがメモリ内にないことを示す#。

```

136 _segment_not_present:

```

```
137    pushl $do_segment_not_present  
138    jmp error_code
```

139

```
# Int12 -- スタックの例外です。タイプです。エラー；エラーコードがあります。
# 命令操作がスタックセグメントの範囲を超えようとしたか、スタック
# セグメントはメモリ内にありません。これは、例外11と13の特別なケースです。オペレーティング
# システムの中には、この例外を利用して、より多くの # スタックスペースを確保すべきかどうかを
# 判断するものもあります。
# for the program.
```

140 _stack_segmentです。

141 pushl \$do_stack_segment

142 jmp error_code

143

```
# Int13 -- 一般的な保護の例外。タイプです。エラー；エラーコードがあります。
# 他のどのクラスにも属さない保護違反を示す。対応する例外ベクトル(0-16)を持たない例外が発生
# した場合、通常は # このクラスにフォールバックします。
```

144 _general_protection:

145 pushl \$_do_general_protection

146 jmp error_code

147

```
# Int17 -- バウンダリ・アライメント・チェック・エラー。
この例外は、メモリ境界チェックが有効なときに、特権レベル3（ユーザーレベル）の # データ
が非境界的にアラインされた場合に発生します。
```

148 _alignment_check:

149 pushl \$_do_alignment_check

150 jmp error_code

151

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

8.2.3 インフォメーション

8.2.3.1 インテル予約済み割り込みベクター定義

ここでは、表8-1に示すように、インテルの予約済み割り込みベクターの概要を説明します。

表8-1 インテル社が予約した例外とインタラプト

ベクター いいえ	名前	タイプ	エラー コード	シグナル	ソース
0	デバイドエラー	フォールト (エラー)	いいえ	SIGFPE	DIVとIDIVの命令。
1	デバッグ	不具合/ トラップ	いいえ	SIGTRAP	あらゆるコードやデータの参照、またはINT命令。
2	nmi	インタラプト	いいえ		ノンマスカブルな外部割り込み。
3	ブレイクポイント	トラップ	いいえ	SIGTRAP	INT 3命令。
4	オーバーフロー	トラップ	いいえ	SIGSEGV	INTO命令。

5	バウンドチェック	フォールト	いいえ	SIGSEGV	BOUNDの指示。
6	無効なオペコード	フォールト	いいえ	SIGILL	UD2命令または予約済みオペコード。
7	デバイスが利用できない	フォールト	いいえ	SIGSEGV	浮動小数点またはWAIT/FWAIT命令。

8	ダブルフォールト	アボート	はい (0)	SIGSEGV	例外を発生させることができるあらゆる命令。 NMI、またはINTRである。
9	コプロセッサ・セグ オーバーラン	アボート	いいえ	SIGFPE	浮動小数点演算命令。
10	無効なTSS	フォール ト	はい。	SIGSEGV	タスクスイッチまたはTSSアクセス。
11	セグメントが存在しない	フォール ト	はい。	SIGBUS	セグメントレジスタのロードやシステムへのアクセス のセグメントがあります。
12	スタックセグメント	フォール ト	はい。	SIGBUS	スタック操作とSSレジスタのロード
13	一般的な保護	フォール ト	はい。	SIGSEGV	あらゆるメモリ参照やその他の保護チェック
14	ページフォルト	フォール ト	はい。	SIGSEGV	任意のメモリ参照。
15	インテル予約		いいえ		
16	コプロセッサのエラー	フォール ト	いいえ	SIGFPE	浮動小数点またはWAIT/FWAIT
17	アライメントチェック	フォール ト	はい (0)		メモリ内の任意のデータ参照。
20-31	インテルの予約				
32-255	ユーザー定義のインター ラプト	インタラ プト			外部割込みまたはINT n命令。

8.3 traps.c

8.3.1 機能説明

traps.cプログラムには、主に例外処理に使用されるC言語の関数が含まれており、これらの関数は、例外処理の低レベルコードasm.sから呼び出されて、エラー箇所やエラーコードなどのデバッグメッセージを表示するために使用されます。汎用関数die()は、割り込み処理の際にエラーの詳細情報を表示するために使用されます。プログラムの最終的な初期化関数trap_init()は、前のinit/main.cで呼び出され、ハードウェアの例外処理用割込みベクトル（トラップゲート）を初期化し、割込み要求信号の到来を可能にします。このプログラムを読むときは、前のasm.sファイルを参照してください。

このプログラムの最初から、C言語プログラムに埋め込まれた多くのアセンブリ文に遭遇することになります。埋め込まれたアセンブリ文の基本的な構文については、セクション3.3.2を参照してください。

8.3.2 コードアノテーション

プログラム 8-2 linux/kernel/traps.c

```

1 /*
2  *linux/kernel/traps.c
3  *
4  *(C)      1991Linus Torvalds
5  */
6

```



```
7 /*  
8  * 'Traps.c' は、いくつかの保存をした後のハードウェアのトラップとフォールトを処理します。  
9  asm.s'の状態。現在は主にデバッグ用の補助ツールですが、今後拡張される予定です。  
10 * 主に問題のあるプロセスを殺すために（おそらくシグナルを与えることで。  
11 しかし、必要に応じて完全に殺してしまうこともあるでしょう。）  
12 */  
// <string.h> というヘッダーファイルがあります。主に、文字列操作に関するいくつかの組み込み関数を定義して  
います。
```

```

// <linux/head.h> ヘッドのヘッダーファイルです。セグメントディスクリプターの簡単な構造は
// また、いくつかのセレクト定数も定義されています。
// <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や
// 初期タスク0のデータと、組み込みのアセンブリ関数マクロ文
// ディスクリプタのパラメータ設定と取得 について。
// <linux/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 でよく使われる機能
// <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。
// 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。
// <asm/segment.h> セグメント操作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の操作 のために
// <asm/io.h> Io のヘッダーファイルです。という形で、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ラ の

13 #include <string.h>
14
15 #include <linux/head.h>
16 #include <linux/sched.h>
17 #include <linux/kernel.h> (日本語)
18 #include <asm/system.h>
19 #include <asm/segment.h>
20 #include <asm/io.h>
21
// 以下のステートメントは、3つの組み込みアセンブリマクロ関数を定義しています。
// 括弧で囲まれたコンポジット・ステートメント（中括弧のステートメント）は
// を式とし、最後のresを出力値とします。23行目では、レジスタ
// 変数 res. この変数は、素早くアクセスして操作できるように、レジスタに保存されます。
// レジスタ（eaxなど）を指定したい場合は、次のように文章を書きます。
// "register char res asm("ax");". 詳細な説明は3.3.2項をご参照ください。
//
// Function:セグメントsegのアドレスaddrにバイトを取ります。
// パラメータ: seg - セグメントセレクト、addr - セグメント内の指定されたアドレス。
// 出力します。0 - eax ( res ); Input: %1 - eax (seg); %2 - メモリアドレス (*(addr)).
22 #define get_seg_byte(seg, addr)
({ 23 register char res; ;^w^` )
24 asm ("push %%fs;mov %%ax, %%fs;movb %%fs:%2, %%al;pop %%fs") -
25 : "=a" ( res): "0" (seg), "m" (*(addr)); ;^w^` )
26 res;})
27
// 機能です。セグメントsegのアドレスaddrにロングワード（4バイト）を取ります。
// パラメータ: seg - セグメントセレクト、addr - セグメント内の指定されたアドレス。
// 出力します。0 - eax ( res ); Input: %1 - eax (seg); %2 - メモリアドレス (*(addr)).
28 #define get_seg_long(seg, addr)
({ 29 register unsigned long res; ;^w
^` )
30 asm
("push %%fs;mov %%ax, %%fs;movl %%fs:%2, %%eax;pop %%fs") %31
: "=a" ( res): "0" (seg), "m" (*(addr)); ;^w^` )
32 res;})
33
// Function:fsセグメントレジスタの値(セレクト)を取得する。
// 出力します。%0 - eax ( res).
34 #define fs() ({ })
35 register unsigned short res; eldest
36 asm ("mov %%fs, %%ax": "=a" ( res):); ;^w

```

\hat{w})

```

38 // いくつかの関数のプロトタイプを以下に定義します。
39 void page\_exception (void); // page_fault
   (mm/page.s, 14). 40
41 ボイ divide\_error(void)です。 // Int0 (kernel/asm.s,
   ド 20)。
42 ボイ debug(void)です。 // int1 (kernel/asm.s,
   ド 54)。
43 ボイ nmi(void)です。 // int2 (kernel/asm.s,
   ド 58)。
44 ボイ int3(void)です。 // int3 (kernel/asm.s,
   ド 62)。
45 ボイ overflow(void)です。 // int4 (kernel/asm.s,
   ド 66)。
46 ボイ bounds(void)です。 // int5 (kernel/asm.s,
   ド 70)。
47 ボイ invalid\_op(void)です。 // int6 (kernel/asm.s,
   ド 74)。
48 ボイ device\_not\_available(void)です。 // int7 (kernel/sys_call.s, 158).
   ド
49 ボイ double\_fault(void)です。 // int8 (kernel/asm.s,
   ド 98)。
50 ボイ coprocessor\_segment\_overnrun(void)で // int9 (kernel/asm.s,
   ド す。 78)。
51 ボイ invalid\_TSS(void)です。 // int10 (kernel/asm.s, 132)。
   ド
52 ボイ segment\_not\_present(void); // int11 (kernel/asm.s, 136)。
   ド
53 ボイ stack\_segment(void)です。 // int12 (kernel/asm.s, 140)。
   ド
54 ボイ general\_protection(void)です。 // int13 (kernel/asm.s, 144)。
   ド
55 ボイ page\_fault(void)です。 // int14 (mm/page.s, 14)。)
   ド
56 ボイ coprocessor\_error(void)です。 // int16 (kernel/sys_call.s, 140)。
   ド
57 ボイ reserved(void)です。 // int15 (kernel/asm.s, 82)。
   ド
58 ボイ parallel\_interrupt(void)です。 // int39 (kernel/sys_call.s,
   ド 295)。
59 void irq13 (void); // int45 (kernel/asm.s, 86) コプロセッサ
   の処理です。 60 void alignment\_check( void); // int46 (kernel/asm.s, 148).
61
   本サブルーチンでは、エラー名、エラーコード、プログラムのCS:EIP、EFLAGSを表示します。
   // ESP、fsセグメント、セグメントメッセージ、プロセスpid、タスクno、10バイト命令コード。
   // スタックがユーザーデータセグメントにある場合は、スタックの16バイトの内容も印刷されます。
   //アウトです。これらの情報は、デバッグに利用できます。
   // パラメータ。
   // str - エラー名の文字列ポインタです。
   // esp_ptr - スタック上の中断されたプログラムの情報へのポインタ（図8-4のesp0を参照）。
   // nr - エラーコード。エラーコードがない例外では、このパラメータは常に0です。
62 static void die(char * str, long esp_ptr, long nr)
63 {。
64     long * esp = (long *) esp_ptr;
65     int i;
66
67     printk ("%s: %04xn|r", str, nr&0xffff)。

```

// 次のステートメントは、現在呼び出されているプロセスのCS:EIP、EFLAGS、およびSS:ESPを表示します。
 // 図8-4からわかるように、ここではesp[0]が図ではesp0になっています。そこで、この
 // の文をバラバラにして見てみましょう。

// (1) EIP:\04x:%p\n -- esp[1] segment selector(CS), esp[0] is EIP;
 // (2) EFLAGS:\t%p -- esp[2]はEFLAGSです。
 esp[4]はSS、esp[3]はESPです。

```

68     printk("EIP: \t%04x:%p\nEFLAGS: \t%p\nESP: \t%04x:%p\n",
69     esp[1], esp[0], esp[2], esp[4], esp[3]);
70     printk("fs: %04x\n", fs());
71     printk("base: %p, limit: %p\n", get_base(current->ldt[1]), get_limit(0x17)).
72     if (esp[4] == 0x17) { // SS == 0x17の場合、ユーザーセグメント    にあることを意
        味します。
73         printk("Stack          : "); // ユーザースタックにも16バイトのデ
        ータを表示します。74         for (i=0; i<4; i++)
75             printk("%p ", get_seg_long(0x17, i+(long *)esp[3]))となります。
  
```

```

76         printk ("^^^").
77     }
78     str                                     (i); // タスクNo. を取得(include/linux/sched.h, 210)
79     printk ("Pid: %d, process nr: %d\n\r", current->pid, 0xffff &
80 i); 80     for(i=0; i<10; i++)
81         printk ("%02x ", 0xff & get_seg_byte(esp[1], (i+(char *)esp[0])));
82     printk ("^^^").
83     do_exit(11                               ); /* セグメント例外を再生する */ (d)
84 }
85
86 // asm.sの割込みハンドラから以下の関数が呼び出されます。
87 void do_double_fault(long esp, long error_code)
88 {
89     die("double fault", esp, error_code);
90 }
91 void do_general_protection(long esp, long error_code)
92 {
93     die("一般保護", esp, error_code); 94 }.
95
96 void do_alignment_check(long esp, long error_code)
97 {
98     die("alignment check", esp, error_code);
99 }.
100
101 void do_divide_error(long esp, long error_code)
102 {
103     die("divide error", esp, error_code);
104 }.
105
106 // これらのパラメータは、スタックに順次プッシュされるレジスタの値である
107 // 割り込みを入力した後。asm.sファイルの24~35行目を参照してください。
108 ボイ do_int3 (long * esp, long error_code,
109 ト
110     長いfs、長いes、長いds。
111     long ebp, long esi, long edi,
112     long edx, long ecx, long ebx, long eax)
113 {
114     int tr;
115
116     // タスクレジスタの取得 TR -> tr
117     printk ("eax|t|tebx|t|tecx|t|tedx|n|r%8x|t%8x|t%8x|t%8x|n\r",
118     eax, ebx, ecx, edx) となります。
119     printk ("esi|t|tedi|t|tebp|t|tesp|n|r%8x|t%8x|t%8x|t%8x|n\r",
120     esi, edi, ebp, (long) esp) となります。
121     printk ("|n|rds|tes|tfs|ttr|n|r%4x|t%4x|t%4x|t%4x|n\r",
122     ds, es, fs, tr) である。
123     printk ("EIP          : %8xCS: %4x EFLAGS: %8x|n\r", esp[0], esp[1], esp[2]) となります。
124 }
125
126 ボイ do_nmi(long esp, long error_code)
127 ト
128 {
129     die("nmi", esp, error_code) とな
130     ります。

```

```

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

```

```
174 {  
175     if (last task used math != current)  
176         を返すことができます。  
177     die ("coprocessor error", esp, error_code);  
178 }.
```



```

179
180 void do_reserved(long esp, long error_code)
181 {。
182     ダイ ("reserved (15, 17-47)
183     error", esp, error_code); 183 }。
184
    // 以下は、その割り込みコールゲートを設定するための例外（トラップ）イニシャライザです。
    // （ベクター）を別々に使用しています。set_trap_gate()とset_system_gate()はどちらも、トラップゲートを
    // 割り込みディスクリプターテーブルIDTです。両者の主な違いは、前者が
    // このため、ブレークポイントトラップint3を設定します。
    // オーバーフロー割り込み、バウンズエラー割り込みは、どのプログラムからでも呼び出すことができます。の両
    // 方が
    // これらの関数は、include/asm/system.hの36行目、39行目にあるアセンブリマクロに組み込まれています。
185 ボイ trap_init(void)
186 ド
187 {
188     int i;
189
190     set_trap_gate(0, & divide_error) で
191     す。
192     set_trap_gate(1, & debug) です。
193     set_trap_gate(2, & nmi) です。
194     set_system_gate(3, & int3) です。      /* int3-5は全ての人から呼び出せま
195                                             す。
196     set_system_gate(4, & overflow)。
197     set_system_gate(5, & bounds)。
198     set_trap_gate(6, & invalid op) で
199     す。
200     set_trap_gate(7, & device not available) です。
201     set_trap_gate(8, & double fault) です。
202     set_trap_gate(9, & coprocessor segment overrun) です。
203     set_trap_gate(10, & invalid TSS) です。
204     set_trap_gate(11, & segment not present);
205     set_trap_gate(12, & stack segment) です。
206     set_trap_gate(13, & general protection) です。
207     set_trap_gate(14, & page fault) です。
208     set_trap_gate(15, & reserved) です。
209     set_trap_gate(16, & coprocessor error) です。
210     set_trap_gate(17, & alignment check) です。
211
    // int17--int47のトラップゲートは、すべて最初に予約済みに設定され、再設定されます
    // ハードウェアの初期化のたびに
212     for (i=18; i<48; i++)
213         set_trap_gate(i, & reserved) です。
214
    // コプロセッサのint45(0x20+13)トラップゲート記述子を下に設定し、生成させる
    // 割り込み要求です。コプロセッサのIRQ13は、8259スレーブチップのIR5ピンに接続されています。
    // 210~211行目は、コプロセッサが割り込み要求信号を送るためのものです。で
    // また、パラレルポート1のint39(0x20+7)のゲートディスクリプタも設定されています。
    // ここで、割り込み要求番号IRQ7は、8259のIR7ピンに接続されている
    // 図2-6のメインチップを参照してください。
    //
    // 210行目のステートメントでは、操作コマンドワードOCW1を8259に送信します。このコマンドは
    // 8259 Interrupt Mask Register IMRの設定に使用されます。0x21はメインチップのポートです。マスクは
    // コードが読み込まれ、AND 0xfb (0b11111011)の直後に書き込まれたことを示しています。

```

```
// 割り込み要求IR2に対応する割り込み要求マスクビットM2が  
// クリアします。図2-6に示すように、スレーブチップのリクエスト端子INTには  
// マスターチップのIR2端子を利用しているので、文中では、割り込み要求の
```

```

// スレーブチップからの信号が有効になります。
// 同様に、211行目のステートメントでは、スレーブチップに対して同様の操作を行います。0xA1
//はスレーブチップのポートです。ここからマスクコードを読み取り、ANDの直後に書き込みます。
スレーブチップ上のIR5の割り込みマスクビットM2を示す // 0xdf (0b11011111)
//がクリアされます。コプロセッサはIR5端子に接続されているので、この記述によって
// コプロセッサに割り込み要求信号IRQ13を送信します。
209      set_trap_gate(45, & irq13) です。
210      outb_p(inb_p                                (0x21)&0xfb, 0x21); // マスターチップ    のIRQ2を有効に
      する。
211      outb(inb_p                                (0xA1)&0xdf, 0xA1); // スレーブチップ    のIRQ13を有効に
      する。
212      set_trap_gate(39, & parallel_interrupt ); // パラレル1のゲー
ト      を設定 213 }。
214

```

8.4 sys_call.s

Linuxでは、ユーザとカーネル資源の間のアクセスインタフェースを実装するために、割り込み呼び出し方式を採用しています。sys_call.sプログラムは、主にシステムコールINT 0x80の入力処理と信号検出処理を実装し、2つのシステム関数であるsys_execveとsys_forkの基本的なインタフェースを与えています。また、コプロセッサのエラー（INT16）、デバイスが存在しない（INT7）、クロック割り込み（INT32）、ハードディスク割り込み（INT46）、フロッピーディスク割り込み（INT38）の割り込みハンドラも記載されています。

8.4.1 機能説明

Linux 0.12では、アプリケーションプログラムは、INT 0x80とレジスタEAX内の関数番号を用いて、カーネルが提供する各種サービスを利用しており、これらのサービスをシステムコール（syscall）サービスと呼んでいる。通常、ユーザーはシステムコールサービスを直接利用することではなく、一般的なライブラリ（libcなど）で提供されているインタフェース関数を介して利用します。例えば、プロセスを生成するシステムコールforkは、ライブラリ内の関数fork()を直接利用することができます。INT 0x80の呼び出しはこの関数で実行され、その結果がユーザープログラムに返されます。

カーネルでは、すべてのシステムコールサービスのC関数実装コードがカーネル内に分散しています。カーネルはそれらをシステムコール関数番号に応じて関数ポインタ（アドレス）テーブルに順次並べ、INT 0x80の処理時に対応するシステムサービス関数を呼び出します。

さらに、このソースファイルには、他のいくつかの割り込み呼び出しのエントリー処理コードも含まれています。これらの割り込みエントリコードの実装処理や手順は基本的に同じです。ソフト割り込み（system_call、coprocessor_error、device_not_available）の場合、処理は基本的に以下の2つのステップに分けられます。まず、対応するC関数ハンドラを呼び出す準備をし、いくつかのパラメータをスタックにプッシュします。システムコールは最大で3つのパラメータを取ることができ、これらはレジスタEBX、ECX、EDXを介して渡されます。その後、C関数を呼び出して対応する関数を処理します。処理が戻ると、現在のタスクのシグナルビットマップが検出され、値が最も小さい(優先度が最も高い)シグナルが処理され、シグナルビットマップのシグナルがリセットされます。

ハードウェアIRQによる割り込みに対しては、まず割り込み制御チップ8259Aに割り込み終了命令EOIを送信し、その後対応するCファンクションプログラムを呼び出します。また、クロック割り込みでは、現在のタスクの信号ビットマップを検出します。

8.4.1.1 インタラプトサービスエントリー処理

システムコール（int 0x80）の割り込み処理については、プログラムは「インターフェイス」と考えることができます。実際、各システムコールの処理は、基本的に対応するC言語の関数を呼び出すことで行われます。全体の

システムコールのプロセスを図8-5に示します。

このプログラムは、まずEAXに入力されたシスコール関数番号が有効かどうか（指定された範囲内かどうか）をチェックし、次に使用されるレジスタの一部をスタックに保存します。デフォルトでは、Linuxカーネルは、カーネルデータセグメントにはセグメントレジスタDS、ESを、ユーザデータセグメントにはFSを使用します。次に、上記のアドレスジャンプテーブル（`sys_call_table`）を介して、対応するsyscallのC関数を呼び出します。C関数がリターンした後、プログラムはリターン値をスタックにプッシュして保存する。

次に、プログラムはこの呼び出しを行ったプロセスの状態を調べます。上記C関数の動作などにより、プロセスの状態が実行中の状態から別の状態に変化した場合や、タイムスライスがなくなった（`counter==0`）場合には、プロセススケジューリング関数`schedule()`（`'jmp _schedule'`）が呼び出されます。`_schedule`を実行する前に、リターンアドレス「`ret_from_sys_call`」がスタックにプッシュされているため、`schedule()`の終了後、最終的に「`ret_from_sys_call`」に戻り、実行が継続されます。

`ret_from_sys_call`ラベルから始まるコードでは、いくつかの後処理を行います。主な処理は、現在のプロセスが初期プロセス0であるかどうかを判断し、初期プロセス0であればシステムコールを直接終了し、割り込みが戻ります。そうでなければ、コードセグメント記述子と使用されているスタックに従って、そのプロセスが通常のプロセスであるかどうかを判断し、そうでなければカーネルプロセス（例えば、初期プロセス1）などであると判断します。スタックの内容もすぐにポップアウトされ、シスコールの割り込みを終了します。最後にあるコードの一部は、プロセスのシグナルを処理するために使われます。プロセス構造体のシグナルビットマップが、プロセスがシグナルを受け取ったことを示している場合、シグナルハンドラの`do_signal()`が呼び出されます。

最後に、保存されているレジスタの内容を復元し、割り込み処理を終了して、割り込まれたプログラムに戻ります。信号があった場合、プログラムはまず対応する信号処理関数に「リターン」して実行し、次に`system_call`を呼び出したプログラムに戻ります。

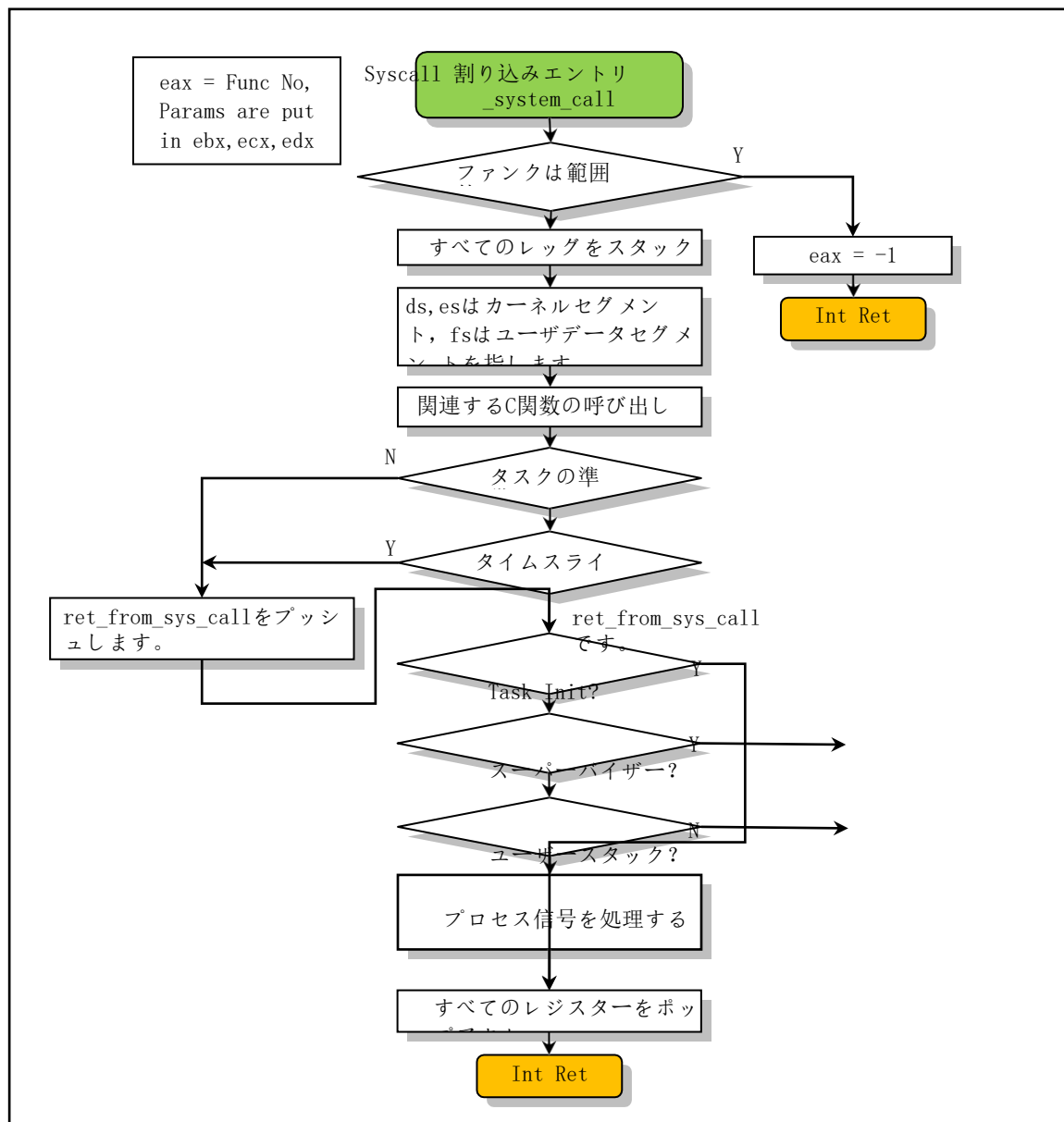


図8-5 システムコール処理フロー図

8.4.1.2 Syscallのパラメータ渡し方法

システムコールINT

0x80におけるパラメータ受け渡しの問題について、Linuxカーネルでは、いくつかの汎用レジスタをパラメータ受け渡しのチャンネルとして使用しています。Linux

0.12系では、プログラムはパラメータの受け渡しにレジスタEBX、ECX、EDXを使用しており、システムコールのサービスプロシージャに3つのパラメータを直接渡すことができます（EAXレジスタのsyscall関数番号は含まれません）。ユーザー空間のデータブロックへのポインタを使用すると、ユーザープログラムはより多くのデータ情報をシステムコールプロシージャに渡すことができます。

前述のように、システムコールの処理では、セグメントレジスタDSおよびESがカーネルデータ空間を指し、FSがユーザーデータ空間に設定される。したがって、実際のデータブロック転送手順では、LinuxカーネルはFSレジスタを用いてカーネルデータ空間とユーザーデータ空間間のデータコピーを行うことができ、カーネルプログラムはコピー処理中にデータ境界範囲のチェック動作を行う必要はない。境界チェックはCPUによって自動的に行われます。カーネル内での実際のデータ転送作

業は、`get_fs_byte()`や `put_fs_byte()`などの関数を用いて行うことができます。これらの関数の実装については、`include/asm/segment.h`ファイルをご参照ください。

このようにレジスタを使ってパラメータを渡す方法には、明確なメリットがあります。それは、システム割り込みサービスルーチンに入ると、パラメータを渡すレジスタも自動的にカーネルの状態スタックに置かれ、割り込み呼び出しからプロセスが終了すると、カーネルの状態スタックもポップされるので、カーネルがそれらを特殊化する必要がないということです。この方法は、当時リーナス氏が知っていたパラメータ転送の方法の中で、最もシンプルで高速な方法です。

8.4.2 コードアノテーション

プログラム 8-3 linux/kernel/sys_call.s

```

1 /*
2  * linux/kernel/system_call.s
3  *
4  * (C) 1991 Linus Torvalds 5
5  */
6
7 /*
8  * system_call.s には、システムコールの低レベル処理ルーチンが含まれています。
9  * これには、コードの一部として、タイマー割り込みハンドラも含まれています。
10 * 同じです。hd-とfloppy-interruptsもここにあります。11 *
12 注：このコードでは、毎行われる信号認識を処理しています。
13 タイマー割り込みの後と、システムコールの後に、それぞれ実行されます。通常の割り込み
14 信号認識を扱うと、全体的にごちゃごちゃしてしまうため
15 * むやみに16 *
17 * 「ret_from_system_call」でのスタックレイアウト。
18 *
19 *      0(%esp) - %eax
20 *      4(%esp) - %ebx
21 *      8(%esp) - %ecx
22 *      C(%esp) - %edx
23 *      10(%esp) - オリジナル %eax          (システムコールでない場合は-1)
24 *      14(%esp) - fs
25 *      18(%esp) - %es
26 *      1C(%esp) - %ds
27 *      20(%esp) - %eip
28 *      24(%esp) - %cs
29 *      28(%esp) - %eflags
30 *      2C(%esp) - %oldesp
31 *      30(%esp) - %olds

```

Linus氏のコメントにある一般的な割り込み手順とは、システムコール # (int 0x80)とクロック割り込み (int 0x20)以外の割り込みのことです。これらのインタラプトは

カーネル状態やユーザー状態でランダムに発生します。これらの割り込みの際にも信号の認識が # 処理されると、システムコールやクロック割り込みの際の信号の認識の処理と # 衝突する可能性があります。これは、カーネルコードのノンプリエンプティブな # 原則に反するものです。そのため、これらの「他の」割り込みで信号を処理することは、システムにとっても # 必要ではありませんし、そうする必要もありません。

```

33
34          SIG_CHLD=          17# 信号 SIG_CHLD (子機の停止または終
了)。35
36          EAX=                0x00# スタック 内の各レジスタのオフセット。

```



```

37 EBX                = 0x04
38 ECX                = 0x08
39 EDX                = 0x0C
40 ORIG_EAX           = 0x10          # syscallでない場合(他の割り込み)は-1
41 FS                 = 0x14
42 ES                 = 0x18
43 DS                 = 0x1C
44 EIP                = 0x20          # 44-48行目は、CPUによって自動的にスタックにプッシュされ
                                     # ます。
45 CS                 = 0x24
46 EFLAGS             = 0x28
47 OLDESP             = 0x2C          # 古いSS:ESPも権限レベルが変わるとプッシュされる
48 OLDSS              = 0x30
49
# アセンブリでのデータ構造へのアクセスを容易にするために、# taskとsignalの構造体のフ
# ィールドのオフセットをここに示す。
# これらはtask_structのフィールドオフセットで、include/linux/sched.hの105行目を参照してください。
50 state=             0# これらはタスク構造へのオフセットです。
51 counter=           4# タスク実行時間のカウント(ティック)、
タイムスライス。
52 priority =         8# counter=タスクの実行開始時の優先度、実行      時間が長いほど
優先度が高くなる。53 signal = 12# シグナルのビットマップ、シグナル = ビットオフセット +
1
54 sigaction =        16# MUST be 16 (=sigactionのlen)
55 blocked            = (33*16)# ブロックされた信号の
オフセット
56
57 #                  sigaction内のオフセット #
include/signal.hの55      行目を参照。58 sa_handler = 0
59 sa_mask = 4
60 sa_flags = 8
61 sa_restorer =      12# kernel/signal.c の記述を参照 62
63 nr_system_calls =  82# Linux 0.12 のシステムコールの総数です。
64
65 ENOSYS =           38# システムコール番号のエラー
コード 66
67 /*
68 * OK、フロッピーを使っているときにパラレルプリンターの割り込みが発生します。
69 奇妙な理由。ウルフル。今は無視しています。70 */
71 .globl _system_call, _sys_fork, _timer_interrupt, _sys_execve
72 .globl _hd_interrupt, _floppy_interrupt, _parallel_interrupt
73 .globl _device_not_available, _coprocessor_error
74
# システムコール番号が正しくない場合は、エラーコード -ENOSYS が返されます。
75 .align              2# メモリは4バイトアライン      されています。
76 bad_sys_callです。
77     pushl            $-ENOSYS# set -ENOSYS in eax
78     jmp ret_from_sys_call

# スケジューラの再実行エントリ。スケジューラは(kernel/sched.c, line 119)で起動し
# ます。# スケジューラの schedule() が戻ると、ret_from_sys_call から実行を続けます。
79 .align 2
80 をリスケしています。
81     pushl $ret_from_sys_call

```

[82](#) jmp _schedule

Int 0x80 -- Linuxシステムコールのエントリポイント (int 0x80、eaxのコール番号)。

```

83 .align 2
84 _system_call:
85     push                    %ds# save original seg registers.
86     プッシュ %es
87     push %fs
88     pushl                   %eax# save the orig_eax

```

システムコールは最大で3つのパラメータを取ることができますが、パラメータはありません。スタックにプッシュされたEBX, ECX, EDXには、対応するC関数のパラメータが # 読み込まれます (99行目参照)。# これらのレジスタがプッシュされる順序は、GNU gccによって指定されます。最初のパラメータはEBXに、2番目はECXに、3番目はEDXに # 格納することができます。

システムコールは、include/unistd.hファイルの150--200行目のマクロにあります。

```

89     pushl %edx
90     pushl                    %ecx# push %ebx,%ecx,%edx as parameters
91     システムコール         への pushl      %ebx#

```

上記でセグメントレジスタを保存した後、ここではDS, ESをカーネルデータ # セグメントに、FSをこのシスコールを実行するユーザプログラムの現在の # ローカルデータセグメントに設定しています。なお、Linux 0.12では、タスクに割り当てられたコードメモリセグメントと # データメモリセグメントは重複しており、それらのセグメントベースアドレスと # リミットは同じである。

```

92     movl                    $0x10,%edx# ds,esをカーネル空間 にセットアップする
93     mov %dx,%ds
94     mov %dx,%es
95     movl                    $0x17,%edx# fs はローカルデータ領域   を指します。
96     mov %dx,%fs
97     cmpl                    _NR_syscalls,%eax# syscall nr is valid ?
98     jae bad_sys_call

```

次の文のオペランドの意味は[_sys_call_table + %eax * 4] です。# プログラムの後の説明と3.2.3項を参照してください。Sys_call_table[]はinclude/linux/sys.hで定義されている # ポインタの配列です。この配列には82個すべての # syscall Cハンドラのアドレスが格納されています。

```

99     call                    _sys_call_table(,%eax,4)# C関数を間接的に   呼び出す。
100    pushl                    %eax# 戻り値。

```

以下の101~106行目では、現在のタスクの状態をチェックします。タスクが実行中でなければ (状態が0でなければ)、 # スケジューラを実行します。タスクが実行中であれば実行中の状態で、そのタイムスライスを使い切った (counter=0) 場合には、 # スケジューラも実行されます。例えば、バックグラウンドプロセスグループのプロセスが # 端末の読み書き操作を制御すると、バックグラウンドグループの全プロセスが # デフォルトでSIGTTINまたはSIGTTOUシグナルを受信し、プロセスグループの全プロセスが # 停止状態になり、現在のプロセスは直ちに復帰します。

```

101 2:
102     movl                    _current,%eax# 構造体ポインタ -> eax
103     cmpl                    $0,state(%eax)# 状態
104     jne reschedule
105     cmpl                    $0,counter(%eax)# カウンター
106     je reschedule

```

以下のコードは、C # 関数から戻った後、シグナルの認識を実行します。他の割込みサービスルーチンが終了する際にも、ここにジャンプして

割り込み処理を終了する前に # 処理を行います。例えば、以下の131行目でプロセッサエラー
一 # 割り込みint16が発生したとします。

ここでは、まず現在のタスクが初期のtask0であるかどうかを判断し、# そうであれば信号処理
を行う必要はないと判断して、直接リターンする。なお、109行目の # _task は、Cプログラムの
task[] 配列に対応しており、直接の参照は
これに#をつけることは、task[0]を参照することと同じです。

```
107 ret_from_sys_callです。
108     movl _current,%eax
109     cmpl     _task,%eax# task[0]はシグナル   を持つことができません。
110     je      3f# フォワードジャンプでラベル3（129行目）に移動、終了
```

元々の呼び出しプログラムのコードセグメントセクタをチェックして # ユーザータスクである
かどうかを確認し、そうでなければ直接割り込みを終了させる（タスクが
カーネルモード中に # プリエンプトされた。）それ以外の場合は、タスクのシグナルをチェックします。
ここで、セクタを0x000fと比較して、ユーザータスクであるかどうかを判断します。# 値
0x000fは、ユーザーコードセグメント（RPL=3、ローカルテーブル、# コードセグメント）の
セクタを表します。そうでない場合は、割り込みハンドラ（INT16など）が107行目にジャン
プして # ここで実行されていることを意味します。この場合、ジャンプして割り込みを終了し
ます。また、オリジナルのスタックセグメントセクタが # 0x17でない（ユーザーセグメント
でない）場合も、システムコールの # 呼び出し元がユーザータスクでないことを示しており、
こちらでも終了します。

```
111     cmpw      $0x0f,CS(%esp)# 古いコードセグメントのスーパーバイザーだった？
112     jne 3f
113     cmpw      $0x17,OLDSS(%esp)# was stack segment = 0x17 ?
114     jne 3f
```

以下のコード（115-128行目）は、現在のタスクのシグナルを処理するために使用されます。#
ここでは、まず現在のタスク構造体のシグナルビットマップ（32ビット、各ビットは1種類のシグ
ナルを表す）を取得し、シグナルブロックコードを使って、# 許されないシグナルをブロックし
ます。その後、最小値の信号を取り、リセットして
元のビットマップの信号の#対応するビット。最後に、この信号を使って
do_signal() (in kernel/signal.c, 128)を呼び出します。do_signal()またはシグナルハンドラが戻
ってきた後、戻り値が0でなければ、# プロセスを切り替える必要があるか、他のシグナルの処理
を続ける必要があるかを確認してください。

```
115     movl      signal(%eax),%ebx# 信号のビットマップ→ebx。
116     movl      blocked(%eax),%ecx# signals blocked -> ecx。
117     NOTL %ECX
118     andl      %ebx,%ecx# 許可された信号   のビットマップを得る
119     bsfl      %ecx,%ecx# ビットマップをbit0からスキャンし、none zero bit   に位置す
        る。
120     je      3f# exit if none。
121     btrl      %ecx,%ebx# 信号   をリセットします。
122     movl      %ebx,signal(%eax)# 新しいビットマップを格納する -> current->signal
123     incl      %ecx# adjust signal from 1 (1--32)。
124     pushl     %ecx#   をパラメータ   として使用します。
125     call      _do_signal# do_signal() (kernel/signal.c, 128)
126     popl      %ecx# パラメータを破棄します。
127     testl     %eax
        , %eax# 戻り値   をチェックします。
128     jne      2b# タスクを切り替えるか、より多くのシグナル   を出す必要があるかどう
        かを確認します。
```

```
129 3:popl %eax#には、100   行目でプッシュされたretコードが含まれています。
130     popl %ebx
131     popl %ecx
```

[132](#) popl %edx[133](#) addl \$4 , %esp# skip orig_eax

```

134     ポッ
135     プ %fs
136     ポッ
137     プ %es
138     ポッ
139     プ %ds
140     アイレ
141     ット
142
143     ##### Int16 --- プロセッサ・エラー・インタラプト。タイプ。エラー; エラーコードなし。
144     # これは、外部のハードウェア例外です。コプロセッサはエラーが発生したことを # 検知する
145     と、ERROR端子を介してCPUに通知します。以下のコードは、コプロセッサから発行されたエラ
146     ー信号を # 処理し、C言語の関数を実行するためにジャンプしています。
147     # math_error() を実行します。リターン後は、ラベル「ret_from_sys_call」にジャンプ
148     して # 実行を継続します。
149
150     .align 2
151     _coprocessor_error:
152         push %ds
153         プッシュ %es
154         push %fs
155         pushl $-1# orig_eaxに -1を埋める #
156         syscallではない
157         pushl %edx
158         pushl %ecx
159         pushl %ebx
160         pushl %eax
161         movl $0x10,%eax# ds, esはカーネル・データ・セグメント を指します。
162         mov %ax,%ds
163         mov %ax,%es
164         movl $0x17,%eax# fs point to(user data seg)
165         mov %ax,%fs
166         pushl $ret_from_sys_call
167         jmp _math_error#
168
169     math_error() (kernel/math/error.c, 11). 156
170     ##### Int7 --- デバイスやコプロセッサが 存在しない。 タイプです。
171     エラー; エラーコードはありません。# コントロールレジスタCROのEM(アナログ)フラグがセッ
172     トされている場合、以下の時に割り込み が発生します。
173     # CPUはコプロセッサ命令を実行するので、割り込み # ハンドラにコプロセッサ命令をエミュレ
174     ートさせるチャンスがある(181行目)。
175     # CROのフラグTSは、CPUがタスク切り替えを行う際に設定されます。TSは、コプロセッサ内の内容
176     とCPUが実行しているタスクが一致していないことを # 判断するために利用できます。# この割り
177     込みは、CPUがコプロセッサのエスケープ命令を実行しているときに発生し
178     # はTSが設定されていることを発見します。この時点で、前のタスクのコプロセッサの内容を保存し、
179     # 新しいタスクのコプロセッサの実行状態を復元することができます(176行目)。kernel/sched.cの
180     # 92行目を参照してください。割り込みは最終的にラベル「ret_from_sys_call」に転送されます。
181     実行のための# (検出 と信号の処理)を行います。)
182
183     .align 2
184     _device_not_available:
185         push %ds
186         プッシュ %es
187         push %fs
188         pushl $-1 # orig_eaxの-1を埋める
189         pushl %edx
190         pushl %ecx
191         pushl %ebx

```

```
166      pushl %eax
167      movl $0x10,%eax      # ds, es をカーネルデータの
                             セグに。
168      mov %ax,%ds
169      mov %ax,%es
```

```

170      movl $0x17,%eax      # ユーザーデータのセグ
                             にfs
171      mov %ax,%fs

# 以下のコードは、フラグTSをクリアし、CR0を取得します。コプロセッサエミュレーションフラ
グEM # がセットされておらず、EMによる割り込みではないことを示している場合、タスクコプロ
セッサ # の状態が復元され、C関数math_state_restore()が実行され、以下のコードが実行されま
す。
# ret_from_sys_call はリターン時に実行されます。
172      pushl $ret_from_sys_call
173      clts# 数学を使える      ようにTSをクリアする
174      movl %cr0,%eax
175      testl      $0x4,%eax# EM (math emulation bit)
176      je _math_state_restore

# EMフラグが設定されている場合は、数学シミュレーション関数math_emulate()を実行します。
177      プッシュ %ebp
           ル
178      プッシュ %esi
           ル
179      プッシュ %edi
           ル
180      プッシュ $0          # ORIG_EIPのための一時的なストレ
           ル              ージ
181      コール _math_emulate  # (math/math_emulate.c, line 476)
182      addl $4,%esp        # 一時的なデータを破棄します。
183      popl %edi
184      popl %esi
185      popl %ebp
186      レット              # ret_from_sys_callへの復帰
187

#### Int32 -- (int 0x20) クロック割り込みハンドラ。
# クロックの割り込み周波数は100Hzに設定されています(include/linux/sched.h, 4)。タイミ
ング # チップ8253/8254は初期化されている(kernel/sched.c, 438)。ここでは、jiffiesは10ご
とに1を追加します。
# ミリ秒。このコードはjiffiesを1だけインクリメントし、8259コントローラにEOIを送り、 # 現
在の特権レベルをパラメータとしてC関数do_timer(long CPL)を呼び出す。
188 .align 2
189 _timer_interrupt:
190     push %ds              # save ds,es and put kernel data space
191     プッシュ %es        # を入れています。fsは_system_callで使用
                           されます。
192     push %fs
193     pushl $-1            # orig_eaxの-1を埋める
194     pushl %edx           # gccでは保存しないので、%eax,%ecx,%edxを
                           保存します。
195     pushl %ecx           # 関数呼び出しを越えてそれらを保存す
                           る。%ebx
196     pushl %ebx          #は、ret_sys_callで使用するために保存され
                           ます。
197     pushl %eax
198     movl $0x10,%eax      # ds,es to kernel
199     mov %ax,%ds
200     mov %ax,%es
201     movl $0x17,%eax      # ユーザーへのfs
202     mov %ax,%fs

```



```
203      incl _jiffies
204      movb $0x20,%al          # EOIを割り込みコントローラ#1に
205      outb %al,$0x20
```

システムコールを実行するセクタ (CSセグメント) の現在の特権レベル (0または3) を # 取得し、`do_timer` のパラメータとしてスタックにプッシュします。その際にはタスクの切り替えやタイミングなどを行う # `do_timer()` 関数は、# `kernel/sched.c` の 324 行目に実装されています。

```

206      movl CS(%esp),%eax
207      andl $3,%eax          # %eax は CPL (0 または 3, 0=supervisor)
208      pushl %eax
209      コール _do_timer      # 'do_timer(long CPL)' は、以下のことを行いま
                           す。
210      addl $4,%esp          # タスクを会計に切り替える ...
211      jmp ret_from_sys_call
212
##### これはsys_execve() のシスコールです。C関数のdo_execve() が呼び出されるのは 呼び出し側の
# コードポインタをパラメータと 関数do_execve() は、fs/exec.cの207行目にありま
して ず。
213 .align 2
214 _sys_execveです。
215      lea EIP(%esp),%eax    # eaxはスタック上の古いeipを指しています。
216      pushl %eax
217      コール _do_execve
218      addl $4,%esp          # 押されたeipを破棄します。
219      レット
...
# まずC関数のfind_empty_process()を呼び出してプロセスのlast_pidを取得します。負の数が返さ
れた場合、現在のタスク配列は満杯である。そうでなければ copy_process() を呼び出して
# プロセスをコピーします。
221 .align 2
222 _sys_forkで
す。
223      コール _find_empty_process  # last_pidの取得 (kernel/fork.c,
                                   143)
224      テストル %eax,%eax          # pid in eax, if negative then ret.
225      js 1f
226      プッシュ %gs
227      プッシュル %esi
228      プッシュル %edi
229      プッシュル %ebp
230      プッシュル %eax
231      コール _copy_process        # copy_process() (kernel/fork.c, 68).
232      アドル $20,%esp            # discard
233      1:Ret
234
##### Int 46 -- (int 0x2E) IRQ14に応答するハードディスクの割り込みハンドラです。
# この割り込みは、要求されたハードディスクの操作が完了したとき、または # エラーが発生し
たときに発生します。(kernel/blk_drv/hd.c 参照)。
# このコードは、まず 8259A スレーブチップに EOI 命令を送り、次に変数 do_hd の関数ポイン #
タを EDX に取り込み、do_hd を NULL に設定します。次にEDXの関数がNULLかどうかをチェックしま
す。
ポインタはnullです。NULLの場合は、edxにextrant_hd_interrupt()を指定してエラーメッセージ
を表示させる。その後、EOI命令が8259Aマスターチップに送られて
EDX が指す関数が呼び出された: read_intr(), write_intr(), または unexpected_hd_interrupt().
235 _hd_interrupt:
236      プッシュ %eax
      ル
237      プッシュ %ecx
      ル
238      プッシュ %edx
      ル
239      push %ds
240      プッシュ %es

```

```
241      push %fs
242      movl $0x10,%eax      # ds, es はカーネルのデータセグメントにポインティングします。
243      mov %ax,%ds
244      mov %ax,%es
```

```

245      movl $0x17,%eax          # fsはユーザデータを指します
                                seg.
246      mov %ax,%fs
247      movb $0x20,%al
248      outb %al,$0xA0          # EOIを割り込みコントローラ#1に
249      jmp 1f                  # give port opportunity to
                                breathe
250 1:      jmp 1f

do_hdは、Read_intr() またはwrite_intr()関数のアドレスを # 割り当てる関数ポインタとして定
義されている。do_hd ポインタ変数は、edx レジスタに格納された後、NULL に設定される。そ
の後、生成された関数ポインタがテストされる。もし、そのポインタが
NULLの場合、このポインタは、未知のハードディスクの割り込みを処理するために、 # C関数
のunexpected_hd_interrupt()に割り当てられる。
251 1: xorl %edx,%edx
252      movl                                %edx,_hd_timeout# hd_timeoutを0に設定すると、コントローラは時間
                                内にINTを生成します。
253      xchgl _do_hd,%edx
254      testl %edx,%edx
255      jne                                1f# if null, point to unexpected_hd_interrupt().
256      movl $_unexpected_hd_interrupt,%edx
257 1: outb %al,$0x20# EOIを8259Aマスターチップ に送る。
258      call                                %edx# イントラ を処理する「面白い」方法。
259      popb %fs
260      popb %es
261      popb %ds
262      popl %edx
263      popl %ecx
264      popl %eax
265      アイレット
266

##### Int38 -- (int 0x26) フロッピードライブの割り込みハンドラで、割り込み要求IRQ6を処理し
ます。# 処理は基本的にハードディスクの上記と同じです。(kernel/blk_drv/floppy.c).# 以下のコ
ードは、まず 8259A 割り込みコントローラに EOI 命令を送信します。
# マスターチップになります。次に、変数do_floppyの関数ポインタをeaxレジスタに格納し、 #
do_floppyをNULLに設定します。次に、eaxの関数ポインタがNULLかどうかをチェックします。NULLで
あれば、 # eaxにexcellent_floppy_interrupt()を指定してエラーメッセージを表示させます。そし
て # eax が指す関数 (rw_interrupt, seek_interrupt, recal_interrupt) を呼び出します。
# reset_interrupt または unexpected_floppy_interrupt。
267 _floppy_interrupt:
268     pushl %eax
269     pushl %ecx
270     pushl %edx
271     push %ds
272     プッシュ %es
273     push %fs
274     movl $0x10,%eax          # ds, es ポイント カーネルデータセグ
275     mov %ax,%ds
276     mov %ax,%es
277     movl $0x17,%eax          # fsはユーザデータを指します seg.
278     mov %ax,%fs
279     movb $0x20,%al          # 8259AマスターチップにEOIを送る。
280     outb %al,$0x20          # EOIを割り込みコントローラ#1に
281     xorl %eax,%eax
282     xchgl _do_floppy,%eax

```

283	testl %eax,%eax	# function pointer NULL ?
284	jne 1f	# yes, point to unexpected_floppy_interrupt()

```

285      movl $_unexpected_floppy_interrupt,%eax
286 1:      コール *%eax          # "interest" way of handling intrins
287      ポップ %fs            # do_floppyが指す関数
288      ポップ %es
289      ポップ %ds
290      popl %edx
291      popl %ecx
292      popl %eax
293      アイレット
294
      ##### Int 39 -- (int 0x27) IRQ7 に対応するパラレルポート割り込みハンドラ。
      # カーネルはこのハンドラを実装しておらず、EOI命令だけがここに送られます。
295 _parallel_interrupt:
296      pushl %eax
297      movb $0x20,%al
298      outb %al,$0x20
299      popl %eax
300      アイレット

```

8.4.3 参考情報

8.4.3.1 GNUアセンブリ言語の32ビットアドレッシング

GNUアセンブリ言語では、AT&Tのアセンブリ構文を使用しています。この詳しい紹介と例については、第3.2.3節を参照してください。ここでは、アドレス指定方法の紹介と例を紹介するだけです。AT&Tおよびインテルのアセンブリ言語のアドレス指定オペランドの形式は次のとおりです。

AT&T: disp (base, index, scale)
 Intel: [ベース + インデックス * スケール + ディスプレー] となります。

dispはオプションのオフセット、baseは32ビットのベースアドレス、indexは32ビットのインデックスレジスタ、scaleはスケールファクター（1、2、4、8、デフォルトは1）です。この2つのアセンブリ言語のアドレス形式は若干異なりますが、実際には具体的なアドレス位置は同じです。上記フォーマットでのアドレス位置の計算方法は、 $\text{disp} + \text{base} + \text{index} * \text{scale}$

応募の際にこれらの項目をすべて書く必要はありませんが、ディスプレイとベースには必ず1つずつ書く必要があります。以下はその例です。

表8-2 メモリアドレッシングの例

要求事項への対応	AT&Tフォーマット	インテルフォーマット
指定されたC変数「booga」へのアドレス指定	_booga	[_booga]
レジスターが指す位置へのアドレス指定	(%eax)	[eax]
の内容を利用して変数をアドレス指定します。 レジスタをベースアドレスとして	_variable(%eax)	[eax + _variable] です。
int配列の値を指定する（スケール値は4	_array(,%eax, 4)	[eax*4 + _array] です。
ダイレクトアドレッシングのオフセット*(p+1)を使用し、pは	1(%eax)	[eax+1]

charのポインタは、%eaxに配置されます。		
指定した文字を8バイトの配列でアドレス指定する のレコードがある。ここで、eaxはインデックス、ebxは	<code>_array(%ebx,%eax,8)</code>	<code>[ebx + eax * 8 + _array]</code> となります。

レコード内の指定された文字のオフセット。		
----------------------	--	--

8.4.3.2 システムコールの追加

新しいシステムコールをカーネルに追加するには、まずその正確な目的を決める必要があります。Linuxシステムでは、1つのシステムコールを複数の目的のために推進することはありません (ioctl()システムコールを除く)。さらに、新しいシステムコールのパラメータ、戻り値、エラーコードを決定する必要があります。システムコールのインターフェイスはできるだけシンプルであることが望ましいので、パラメータはできるだけ少なくします。また、システムコールの汎用性や移植性を考慮して設計する必要があります。Linux

0.12に新しいシステムコールを追加したい場合、以下のことを行う必要があります。

まず、システムのコンピュータ名を変更するためのsys_sethostname()という関数のような、新しいシステムコールのハンドラに関連するプログラムに記述します。通常、このC関数はkernel/sys.cプログラムの中に置くことができます。また、thisname構造体が使用されているので、sys_uname()内のthisname構造体 (218~220行目) を関数の外に移動する必要もあります。

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

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

そして、include/unistd.hファイルに新しいシステムコール番号とプロトタイプ定義を追加します。例えば、149行目の後に関数番号を追加し、279行目の後にプロトタイプ定義を追加します。

```
// 新しいシステムコール番号で
す。#define NR_sethostname87
int sethostname(char *name, int len);
```

次に、include/linux/sys.hファイルに外部関数宣言を追加し、以下のように、関数ポインタテーブルsys_call_tableの最後に新しいシステムコールハンドラの名前を挿入します。なお、関数名は、システムコール関数番号の厳密な順に並べる必要があります。

```
extern int sys_sethostname();
```

```
// 関数ポインタの配列表。  
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
```

```
....,
sys_lstat, sys_readlink, sys_uselib, sys_sethostname } です。
```

その後、sys_call.sファイルの63行目を修正し、nr_system_callsの総数を1つ増やします。この時点で、カーネルを再コンパイルすることができます。最後に、lib/ディレクトリにあるライブラリ関数の実装を参照して、新しいシステムコールライブラリ関数sethostname()をlibcライブラリに追加します。

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

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

8.4.3.3 アセンブリファイルでシステムコールを直接使用する

以下は、As86とGNU

asの関係と違いを説明する際に、Linus氏が与えた簡単なアセンブリ例asm.sです。この例は、Linuxシステム上で、スタンドアロンのプログラムをアセンブリ言語でプログラムする方法を示しています。つまり、スタートコードモジュール（crt0.oなど）やライブラリの関数を使用する必要はありません。その手順は以下の通りです。

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

のメッセージが表示されます。

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

使用されるシステムコールは2つあります。書き込みシステムコールで実行されるC関数は、sys_write(int fd, char *buf, int len)と宣言されています（プログラムfs/read_write.cの83行目を参照）。このコールには3つのパラメータがあります。この3つのパラメータは、システムコールを呼び出す前に、レジスタEBX、ECX、EDXに格納される。このプログラムのコンパイルと実行の手順は以下のとおりです。

```
[/usr/root]# as -o asm.o asm.s
[/usr/root]# ld -o asm asm.o
[/usr/root]# ./asm
ハローワールド
[/usr/root]#
```

8.5 mktime.c

mktime.cプログラムは、カーネル固有のUNIXカレンダータイムのブートタイムを計算するために使用されます。

8.5.1 機能

このプログラムには、カーネルでのみ使用される関数kernel_mktime()が1つだけあり、1970年1月1日0:00から起動する日までの秒数（カレンダー時間）をブートタイムとして計算するために使用されます。この関数は、標準Cライブラリで提供されているtm構造体が表す時間をUNIXカレンダー時間に変換するmktime()関数と全く同じものである。ただし、カーネルは通常のプログラムではないので、開発環境ライブラリの関数を呼び出すことはできず、自分で記述する必要があります。

8.5.2 コードアノテーション

ヨ ン

プログラム 8-4 linux/kernel/mktime.c

```

1 /*
2  * linux/kernel/mktime.c
3  *
4  * (C)      1991 Linus Torvalds
5  */
6
7 // 時間型のヘッダーファイルです。この中で最も重要なのは、tmの定義です。
8 #include <time.h> 時刻に関する構造体といくつかの関数のプロ
9 トタイプ。7 #include <time.h>
10
11 /*
12  * これはライブラリのルーチンではなく、カーネルでのみ使用されます。
13  * このように、1970年以下の年号は気にせず、すべてを想定しています。
14  * *はOKです。同様に、TZなども喜んで無視されます。私たちは、すべてを
15  * なるべく簡単に。ライブラリーのために何か公共のものを見つけよう
16  * ルーチン (minix timesは公開されていると思いますが)。
17 */
18 /*
19  * PS. 私は1970年という年号を作った人が嫌いです。
20  * 代わりにうるう年? 私はグレゴリウスも嫌いだ。私は不機嫌です。
21 */
22 #define MINUTE 60 // 1分を秒 単位で表します。
23 #define HOUR (60*MINUTE) // 1時間を秒 単位で表します。
24 #define DAY (24*HOUR) // 1日を秒 単位で表します。
25 #define YEAR (365*DAY) // 1年を秒 単位
26 で表します。24
27
28 興味深いことに、うるう年を想定しています。
29 // 各月の初めの開始時間秒数は、年限で定義されています。
30 static int month[12] = {
31     0,
32     DAY*(31)です。
33     DAY*(31+29)です。
34     day*(31+29+31)で

```

31 す。
 day*(31+29+31+30)
 です。

```

32     day*(31+29+31+30+31),
33     day*(31+29+31+30+31+30),
34     day*(31+29+31+30+31+30+31),
35     day*(31+29+31+30+31+30+31+31),
36     day*(31+29+31+30+31+30+31+31+30),
37     day*(31+29+31+30+31+30+31+31+30+31),
38     day*(31+29+31+30+31+30+31+31+30+31+30)
39 };
40
41 // この関数は、1970年1月1日の0:00からの経過秒数を計算します。
42 // マシンが起動した日を起動時間とする。tmのフィールドが割り当てられているのは
43 // init/main.cで、CMOSから情報を取得しています。
44 long kernel_mktime(struct tm * tm)
45 {
46     ロングレスって
47     int year;
48
49     // まず、1970年からの経過年数を計算します。があるからです。
50     // ここで2桁の表現をすると、2000年問題が発生します。これを単純に解くと
51     if (tm->tm_year<70) tm->tm_year += 100;
52     // UNIXのyear yは1970年から計算されているので。1972年まではうるう年なので
53     // 3年目（71, 72, 73）は最初のうるう年なので、うるう年の計算方法は以下の通りです。
54     // 1970年からの1年間は、1 + (y - 3) / 4、つまり(y + 1)/4となります。
55     // res = これらの年の秒数 + 各うるう年の秒数 + 秒数
56     // 現行の年から現行の月まで。の2月の日数は、現在の月に比べて
57     // month[]配列には、うるう年の日数、つまり
58     // 2月は1日多い。したがって、その年がうるう年ではなく、現在の
59     // の月が2月より大きい場合は、この日を差し引きます。1970年から数えているので
60     // うるう年の検出方法は。(y + 2)を4で割ることができ、そうでない場合は
61     // うるう年ではありません。
62     // if (tm->tm_year<70) tm->tm_year += 100;
63     year = tm->tm_year - 70;
64     うるう年を正しく設定するために必要な /* マジックオフセット(y+1)です。
65     res = YEAR*year + DAY*((year+1)/4);
66     res += month[tm->tm_mon];
67     ここでは /* と (y+2) を使用しています。うるう年ではなかった場合は、*/を調整する必要があります。
68     if (tm->tm_mon>1 && ((year+2)%4))
69         res -= DAY;
70     res += DAY*(tm->tm_mday-1); // 過去1ヶ月間の日数（秒 単位）。
71     res += HOUR*tm->tm_hour; // 1日の過去の時間を秒 単位で表示します。
72     res += MINUTE*tm->tm_min; // 1時間のうちの過去分を秒 単位で表示します。
73     res += tm->tm_sec; // 1分間に経過した秒数です。
74     return res; // 1970年からの経過秒数です。58 }
75 }

```

8.5.3 インフォメーション

8.5.3.1 うるう年の計算方法

うるう年の基本的な計算方法は

yが4で割り切れて、100で割り切れない、または400で割り切れる場合、yはうるう年である。

8.6 sched.c

8.6.1 機能説明

sched.cのソースファイルには、カーネル内のタスクをスケジューリングするためのコードが含まれています。このプログラムには、スケジューリングのためのいくつかの基本的な関数（`sleep_on()`、`wakeup()`、`schedule()`など）や、いくつかの簡単なシステムコール関数（`getpid()`など）が含まれています。また、システムクロック割り込みサービスルーチンのタイマー関数`do_timer()`もこのプログラムに含まれています。また、フロッピーディスクドライブのタイミング処理のプログラミングを容易にするために、ライナス氏はフロッピーディスクのタイミングに関連するいくつかの関数をこのプログラムに入れている。

これらの基本機能のコードは長くはありませんが、やや抽象的で理解しにくいものです。幸いなことに、より詳細な紹介や議論をしている教科書がすでにたくさんあります。そのため、勉強する際には、これらの関数の説明を他の本で参照することができます。コードの注釈や分析を始める前に、スケジューラ、スリープ、ウェイクアップの各機能の原理を見てみましょう。

8.6.1.1 スケジュール機能

スケジュール関数`schedule()`は、システム内で次に実行するタスク（プロセス）を選択する役割を担っています。まず、すべてのタスクをチェックし、シグナルを受信したタスクを起こします。具体的な方法としては、タスク配列の各タスクのアラームタイミング値「alarm」をチェックします。タスクのアラーム時間が経過していれば（`jiffies > alarm`）、そのシグナルビットマップにSIGALRMシグナルを設定し、アラーム値をクリアします。`jiffies`はマシンのブートタイムからのティック数です（`10ms/tick`、`sched.h`で定義）。タスクのシグナルビットマップにblockedシグナル以外のシグナルがあり、タスクが割り込み可能なスリープ状態(`TASK_INTERRUPTIBLE`)であれば、タスクはレディ状態(`TASK_RUNNING`)に設定されます。

これに続いて、スケジューリング機能のコア処理部分があります。この部分では、タスクのタイムスライスと優先度メカニズムに基づいて、後で実行するタスクを選択します。まず、タスク配列にあるすべてのタスクをループして、残りの実行時間のカウンタ値が最大のタスクを選択し、`switch_to()`関数でそのタスクに切り替えます。

すべての実行準備が整ったタスクのカウンタ値がゼロに等しい場合は、すべてのタスクのタイムスライスがその時点で使い果たされたことを意味します。そこで、タスクの優先度の値「priority」に従って、各タスクの実行タイムスライスの値「counter」をリセットし、再びすべてのタスクの実行タイムスライスの値を循環させます。

8.6.1.2 スリープ&ウェイクアップ機能

この2つの関数は非常に短いものですが、`schedule()`関数よりも理解するのが難しいものです。コードを見る前に、図を使って説明しておきましょう。簡単に言うと、`sleep_on()`の主な機能は、プロセス(タスク)が要求するリソースが使用中であったり、メモリに存在しない場合に、一時的にプロセスを待ち行列に切り替えることです。切り替えて戻ってくると、そのプロセスは継続して実行されます。待ち行列に入れる方法は、関数内の`tmp`ポインタを各待ちタスクのリンクとして利用しています。

この関数では、3つのタスクポインタの操作を行います。`*p`、`tmp`、`current`です。`*p`は、ファイルシステムのメモリiノードの`i_wait`ポインタ、メモリバッファ演算の`buffer_wait`ポインタなど、待ち行

列の先頭ポインタ，`tmp`は，関数スタック上に設けられた一時的なポインタで，現在のタスクのカーネル状態スタックに格納されている，「`current`」は，現在のタスクへのポインタである。これらのポインタのメモリ上での変化については、図8-6の模式図を使って説明することができます。図中の長い棒は、メモリバイトの並びを表しています。

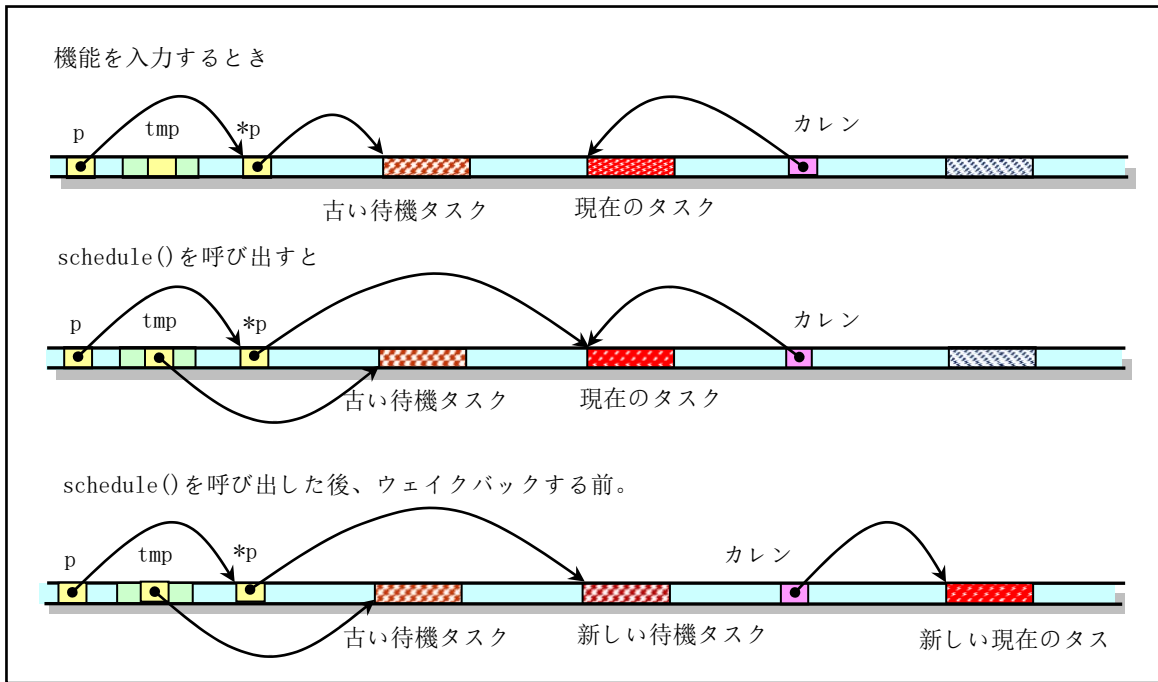


図8-6 sleep_on()におけるポインタの変化の模式図。

この関数に入ると、キューヘッドポインタ $*p$ は、待ち行列で待機していたタスク構造体（プロセス記述子）を指しています。もちろん、システムの実行開始当初は、待ち行列に待ちタスクは存在しません。そのため、上図の本来の待ちタスクは最初から存在せず、 $*p$ はNULLを指しています。

ポインタ操作により、スケジューラ関数が呼び出される前に、キューヘッドポインタは現在のタスク構造を指し、関数内の一時ポインタ「tmp」は元の待ちタスクを指します。スケジューラを実行する前と、タスクが起こされて実行に戻される前に、現在のタスクポインタは新しい現在のタスクに向けられ、CPUは新しいタスクでの実行に切り替わります。このように、sleep_on()関数の実行により、tmpポインタはキューの中のキューヘッドポインタが指す元の待ちタスクを指し、キューヘッドポインタは新たに追加された待ちタスク、つまりこの関数を呼び出すタスクを指していることになります。このように、スタック上の一時ポインタtmpのリンク機能により、複数のプロセスが同一の資源を待つためにこの関数を呼び出すと、カーネルプログラムは暗黙のうちに待ち行列を構築することになる。図8-7の待ち行列の図をご覧ください。この図では、待ち行列の先頭に3番目のタスクが挿入されたときの状況を示しています。この図から、sleep_on()関数の待ち行列形成プロセスをよりわかりやすく理解することができます。

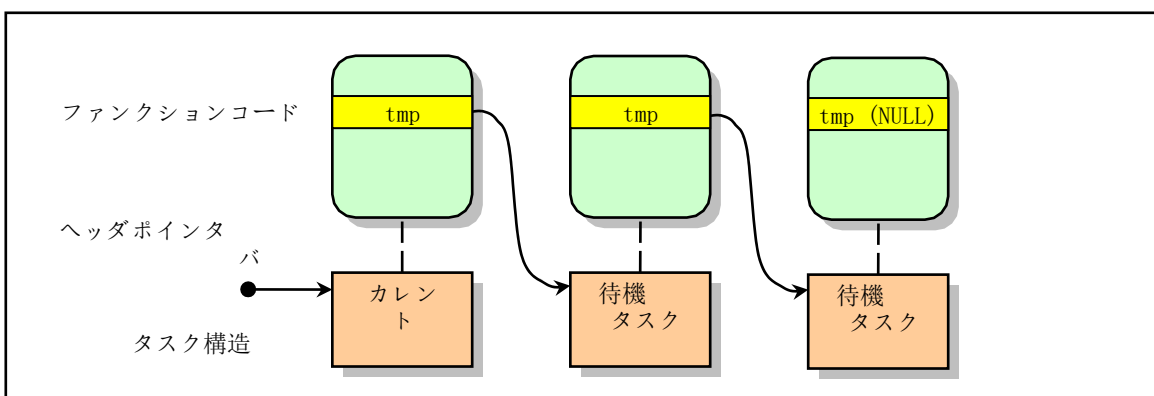


図8-7 sleep_on()の暗黙のタスク待ち行列の様子

sleep_on()関数は、プロセスを待ち行列に挿入した後、schedule()関数を呼び出して別のプロセスを実行します。プロセスが覚醒して再実行されると、それ以降の文が実行され、それよりも早く待ち行列に入ったプロセスが覚醒します。なお、ここでのいわゆるウェイクアップは、プロセスが実行状態にあることを意味するのではなく、実行をスケジューリングできる準備状態にあることを意味しています。

ウェイクアップ関数wake_up()は、利用可能なリソースを待っている指定されたタスクをレディ状態(TASK_RUNNING)にするための関数です。この関数は汎用的なウェイクアップ関数です。ディスク上のデータブロックを読み出す場合など、待ち行列にあるどのタスクも先に目覚めてしまう可能性があるため、ウェイクアップタスク構造体のポインタを空にすることも必要です。こうすることで、スリープ状態になったプロセスが起こされ、sleep_on()が再実行されたときに、そのプロセスを起こす必要がなくなります。

また、interruptible_sleep_on()という関数があり、その構造は基本的にsleep_on()と似ていますが、スケジューリングの前に現在のタスクを割り込み可能な待機状態にし、タスクを目覚めさせた後に、後から入力されたタスクがあるかどうかを判断する必要があります。もしあれば、それらを先に実行するようにスケジューリングします。カーネル0.12からは、この2つの関数は1つにまとめられ、2つのケースを区別するためのパラメータとして、タスクの状態のみを使用するようになりました。

このファイルのコードを読む際には、include/linux/sched.hファイルのコメントを参考にすると、カーネルのスケジューリング機構をより深く理解することができます。

8.6.2 コードアノテーション

ヨ ン

プログラム 8-5 linux/kernel/sched.c

```

1  /*
2      *linux/kernel/sched.c
3      *
4      *(C) 1991 Linus Torvalds
5      */
6
7  /*
8  sched.cはメインのカーネルファイルです。スケジューリングのプリミティブが含まれています。
9  * (sleep_on、wakeup、scheduleなど)や、いくつかのシンプルなシステム
10 関数を呼び出します (getpid())というタイプで、単にフィールドを
11 * カレントタスク
12 */
// <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、組み込みのアセンブリ関数マクロ文
// ディスクリプタのパラメータ設定と取得 について。
// <linux/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 でよく使われる機能
// <linux/sys.h> システムコール用のヘッダーファイルです。72個のシステムコールC関数を含む
// sys_ で始まるハンドラーです。
// <linux/fdreg.h> フロッピーディスクのファイルです。フロッピーディスクコントローラの定義が含まれていま
// す。

```

```
//パラメータ
// <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。
// ディスクリプター/割り込みゲートなどが定義 されています。
// <asm/io.h> Io のヘッダーファイルです。という形で、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ラ の
```

```

// <asm/segment.h> セグメント操作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の操作のために
// <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
// 関数のプロトタイプ
13 #include <linux/sched.h>
14 #include <linux/kernel.h> (日本語)
15 #include <linux/sys.h>
16 #include <linux/fdreg.h> (日本語)
17 #include <asm/system.h>
18 #include <asm/io.h>
19 #include <asm/segment.h>
20
21 #include <signal.h> (英語)
22
// このマクロは、信号 nr の対応するビットのバイナリ値を
// シグナルのビットマップです。シグナル番号の範囲は1〜32です。例えば、ビットマップの値が
// 信号5は1<<(5-1)=16=00010000b。
// SIGKILL信号とSIGSTOP信号以外のすべての信号はブロック可能です。
// BLOCKABLE = (11111111, 111111011, 111111110, 111111b)。
23 #define _S(nr) (1<<((nr)-1))
24 #define BLOCKABLE (~( _S(SIGKILL) | _S(SIGSTOP)))
25
// カーネルデバッグ関数。pid、プロセスの状態、カーネルスタックのフリーバイトを表示します。
// 指定されたタスクの nr. のための // (およそ) とタスクの弟妹・兄妹
// タスクのデータとカーネルの状態スタックが同じメモリページ (4096バイト) にあるので
// (1ページあたり)、カーネルの状態スタックはページの終わりから下に向かって始まるので
// 28行目の変数jは、カーネルスタックの最大容量、つまり一番下のトップを表しています。
// タスクのカーネルスタックの位置。
// パラメータ
// nr - タスク 番号、p - タスク構造体ポインタ。
26 void show_task(int nr, struct task_struct * p)
27 {。
28     int i, j = 4096-sizeof(struct task_struct);
29
30     printk("%d: pid=%d, state=%d, father=%d, child=%d, ", nr, p->pid. ")。
31     p->state, p->p_pptr->pid, p->p_cptr ? p->p_cptr->pid : -1); 32
32     i=0;
33     while (i<j && !((char *) (p+1))[i]) // タスク構造体の後のゼロバイトの
34     数を検出します。 34 i++;
35     printk("%d/%d chars free in kstack\n", i, j)。
36     printk("PC=%08X", *(1019 + (unsigned long *) p));
37     if (p->p_ysptr || p->p_osptr)
38         printk("Younger sib=%d, older sib=%d\n",
39             p->p_ysptr ? p->p_ysptr->pid : -1,
40             p->p_osptr ? p->p_osptr->pid : -1)
41             となります。)
42     その他
43     printk("%^w^")。
44 }
45
// システム内のすべてのタスクのステータス情報を表示します。
NR_TASKS はシステム内のタスクの最大数 (64) で、linux/sched.h の 6 行目で定義されています。
45 void show_state(void)
46 {

```

```

47     int i;
48
49     printk(" ");
50     for (i=0;i<NR_TASKS;i++)
51         if (task[i])
52             show_task(i,task[i])となります。
53 }
54
55 // 8253カウンタ/タイマチップの入力クロック周波数は約1.193180MHzです。また、Linux
56 // カーネルはタイマー割り込みの周波数が100Hzであることを期待する、つまりクロック割り込み
57 // は10msごとに発行されます。つまり、ここではLATCHは8253チップを設定するための初期値であり、参照してくだ
58 // さい。
59 // ライン438。
60 #define LATCH (1193180/HZ)
61
62 extern void mem_use (void); // [??]どこにも 定義
63   されていない。
64 extern int timer_interrupt (void); //
65 kernel/system_call.s, 189
66 extern int system_call (void); //
67 kernel/system_call.s, 84
68
69 // 各タスク（プロセス）は、独自のカーネルステートスタックを持っています。これがタスクユニオンを定義しま
70 す。
71 // タスク構造とスタック配列で構成されています。のデータ構造は、タスク構造とスタック配列で構成されていま
72 すタスクとそのカーネル状態スタックは、同じメモリページ（データセグメント）に配置されます。
73 // セレクタは、スタックセグメントレジスタSSから取得できます。
74 // 以下の67行目では、初期タスクのデータを設定しています（初期データはlinux/sched.hの156にあります）。
75
76 union task_union {
77     struct task_struct task;
78     char stack[PAGE_SIZE];
79 };
80
81 static union task_union init_task = {INIT_TASK,};
82
83 // システムスタートからの秒数（10ms/tick）。ティックは、タイマーが
84 // チップの割り込みが発生します。
85 // 「volatile」という修飾語、その英語の意味は、変化しやすい、不安定である。という意味を持ちます。
86 // この修飾子の目的は、その変数の内容が次のような可能性があることをコンパイラに示すことです。
87 // は、他のプログラムによる修正の結果、変化します。通常、変数が
88 // をプログラムで宣言すると、コンパイラはそれを汎用のレジスタに入れようとします。
89 // のように、アクセス効率を上げるために、EBXのようなそれ以降は、一般的に
90 // メモリ内の変数の元の値を表示します。他のプログラムやデバイスが変数を変更すると
91 // この時点でメモリ内のこの変数の//値は、EBXの値は更新されません。になります。
92 // この問題を解決するために、volatile修飾子を作成し、コードがその値を取らなければならないようにします。
93 // 変数を参照する際には、指定したメモリ位置から // を実行します。ここでは、gccが必要です
94 // ジフティを最適化するのでもなく、場所を移動するのでもなく、記憶からその値を取ること。
95 // カウンタ/タイマチップの割り込み処理プロセスや他のプログラムが変更されるため
96 // その値。
97
98 unsigned long volatile jiffies = 0; // カーネルパルス（ティック）
99 unsigned long startup_time = 0; // 1970:0:0:0からのトータル秒
100
101 int jiffies_offset = 0; /* # 真を得るために加える時計の秒数
102   時間」です。常に以下 でなければなりません。
103   1秒の 価値。時間マニアのための
104   自分のマシンを同期させたい人
105   to WWV :-)* */

```



```

// 現在のタスクポインタで、初期化時にはタスク0を指します。
77 struct task_struct *current = &(init_task.task);
78 struct task_struct *last_task_used_math = NULL;
79
// タスクポインタの配列を定義します。最初の項目はタスクデータに初期化されます。
// 初期タスク（タスク0）の構造。
80 struct task_struct *task[NR_TASKS] = {&(init_task.task), };
81
// ユーザースタック（配列）を定義します。合計1K個のアイテム、サイズは4Kバイトです。カーネルとして使用
// カーネルの初期化の際に、 // スタックを使用します。初期化が完了した後、これは
// タスク 0のユーザーモードスタックとして//タスク 0を実行する前のカーネ
// ルスタックであり
// 後にタスク0と1のユーザーステートスタックとして使用されます。
// 次の構造体は、スタックSS: ESPを設定するために使用されます。head.sの23行目を参照してください。SSは
// カーネルのデータセグメントセクタ（0x10）に設定され、ESPは、データセグメントセクタの最後を指すように
// 設定されています。
// user_stackの配列の最後の項目です。これは、Intel CPUがスタック
// の内容を保存しています。
// SPポインタでスタックを
82 long user_stack [ PAGE_SIZE>>2 ] ;
83
84 構造体 {
85     long * a;
86     short b.
87     } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
88 /*
89  * 'math_state_restore()'は、現在の数学情報を
90  * 古い演算状態の配列、および現在のタスクから新しい演算状態を取得する
91  */
// タスクの交換が予定された後、この関数を使って数学を保存します
// 元のタスクのコプロセッサの状態（コンテキスト）を復元し、コプロセッサのコンテキストを復元する
// スケジュールされた新しいタスクの
92 void math_state_restore()
93 {
// タスクが変更されていない（前のタスクが現在のタスクである）場合に返します。ここでは
// 「前のタスク」とは、直前に交換されたタスクのことです。
// さらに、WAIT命令はコプロセッサ命令の前に実行する必要があります。
// 前のタスクがコプロセッサを使用していた場合、その状態はTSSのタスクのフィールドに保存されます。
94     if (last_task_used_math == current)
95         を返すことができます。
96         asm ("fwait" )。
97     if (last_task_used_math) {
98         __asm ("fnsave %0":: "m" (last_task_used_math->tss.i387))。
99     }
// ここで、'last_task_used_math'は現在のタスクを指しており、現在のタスクが
// スワップアウトされます。この時点で、現在のタスクがコプロセッサを使用していた場合、その状態は
// 復元します。それ以外の場合は、初めての使用なので、初期化コマンドを送信します。
// をコプロセッサに送信し、コプロセッサフラグを設定します。
100     last_task_used_math=current;
101     if (current->used_math) {
102         asm ("frstor %0":: "m" (current->tss.i387))。
103     } else
{ 104         asm ("fninit ">::); // math に初期コマンドを送る。
105         current->used_math=1;// 使用済みの数学フラグ を設定します。

```

106

}

```

107 }
108
109 /*
110  * 'schedule()' は、スケジューラ関数です。これはGOOD CODE! また
111  * おそらく、これを変更する理由はないでしょう。
112  * どんな状況でも（つまり、IOバウンドのプロセスに良いレスポンスを与えるなど）。
113  * 1つは、ここのシグナルハンドラのコードに注目してください。
114  *
115  * 注! タスク0は「アイドル」タスクで、他のタスクがないときに呼び出されます。
116  * タスクを実行することができます。また、殺されることも、眠ることもできません。状態は
117  * task[0]の情報は使われません。
118 */
119 void schedule(void)
120 {
121     int i, next, c;
122     struct task_struct ** p; // タスク構造体のポインタのポイン
タ。123
124 /* アラームをチェックして、シグナルを受け取った割り込み可能なタスクを起こす */
125
126 // タスク配列の最後のタスクからアラームのチェックを開始します。空のポインタの項目をスキップする
127 // ループするとき。
128 for(p = &LAST_TASK; p > &FIRST_TASK; --p)
129     if (*p) {
130         // タスクのタイムアウトが設定されていて期限切れになった場合 (jiffies>timeout)、タイムアウトを
131         // 0で、タスクがTASK_INTERRUPTIBLEのスリープ状態であれば、readyにする
132         // 状態 (TASK_RUNNING) です。
133         if ((*p)->timeout && (*p)->timeout < jiffies) {
134             (*p)->timeout = 0;
135             if ((*p)->state == TASK_INTERRUPTIBLE)
136                 (*p)->state = TASK_RUNNING;
137         }
138         // タスクのSIGALRMシグナルのタイムアウト・アラーム値が設定されていて、かつ期限が切れた場合
139         // (alarm<jiffies)の場合、シグナルビットマップにSIGALRMシグナルが設定される、つまりSIGALARM
140         // 信号がタスクに送信され、アラームが解除されます。この場合のデフォルトアクションは
141         // タスクを終了させるためのシグナルです。
142         if ((*p)->alarm && (*p)->alarm < jiffies) {
143             (*p)->signal |= (1<<(SIGALRM-1));
144             (*p)->alarm = 0;
145         }
146         // シグナルビットマップに、ブロックされたシグナルの他に、他のシグナルがある場合や
147         // タスクが割り込み可能な状態であれば、タスクをレディ状態 (TASK_RUNNING) にします。
148         // '~(BLOCKABLE & (*p)->blocked)' はブロックされたシグナルを無視するために使用されますが、SIGKILLは
149         // とSIGSTOP信号をブロックすることはできません。
150         if (((*p)->signal & ~(BLOCKABLE & (*p)->blocked)) &&
151             (*p)->state == TASK_INTERRUPTIBLE)
152             (*p)->state = TASK_RUNNING; // レディを設定します。
153     }
154 }
155 /* これがスケジューラの本体です。*/
156
157 while (1) {
158     c = -1;
159     next = 0;

```



```

147         i = NR_TASKS;
148         p = & task[NR_TASKS];
// このコードは、タスク配列の最後のタスクからもループし、空の
// スロットになっています。のカウンタ（タスク実行時間のカウントダウン数）を比較して、各レディ
// 状態のタスク。どの値が大きいかというと、それはまだ多くの実行時間の
// タスクを指し、nextはそのタスクのタスク番号を指します。
149         while (--i) {
150             if (!*--p)
151                 を続けています。
152             if ((*p)-> state == TASK_RUNNING && (*p)->counter > c)
153                 c = (*p)->counter, next = i. です。
154         }
// 比較の結果、カウンタの値が0にならない場合、または
// システム内に実行可能なタスクがない場合（cが-1のまま、next=0）、外側のwhileを終了します。
// のループ(144行)の中で、後者のタスク切り替えマクロ(161行)を実行します。 それ以外
// の場合は
// 各タスクのカウンタ値は、各タスクの優先度に応じて更新され、その後
// を144ラインに戻して再比較します。カウンタの値は次のように計算されます。
//     カウンタ = カウンタ /2 + プライオリティ
// ここでの計算過程では、プロセスの状態を考慮していないことに注意してください。
155         if (c) break;
156         for(p = & LAST_TASK; p > & FIRST_TASK; --p)
157             if (*p)
158                 (*p)->counter = ((*p)->counter >> 1) + です。
159                 (*p)->priorityです。
160     }
// 次のマクロ(sched.h)は、選択されたタスクnextをカレントタスクとし
// 実行するタスクに切り替えます。146行目でnextが0に初期化されているので、nextは
// システム内に実行すべき他のタスクがない場合は常に0です。そのため、スケジューラーの
// は、システムがアイドル状態のときにタスク0を実行します。この時、タスク0はpause()を実行するだけです。
// syscallで、この関数が再び呼ばれるようになります。
161         switch to(next);
162     }.
163
// これは pause() システムコールで、現在のタスクの状態を
// 割り込み可能な待機状態(TASK_INTERRUPTIBLE)にして、スケジュールを変更します。
// このシステムコールは、シグナルを受信するまでプロセスをスリープ状態にします。この
// シグナルは、プロセスを終了させたり、プロセスにシグナルキャプチャーを呼び出させるために使用されます。
// 関数です。Pause()は、シグナルが捕捉され、シグナル捕捉ハンドラが
// を返します。この時点でpause()の戻り値は-1となり、errnoにはEINTRが設定されます。
// まだ完全には実装されていません（カーネル0.95まで）。
164 int sys_pause(void)
165 {
166     current-> state = TASK_INTERRUPTIBLE;
167     schedule() です。
168     return 0;
169 }.
170
// 次の関数は、現在のタスクを中断可能または中断不可能に設定します。
// スリープ状態にして、スリープキューのヘッドポインタを現在のタスクに向けさせる。
// 関数のパラメータ p は、タスクキューの待ち受けポインタであり、パラメータの状態は
// タスクが使用する状態 sleep: TASK_UNINTERRUPTIBLEまたはTASK_INTERRUPTIBLEです。のタスクは
// を使ってカーネルを明示的に目覚めさせる必要があります。
// 割り込み可能なスリープ状態にあるタスクは、信号によって起動することができます。

```

```

// タスクのタイムアウトなど。(レディ状態TASK_RUNNINGに設定)。
// *** なお、このコードはあまり成熟していないため、いくつかの問題点があります。
171 static inline void sleep_on(struct task_struct **p, int state)
172 {
173     struct task_struct *tmp;
174
175     // まず、ポインタが無効であれば、終了します。(ポインタが指すオブジェクトは
176     // はNULLでも構いませんが、ポインタ自体が0になることはありません。)現在のタスクがタスク 0 であれば
177     // カーネルパニック。
178     if (!p)
179         を返すことができます。
180         もし (current == &(init_task.task))
181             panic ("task[0] trying to sleep")が発生します。
182
183     // そして、inode->i_waitのように、(もしあれば) tmpにすでに待ち行列に入っているタスクを指定させます。
184     // そして、スリープキューの先頭を現在のタスクに向けます。これにより、現在のタスクが
185     // *pの待ち行列になります。その後、現在のタスクは指定された待機状態になり
186     // 再スケジューリングを行います。
187
188     tmp = *p;
189     *p = 電流。
190     current-> state = state;
191 repeat: schedule();
192
193     // この待機中のタスクが目覚めた時にのみ、プログラムは次の実行を続けます。
194     // ここでプロセスが明示的に起こされて実行されたことを示す。
195     // キューに待ちタスクが残っていて、キューが指すタスクが
196     // ヘッダー *p が現在のタスクではない場合、次のキューに入るタスクがまだ存在します。
197     // タスクが          挿入されます。そのため、入力された          これらの後続タスクも覚醒させる必要が
198     // あります。
199     //を後回しにします。そのため、キューのヘッダで示されるタスクが先にレディ状態になり
200     // 現在のタスク自体を中断しない待機状態にする。つまり、待機後
201     // このような後続のキューイングされたタスクがアウェイクンされるためには、現在のタスク自体を
202     // wake_up()関数を使って起こされます。その後、ラベルリピートにジャンプし、再実行して
203     // schedule()関数。
204
205     if (*p && *p != current) {...
206         (*p). 状態 = 0;
207         current-> state = TASK_UNINTERRUPTIBLE;
208         goto リピート。
209     }
210
211     // ここで実行、このタスクが本当に実行に目覚めたことを示す。この時点で
212     // キューヘッドは、このタスクを指すべきです。それが空の場合、それはタスクの中に
213     // スケジュールに問題があると、警告メッセージが表示されます。最後に、ヘッドに
214     // 目の前でキューに入ったタスクを指します(*p = tmp)。そのようなものがあれば
215     // タスク、つまり、キューにタスクがある (tmpが空ではない) 場合、そのタスクが起こされます。
216     // そのため、最初にキューに入ったタスクが最終的に待ち行列を設定します
217     // ウェイクアップ後に実行されるときは、ヘッドをNULLにする。
218
219     もし (!*p)
220         printk ("Warning: *P = NULL\|r")となります。
221     if (*p = tmp)
222         tmp-> state=0;
223 }
224
225 // 現在のタスクを割り込み可能な待機状態(TASK_INTERRUPTIBLE)にし、それを
226 // ヘッドポインタ*pで指定された待ち行列に入ります。この待機状態のタスクは

```

シグナルやタスクのタイムアウトなどの手段で目覚めさせます。

```

194 void interruptible_sleep_on(struct task_struct **p)
195 {
196     __sleep_on(p, TASK_INTERRUPTIBLE);
197 }
198
// 現在のタスクを割り込み可能な待機状態(TASK_UNINTERRUPTIBLE)にし、そのタスクに
// それをヘッドポインタ*pで指定された待ち行列に入れる。この待機状態のタスクができるのは
// wait_up()関数によって起こされます。
199 void sleep_on(struct task_struct **p)
200 {
201     __sleep_on(p, TASK_UNINTERRUPTIBLE);
202 }
203
// 割り込み禁止の待機タスクを起こします。*p はタスク待ち行列の先頭ポインタです。以降は
// 新しい待機タスクが待機キューの先頭に挿入されると、ウェイクアップは最後のタスクとして
// 待ち行列に入ります。タスクがすでに停止状態またはゾンビ状態にある場合は、警告
// のメッセージが表示されます。
204 void wake_up(struct task_struct **p)
205 {...
206     if (p && *p) {
207         if ((*p).state == TASK_STOPPED)
208             printk("wake_up: TASK_STOPPED");
209         if ((*p).state == TASK_ZOMBIE)
210             printk("wake_up: TASK_ZOMBIE");
211         (*p).state = 0; // TASK_RUNNING
212     }
213 }
214
215 /*
216  * OK, here are some floppy things that should be in the kernel
217  * 適切です。これらは、フロッピーにタイマーが必要なために存在しています。
218  * それが一番簡単な方法でした。
219  */
// 以下の220--281行のコードは、フロッピードライブのタイミングを処理するために使用されます。
// このコードを読む前に、以下の章の説明に目を通してください。
// フロッピードライブ用のブロックデバイス(floppy.c)を使用するか、またはこのコードを見て
// フロッピーブロックデバイスのドライバです。
//
// 配列wait_motor[]は、駆動モーターを待つプロセスポインタを格納するために使用されます。
// で通常速度で起動します。配列のインデックス0~3は、フロッピードライブA~Dに対応しています。
// 配列mon_timer[]には、各フロッピードライブのモーターに必要なティック数が格納されています。
// で起動します。プログラムのデフォルトの起動時間は50ティック(0.5秒)です。
// 配列moff_timer[]には、各フロッピードライブが以下の状態になるまでの時間が格納されています。
// モーターが停止します。10,000ティック(100秒)に設定されています。
220 static struct task_struct * wait_motor[4] = {NULL, NULL, NULL, NULL};
221 static int mon_timer[4]={0,0,0,0};
222 static int moff_timer[4]={0,0,0,0};

// 以下の変数は、現在のデジタル出力レジスタ(DOR)に対応しています。
// フロッピーディスクドライブコントローラ。本レジスタの各ビットの定義は以下の通りです。
// ビット7-4: ドライブD-Aモーターの起動を個別に制御します。1-スタート; 0-クローズ。

```

```

// Bit 3:1 - DMAおよび割り込み要求を有効にし、0 - DMAおよび割り込み要求を無効にする。
// Bit 2:1 - フロッピー・ドライブ・コントローラ(FDC)の起動; 0 - FDCのリセット。
ビット1-0: フロッピーディスクドライブA~Dの選択に使用します。
// ここで設定される初期値はDMAと割り込み要求を許可し、FDCを開始する。
223 unsigned char current_DOR = 0x0C;
224
// フロッピードライブが正常に動作し始めるまでの待ち時間を指定します。
// パラメータ nr はフロッピーディスクのドライブ番号 (0-3) で、戻り値は nr のティック数で
// す。
// 選択された変数は、選択されたフロッピードライブフラグです (blk_drv/floppy.c、123行目)。
// マスクは、選択されたフロッピードライブDORのスタートモータービット、上位4ビットの
// は、フロッピードライブのスタートモーターフラグです。
225 int ticks_to_floppy_on(unsigned int nr)
226 {
227     extern unsigned char selected;
228     unsigned char mask = 0x10 << nr;
229
// システムには最大4台のフロッピードライブが搭載されています。最初に、その時間 (100秒) を
// 設定します。
// 指定されたフロッピー・ドライブnrが停止する前に、停止する必要があります。次に、現在のDOR
// を
// の値を変数maskに格納し、指定されたフロッピーディスクドライブのモータースタートフラグを
// の中に入っています。
230     if (nr>3)
231         panic ("floppy_on: nr>3")となります。
232     mon timer[nr]=10000 ;/* 100秒=とても大きい :-)*/*
233     cli(); /* floppy_off を使ってオフにする */。
234     mask |= current_DOR;
// 現在、フロッピーディスクドライブが選択されていない場合は、まず、他の
// フロッピードライブの選択ビットを設定してください。
235     if (! selected) {
236         mask &= 0xFC;
237         マスク |= nr;
238     }
// DORの現在の値が必要な値と異なる場合、新しい値 (マスク) を
// FDCのデジタル出力ポートに出力され、起動に必要なモーターがあれば
// まだ起動していない場合は、対応するフロッピーディスクドライブのモーター起動タイマーの値
// が設定されます。
// (HZ/2 = 0.5) 秒または50ティック)を設定します。)起動している場合は、起動タイミングを2
do_floppy_timer()での以下の減価償却の要件を満たすことができる、 // ティック。
その後、現在のデジタル出力レジスタ current_DOR が更新されます。
239     if (mask != current_DOR) {...
240         outb(mask, FD_DOR) です。
241         if ((mask ^ current_DOR) & 0xf0)
242             mon timer[nr] = HZ/2;
243         else if (mon timer[nr] < 2)
244             mon timer[nr] = 2;
245         current_DOR = mask;
246     }
247     sti (); // int を有効にする。
248     return mon timer [nr]; // モーターの 起動に必要な時間値を返す。
249 }
250
// フロッピーディスクドライブモーターの起動に必要な時間を待つ。
// 指定したフロッピーディスクドライブのモーターが起動してから動作するまでの遅延時間を設定

```

します。

// 通常の速度で動作し、その後スリープします。タイマー割り込み処理中は、ここで設定した遅延値が
// がデクリメントされます。遅延時間が経過すると、ここで待機中のプロセスを起こします。

```

251 void floppy_on(unsigned int nr)
252 {...
    // 割り込みを禁止する。モータ起動タイマが満了していない場合、現在の処理は
    // 常に中断できないスリープ状態にして、待機中のキューに入れる
    // モーターを動作させます。そして、割り込みを開きます。
253     cli()です。
254     while (ticks_to_floppy_on(nr))
255         sleep_on(nr+wait_motor)です。
256     sti();
257 }
258
    // モーターストールタイマー（3秒）をオフにする場合に設定します。
    // この関数を使って、指定したフロッピードライブの電源を明示的にオフにしない場合
    // モーターを100秒間ONにするとOFFになります。
259 void floppy_off(unsigned int nr)
260 {
261     moff_timer[nr]=3*HZ;
262 }。
263
    // フロッピーディスクのタイマーサブルーチンです。モータスタートタイミング値の更新とモータオフ
    // ストールカウント値。このサブルーチンは、システムタイマーの割り込み時に呼び出されるので
    // システムは、1ティック（10ms）が経過するたびに1回呼び出され、モーターのオンの値が
    // またはOFFのタイマーは、その都度更新されます。モーターストールのタイミングが切れた場合、DORモータース
    // タートビットがリセットされます。
264 void do_floppy_timer(void)
265 {
266     int i;
267     unsigned char mask = 0x10;
268
    // システムに搭載されている4つのフロッピードライブについて、使用されているフロッピードライブを1つずつ確
    // 認します。
    // 1つです。
    // DORで指定されたモーターでない場合はスキップしま
    // す。
    // モータースタートのタイマーが切れ たら
    // プロセスを起こします。モーターオフタイマーが切れると、モーターのスタートビットをリセットする。
269     for (i=0 ; i<4 ; i++,mask <=< 1) {。
270         if (!(mask & current_DOR))
271             を続けています。
272         if (mon_timer[i]) {...
273             if (!mon_timer[i])//モーターオンのタイマーが切れる と
274                 wake_up(i+wait_motor); // プロセスを起動する。
275         } else if (! moff_timer[i]) {...
276             current_DOR &= ~mask;          モーターのリセッ スタート
                                                ト ビット
277             outb(current_DOR, FD_DOR)。    // DORを更新しま
                                                す。
278         } else
279             moff_timer[i]--。
280     }
281 }
282
    // 以下は、カーネルタイマーのコードです。タイマーは最大64個まで設定できます。
    // 285～289行目では、リンクされたタイマーリスト構造とタイマー配列を定義しています。リンクされたタイマー
    // このリストは、フロッピーディスクドライブがタイミングを計るためにモーターをオン・オフするためのものです
    // このタイプのタイマーは、最近のLinuxシステムのダイナミックタイマーに似ていて

```

// カーネルでのみ使用されるようになっていきます。

[283](#) #define [TIME_REQUESTS](#) 64

[284](#)

```

285 static struct timer\_list {
286     long jiffies                // タイマーの秒数
287     void (*fn) ()               // タイマーハンドラー
288     struct timer\_list * next;    // 次のタイマーを指します。
289 } timer\_list[TIME REQUESTS], * next\_timer = NULL ; // next\_timer はタイマーキューの
先頭です。290
// タイマーサブルーチンを追加します。入力パラメータは、指定したタイミングの値（ティック）と
// 関連するハンドラーを表示します。フロッピーディスクドライバは、この関数を使って、遅延
// モーターの起動や停止を行う // 操作です。
// jiffies - 時限数; *fn() - 時間切れの際に実行される関数。
291 void add\_timer(long jiffies, void (*fn)(void))
292 {。
293     タイマーリスト構造体 *
p; 294
// 追加されるタイマーハンドラポインタがNULLの場合、この関数は終了します。
295     if (!fn)
296         を返すことができます。
297         cli()です。
// タイマーの時間値 <=0 の場合、そのハンドラは直ちに呼び出され、タイマーは
// リンクリストに追加されます。
298     if (jiffies <= 0)
299         (fn) () となっています。
300     else {
// それ以外の場合は、タイマー配列から空いているエントリを探します。
301         for (p = timer\_list ; p < timer\_list + TIME REQUESTS ; p++)
302             if (!p->fn)
303                 ブレークします。
// タイマー配列を使い切った場合、システムはクラッシュします :-)。そうでなければ、タイマーデータの
// 構造体に情報を詰め込み、リストのヘッダーにリンクさせる。
304         if (p >= timer\_list + TIME REQUESTS)
305             panic("No more time requests free")。
306         p->fn = fn;
307         p->jiffies = jiffiesです。
308         p->next = next\_timer;
309         next\_timer = p;
// リンクされたリストのアイテムは、時間値に応じて早いものから遅いものへとソートされます。
// ソートする前に必要なティック数を引きます。このようにして、処理時に
// タイマーの場合は、最初の項目のタイミングが切れたかどうかを確認すればよい。
310         while (p->next && p->next-> jiffies < p-> jiffies) {。
311             p->jiffies -= p->next->jiffies。
312             fn = p->fn;
313             p->fn = p->next->fn となります。
314             p->next->fn = fn;
315             jiffies = p-> jiffies です。
316             p->jiffies = p->next->jiffies です。
317             p->next-> jiffies = jiffies です。
318             p = p->nextとなります。
319         }
320     }
321     sti();
322 }
323
//// タイマー割り込みハンドラで呼び出されるC関数です。_timer_interruptで呼び出される

```


sys_call.sファイルの//(189,209行目)です。パラメータcplは現在の特権
実行されているコードセクターの特権レベルである、レベル0または3。

// 割り込みが発生します。Cpl=0は、割り込みが発生したときにカーネルコードが実行されていることを意味しま
す。

cpl=3は、割り込みが発生したときにユーザーコードが実行されていることを意味します。

が発生します。タスクの場合、その実行時間スライスを使い切ってしまうと、タスクが切り替えられます。で
この時点で、この機能はタイミングの更新を行います。

324 void [do_timer](#)(long cpl)

325 {。

326 static int blanked = 0;

327

// まず、画面のブランクアウト操作を行う必要があるかどうかを判断します。もしblankcountが

// が0でない場合、または黒画面の遅延間隔blankintervalが0の場合は、画面の

// がすでに黒画面になっている場合（黒画面フラグblanked = 1）、画面を復元します。

// blankcountが0でない場合は、デクリメントされ、ブラックスクリーンフラグがリセットされます。

328 if ([blankcount](#) || ! [blankinterval](#)) {...

329 if (ブランク)

330 [unblank_screen](#)()です。

331 if ([blankcount](#))

332 [blankcount](#)--。

333 blanked = 0となります。

// それ以外の場合は、ブラックスクリーンフラグが設定されていなければ、画面は真っ白にな

// り、フラグは

334 が設定されます。

 } else if (!blanked) {

335 [blank_screen](#)()です。

336 blanked = 1;

337 }

// 次に、ハードディスクの操作によるタイムアウトの問題を処理します。もし、ハードディスクのタイムアウトカ
ウントが

をデクリメントして0にすると、ハードディスクのアクセスタイムアウト処理を行います。

338 if ([hd_timeout](#))

339 if (!--[hd_timeout](#))

340 [hd_times_out](#) (); // blk_drv/hdc, line

318 341

// ビープ音のカウンタ数に達した場合、ビープ音をオフにする。（ポート0x61にcmdを送信、リセット）

ビット0は8253チップのカウンタ2を制御し、ビット1はスピーカーを制御します。）。

342 も ([beepcount](#)) // ビープ音が鳴る (chr_drv/console.c, 950)

し

343 if (!--[beepcount](#))

344 [sysbeepstop](#)(); // chr_drv/console.c, 944.

345

// 現在の特権レベル(cpl)が0(最高)の場合(カーネルが非特権であることを示す)

// プログラムが動作している)、その後、カーネルコードのランタイムstimeがインクリメントされます。

// 一般ユーザープログラムが動作していることを意味し、utimeを追加しています。

346 if (cpl)

347 [current](#)-> [utime](#)++となります。

348 その他

349 [current](#)-> [stime](#)++;

350

// タイマーが存在する場合、リンクリストの最初のタイマーの値がデクリメントされて

// 1です。0となった場合、対応するハンドラが呼び出され、ハンドラポインタの

//がnullに設定され、その後、タイマーが削除されます。

```
351     if (next_timer                                ) { //タイマーリストのヘッダー。  
352         next_timer->jiffies--。  
353         while (next_timer && next_timer-> jiffies <= 0) {。
```

```

354 void (*fn)(void); // 関数ポインタの定
義です。 355
356 fn = next_timer->fn;
357 next_timer->fn = NULL;
358 next_timer = next_timer->next;
359 (fn)(); // タイマーハンド
ラ
を呼び出します。
360 }
361 }
// 現在のフロッピーディスクコントローラFDCのDORのモーターイネーブルビットが設定されている場合。
// フロッピーディスクのタイマールーチンが実行されます。
362 if (current_DOR & 0xf0)
363 do_floppy_timer()です。
// タスクにまだ実行時間がある場合は、ここで終了してタスクの実行を継続します。そうでない場合は
// 現在のタスク実行カウンタは0に設定されます。 また、カーネルコードで実行されている場合には
// 割り込みが発生すると戻り、そうでない場合は、ユーザープログラムが
// が実行されたので、スケジューラーを呼び出して、タスク切り替え操作を試みます。
364 if ((--current->counter)>0) return;
365 current->counter=0となります。
366 if (!cpl) return; // カーネルコード
367 schedule();
368 }。
369
// システムコール機能 - アラームタイマーの値(秒単位)を設定します。
// パラメータseconds > 0の場合、新しいタイミングが設定され、残りの間隔は
// 元のタイミングに戻し、そうでなければ0を返します。
// プロセスデータ構造のアラームフィールドの単位はティックで、以下の合計になります。
// システムの目盛り値jiffiesとタイミング値、すなわち「jiffies + HZ* 秒」、ここでは
// 定数 HZ = 100 です。この機能の主な動作は、アラームフィールドを設定することです。
// と2つの時間単位の間で変換することができます。
370 int sys_alarm(long seconds)
371 {
372 int old = current-> alarm;
373
374 if (old)
375 old = (old - jiffies) / HZ;
376 current-> alarm = (seconds>0)?(jiffies+HZ*seconds):0;
377 return (old);
378 }。
379
// 現在のプロセスのpidを取得します。
380 int sys_getpid(void)
381 {
382 return current->pid;
383 }。
384
// 親のpidを取得する - ppid。
385 int sys_getppid(void)
386 {
387 return current->p_pptr->pid;
388 }。
389
// 現在のユーザーIDを取得します。
390 int sys_getuid(void)

```

```

391 {
392     return current->uid;
393 }.
394
395 // 有効なユーザーID - euid を取得します。
396 int sys\_geteuid(void)
397 {
398     return current->euid;
399 }.
400
401 // グループIDの取得 -
402 int sys\_getgid(void)
403 {
404     return current->gid;
405 }.
406
407 // 有効なグループID - egidを取得します。
408 int sys\_getegid(void)
409 {
410     return current->egid;
411 }.
412
413 // システムコール機能 -- CPU使用の優先度を下げる（誰かが使う？
414 // パラメータの増分は、0より大きい値に制限する必要があります。
415 int sys\_nice(long increment)
416 {
417     if (current->priority-increment>0)
418         current->priority -= increment;
419     0を返す;
420 }.
421
422 // カーネルスケジューラの初期化サブルーチンです。
423 void sched\_init(void)
424 {
425     int i;
426     struct desc\_struct * p; // ディスクリプタ構造体ポイ
427     ンタ 421
428     // Linux開発当初、カーネルは成熟していませんでした。カーネルのコードは
429     // を修正することが多い。Linus氏は、これらの重要な部分を誤って修正してしまったのではないかと心配しました。
430     // データ構造がPOSIX規格との互換性を欠く原因となるため、データ構造に
431     // ここで次のように述べています。これは必要なことではなく、純粋に自分自身を戒めるためと
432     // カーネルのコードを変更する人。
433     if (sizeof(struct sigaction) != 16) // シグナル構造体
434         panic("Struct sigaction MUST be 16 bytes").
435
436     // TSS（タスクステートセグメント）記述子とLDT（ローカルデータテーブル）記述子の
437     // グローバルディスクリプターテーブル（GDT）には、初期タスク（タスク0）が設定されています。
438     // FIRST_TSS_ENTRYとFIRST_LDT_ENTRYの値は、それぞれ4と5で、以下のように定義されています。
439     gdtは記述子配列（linux/head.h）であり、それは
440     ファイル head.s の 234 行目にある // ベースアドレス _gdt です。したがって、 gdt + FIRST_TSS_ENTRY は
441     // gdt[FIRST_TSS_ENTRY]（つまりgdt[4]）は、gdt配列のアイテム4のアドレスです。
442     // asm/system.hの65行目を参照してください。
443     set\_tss\_desc(gt+FIRST_TSS_ENTRY,&(init_task.task.tss));

```

```

425     set_ldt_desc(gt+FIRST_LDT_ENTRY,&(init_task.task.ltt));
// タスク配列とディスクリプターテーブルのエントリをクリアする (i=1から始まるので、初期タ
// スクのディスクリプターが残っていることに注意)。
426     p = gdt+2+FIRST_TSS_ENTRY;
427     for(i=1;i<NR_TASKS;i++) {...
428         task[i] = NULLとなります。
429         p->a=p->b=0となります。
430         p++;
431         p->a=p->b=0となります。
432         p++;
433     }
434     /* NTをクリアすることで、後でトラブルが起きないようにするためです。
// EFLAGSのNTフラグは、タスクのネストした呼び出しを制御するために使用されます。NTが設定さ
// れると
// 現在の割込みタスクは、IRET命令を実行するとタスクスイッチが発生します。
// NTは、TSSのback_linkフィールドが有効かどうかを示す。NT=0の場合は無効です。
435     asm ("pushfl ; andl $0xffffbfff, (%esp) ; popfl"); // NTのリセット

// タスク0のTSSセグメントセレクトがタスクレジスタ(TR)にロードされる。また、LDT
// セグメントセレクトは、ローカルディスクリプターテーブルレジスタ (LDTR) にロードされます。注意!は
// GDT内の対応するLDT記述子の//セレクトがLDTRに読み込まれます。それは
// 今回は明示的にロードするだけです。その後、新しいタスクLDTのロードは
// TSSのLDTエントリに応じて、CPUが自動的にロードする。
436     ltr(0); // include/linux/sched.h, 157-158
437     lldt(0); // 0はタスクNo.

// 以下のコードは、8253タイマーを初期化するために使用されます。チャンネル0、作業を選択
// モード3、バイナリカウントモード。チャンネル0の出力端子は、IRQ0に接続されています。
// 10ミリ秒ごとにIRQ0のリクエストを発行する割り込み制御マスターチップ。
// LATCHは、初期のタイミングカウント値です。
438     outb_p(0x36, 0x43); /* バイナリー、モード3、LSB/MSB、ch 0 */
439     outb_p(LATCH & 0xff, 0x40); /* LSB */ (LSB)
440     outb(LATCH >> 8, 0x40); /* MSB */ (LATCH >> 8, 0x40)

// タイマー割り込みハンドラを設定する。割り込みコントローラのマスクコードを変更して
// タイマー割り込みが発生します。その後、システムコールの割り込みゲートを設定します。のマクロ定義は
// asm/system.hファイルの33行目と39行目に記述されています。
441     set_intr_gate(0x20, & timer_interrupt);
442     outb(inb_p(0x21)&~0x01, 0x21); // intマスクを変更し、タイマー
// を有効にする。
443     set_system_gate(0x80, & system_call);
444 }.
445

```

8.6.3 インフォメーション

8.6.3.1 フロッピーディスクコントローラ

ここでは、上記プログラムの応用として、フロッピーディスクコントローラ (FDC) が使用する I/Oポートのみを簡単に紹介します。FDCのプログラミングの詳細については、第9章のfloppy.c以降の説明を参照してください。FDCのプログラミングでは、4つのポートにアクセスする必要があります。これらのポートは、コントローラ上の1つまたは複数のレジスタに対応しています。通常のフロッピーディスクコントローラの場合、表8-3のようなポートがあります。

表8-3 フロッピーディスクコントローラのポート

I/Oポートポート	名Read/Writeレジスター名
-----------	-------------------

0x3f2	fd_dor	ライトオ	デジタル出力レジスタ（デジタルコントロー
0x3f4	fd_status	ンリー リ	ラレジスタ）FDCメインステータスレジス
0x3f5	fd_data	ードオン	タ
		リー リー	FDCデータレジスタ
		ド/ライト	
		0x3f7FD_DIR読み取り	専用デジタル入力レジスタ
		0x3f7FD_DCRWrite	onlyDrive Control Register (Transfer Rate Control)

デジタル出力レジスタDOR（またはデジタルコントロール）は、ドライバモータのターンオン、ドライバセレクト、FDCのスタート／リセット、DMAや割り込み要求の有効／無効を制御する8ビットのレジスタです。

FDCのメインステータスレジスタも8ビットのレジスタで、FDCとフロッピーディスクドライブFDDの基本的な状態を反映しています。通常、メイン・ステータス・レジスタのステータス・ビットは、CPUがFDCにコマンドを送る前、あるいはFDCが動作結果を取得する前に読み込まれ、現在のFDCデータ・レジスタの準備ができているかどうか、またデータ転送の方向を決定する。

FDCのデータポートは、複数のレジスタ（書き込み可能なコマンド・レジスタとパラメータ・レジスタ、読み出し可能なリザルト・レジスタ）に対応していますが、データポート0x3f5には、常に1つのレジスタしか表示できません。書き込み専用のレジスタにアクセスするときは、主状態制御のDIO方向ビットが0（CPU→FDC）でなければならない、読み取り専用のレジスタにアクセスするときはその逆となります。結果を読み出す場合、FDCがビジー状態でない場合にのみ結果が読み出されます。通常、結果データは最大7バイトです。

フロッピーディスクコントローラは、合計15個のコマンドを受け付けることができます。各コマンドは、「コマンドフェーズ」「実行フェーズ」「結果フェーズ」の3つのフェーズを経ます。

コマンドフェーズとは、CPUがFDCにコマンドバイトとパラメータバイトを送信することである。最初のバイトは常にコマンドバイト（コマンドコード）で、その後0～8バイトのパラメータが続きます。

実行フェーズとは、FDC実行コマンドで指定された動作のことです。実行フェーズでは、CPUは介入しません。通常、FDCはコマンド実行の終了を知るために、割り込み要求を発行する。CPUから送られてくるFDCコマンドがデータを転送するものである場合、FDCは割り込みモードでもDMA方式でも実行可能です。割り込みモードでは、1バイトずつの転送を行います。DMAモードはDMAコントローラの管理下であり、FDCとメモリはすべてのデータが転送されるまでデータを転送します。この時、DMAコントローラはFDCに転送バイト数終了信号を通知し、最後にFDCが割り込み要求信号を発行してCPUに実行フェーズの終了を知らせます。

リザルトフェーズは、CPUがFDCデータレジスタの戻り値を読み、FDCコマンドの実行結果を得ることである。返される結果データは、0～7バイトの長さです。リザルトデータを返さないコマンドの場合は、FDCに検出割り込みステータスコマンドを送って動作状況を把握する必要があります。

8.6.3.2 プログラマブル・タイマ/カウンタ・コントローラ

1. インテル8253（8254）チップ

インテル8253(または8254)は、プログラマブル・タイマ/カウンタ・チップで、コンピュータで一

一般的に遭遇する時間制御の問題を解決し、ソフトウェアの制御下で正確な時間遅延を生成します。このチップは、3つの独立した16ビットカウンタチャンネルを備えています。各チャンネルは異なる動作モードで動作し、これらはソフトウェアで設定できます。8254は、8253チップの改良版です。主な機能は基本的に同じですが、8254チップにはリードバックコマンドが追加されています。以下の説明では、8253チップと8254チップを総称して「8253」とし、機能の違いを指摘するにとどめます。

8253チップのプログラミングは比較的簡単で、さまざまな長さの時間の遅延を作り出すことができます。8253(8254)チップのブロック図を図8-8に示します。

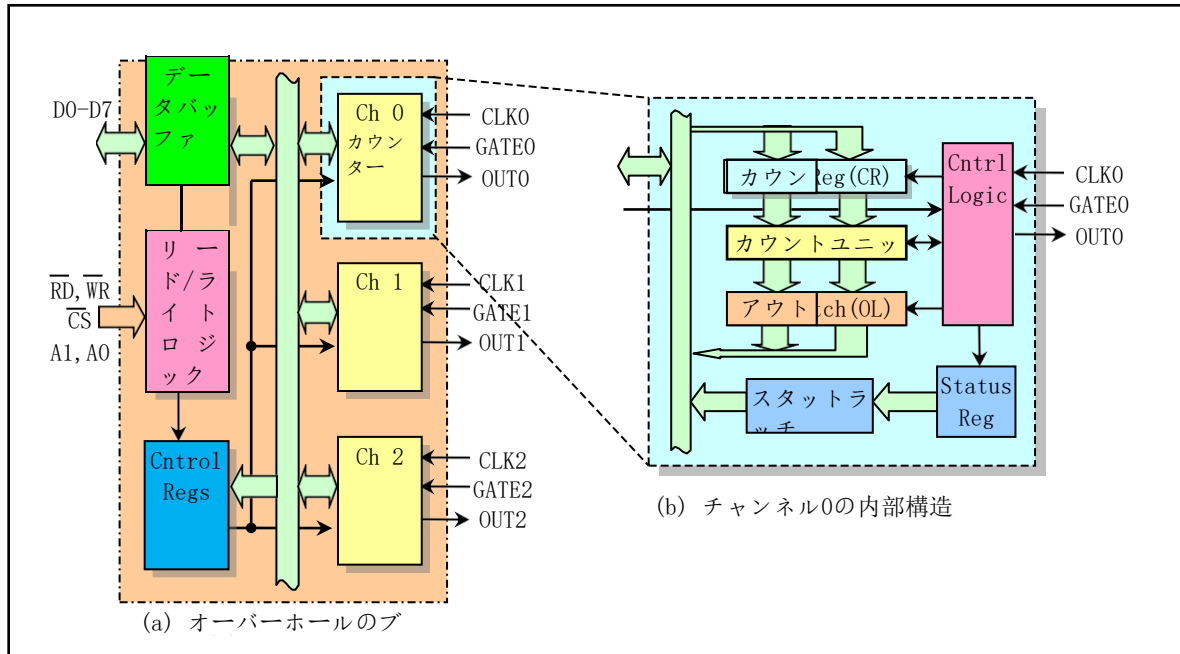


図 8-8 8253(8254)タイマ/カウンタチップの内部構造

3-state, bidirectional 8-bit Data Bus

Bufferは、システムデータバスとのインターフェースに使用されます。リード/ライトロジックは、システムバスからの入力信号を受信し、他の部分に入力される制御信号を生成するために使用されます。アドレスラインA1,A0は、読み書きが必要な3つのカウンタチャンネルまたはコントロールワードレジスタの1つを選択するために使用されます。通常はシステムのアドレスラインA0, A1に接続されています。リード/ライトピンRD, WRとチップセレクトピンCSは、CPUが8253チップのリード/ライト動作を制御するために使用されます。コントロール・ワード・レジスタは、CPUが指定されたカウンタの動作を設定するために使用します。これは書き込み専用のレジスタです。しかし、8254チップでは、リードバックコマンドを使ってステータス情報を読み出すことができます。3つの独立したカウンタチャンネルは全く同じように機能しますが、それぞれが異なる方法で動作することができます。各カウンタがどのように動作するかは、コントロールワードレジスタで決定します。各カウンタのCLK端子は、クロック周波数発生器（水晶振動子）に接続されています。クロック入力周波数は、8253は最大2.6MHz、8254は最大10MHzです。GATE端子は、カウンタのゲート制御入力で、カウンタの起動・停止や出力状態の制御に使用される。ピンOUTは、カウンタの出力信号端子である。

図8-8(b)は、カウンタチャンネルの1つの内部論理ブロック図です。ステータス・レジスタには、コントロール・ワード・レジスタの現在の内容と、ロック時の出力およびNull Count Flagのステータスが格納されます。実際のカウンタは、図中のCE（カウンティング・ユニット）です。これは16ビットの事前設定可能な同期式ダウンカウンタである。出力ラッチOL（Output Latch）は、2つの8ビットラッチOLmとOL1で構成されており、それぞれがラッチのハイバイトとローバイトを表しています。通常、2つの出力ラッチの内容は、カウントユニットCEの内容変化に追従して変化しますが、チップがカウンタラッチコマンドを受信すると、その内容はロックされます。CPUがその内容を読み出すまでは、CEの内容変化に追従し続けることになります。なお、CEの値は読めません。カウント

値を読み取る必要がある場合は、常にラッチOLの内容が出力されます。図8-8(b)の他の2つは、カウントレジスタ（CR）と呼ばれる8ビットのレジスタです。CPUがカウンタチャンネルに新しいカウント値を書き込むと、初期のカウント値がこの2つのレジスタに格納され、その後、カウントユニットCEにコピーされます。これらの2つのレジスタは、カウンタがプログラムされるとクリアされます。したがって、初期カウント値がカウントレジスタCRに保存された後、カウントユニットCEに送られます。GATEがイネーブルになると、カウントユニットはクロックパルスCLKの作用下でカウントダウン動作を行います。カウント値がゼロにデクリメントされるまで、1つデクリメントされるたびに、

a

信号がOUT端子に送られます。

2. 8253（8254）チップのプログラミング

システムの電源を入れたばかりの状態では、8253の状態は不明です。8253にコントロールワードと初期カウント値を書き込むことで、使用したいカウンタをプログラムすることができます。使用しないカウンタについては、プログラムする必要はありません。表8-4にコントロールレジスタの内容のフォーマットを示します。

表 8-4 8253(8254)チップコントロールワードフォーマット

ビット	名前	説明
7	SC1	SC1,SC0は、カウンタ・チャンネル0-2の選択、またはコマンドの読み出しに使用します。
6	SC0	
		00 - チャンネル0、01 - チャンネル1、02 - チャンネル2、11 - リードバックコマンド（8254のみ）。
5	RW1	RW1とRW0は、カウンタのリード/ライト動作の選択に使用します。
4	RW0	
		00 - レジスタラッチコマンドを示す; 01 - 低バイト（LSB）を読み書きする; 10 - 高バイト（MSB）を読み書きする; 11 - 低バイトを最初に、次に高バイトを読み書きする。
3	M2	M2-M0は、指定したチャンネルの動作モードを選択するため に使用します。000 - モード0、001 - モード1、010 - モード 2。 011-モード3、100-モード4、101-モード5。
2	M1	
1	M0	
0	BCD	
		カウント値のフォーマット選択。0 - 16ビットバイナリカウント、1 - 4BCDコードカウント。

CPUが書き込み動作を行う際、A1、A0ラインが11（ここでは、PCマイコンの対応するポート0x43）であれば、コントロールワードレジスタにコントロールワードが書き込まれます。コントロールワードの内容は、プログラムされるカウンタ・チャンネルを指定します。指定されたカウンタには、初期カウント値が書き込まれます。A1とA0が00、01、10（それぞれPCポート0x40、0x41、0x42に対応）の場合、3つのカウンタのうち1つが選択されます。書き込み操作では、まずコントロールワードを書き込み、次に初期カウント値を書き込む必要があります。初期カウント値は、コントロールワードで設定されたフォーマット（バイナリまたはBCDコードフォーマット）で書き込む必要があります。カウンタが動作を開始しても、いつでも指定したカウンタに新しい初期値を書き換えることができます。設定されているカウンタの動作には影響しません。

読み出しの際、8254チップのカウンタの現在のカウント値を読み出すには3つの方法があります。

(1)単純な読み出し動作 (2)カウンタラッチコマンドの使用 (3)リードバックコマンドの使用。(1)の方法では、読み出し中にGATE端子または対応する論理回路でカウンタのクロック入力を一時的に停止させる必要があります。そうしないと、カウント動作が行われている可能性があり、その結果、読み取り結果が正しくないものとなります。 2つ目の方法は、カウンタのラッチコマンドを使用する方法です。このコマンドは、読み出し動作の前にまずコントロールワードレジスタに送信され、D5、D4の2ビット（00）は、コントロールワードコマンドの代わりにカウンタラッチコマンドが送信されたことを示します。カウンタは、このコマンドを受け取ると、カウントユニットCEのカウント値を出力ラッ

チレジスタOLにラッチします。この時点で、CPUがOLの内容を読み込まなければ、再度カウンタラッチコマンドを送信してもOLの値は変わりません。CPUがカウンタ動作の読み出しを実行して初めて、OLの内容は自動的にカウントユニットCEに追従して変化していきます。3つ目の方法は、リードバック・コマンドを使うことです。しかし、この機能を持っているのは8254だけです。このコマンドを使えば、現在のカウント値、カウンタの動作状況、現在の出力状態やNULLカウントフラグなどをプログラムで検出することができます。2つ目の方法と同様に、カウント値がロックされた後、CPUがカウンタ動作の読み出しを行った後、OLの内容は自動的に再びカウントユニットCEに追従します。

3. カウンタ動作モード

8253/8254の3つのカウンタチャンネルが独立して動作します。6種類の方法があります。

(1) モード0 - 端子カウントで割り込み

このモードが設定されると、出力端子OUTはLOWとなり、カウントが0にデクリメントされるまでLOWのままとなります。この時、OUTはHighになり、新しいカウント値が書き込まれるか、コントロールワードがモード0にリセットされるまでHighのままです。この方法は、通常、イベントカウントに使用されます。このモードの特徴は、GATE端子を使用してカウントの一時停止を制御すること、カウント終了時に割り込み信号として出力がHighになること、カウント中に初期のカウント値をリロードして、カウントHighバイトを受信した後に再実行することができることです。

(2) モード1: ハードウェアトリガー可能なワンショット

このモードで動作するとき、OUTは初期状態ではハイレベルになっています。CPUが制御ワードと初期カウント値を書き込んだ後、カウンタの準備が整います。この時点で、GATE端子の立ち上がりエッジにより、カウンタの動作が開始され、OUTがLowになります。カウント終了(0)まで、OUTはHighになります。カウント期間中またはカウント終了後、GATEは再びHighとなり、カウンタが初期カウント値をロードするトリガーとなり、カウント動作を再開します。この動作モードでは、GATE信号は動作しません。

(3) モード2: レートジェネレータ

このモードの機能は、N分周器に似ています。通常は、リアルタイムクロックの割り込みを生成するために使用します。初期状態では、OUTはHighです。カウント値が1にデクリメントされると、OUTはLOWになり、その後HIGHになります。その間隔はCLKの1パルス幅です。この時点でカウンタは初期値を再読み込み、上記の処理を繰り返します。したがって、初期値をNとした場合、Nクロックごとにローレベルのパルス信号が出力されます。このようにして、GATEはカウントの一時停止と継続を制御することができます。GATEがHighになると、カウンタは初期値で再ロードされ、再カウントを開始します。

(4) モード3 - 矩形波モード

この方法は、通常、ボーレート・ジェネレーターに使用されます。このモードはモード2と似ていますが、OUTの出力が矩形波になります。初期カウント値をNとすると、矩形波の周波数は入力クロックCLKの1/Nとなります。このモードの特徴は、矩形波のデューティサイクルが約1対1（Nが奇数の場合は若干異なる）であることと、カウンタのデクリメント中に新しい初期値をリセットした場合、前のカウントが終了してから新しい初期値が有効になることです。

(5) モード4 - ソフトウェア・トリガ・ストローブ

初期状態ではOUTはハイレベルです。カウントが終了すると、OUTはクロックパルス幅のローレベルを出力した後、ハイレベルになります（ローレベルゲート）。カウント動作は、カウント初期値を書き込むことで「トリガ」されます。この動作モードでは、GATE端子はカウントの一時停止（1許可カウント）を制御できますが、OUTの状態には影響しません。カウント動作中に新しい初期値が書き込まれた場合、カウンタは1クロックパルス後に新しい値を使って再カウントします。

(6) モード5 - ハードウェアトリガストローブ

OUTは、初期状態ではHighです。カウント動作は、GATE端子の立ち上がりエッジで開始されます。カウントが終了すると、OUTはクロックCLKのパルス幅のローレベルを出力した後、ハイレベルになります。コントロールワードと初期値を書き込んだ後、カウンタは直ちにカウント初期値をロードして動作を開始するわけではありません。GATE端子がHighになってからCLKクロックパルスを

経て初めて動作を開始します。

PC/ATおよびその互換マイクロコンピュータシステムには、8254チップを使用しています。クロックタイミングの割り込み信号、ダイナミックメモリのDRAMリフレッシュタイミング回路、ホストスピーカの音色合成に、3つのタイマ/カウンタチャネルを使用しています。3つのカウンタの入力クロック周波数はいずれも1.193180MHzです。PC/ATマイクロコンピュータに搭載された8254チップの接続図を図8-9に示す。A1、A0端子はシステムアドレスラインA1、A0に接続されており、システムアドレスラインA9--A2信号が0b0010000のときに8254チップが選択されます。したがって、8254チップのI/Oポートの範囲は0x40--0x43となります。その中でも

0x40--0x42はセレクトカウンタチャンネル0--2に対応し、0x43はコントロールワードレジスタの書き込みポートに対応します。

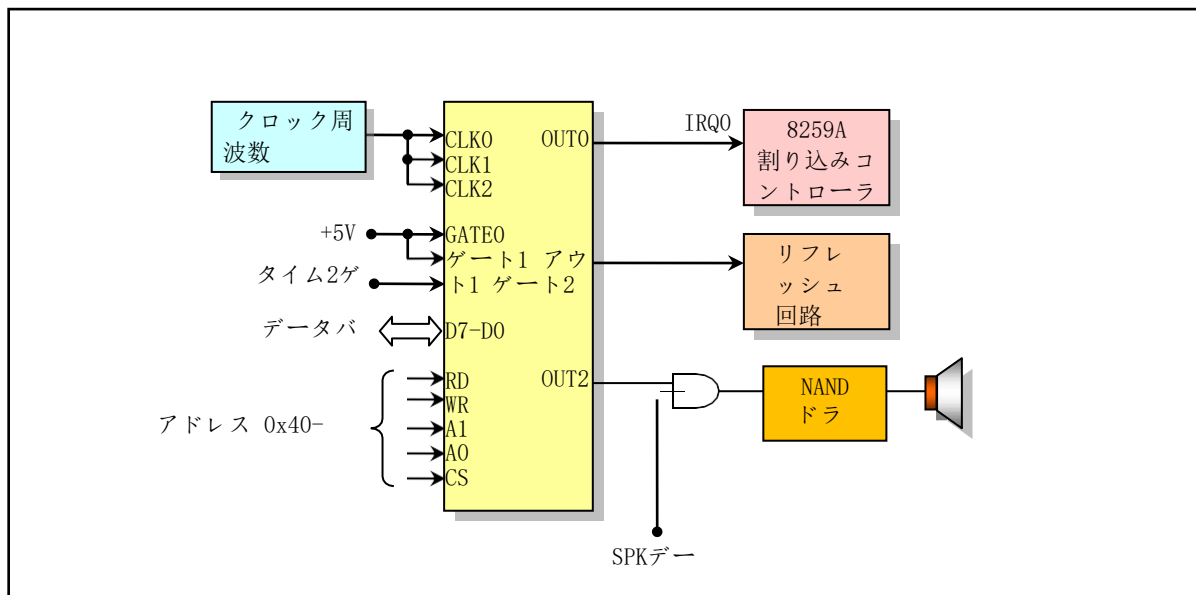


図8-9 PC内のタイマ/カウンタチップ接続図

カウンタチャンネル0は、そのGATE端子がHighに固定されています。電源投入時には、モード3（矩形波生成モード）で動作するように設定されており、初期設定ではカウント値が0に設定されているため、カウント値は65536（0--65535）となります。したがって、OUT0端子からは、1秒間に18.2HZ（ $1.193180\text{MHz}/65536$ ）の周波数の矩形波信号が出力されることになる。OUT0は、プログラマブルインタラプトコントローラ8259チップのレベル0インタラプトリクエストに接続されている。したがって、矩形波の立ち上がりエッジを利用して割り込み要求を出すことができ、54.9ms（ $1000\text{ms}/18.2$ ）ごとに割り込み要求を出すことになります。

カウンタチャンネル1のGATE端子も直接ハイレベルに接続されているので、カウント許容範囲内の状態になっています。モード2（周波数発生器モード）で動作し、初期値は通常18に設定されています。このカウンタは、PC/XTシステムまたはPC/ATシステムのDMAコントローラチャンネル2のリフレッシュ回路にRAMリフレッシュ信号を送るために使用されます。信号は約15マイクロ秒ごとに出力され、出力周波数は $1.19318/18=66.288\text{KHz}$ となります。

カウンタチャンネル2のGATEピン（TIME2GATE）は、8255AチップポートBのD0ピンまたは同等のロジックに接続されています。図8-9のSPK DATAは、8255AチップポートB（0x61）のD1ピンまたは同等のロジックに接続されています。このカウンタチャンネルは、メインスピーカーを鳴らすために使用していますが、8255Aチップ（または同等品）を使用して、通常のタイマーとして使用することもできます。

Linux 0.12では、カーネルは8254のカウンタチャンネル0を再初期化するだけで、カウンタはモード3で動作し、初期カウント値はバイナリ、初期カウント値はLATCH（ $1193180/100$ ）に設定されます。つまり、カウンタ0は、10ミリ秒ごとに矩形波の立ち上がり信号を送り、割り込み要求信号（IRQ0）を発生させます。このため、8254に書き込むコントロールワードは0x36（0b00110110）とし、次に初

期カウント値の下位バイトと上位バイトを書き込みます。初期カウント値の下位バイトと上位バイトの値は、それぞれ(LATCH & 0xff)と(LATCH >> 8)です。このインターバルタイミグで発生する割り込み要求は、Linux 0.12カーネルが動作する際のパルスとなります。現在実行中のタスクを定期的に切り替えたり、各タスクが使用しているシステムリソース（時間）をカウントしたり、カーネルのタイミグ操作を実現するために使用されます。

8.7 signal.c

8.7.1 機能説明

signal.cプログラムは、カーネル内のシグナル処理に関するすべての機能を含んでいます。UNIX系のシステムでは、シグナルは「ソフトウェア割込み」の処理機構である。シグナルを使用するもっと複雑なプログラムもたくさんあります。シグナル機構は、非同期イベントを処理する方法を提供します。これは、プロセス間の通信のための単純なメッセージ機構として利用でき、あるプロセスが別のプロセスにシグナルを送信することができます。シグナルは通常、正の整数で、自身のシグナルクラスを示す以外の情報は持ちません。例えば、子プロセスが終了または終了すると、SIGCHILDシグナルが親プロセスに送信され、子プロセスの現在の状態が通知されます。システム関数のkill()を使用すると、同じグループのすべての子プロセスに終了実行シグナルが送信されます。ctrl-CをタイプするとSIGINTシグナルが生成され、フォアグラウンドプロセスに送信されて終了します。また、プロセスが設定したアラームタイマーが切れると、システムはプロセスにSIGALRM信号を送ります。ハードウェア例外が発生すると、システムも実行中のプロセスに対応する信号を送ります。

信号処理の仕組みはごく初期のUNIXシステムにも存在していたが、初期のUNIXカーネルの信号処理の方法は信頼性が低かった。シグナルが失われることがあり、クリティカルなエリアコードを処理している最中に、プロセスが指定されたシグナルをクローズすることが困難な場合もありました。後のPOSIXでは、シグナルを確実に処理する方法が提供されています。互換性を保つために、Linuxカーネルは両方のシグナル処理方法を提供しています。

8.7.1.1 Linuxにおけるシグナル

Linuxカーネルのコードでは、符号なし長整数（32ビット）のビットは、通常、さまざまな異なる信号を表現するために使用されるため、システムには最大で32種類の信号が存在することになります。本バージョンのLinuxカーネルでは、22種類のシグナルが定義されています。このうち20種類はPOSIX.1規格で規定されているシグナルで、残りの2種類はLinux固有のシグナルです。SIGUNUSED(未定義)とSIGSTKFLT(スタックエラー)です。前者は、システムが現在サポートしていない他のすべてのシグナルタイプを表すことができます。これら22個のシグナルの具体的な名称と定義については、プログラム後のシグナルリストの表8-4を参照してください。また、ヘッダファイルinclude/signal.hの内容も参照してください。

プロセスがシグナルを受信した場合、処理や動作には3つの異なる方法があります。1つはシグナルを無視する方法ですが、2つのシグナルは無視できません（SIGKILLとSIGSTOP）。2つ目の方法は、プロセスが独自のシグナルハンドラを定義して、シグナルを処理することです。3つ目は、システムのデフォルトの信号処理動作を行うことです。

1. このシグナルを無視します。ほとんどのシグナルは、プロセスによって無視することができます。しかし、無視できないシグナルが2つあります。SIGKILLとSIGSTOPです。その理由は、スーパーユーザが指定されたプロセスを終了または停止させる明確な方法を与えるためです。また、いくつかのハードウェア例外によって生成されたシグナルを無視すると（例えば、0で割るなど）、プロセスの動作や状態が不可知になることがあります。
2. シグナルをキャプチャする。キャプチャー操作を行うためには、まず、指定したシグナルが発生したときに、カスタムのシグナルハンドラを呼び出すようにカーネルに指示する必要があります。

このハンドラの中では、何でもできます。もちろん、何もしないで、シグナルを無視するのと同じ役割を果たすこともできます。カスタムシグナル処理関数の例としては、プログラム実行中に一時ファイルを作成する場合、**SIGTERM**（実行終了）シグナルを捕捉する関数を定義し、その関数内でクリーンアップを行うことができます。**SIGTERM**シグナルは、killコマンドが送るデフォルトのシグナルです。

3. デフォルトのアクションを実行します。プロセスはシグナルを処理せず、シグナルはシステムの対応するデフォルトのシグナルハンドラで処理されます。カーネルは、シグナルの種類ごとにデフォルトアクションを用意しています。通常、これらのデフォルトアクションは、プロセスの実行を終了します。の説明を参照してください。

ポストプログラムのシグナルリスト（表8-4）を参照してください。

8.7.1.2 信号処理の実装

signal.cプログラムは主に以下の内容を含んでいます。1) アクセスシグナルのブロックコードシステムコールsys_ssetmask()とsys_getmask()、2) シグナルハンドリングsyscall sys_signal() (伝統的なシグナルハンドリング関数)、3) 修正シグナルアクションのハンドラsyscall sys_sigaction() (信頼性の高いシグナルハンドラ)、4) そしてシステムコール割り込みハンドラでシグナルを処理する関数do_signal()です。また、送信シグナル関数send_sig()と通知親プロセス関数tell_father()は、別のソースファイル(exit.c)に含まれています。なお、コード中の名前のプレフィックス「sig」はシグナルの略です。

signal()とsigaction()の機能は類似しており、シグナルハンドラを変更するために使用することができます。しかし、signal()は、カーネルがシグナルを処理するための伝統的な方法であり、ある特別なタイミングでシグナルが失われることがあります。ユーザが自分自身のシグナルハンドラ（シグナルハンドラ）を使いたい場合、ユーザはsignal()またはsigaction()のシスコールを使って、まずタスクのデータ構造にsigaction[]構造体の配列項目を設定し、自分自身のためにシグナルハンドラといくつかの属性を構造体に入れる必要があります。カーネルは、システムコールやいくつかの割り込み手続きから戻ってきたときに、現在のプロセスがシグナルを受け取ったかどうかを検出します。ユーザが指定した特定のシグナルを受信した場合、カーネルはプロセスタスクのデータ構造のsigaction[]の構造項目に従って、ユーザ定義のシグナル処理サービスプログラムを実行します。

1. signal()関数

ヘッダファイルinclude/signal.hの62行目では、signal()関数のシグネチャが以下のように宣言されています。

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

このsignal()関数は2つの引数を取ります。1つは捕捉するシグナルsignrを指定するもので、もう1つは新しいシグナルハンドラポインタvoid

(*handler)(int)です。この新しいシグナルハンドラは、戻り値のない関数ポインタで、指定されたシグナルが発生したときにハンドラに渡される整数のパラメータです。

signal()関数のプロトタイプ宣言は複雑に見えますが、次のように型を定義すると、このようになります。

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

そこで、signal()関数のプロトタイプを以下のようなシンプルな形に書き換えます。

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

signal()関数は、シグナル署名者のために新しいシグナルハンドラをインストールします。シグナルハンドラは、ユーザが指定したシグナル処理関数であったり、カーネルが提供する特定の関数ポインタSIG_IGNやSIG_DFLであったりします。指定されたシグナルが到着すると、関連するシグナル処理ハンドラがSIG_IGNに設定されている場合、そのシグナルは無視されます。シグナル処理ハンドラがSIG_DFLであれば、そのシグナルのデフォルトの処理が行われます。そうでない場合、シグナルハンドラがユーザのシグナルハンドラに設定されていると、カーネルはまずシグナルハンドラをデフォ

ルトのハンドラにリセットし、実装に関連したシグナルのブロック処理を行った後、指定されたシグナルハンドラを起動します。

`signal()`関数は元のシグナルハンドラを返しますが、返されたハンドラも戻り値のない関数ポインタで、整数の引数を持ち、新しいハンドラが1回呼ばれた後、デフォルトの処理ハンドラ値 `SIG_DFL`に戻されます。

include/signal.hファイル（46行目から）では、デフォルトのハンドルSIG_DFLと無視のハンドルSIG_IGNが次のように定義されています。

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

これらはすべて、`signal()`関数の第2パラメータで要求されているように、それぞれ戻り値のない関数ポインタを表しています。ポインタ値はそれぞれ0と1です。この2つのポインタ値は、論理的には実際のプログラムではありえない関数のアドレス値となります。したがって、`signal()`関数では、この2つの特殊なポインタ値をもとに、デフォルトの信号処理ハンドルを使用するか、信号の処理を無視するかを判断することができます。もちろん、SIGKILLやSIGSTOPは無視できない。関連するコードとして、以下のプログラムリストの155～162行目の処理文を参照してください。

プログラムが実行を開始すると、システムはすべてのシグナルをSIG_DFLまたはSIG_IGNとして処理する方法を設定します。また、プログラムが子プロセスを`fork()`した場合、子プロセスは親プロセスのシグナル処理モード(シグナルマスク)を継承します。そのため、親プロセスのシグナルの設定・処理方法は、子プロセスでも同様に有効です。

指定した信号を連続して取り込むために、ユーザープログラムで`signal()`関数を通常使用する例を以下に示します。

```
void sig_handler(int signr)
{
    signal(SIGINT,                sig_handler); // 次のキャプチャ    のためにハンドラを再インストールしま
    す。                                     ...// 何か    をする。
}

メイン ()
{
    signal(SIGINT,                sig_handler); // mainでシグナルハンドラを設定します。
    ...
}
```

`signal()`関数が信頼できない理由は、シグナルがすでに発生して独自の信号処理関数のセットに入っているとき、独自のハンドラを再設定するまでのこの間に、別のシグナルが発生する可能性があるからです。しかし、この時点でシステムはハンドルをデフォルト値に設定しています。そのため、信号切れを起こす可能性があります。

2. sigaction()関数

`sigaction()`関数は、`sigaction`データ構造を使用して、指定されたシグナルの情報を保持します。これは、カーネルがシグナルを処理するための信頼できるメカニズムです。これにより、指定されたシグナルの処理ハンドルを簡単に見たり変更したりすることができます。この関数は、`signal()`のスーパーセットです。include/signal.hヘッダーファイルでのこの関数の宣言（73行目）は次のとおりです。

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

パラメータ **sig** は、信号処理ハンドラを表示または変更するために必要な信号であり、最後の2つのパラメータは **sigaction** 構造体へのポインタである」と述べています。パラメータ **act** のポインタが **NULL** でない場合、その動作は

の場合、指定されたシグナルはact構造体の情報に応じて変更することができます。oldactが空でない場合、カーネルは構造体のシグナルの元の設定を返します。sigaction構造体は以下の通りである。

```

48 struct sigaction {
49     void (*sa_handler)(int); // シグナルハンドラーです。
50     sigset_t sa_mask; // シグナルマスク。
51     int sa_flags;
52     void (*sa_restorer)(void); // 内部
    のrestorerポインタ。53 };

```

信号処理方法を変更するとき、処理ハンドラsa_handlerがデフォルトのSIG_DFLまたは無視ハンドラSIG_IGNでない場合、sa_handlerが呼び出される前に、sa_maskフィールドは、プロセスの信号マスクビットマップに追加する必要がある1つの信号セットを指定します。シグナルハンドラーが戻ると、システムはプロセスの元のシグナルマスクビットマップを復元します。このようにして、シグナルハンドラーが呼ばれたときに、指定したシグナルの一部をブロックすることができます。シグナルハンドラーが呼ばれると、新しいシグナルマスクビットマップには、現在送信されているシグナルが自動的に含まれ、そのシグナルの継続的な送信がブロックされます。このように、指定された信号の処理中に、処理が完了するまで同じ信号が失われることなくブロックされるようにすることができます。また、信号がブロックされて複数回発生した場合、通常は1つのサンプルしか保存されません。つまり、ブロックが解除されると、ブロックされた同じ複数の信号に対して1回だけ信号処理ハンドラーが再度呼び出されます。シグナルの処理ハンドラを変更した後は、再び変更されるまでそのハンドラが使用されます。これは、従来のsignal()関数とは異なります。signal()関数は、ハンドラの処理が終了すると、そのシグナルのデフォルトのハンドラに戻します。

sigaction構造体のsa_flagsは、シグナル処理の他のオプションを指定するために使用されます。これらのオプションは、シグナル処理におけるデフォルトの処理の一部を変更するために使用されます。これらのオプションの定義については、include/signal.hファイルの記述（37～40行目）を参照してください。

```

// シグアクション構造体のsa_flagsフィールドが取ることのできるシンボル定数値。
37 #define SA_NOCLDSTOP 1 // 子機が停止した場合、SIGCHLDを無視する。
38 #define SA_INTERRUPT 0x20000000 // sigによる割り込みの後、syscallを再起動しない。
39 #define SA_NOMASK 0x40000000 // 現在の信号をマスクしてはいけません。
40 #define SA_ONESHOT 0x80000000 // 終了後にdefaultハンドラに戻す。

```

sigaction構造体の最後のフィールドと、sys_signal()関数のパラメータ・レストラーは、どちらも関数ポインタです。プログラムのコンパイルやリンクの際にlibcライブラリから提供され、シグナルハンドラの終了後にユーザーモードのスタックをクリーンアップしたり、以下に詳述するようにeaxに格納されたシステムコールの戻り値を復元したりします。

3. do_signal()関数

do_signal()関数は、カーネルのシステムコール(int 0x80)割り込みハンドラのシグナルの前処理関数です。プロセスがシステムコールを呼び出したり、タイマー割り込みが発生したりするたびに、そ

のプロセスがシグナルを受け取っていれば、この関数はシグナルの処理ハンドル（つまりシグナルハンドラ）をユーザプログラムのスタックに挿入します。このようにして、図8-10に示すように、現在のシステムコールが戻った直後にシグナルハンドラが実行され、その後、ユーザのプログラムが実行されます。

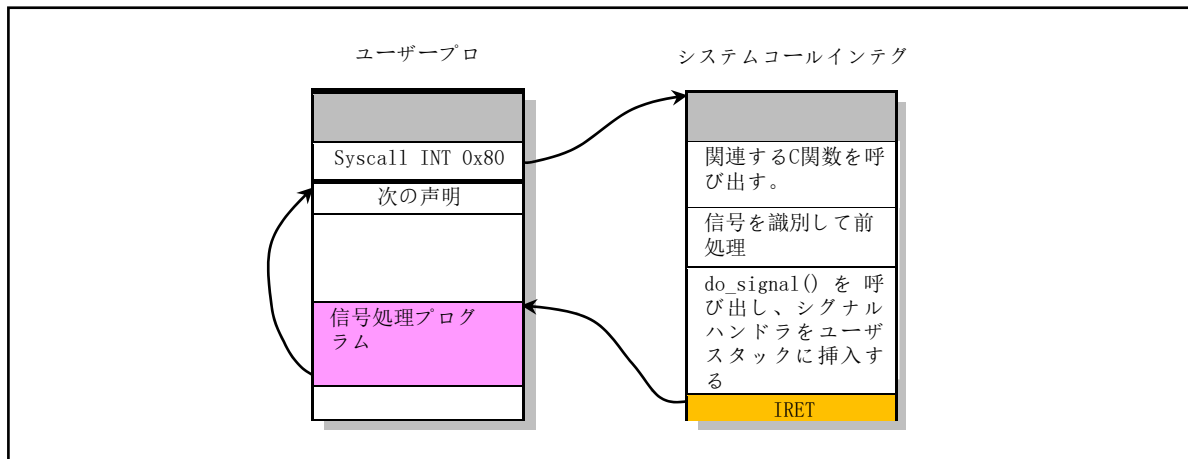
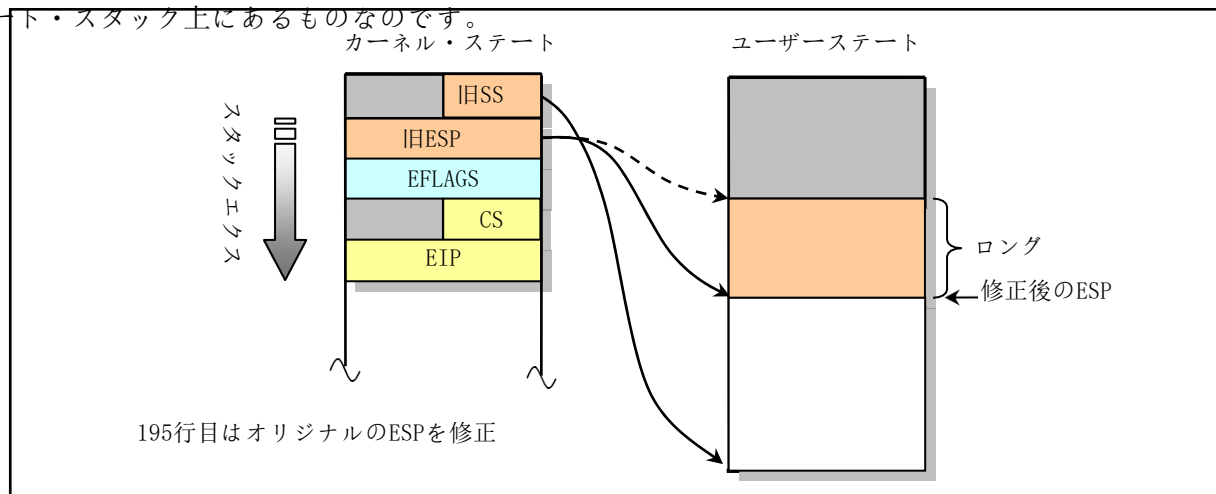


図8-10 シグナルハンドラーの呼び出され方

シグナル・ハンドラのパラメータをユーザ・スタックに挿入する前に、`do_signal()`関数はまずユーザ・プログラム・スタック・ポインタを長さ方向に展開し（以下のプログラムの195行目を参照）、次に関連するパラメータをそこに追加します。図8-11をご覧ください。`do_signal()`関数の193行目から始まるコードは理解しにくいので、以下に詳しく説明します。

ユーザープログラムが、カーネルに入ったばかりのシステムコールを呼び出すと、図8-11のように、プロセスのカーネル状態スタックが、CPUによって自動的にコンテンツに押し込まれます。すなわち、SS、ESPと、ユーザープログラムの次の命令のCS、EIPです。指定されたシステムコールを処理し、`do_signal()`を呼び出す準備をした後（つまり、ファイル`sys_call.s`の124行目以降）、カーネルの状態スタックの内容は、図8-12の左側ようになります。つまり、`do_signal()`の引数はカーネル・ステート・スタック上にあるものなのです。

図 8-11 `do_signal()`によるユーザ・スタックの変更

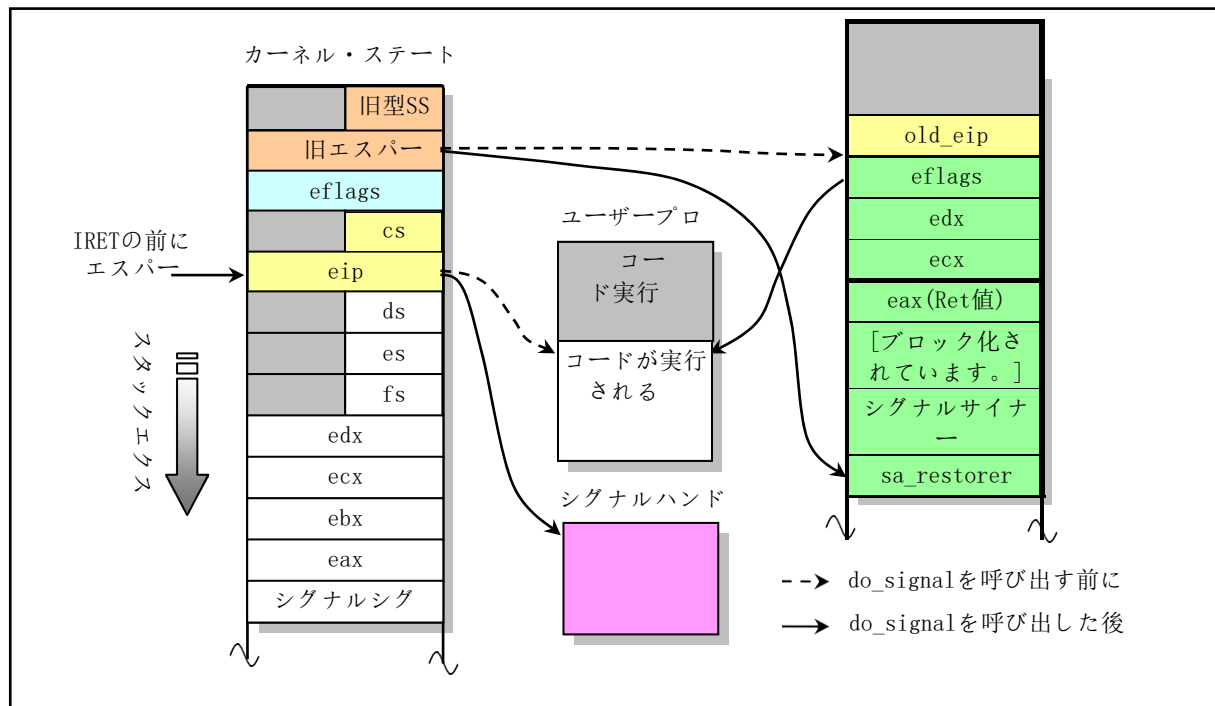


図8-12 ユーザー・ステート・スタックを変更する具体的なプロセス

`do_signal()`が2つのデフォルトのシグナルハンドラ(`SIG_IGN`と`SIG_DFL`)を決定して処理した後、ユーザがシグナルハンドラをカスタマイズした場合、193行目から`do_signal()`はユーザ定義のハンドラをユーザの状態スタックに挿入する準備を始めます。まず、カーネル状態スタックにある元のユーザプログラムのリターン実行点ポインタ`eip`をユーザスタックの`old_eip`として保存し、その`eip`をカスタムハンドラ`sa_handler`に置き換えます。つまり、図のカーネル状態スタックの`eip`は`sa_handler`を指しています。次に、カーネルステートに保存されている「オリジナル`esp`」から`long`の値を差し引くことで、ユーザーステートスタックを7~8個の`long`ワード分だけ下方に拡張します。最後に、図8-12の右側に示されているように、カーネルスタック上のレジスタコンテンツの一部がこのスペースにコピーされます。

カーネルコードは、合計7~8個の値をユーザーステートスタックに配置します。`old_eip`は、元のユーザプログラムのリターンアドレスで、`eip`がカーネルスタック上のシグナルハンドラのアドレスに置き換わる前に確保されます。`eflags`、`edx`、`ecx`は、システムコールが呼び出される前の元のユーザプログラムの値です。これらは基本的にシステムコールが使用するパラメータです。システムコールが戻ってきた後も、これらのユーザプログラムのレジスタ値を復元する必要があります。システムコールの戻り値は`eax`に格納されます。処理されたシグナルが自分自身の受信も許可していた場合、その処理のためにブロックされたコードもスタックに格納されます。次は、シグナル`signr`です。

最後の1つは、シグナル活動回復関数のポインタ`sa_restorer`です。ユーザが`signal()`関数を定義する際には、1つの信号値`signr`と1つの信号ハンドラのみが提供されるため、この回復関数はユーザーによって設定されることはありません。

以下は、`SIGINT`にカスタム・シグナル処理ハンドラを設定する簡単な例です。デフォルトでは、`Ctrl-C`キーの組み合わせを押すと`SIGINT`シグナルが生成されます。

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

```
void handler(int sig) // ユーザー定義のシグナルハンドラ。
```

```

{
    printf("The signal is %d\n", sig)と表示されます。
    (void) signal(SIGINT, SIG_DFL); // SIGINT のデフォルトハンドラを復元する
    } //一部のシステムは自動的に 復元されます

int main()
{
    (void) signal(SIGINT, handler); // SIGINTに対するユーザ定義の
    ハンドラを設定 while (1) {
        printf("Signal test.n").
        sleep(1); // 1秒 待つ。
    }
}

```

その中でも、シグナルハンドラ関数 () は、SIGINTという信号が現れたときに呼び出され、メインプログラムに戻って実行を継続します。この関数は、まずメッセージを出力し、SIGINTシグナルの処理をデフォルトのシグナルハンドラに設定します。そのため、Ctrl-Cキーの組み合わせを2回目に押したとき、SIG_DFLはプログラムの実行を終了させます。

4. sa_restorer機能

では、sa_restorerという関数はどこから来ているのでしょうか？ 実は、関数ライブラリから提供されています。この関数は、LinuxのLibc-2.2.2のライブラリファイル（misc/subdir）にあり、次のように定義されています。

```

.globl sig_restore
.globl masksig_restore
# ブロックがない場合はこの復元機能を使う
__sig_restoreです。
    addl    $4,%esp# 信号を破棄する signr
    popl    %eax# eax にシステムコールのRet値を復元する
    popl    %ecx# restore user original
    registers popl %edx
    popfl# restore user
    eflags. ret
# ブロックされている場合は、以下の復元機能を使用してください。ssetmaskで使用するためにブロックされています
__masksig_restoreです。
    addl    $4,%esp# ディスカード・シグナル・サインル
    call    ssetmask# set signal mask old blocking addl
    $4,%esp# discard blocked.
    popl    %eax
    popl    %ecx
    popl    %edx
    popfl ret

```

この関数の主な目的は、シグナル・ハンドラ終了後にユーザ・プログラムがsyscallを実行した後、戻り値や一部のレジスタの内容を復元し、シグナル値signrをクリアすることです。ユーザ定義のシグナル・ハンドラをコンパイルする際、コンパイラはlibcライブラリのsignal syscallを呼び出し、sa_restorer()関数をユーザ・プログラムに挿入します。ライブラリファイル内のsignal syscallの関数実装を以下に示します。

```

01 #define LIBRARY
02 #include <unistd.h>
03
04 extern void sig_restore();
05 extern void masksig_restore();
06
07 // ユーザーがライブラリで呼び出すラッパー関数signal()。
08 void (*signal(int sig, sighandler_t func))(int)
09 {
10     // void (*res)() です。
11     register int foebx asm ("bx") = sig;
12     asm ( "int $0x80" : "=a" (res)
13         : "0" (NR_signal), "r" (foebx), "c" (func), "d" ((long) sig_restore) );
14     return res;
15 }
16
17 // ユーザーが呼び出したsigaction()関数です。
18 int sigaction(int sig, struct sigaction * sa, struct sigaction * old)
19 {
20     register int foebx asm ("bx") = sig;
21     if (sa->sa_flags & SA_NOMASK)
22         sa->sa_restorer = sig_restore;
23     その他
24     sa->sa_restorer = masksig_restore;
25     asm ( "int $0x80" : "=a" (sig)
26         : "0" (NR_sigaction), "r" (foebx), "c" (sa), "d" (old) );
27     if (sig >= 0)
28         0を返す。
29     errno = -sig;
30     return -1;
31 }

```

`sa_restorer()`は、シグナルハンドラを実行した後のユーザプログラムのレジスタ値やシステムコールの戻り値を、シグナルハンドラを実行せずにシステムコールから直接戻ってきたかのようにクリーンアップする役割を担っています。

最後に、シグナルが処理されるまでの流れを説明します。`do_signal()`が実行された後、`sys_call.s`は、プロセスのカーネル状態スタック上の`eip`以下の値をすべてスタックにポップします。`IRET`命令を実行した後、CPUはカーネル状態スタック上の`cs:eip`、`eflags`、`ss:esp`をポップし、ユーザー状態に戻ってプログラムを実行します。しかし、今回は`eip`がシグナルハンドラへのポインタに置き換えられているため、ユーザ定義のシグナルハンドラが直ちに実行されます。シグナルハンドラが実行された後、CPUは`RET`命令によって`sa_restorer`が指す回復プログラムに制御を移す。`sa_restorer`プログラムは、ユーザレベルのスタッククリーンアップを行い、スタック上のシグナル値`signr`をスキップし、システムコール後の戻り値`eax`とレジスタ`ecx`、`edx`、フラグレジスタ`eflags`をポップする。システムコール後の各レジスタやCPUの状態が完全に復元されます。最後に、元のユーザプログラムの`eip`（つまりスタック上の`old_eip`）が`sa_restorer`の`RET`命令によってポップアップされ、ユーザプログラムの実行が復帰する。

8.7.1.3 プロセス停止

`signal.c`プログラムには、プロセスのシグナルマスクを引数で与えられたセットに一時的に置き

換え、シグナルが受信されるまでプロセスを一時停止する`sys_sigsuspend()`シスコールの実装も含まれています。このsyscallは、3つのパラメータを持つシグネチャとして宣言されています。

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

`restart`は、再起動の指標です。0であれば、現在のマスクを`oldmask`に保存し、シグナルを受け取るまでプロセスをブロックします。0でなければ、保存された`oldmask`から元のマスクを復元し、通常通り戻ります。

`syscall`には3つのパラメータがありますが、一般ユーザプログラムが使用する際には、ライブラリを経由して呼び出します。ライブラリーのこの関数は、パラメータが設定されたフォームのみを使用します。

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

これは、Cライブラリで`syscall`が使用される形式です。最初の2つのパラメータは、`sigsuspend()`ライブラリ関数によって処理されます。このライブラリ関数の一般的な実装は、以下のコードに似ています。

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

int sigsuspend(sigset_t *sigmask)
{
    int res;

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

ここでは、レジスタ変数`foebx`が上記の'`restart`'となっています。`syscall`が初めて呼ばれたときは0で、元のブロッキングコードが保存されており (`old_mask`)、'`restart`'は0以外の値に設定されています。そのため、プロセスが2回目にシスコールを呼び出す際には、プロセスが元々`old_mask`に保存していたブロッキングコードを復元します。

8.7.1.4 信号で中断されたシステムコールの再起動

プロセスが遅いシステムコールの実行中にブロックされた状態でシグナルを受信すると、システムコールはシグナルによって中断され、もはや継続しません。このとき、システムコールはエラーメッセージを返し、対応するグローバルエラーコード変数`errno`には、システムコールがシグナルによって中断されたことを示す`EINTR`が設定されます。例えば、パイプや端末機器、ネットワーク機器の

読み書きを行う場合、読み込んだデータが存在しなかったり、機器がすぐにデータを受け入れられなかったりすると、システムコールの呼び出しプログラムがずっとブロックされてしまう。そのため、一部の低速なシステムコールについては、シグナルを使用することで、必要に応じてそれらを中断し、ユーザープログラムに戻ることができます。これには `pause()` や `wait()` などのシステムコールも含まれます。

しかし、場合によっては、信号によって中断されたシステム・コールをユーザー・プログラムが個人的に処理する必要はありません。なぜなら、ユーザーがそのデバイスが低速デバイスであるかどうかを知らないことがあるからです。プログラムがインタラクティブに実行できれば、低速デバイスの読み書きができるかもしれません。このようなプログラムで信号が取り込まれ、システムがシステムコールの自動再起動機能を提供していない場合、プログラムはシステムコールの読み書きが行われるたびに、エラーの戻りコードを検出する必要があります。また、信号で中断された場合は、再度読み書きを行う必要があります。例えば、読み取り操作を行う際に、信号によって中断された場合、読み取り操作を継続するためには、次のようなコードを書くことが求められます。

再び:

```
if (( n = read(fd, buff, BUFFSIZE)) <
    0 ) { if (errno == EINTR)
        goto          again; /* 中断されたシステムコール */
    }
```

システムコールが中断された場合にユーザープログラムが対処しなくても済むように、信号処理中に中断された一部のシステムコールを再起動（再実行）する機能が導入されています。自動的にリスタートされるシステムコールには、`ioctl`、`read`、`write`、`wait`、`waitpid`がある。最初の3つのシステムコールは、低速デバイスで動作している場合にのみシグナルによって中断されますが、`wait`と`waitpid`はシグナルが捕捉されると常に中断されます。

シグナルを処理する際、`sigaction` 構造体に設定されているフラグに応じて、中断されたシステムコールを再開するかどうかを選択することができます。Linux 0.12 カーネルでは、`sigaction` 構造体に `SA_INTERRUPT` フラグが設定されており（システムコールを中断できる）、関連するシグナルが `SIGCONT`、`SIGSTOP`、`SIGTSTP`、`SIGTTIN`、`SIGTTOU` でない場合、シグナルを受信するとシステムコールが中断される。それ以外の場合、カーネルは中断されたシステムコールを自動的に再実行します。実行方法は、まずシステムコールが呼び出されたときの元のレジスタ `eax` の値を復元し、次にユーザープログラムの `eip` から2バイトを引き、つまり `eip` をシステムコールの `int 0x80` 命令にリダイレクトさせます。

現在の Linux システムでは、フラグ `SA_INTERRUPT` は破棄されています。代わりに、逆の意味を持つフラグ `SA_RESTART` が、シグナルハンドラの実行後に中断されたシステムコールを再開するために必要とされています。

8.7.2 コードコメント

プログラム 8-6 linux/kernel/signal.c

```
1 /*
2  *linux/kernel/signal.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や
//初期タスク0のデータと、組み込みのアセンブリ関数マクロ文
//ディスクリプタのパラメータ設定と取得 について。
```

```
// <linux/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 でよく使われる機能  
// <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。  
// 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。  
// <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
```

```

//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。
// (Linuxはminixから導入されました7
#include <linux/sched.h>.
8 #include <linux/kernel.h> (日本語)
9 #include <asm/segment.h>
10
11 #include <signal.h>
12 #include <errno.h>
13
// 現在のタスクのシグナルマスクのビットマップ（マスク、ブロックコード）を取得します。省略形は
// 「sgetmask」。
// は、「signal-get-mask」に分解できます。
14 int sys_getmask()
15 {
16     return current->blocked;
17 }。
18
// 新しいシグナルマスクのビットマップを設定します。シグナルSIGKILLとSIGSTOPはマスクできません。
// 戻り値は、元のシグナルマスクのビットマップです。
19 int sys_ssetmask(int newmask)
20 {。
21     int old=current->blocked;
22
23     current->blocked = newmask & ~(1<<(SIGKILL-1)) & ~(1<<(SIGSTOP-1));
24     return old;
25 }。
26
// 受信したがブロックされている信号を検出して取得する。まだ処理されていないビットマップ
// 信号はセットに入れておきます。
27 int sys_sigpending(sigset_t *set)
28 {。
29     /* "set"には、保留されているがブロックされているシグナルを記入します。*/
// まず、提供されるユーザーストレージスペースが4バイトであることを確認します。のビットマップは
// 処理されずにブロックされた信号は、次に位置に埋められます。
// セットポインタで示されます。
30     verify_area(set, 4)です。
31     put_fs_long(current->blocked & current->signal, (unsigned long *)set);
32     return 0;
33 }。
34
35 /* アトミックに新しいシグナルマスクをスワップして、シグナルを待ちます。
36 *
37 * syscallの再起動でいくつかのゲームをする必要があります。
私たちは助けを得る
38 * syscallライブラリのインターフェースから。を調整する 必要があることに注意
してください。
39 * libcのルーチンで呼び出し規則を設定します。
40 *
41 * "set"は、1003.1-1988, 3.3.7に記載されている通り、単なるsigmaskです。
42 * sigset_tが32ビットの量として渡されることを前提としています。
43 *
44 * "restart" は再起動の指示を保持します。それが0でない場合は
45 * 古いマスクをインストールして、通常通り戻ります。 ゼロの場合

```

合は

[46](#) *現在のマスク をold_maskに入れて、信号が来るまでブロックします。

[47](#) */

```

// syscallは、一時的にシグナルマスクをパラメータで指定されたセットに置き換えます。
// して、シグナルを受信するまで処理を中断します。
// リスタートは、中断されたシステムコールのリスタート表示です。システムコールが起動されると
// 初めての場合は0となり、元のブロッキングコードが保存されている (old_mask) と
// restartには0以外の値が設定されています。したがって、プロセスがシスコールの
// 2回目には、プロセスがもともと持っていたブロッキングコードを復元します。
// old_maskに保存されます。
// pause() シスコールは、それを呼び出したプロセスが次のようになるまでスリープ状態にします。
// がシグナルを受け取ります。このシグナルは、プロセスの実行を終了させるか
// 対応するシグナルキャプチャ関数をプロセスに実行させます。 48 int
50 sys sigsuspend(int restart, unsigned long old_mask, unsigned long set) 49 {
50 extern int sys pause(void); 51
// restartフラグが0でない場合は、プログラムの再実行を意味します。元の
// old_maskに保存されていたプロセスブロッキングコードが復元され、コード -EINTR
// が返されます (システムコールはシグナルによって中断されます)。
52     if (restart) {
53         /* we're restarting */
54         current->blocked = old_mask;
55         リターン -EINTR;
56     }
// そうでない場合、リスタートフラグは0です。そのため、まず最初に
// リスタートフラグ (1に設定)、現在のブロッキングコードをold_maskに保存して
// プロセスのブロッキングコードをセットします。その後、sys\_pause\(\) を呼んでプロセスをスリープさせて
// シグナルが到着するのを待ちます。プロセスがシグナルを受け取ると、pause\(\) が戻ります。
// そして、プロセスがシグナル・ハンドラを実行すると、呼び出しは -ERESTARTNOINTR を返します。
// コードを入力して終了します。このリターンコードは、シグナルが処理された後、そのシグナルが
// 走り続けるためには、システムコールに戻る必要があります。
57     /* 我々は再起動しない。*/
58     *(&restart) = 1となります。
59     *(&old_mask) = current->blocked;
60     current->blocked = set;
61     (void) sys pause (); /* シグナルが来たら戻る */ 。
62     return -ERESTARTNOINTR; /* シグナルを処理して、戻ってくる */ 。
63 }
64
// fsデータセグメントにシグアクションデータをコピーする。つまり、カーネル空間からのコピー
// ユーザー (タスク) のデータセグメントに
// まず variable to の mem space が十分な大きさであることを確認します。そして、sigactionをコピーします
// 構造体データをfsセグメント(ユーザ)空間に入れる。マクロ put_fs_byte()を実装する。
// include/asm/segment.hにあります。
65 static inline void save\_old(char * from, char * to)
66 {。
67     int i;
68
69     verify\_area(to, sizeof(struct sigaction));
70     for (i=0 ; i< sizeof(struct sigaction) ; i++) {。
71         put\_fs\_byte(*from, to)です。
72         from++;
73         to++;
74     }
75 }

```

```

76 // fsのデータセグメント'from'から'to'にsigactionデータをコピーする。つまり、「から」にコピーします。
// ユーザーデータスペースからカーネルデータセグメントへ。
77 static inline void get\_new(char * from, char * to)
78 {。
79     int i;
80
81     for (i=0 ; i< sizeof(struct sigaction) ; i++)
82         *(to++) = get\_fs\_byte(from++);
83 }。
84
// sigaction()に似たsignal()シスコールです。の新しいシグナルハンドラをインストールします。
// 指定されたシグナルを処理します。シグナルハンドラは、ユーザが指定した関数であったり、あるいは
// SIG_DFL（デフォルトのハンドラ）またはSIG_IGN（無視）。
// パラメータ: signum - 指定されたシグナル、handler - 指定されたシグナルハンドラ。
// restorer - 復元関数のポインタ。この関数は、libc ライブラリによって提供されます。
// いくつかのレジスタの元の値を復元するために使用され、その戻り値は
// の場合と同様に、シグナルハンドラの終了後にシスコンが戻ってきたときには、 //シスコンは
// syscallはシグナルハンドラを実行せず、ユーザプログラムに直接戻ります。
// この関数は、元のシグナルハンドラを返します。
85 int sys\_signal(int signum, long handler, long
restorer) 86 {。
87     struct sigaction tmp;
88
// まず、信号が有効範囲（1〜32）内にあることを確認し、信号であってはならない
// SIGKILL（とSIGSTOP）です。この2つのシグナルはプロセスが捕捉できないからです。
// そして、与えられたパラメータに従って、シグアクション構造のコンテンツを構築します。
// sa_handler は指定されたシグナルハンドラです。 sa_mask はシグナルマスクです。 sa_flags は
// 実行時のフラグの組み合わせ。ここでは、シグナルハンドラをデフォルトの
// 一度だけ使用した後は、そのシグナルを独自のハンドラで受信することができます。
89     if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
90         return -EINVAL;
91     tmp.sa_handler = (void (*)(int)) handler;
92     tmp.sa_mask = 0;
93     tmp.sa_flags = SA\_ONESHOT | SA\_NOMASK;
94     tmp.sa_restorer = (void (*)(void)) restorer;
// その後、元のシグナルハンドラを取り出し、sigaction構造体を設定します。最後に返す
// 元のシグナルハンドラ。
95     handler = (long) current-> sigaction[signum-1].sa_handler;
96     current-> sigaction[signum-1] = tmp;
97     return handler;
98 }。
99
// Sigaction() システムコール。シグナルを受け取ったときにプロセスの動作を変更する。
// SignumはSIGKILL以外のシグナルです。[新しいアクションが空でない場合]は、新しい
// の操作がインストールされます。oldaction ポインタが空でなければ、元の操作の
// はoldactionに保持されます。成功した場合は0を、そうでない場合は-EINVALを返します。
100 int sys\_sigaction(int signum, const struct sigaction * action,
101 struct sigaction * oldaction)
102 {
103     struct sigaction tmp;
104
// まず、信号が有効範囲（1〜32）内にあることを確認し、信号であってはならない

```

```

// SIGKILL (とSIGSTOP) です。この2つのシグナルはプロセスが捕捉できないからです。
// そして、シグナルのsigaction構造体に新しいアクションを設定します。もし古いアクションが
// ポインタが空でない場合は、元の操作ポインタを、指定した場所に保存します。
// オールドアクションで // になります。
105     if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
106         return -EINVAL;
107     tmp = current-> sigaction[signum-1];
108     get_new((char *) action,
109             (char *) (signum-1+current-> sigaction))となります。
110     if (oldaction)
111         save_old((char *) &tmp, (char *) oldaction);
// シグナルを独自のシグナルハンドラで受信できるようにした場合、マスクは0になります。
// そうでなければ、この信号をマスクするためのマスクが設定されます。
112     if (current-> sigaction[signum-1].sa_flags & SA_NOMASK)
113         current-> sigaction[signum-1].sa_mask = 0;
114     その他
115         current-> sigaction[signum-1].sa_mask |= (1<<(signum-1));
116     return 0;
117 }。
118
119 /*
120  * ルーチンは、カレントディレクトリにコアダンプイメージを書き込みます。
121  * 現在は実装されていません。
122  */
123 int core_dump(long signr)
124 {。
125     return(0          ); /* ダンプをしなかった */ (注)
126 }
127
// システムコール割り込みハンドラの実信号前処理コード。
// このコードの主な目的は、信号処理ハンドラをユーザーの
// システムコールが戻ってきたら、すぐにシグナルハンドラを実行してください。
// にして、ユーザープログラムの実行を継続します。この関数のパラメータには
// システムコールハンドラに入ってからスタックに段階的にプッシュされるすべての値は
// この関数を呼び出す前の場所です (sys_call.s, 125行目)。これらの値には
// (sys_call.sの行番号)となります。
// (1) ユーザースタックアドレスssとesp、eflags、リターンアドレスcsとeip
//     割り込み命令      によってカーネル状態のスタックにプッシュされたものです。
// (2) セグメントレジスタds, es, fsおよびレジスタeax(orig_eax), edxの値。
// 85-91 行目でsystem_callを入力した直後に、 //ecx, ebxがスタックにプッシュされました。
// (3) 100行目でsys_call_tableが呼び出された後、その戻り値(eax)は
//     //system-callがプッシュされます。
// (4) 124行目で、現在処理されている信号 (signr) がスタックにプッシュされます。
128 int do_signal(long signr, long eax, long ebx, long ecx, long edx, long orig_eax),
129     long fs, long es, long ds,
130     long eip, long cs, long eflags,
131     unsigned long * esp, long ss)
132 {。
133     unsigned long sa_handler;
134     long old_eip=eip;
135     struct sigaction * sa = current-> sigaction + signr - 1;
136     int                longs; // current->
>sigaction[signr-1] 137

```



```

138     unsigned long * tmp_esp;
139
140     // デバッグ文です。notdefが定義されていると、関連する情報が表示されます。
141     #ifdef notdef
142         printk("pid: %d, signr: %x, eax=%d, oeax = %d, int=%d\n").
143         current->pid, signr, eax, orig_eax,
144         sa->sa_flags & SA_INTERRUPT) となります。
145     #endif
146
147     // orig_eaxの値は、システムコールのイントレプトではなく、以下の時間帯に呼び出された場合は
148     // -1になります。
149     // 他の割り込みサービス (sys_call.sの144行目参照)。そのため、orig_eaxが-1になっていないと
150     // きは
151     // システムコールの処理でこの関数が呼び出されたことを意味します。waitpid()では
152     // kernel/exit.c の関数は、SIGCHLD シグナルを受信した場合、または、データを
153     // パイプラインではあるが、データは読み込まれず、プロセスが何らかのノンブロッキングシグナルを受信すると
154     // には、戻り値として -ERESTARTSYS が返されます。これは、そのプロセスが
155     // が中断されても、実行を続けた後にシステムコールが再開されます。コードを返す
156     // -ERESTARTNOINTR は、信号を処理した後、再び
157     // システムコールを継続して実行する、つまりシステムコールが中断されないようにします。
158     // したがって、以下の記述は、この関数がシステムコールで呼び出された場合に
159     // で、システムコールのリターンコードeaxが-ERESTARTSYSまたは-ERESTARTNOINTRに等しい場合で
160     // す。
161     // 以下の処理を行います（実際にはユーザープログラムに戻っていません）。
162     if ((orig_eax != -1) &&
163         ((eax == -ERESTARTSYS) || (eax == -ERESTARTNOINTR))) {
164
165         // システムコールのリターンコードが「-ERESTARTSYS」で、シグアクションにフラグが含まれている場合
166         // SA_INTERRUPTまたはシグナルがSIGCONTより小さいかSIGTTOUより大きい（つまりシグナルが
167         // がSIGCONT、SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOUのいずれでもない場合）には、修正された戻り値が
168         // システムコールは eax = -EINTR で、これはシグナルによって中断されたシステムコールです。
169         if ((eax == -ERESTARTSYS) && ((sa->sa_flags & SA_INTERRUPT) ||)
170             signr < SIGCONT || signr > SIGTTOU))
171             *(&eax) = -EINTR;
172         else {
173             そうでない場合は、eaxがシステムコールが呼ばれる前の値に復元されて
174             // 元のプログラム命令ポインタは2バイト減算されます。つまり、返すときに
175             // をユーザープログラムに送信し、プログラムを再起動させて先ほどのシステムコールを実行させます。
176             // 信号によって中断されます。
177             *(&eax) = orig_eaxです。
178             *(&eip) = old_eip - 2;
179         }
180     }
181
182     // シグナルハンドルがSIG_IGN（1、ハンドルはデフォルトでは無視される）の場合、シグナルは
183     // 処理されずにそのまま返されます。
184     sa_handler = (unsigned long) sa->sa_handler;
185     if (sa_handler==1)
186         return(1); /* 無視して、他にも信号があるかどうか確認する。*/
187     // ハンドルがSIG_DFL（0、デフォルト処理）の場合、デフォルトの方法で
188     // 信号に応じた処理を行います。
189     if (!sa_handler) {
190         スイッチ (signr) {
191             // シグナルが次の2つであった場合も無視して返します。
192             ケース SIGCONTです。
193             ケース SIGCHLDです。
194             return(1); /*

```

Ignore, .../*

[163](#)

```

// 信号が以下の4つの信号のいずれかである場合、現在のプロセスの状態を
// を停止状態TASK_STOPPEDに設定します。もし、現在のプロセスの親プロセスが「TASK_STOPPED」
// を設定していなければ
// SIGCHLDシグナルのシグアクションフラグSA_NOCLDSTOPを設定した後、SIGCHLDシグナルを送信す
// る。
// を親プロセスに送信します。SA_NOCLDSTOP フラグは、SIGCHLD シグナルが
// 子プロセスが実行を停止したとき、または実行を継続したときに生成されます。
164         case SIGSTOP:
165         case SIGTSTP:
166         case SIGTTIN:
167         case SIGTTOU:
168             current->state = TASK_STOPPED;
169             current->exit_code = signr;
170             if (!(current->p_pptr->sigaction[SIGCHLD-1].sa_flags &
171                 SA_NOCLDSTOP))
172                 current->p_pptr->signal |= (1<<(SIGCHLD-1));
173             return(1); /* 別のイベントを再設定する */ (1)
174
// 信号が以下の6つの信号のいずれかである場合、その信号がコアダンプを生成する場合。
// do_exit()を呼び出し、終了コードをsignr|0x80として終了します。それ以外の場合の終了コード
// は
// シグナルの値を返します。do_exit()のパラメータは、リターンコードと終了ステータスです。
// プログラムが提供する情報です。のステータス情報として利用することができます。
// wait()またはwaitpid()関数です。sys/wait.hファイルの13-19行目を参照してください。 wait()
// または
// waitpid()は、これらのマクロを使って、子プロセスの終了ステータスコードや
// 子プロセスが終了した理由（シグナル）を示します。
175         case SIGQUIT:
176         case SIGILL:
177         case SIGTRAP:
178         case SIGIOT:
179         case SIGFPE:
180         case SIGSEGV:
181             if (core_dump(signr))
182                 do_exit(signr|0x80)
183                 フォールスルー /*/*。
184                 のデフォルトです。
185                 do_exit(signr)です。
186             }
187     }
188     /*
189     * OK, we're invoking a handler
190     */
// シグナルハンドラーが1回しか呼ばれない場合は、ハンドラーは空白にしておきます。備考
// シグナルハンドラーが sa_handler ポインタにあらかじめ保存されていること。
// システムコールがカーネルコードに入ると、ユーザープログラムのリターンアドレス (eip、cs
// がカーネルの状態スタックに保存されます。以下のコードは、コードポインタeipを
// シグナルハンドラーを指すようにカーネルの状態スタックを変更し、sa_restorerもプッシュしま
// した。
// signr, プロセスマスク (SA_NOMASK が設定されていない場合), eax, ecx, edx をパラメータに
// して
// ユーザースタックに格納されます。また、元のプログラムのリターンポインタとeflagレジスタも
// をユーザースタックに格納します。したがって、シスコールがユーザープログラムに戻るときには、
// ユーザーの
// シグナルハンドラーが先に実行され、その後、ユーザープログラムが続けて実行されます。

```

```
191         if (sa->sa_flags & SA\_ONESHOT)  
192             sa->sa_handler = NULL。
```

// カーネルのステータスタック上のユーザの次のコード命令ポインタeipに指し示すようにする。
// シグナルハンドラーです。C言語の関数は値で渡されるので、形式としては

```

// eipに値を割り当てる際に “*(&eip)”と表示される。
// sigaction構造体のsa_maskフィールドは、次のような信号のセットを与えます。
// は、現在のシグナルハンドラの実行中にマスクされます。それと同時に、現在の
// シグナルもブロックされてしまいます。ただし、sa_flagsでSA_NOMASKフラグを使用している場合
// は
// 現在のシグナルはマスクされません。シグナルハンドラが、自分自身の
// シグナルを受信した際には、そのプロセスのシグナルブロックコードもスタックにプッシュする
// 必要があります。
193     *(&eip) = sa_handler;
194     longs = (sa->sa_flags & SA_NOMASK)?7:8;
// 元のユーザプログラムのユーザスタックポインタを7（または8）個のロングワードで拡張（使用
// がシグナルハンドラのパラメーターを格納している場合など）、メモリ使用量のチェック（も
// し
// メモリが境界を超えている場合、新しいページを割り当てるなど。）
195     *(&esp) -= longs;
196     verify_area(esp, longs*4);
// sa_restorer、signal signer、マスクコードblocked(SA_NOMASKが設定されている場合)、eax、
// ecxを格納する。
// edx, eflags, ユーザプログラムコードポインタをユーザスタックの下から上に向かって表示しま
// す。
197     tmp_esp=esp;
198     put_fs_long((long) sa->sa_restorer, tmp_esp++);
199     put_fs_long(signr, tmp_esp++);
200     if (!(sa->sa_flags & SA_NOMASK))
201         put_fs_long(current->blocked, tmp_esp++);
202     put_fs_long(eax, tmp_esp++);
203     put_fs_long(ecx, tmp_esp++);
204     put_fs_long(edx, tmp_esp++);
205     put_fs_long(eflags, tmp_esp++);
206     put_fs_long(old_eip, tmp_esp++);
207     current->blocked |= sa->sa_mask; // sa_maskのビットマップ で埋める
208     return(0); /* Continue, execute handler */
209 }
210

```

8.7.3 インフォメーション

8.7.3.1 信号の説明

プロセス内のシグナルは、プロセス間の通信に使用される単純なメッセージで、通常は1〜31のラベル値であり、他の情報は持ち合わせていません。Linux 0.12カーネルがサポートしているシグナルを表8-5に示します。

表8-5 プロセス信号

シグ	名前	説明	デフォルトの動作
1	SIGHUP	(Hangup) 制御端末がなくなったときや、Xtermの電源を切ったとき、モデムを切断したときに、カーネルがこの信号を生成します。バックグラウンドプログラムは制御端末を持たないので、しばしばSIGHUPを使って、設定ファイルを読み直す必要があることを知らせます。	(アボート) ハングアップ 制御の端末 やプロセス。

2	SIGINT	(Interrupt) キーボードからのインタラプト。通常、ターミナルドライバは を ^C にバインドします。	(アボート) プログラムを終了します。
3	SIGQUIT	(Quit) キーボードからの終了割り込みです。通常、ターミナルドライバはこれを^^にバインドします。	(Dump) 終了して、ダンプコアファイルが を生成しました。
4	SIGILL	(Illegal Instruction) プログラムにエラーや不正な操作があった場合	(Dump)

		コマンドが実行されました。	
5	SIGTRAP	(Breakpoint/Trace Trap) デバッグに使用され、ブレークポイントを追跡します。	
6	SIGABRT	(Abort) 実行を中止して異常終了します。	(Dump)
6	SIGIOT	(IO トラップ) SIGABRT と同じ	(Dump)
7	SIGUNUSED	(Unused) 使用されていません。	
8	SIGFPE	(Floating Point Exception) 浮動小数点の例外。	(Dump)
9	SIGKILL	(Kill) 番組を終了しました。この信号は、キャプチャや無視していました。プロセスを直ちに終了させたい場合は、シグナル9を送ります。プログラムがクリーンアップを行う機会がないことに注意してください。	(アボート)
10	SIGUSR1	(User defined Signal 1) ユーザー定義の信号1。	(アボート)
11	SIGSEGV	(Segmentation Violation) この信号は、プログラムが無効なメモリを参照したときに発生します。例えば、マップされていないメモリをアドレス指定した場合などです。 許諾されていないメモリにアクセスすること。	(Dump)
12	SIGUSR2	(ユーザー定義信号) 2) IPC などのユーザープログラム用に予約されている。 の目的を達成しました。	(アボート)
13	SIGPIPE	(Pipe) プログラムがソケットやパイプに書き込みを行う際に発生する信号です。 とリーダーがいません。	(アボート)
14	SIGALRM	(Alarm) この信号は、ユーザーが以下の方法で設定した遅延時間後に発生します。 のアラームsyscallです。この信号は、しばしばsyscallのタイムアウトを決定するために使用されます。	(アボート)
15	SIGTERM	(Terminate) プログラムの終了を親切に要求するために使用します。これは、killのデフォルトのシグナルです。SIGKILLとは異なり、このシグナルはキャプチャすることができるので 帰る前に掃除をする。	(アボート)
16	SIGSTKFLT	(Stack fault on coprocessor) コプロセッサのスタックエラーです。	(アボート)
17	SIGCHLD	(Child) 子プロセスが発行されました。子プロセスが停止しているかを終了しました。その意味を変えて別の用途に使うことができます。	(無視)Thechild プロセスが停止または終了します。
18	SIGCONT	(Continue) このシグナルは、SIGSTOPで停止したプロセスを再開させる の操作を行います。捕らえることができる。	(Continue)Resume プロセスを実行しています。
19	SIGSTOP	(Stop) プロセスの実行を停止します。この信号は、キャプチャや無視することはできません。	(Stop) プロセスの停止 を実行しています。
20	SIGTSTP	(Terminal Stop) 端末に停止キーシーケンスを送信します。この信号は捕らえたり、無視したり。	(ストップ)。

21	SIGTTIN	(バックグラウンドでのTTY入力) バックグラウンド・プロセスは、制御できなくなった端末からデータを読み取ろうとしますが、このとき、SIGCONT信号を受信するまでプロセスは停止します。この信号には 捕らえたり、無視したり。	(ストップ)。)
22	SIGTTOU	(バックグラウンドでのTTY出力) バックグラウンドのプロセスが、制御不能になった端末にデータを出力しようとする時、その時点でSIGCONT信号を受信するまでプロセスが停止されます。この信号には 捕らえたり、無視したり。	(ストップ)。)

8.8 exit.c

8.8.1 機能説明

exit.cプログラムは、主にプロセスの終了・退出に関する処理を実装しています。プロセスの解放、セッション(プロセスグループ)の終了、プログラムの終了ハンドラのほか、プロセスのキル、プロセスの終了、プロセスのサスペンドなどのシステムコールが含まれます。また、シグナル送信機能であるsend_sig()や、子プロセスの終了を親プロセスに通知する機能であるtell_father()も搭載されています。

リリースプロセス関数release()は、タスク配列中の指定されたタスクポインタを削除し、指定されたタスクポインタに応じて関連するメモリページを解放し、直ちにカーネルに実行中のタスクのリスケジュールを行わせます。プロセスグループ終了関数 kill_session()は、セッション番号が現在のプロセスIDと同じであるプロセスにシグナルを送るために使われます。システムコールのsys_kill()は、任意の指定されたシグナルをプロセスに送るために使われます。パラメータpidに応じて、システムコールは異なるプロセスやプロセスグループにシグナルを送ります。様々な状況での処理は、プログラムコメントに記載されています。

プログラム終了処理関数do_exit()は、exitシステムコールの割込みハンドラで呼び出されます。この関数はまず、現在のプロセスのコードおよびデータセグメントが占有しているメモリページを解放します。カレントプロセスに子プロセスがある場合は、子プロセスの father フィールドを 1 に設定し、つまり子プロセスの親をプロセス 1 (init プロセス) に変更します。子プロセスがすでにゾンビ状態になっている場合は、子プロセスの終了信号SIGCHLDをプロセス1に送信します。その後、現在のプロセスが開いているすべてのファイルを閉じ、使用中の端末デバイス、コプロセッサデバイスを解放します。現在のプロセスがプロセスグループの最初のプロセスである場合、関連するすべてのプロセスを終了させる必要があります。そして、カレントプロセスをゾンビ状態にし、終了コードを設定し、子プロセス終了信号SIGCHLDを親プロセスに送信します。最後に、カーネルが実行中のタスクを再スケジュールしましょう。

システムコールのwaitpid()は、pidで指定された子プロセスがexit(終了)するか、プロセスの終了を要求するシグナルを受け取るか、シグナルハンドラを呼び出す必要があるまで、現在のプロセスを一時停止するために使用されます。pidで指定された子プロセスがすでに終了している(いわゆるゾンビプロセスになっている)場合、このコールは直ちに返されます。子プロセスが使用していたリソースはすべて解放されます。この関数の具体的な動作は、パラメータに応じて異なる処理も行われます。詳細は、コードの関連コメントを参照してください。

8.8.2 コードコメント

```

5  */
6
7  #define DEBUG_PROC_TREE
8
1 /*
2  *linux/kernel/exit.c
3  *
4  *(C) 1991 Linus Torvalds
```

プログラム 8-7 linux/kernel/exit.c

```
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。  
// <signal.h> シグナルのヘッダファイルです。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。
```

```

// <sys/wait.h> Waitのヘッダファイル。システムコール wait() core waitpid() および関連する
// 恒常的なシンボル。
// <linux/sched.h> スケジューラーのヘッダファイルでは、タスク構造体task_struct、データ
// 初期のタスク0 の、そしていくつかの組み込みアセンブリ関数のマクロのステートメント。
// デスクリプタのパラメータ設定と取得
// <linux/kernel.h> カーネルのヘッダファイルです。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 の機能を利用します。
// <linux/tty.h> ttyヘッダファイルは、tty_io、シリアルのパラメータと定数を定義しています。
// コミュニケーション
// <asm/segment.h> セグメント操作のヘッダファイルです。埋め込みアセンブリ関数の定義
// セグメントレジスタの操作のため9
#include <errno.h>
10 #include <signal.h>
11 #include <sys/wait.h>
12
13 #include <linux/sched.h>
14 #include <linux/kernel.h> (日本語)
15 #include <linux/tty.h>
16 #include <asm/segment.h>
17
18 int sys_pause (void); // シグナルを受け取るまでスリープ状態にします
(kernel/sched.c, 164)。19 int sys_close(int fd); // ファイルを閉じる (fs/open.c, 219)。
20
// プロセスが占有していたタスクスロットと、そのタスク構造が占有していたメモリページを解放します。
// パラメータpは、タスクデータ構造へのポインタです。この関数はsys_kill()で呼び出されます。
// および sys_waitpid() に従います。
// プログラムは、タスクポインタ配列 task[] をスキャンして、指定されたタスクを見つけます。見つかった場合
// は
// タスクスロットがまず空になり、次にタスクデータ構造が占有するメモリページが
// をリリースしました。最後にスケジューラーを実行し、戻ってきたらすぐに終了します。もし、ア
// イテム
// 指定されたタスクに対応する // がタスク配列に見つからないと、カーネルパニックになります。
21 void release(struct task_struct * p)
22 {...
23     int i;
24
// 与えられたタスク構造体ポインタが NULL の場合には終了します。ポインタが現在のプロセスを指している場合
// 警告メッセージを表示して終了します。あなたがクリーチャーでなくても、ここでは自殺は許されません！
25     if (!p)
26         を返すことができます。
27     if (p == current) {
28         printk("task releasing itself\n");
29         を返すことができます。
30     }
// 以下のループ文は、タスク構造体のポインタの配列をスキャンして、指定された
// タスク p. 見つかった場合は、タスクポインタ配列の対応する項目を NULL に設定して
// タスク間の関連ポインタが更新され、タスクが占有しているメモリページの
// pデータ構造が解放されます。最後に、スケジューラーが戻ってきたら終了します。タスクpが見つ
// からない場合。
// カーネルのコードが間違っていて、エラーメッセージが表示されたり、カーネルがクラッシュした
// りすることがあります。さらには
// リンクを更新するコードは、タスクpを二重リンクリストから削除します。
31     for (i=1 ; i< NR_TASKS ; i++)

```

```
32         if (task[i]==p) {...  
33             task[i]=NULLとなり ます。  
34             /* リンクの更新 */  
// 次のコードはリンクリストを操作します。pが最後の（最も古い）子プロセスでない場合。
```

```

// 古い兄弟(neighbor)に若い兄弟を指させる。pが最新の子でない場合
// 処理では、新しい方の兄弟が古い方の兄弟を指すようにします。タスクpが最新の子である場合
// プロセスでは、その親の最新の子ポインタcptrを更新して、その古い兄弟を指すようにする必要があります。
// 図5-20を参照してください。
// osptr (old sibling pointer) は、p よりも前に作成された兄弟プロセスを指します。
// ysptr (若い兄弟のポインタ) は、pの後に作られた兄弟のプロセスを指します。
// pptr (親ポインタ)は、pの親プロセスを指します。
// cptr (child pointer) 親プロセスが、最後に作成された子プロセスを指す。
35         if (p->p_osptr)
36             p->p_osptr->p_ysptr = p->p_ysptrとなります。
37         if (p->p_ysptr)
38             p->p_ysptr->p_osptr = p->p_osptr;
39         その他
40             p->p_pptr->p_cptr = p->p_osptr;
41         free_page((long)p)です。
42         schedule()です。
43         を返すことができます。
44     }
45     panic("Trying to release non-existent task")が発生します。
46 }
47
48 #ifdef DEBUG_PROC_TREE
// シンボルDEBUG_PROC_TREEが定義されている場合、以下のコードがコンパイル時に含まれます。
49 /*
50  * task[] の配列に task_struct ポインタが存在するかどうかを確認します。
51  * 見つかった場合は0を、見つからなかった場合は1を返します。
52  */
53 int bad_task_ptr(struct task_struct *p)
54 {...
55     インティ;
56
57     if (!p)
58         0を返す。
59     for (i=0 ; i< NR_TASKS ; i++)
60         if (task[i] == p)
61             0を返す。
62     return 1;
63 }。
64
65 /*
66  * このルーチンは、pidツリーをスキャンして、rep不変であることを確認します。
67  * を保持しています。非常に遅い       ので、デバッグにのみ使用されます。....
68  *
69  * 実際よりもずっと怖く見える.... 我々はこれ以上何もしていない。
70  * p_ysptrとp_osptrにあるダブルリンクリストを検証するよりも。
71  * p_cptrで定義されたプロセスツリーと一致していることを確認したり
72  * p_pptr;
73  */
74 void audit_ptree()
75 {
76     inti; 77
// このループは、システム内のタスク0以外のすべてのタスクをスキャンして、その正しさを

```

```

// 4つのポインタ(pptr, cptr, ysptr,                                osptr)。タスク配列スロットが空の      場合はスキップ
// します。
78     for (i=1 ; i< NR_TASKS ; i++) {...
79         if (! task[i])
80             を続けています。
// タスクの親ポインタ p_pptr がどのプロセスも指していない場合（つまり
// がタスク配列に存在する場合）、警告メッセージが表示されます。以下のステートメントは
// cptr, ysptr, osptr にも同様の操作を行います。
81         if (bad_task_ptr(task[i]->p_pptr))
82             printk("Warning, pid %d's parent link is bad\n",
83                 task[i]->pid)となります。
84         if (bad_task_ptr(task[i]->p_cptr))
85             printk("Warning, pid %d's child link is bad\n",
86                 task[i]->pid)となります。
87         if (bad_task_ptr(task[i]->p_ysptr))
88             printk("Warning, pid %d's ys link is bad\n").
89                 task[i]->pid)となります。
90         if (bad_task_ptr(task[i]->p_osptr))
91             printk("Warning, pid %d's os link is bad\n",
92                 task[i]->pid)となります。
// タスクの親ポインタp_pptrが自分自身を指している場合は、警告メッセージが表示されます。
// 以下のステートメントは、cptr, ysptr, osptrに対して同様の操作を行います。
93         if (task[i]->p_pptr == task[i])
94             printk("Warning, pid %d parent link points to self\n")と表示されます。
95         if (task[i]->p_cptr == task[i])
96             printk("Warning, pid %d child link points to self\n")と表示されます。
97         if (task[i]->p_ysptr == task[i])
98             printk("Warning, pid %d ys link points to self\n")と表示されます。
99         if (task[i]->p_osptr == task[i])
100            printk("Warning, pid %d os link points to self\n")と表示されます。
// タスクに古い兄弟プロセス（自分よりも先に作成されたもの）がある場合、チェックします。
// 共通の親を持つ場合は、バディの ysptr ポインタがこのプロセスを指しているかどうかをチェックする
// 正しく表示されない場合は、警告メッセージが表示されます。
101        if (task[i]->p_osptr) {
102            if (task[i]->p_pptr != task[i]->p_osptr->p_pptr)
103                printk(
104                    "Warning, pid %d older sibling %d parent is %d\n" と表示されました。
105                    task[i]->pid, task[i]->p_osptr->pid,
106                    task[i]->p_osptr->p_pptr->pid)となります。
107            if (task[i]->p_osptr->p_ysptr != task[i])
108                printk(
109                    "Warning, pid %d older sibling %d has mismatched ys link\n" と表示されます。
110                    task[i]->pid, task[i]->p_osptr->pid) となります。)
111        }
// タスクに（自分より後に作成された）若い兄弟プロセスがある場合、チェックします。
// それらが共通の親を持つ場合、若い方のosptrポインタが正しく指しているかどうかをチェックします。
// このプロセスに // を追加しないと、警告メッセージが表示されます。
112        if (task[i]->p_ysptr) {
113            if (task[i]->p_pptr != task[i]->p_ysptr->p_pptr)
114                printk(
115                    "Warning, pid %d younger sibling %d parent is %d\n" となっています。
116                    task[i]->pid, task[i]->p_osptr->pid,
117                    task[i]->p_osptr->p_pptr->pid)となります。
118            if (task[i]->p_ysptr->p_osptr != task[i])

```

```

119                                     printk(
120                                     "Warning, pid %d younger sibling %d has mismatched os
link\n" (Warning, pid %d younger sibling %d has mismatched
os link\n)
121                                     task[i]->pid, task[i]->p_ysptr->pid) となりま
す。)
122                                     }
// タスクの最新の子ポインタcptrが空でない場合、子の親が以下になっているかどうかをチェックします。
// 処理を行い、子の若いポインタ ysptr が空であるかどうかをチェックします。そうでない場合は、警告
// のメッセージが表示されます。
123                                     if (task[i]->p_cptr) {...
124                                     if (task[i]->p_cptr->p_pptr != task[i])
125                                     printk(
126                                     "Warning, pid %d youngest child %d has mismatched parent
link\n" (Warning, pid %d youngest child %d has mismatched
parent link\n)
127                                     task[i]->pid, task[i]->p_cptr->pid) となります。)
128                                     if (task[i]->p_cptr->p_ysptr)
129                                     printk(
130                                     "Warning, pid %d youngest child %d has non-NULL ys link\n" と表
示されました。
131                                     task[i]->pid, task[i]->p_cptr->pid) となります。)
132                                     }
133                                     }
134 }
135 #endif /* DEBUG_PROC_TREE */。
136
// 特権privを持つタスクpにシグナルsigを送信します。
// sig - シグナル、p - タスクへのポインタ、priv - シグナルの送信を強制するフラグ。
// プロセスユーザーの属性やレベルに関係なく、信号を送信する権利。
// この関数は、まずパラメータの正しさをチェックし、次に
// の条件を満たします。条件が満たされた場合、プロセスにシグナルsigを送信し、終了します。
// そうでない場合は、ライセンス違反のエラーコードを返します。
137 static inline int send_sig(long sig, struct task_struct * p, int priv)
138 {。
// パーミッションがなく、現在のプロセスの実効ユーザーID (euid) が
// プロセスpとは異なる、スーパーユーザーではない場合、送信する権利はない。
// suser()は (current->euid==0)と定義され、スーパーユーザーかどうかをチェックするのに使われます。
139         if (!p)
140                 return -EINVAL;
141         もし (!priv && (current->euid!=p->euid) && ! suser())
142                 return -EPERM;
// 送信するシグナルがSIGKILLまたはSIGCONTの場合、プロセスpがシグナルを受信すると
// が停止している場合は、レディ状態 (TASK_RUNNING) に設定されます。その後、修正して
// シグナルビットマッププロセスpが、シグナルSIGSTOP、SIGTSTP、SIGTTIN、を削除 (リセット) す
// るために
// SIGTTOUでプロセスが停止してしまいます。
143         if ((sig == SIGKILL) || (sig == SIGCONT)) {...
144                 if (p->state == TASK_STOPPED)
145                         p->state = TASK_RUNNING;
146                 p->exit_code = 0;
147                 p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1)) |
148                         (1<<(sigttin-1)) | (1<<(sigttou-1)))のようになります。)
149         }
150         /* シグナルが無視されるなら、投稿もしないでください */。

```

```
151     if ((int) p-> sigaction[sig-1].sa_handler == 1)
152         0を返す。
153     /* 順序に依存 SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU */。
// シグナルがSIGSTOP、SIGTSTP、SIGTTIN、SIGTTOUのいずれかである場合には
// は、プロセスpの実行を停止します。したがって（pのシグナルビットマップにSIGCONTが設定され
// ている場合）、それは
```


ビットマップのSIGCONTビットをリセットするためには、 // が必要です。

```

154     if ((sig >= SIGSTOP) && (sig <= SIGTTOU))
155         p->signal &= ~(1<<(SIGCONT-1));
156     /* 実際に信号を送る */
157     p->signal |= (1<<(sig-1));
158     0を返す;
159 }。
160
161 // プロセスグループIDに基づいてセッションIDを取得する pgrp.
162 // コードは、タスク配列をスキャンし、グループID pgrpを持つプロセスを探し、そのセッションを返す
163 // idです。指定されたグループ pgrp に対応するプロセスが見つからない場合は、-1 が返されます。
164 int session_of_pgrp(int pgrp)
165 {。
166     struct task_struct **p;
167     for (p = & LAST_TASK; p > & FIRST_TASK; --p)
168         if ((*p)->pgrp == pgrp)
169             return ((*p)->session)となります。
170     return -1;
171 }。
172
173 // プロセスグループをキルする（グループにシグナルを送る）。
174 // パラメータ: grp - プロセスグループID、sig - シグナル、priv - 特権。
175 // つまり、シグナルsigはプロセスグループpgrpの各プロセスに送られます。限り、それは
176 // プロセスへの送信に成功した場合は0を返します。それ以外の場合は、プロセスが
177 // グループ pgrp の場合、エラーコード -ESRCH が返されます。グループが pgrp であるプロセスが
178 // 見つかったが、信号の送信に失敗した場合は、エラーコードが返される。
179 int kill_pg(int pgrp, int sig, int priv)
180 {。
181     struct task_struct **p;
182     int err,retval = -ESRCH ; // ESRCH - エラー検索。
183     int found = 0;
184
185     // まず、与えられたシグナルとプロセスグループが有効であるかどうかをチェックし、次に、すべてのタスクをス
186     // キャンします。
187     // システムになります。グループID pgrpを持つプロセスがスキャンされた場合、シグナルsigがそのプロセスに送
188     // られます。
189     // シグナルの送信が成功していれば、この関数は最後に0を返します。
190     もし (sig<1 || sig>32 || pgrp<=0)
191         return -EINVAL;
192     for (p = & LAST_TASK; p > & FIRST_TASK; --p)
193         if ((*p)->pgrp == pgrp) {...
194             if (sig && (err = send_sig(sig,*p,priv)))
195                 retval = err;
196             その他
197                 が見つかりました。
198         }
199     return(found ? 0 : retval);
200 }。
201
202 // プロセスをキルする（プロセスにシグナルを送る）。
203 // パラメータ: pid - プロセスID、sig - シグナル、priv - 特権。
204 // つまり、シグナルsigはpidのプロセスに送られます。もし、pidのプロセスが
205 // 見つかった場合、シグナルの送信に成功した場合は0を、そうでない場合はエラーコードを返します。

```

// pid を持つプロセスが見つからない場合は、エラーコード -ESRCH が返されます。

```

189 int kill_proc(int pid, int sig, int priv)
190 {
191     struct task_struct **p;
192
193     if (sig<1 || sig>32)
194         return -EINVAL;
195     for (p = & LAST_TASK; p > & FIRST_TASK; --
196         p)
197         if ((*p)->pid == pid)
198             return(sig ? send_sig(sig,*p,priv) : 0);
199     return(-ESRCH);
200 }。
201 /*
202  * POSIXでは、kill(-1, sig)は不定形となっていますが、私たちが持っているのは
203  * はおそらく間違っています。BSDやSYSV のようにすべきです。
204  */
205 // システムコールの kill() を使って、プロセスやプロセスグループに何らかのシグナルを送ることができます。
206 // パラメータ: pid - プロセスID、sig - 送信する必要があるシグナル。
207 // pid > 0の場合、プロセスIDがpidであるプロセスにシグナルが送られます。
208 // pid = 0の場合、シグナルは現在のプロセスのグループ内のすべてのプロセスに送信されます。
209 // pid = -1の場合は、最初の（初期）プロセスを除くすべてのプロセスにシグナルが送られます。
210 // pid < -1の場合は、グループ内のすべてのプロセスにシグナルが送られます -pid。
211 // sigが0の場合、シグナルは送られませんが、エラーチェックは行われます。成功した場合は 0 を返します。
212 //
213 // この関数は、タスク配列をスキャンして、以下の条件を満たすプロセスにシグナルsigを送信します。
214 // pidに応じた条件を設定します。pidが0の場合は、現在のプロセスが
215 // はグループリーダーなので、シグナルsigを全プロセスに強制的に送信する必要があります
216 グループ内では
217 int sys_kill(int pid, int sig)
218 {
219     struct task_struct **p = NR_TASKS + task ; // 最後のアイテム を指します。
220     int err, retval = 0;
221
222     if (!pid)
223         return(kill_pg(current->pid, sig, 0));
224     if (pid == -1) {
225         一方 (--p > & FIRST_TASK)
226             if (err = send_sig(sig,*p,0))
227                 retval = err;
228         return(retval)になります。
229     }
230     も (pid < 0)
231     し
232         return(kill_pg(-pid, sig, 0))。
233     /* ノーマル・キル */
234     return(kill_proc(pid, sig, 0));
235 }。
236 /*
237  * プロセスグループが「孤児」であるかどうかを、POSIXに基づいて判断します。
238  * 2.2.2.52.Orphaned Process Group is not to be affected by 2 .2.2.52.
239  * の定義。
240  * ターミナルで生成されたストップ シグナルによって 新たに孤児となったプロセスグループは

```

[228](#) *SIGHUP*と*SIGCONT*を受信するために、*。

[229](#) *

230 * “あなたに聞きますが、あなたは孤児であることが何であることを知っていますか？”

231 */

```
// 前述のPOSIX P1003.1の2.2.2.52項は、オーファンプロセスの記述である
//グループになります。いずれの場合も、プロセスが終了すると、プロセスグループが
// 「孤児」プロセスグループとそのグループ外の親との間の接続は、依存しています。
// を親プロセスとその子プロセスの両方に適用します。そのため、グループ外の最後のプロセスが
// 親プロセスに接続されている、または最後の親の直系の子孫である
// プロセスが終了すると、そのプロセスグループはオーファンプロセスグループになります。いずれの場合も
// プロセスの終了により、プロセスグループが孤児グループになった場合、すべての
// グループ内のプロセスは、ジョブコントロールシェルから切断されます。
// ジョブコントロールシェルは、このプロセスの存在についての情報を一切持たなくなる
//グループになります。停止状態にあるグループ内のプロセスは、永遠に消えてしまいます。解決するには
// この問題では、停止状態のプロセスを含む新しく生成されたオーファンプロセスグループが必要です。
// から切断されたことを示すSIGHUPシグナルとSIGCONTシグナルを // 受け取ることができます。
//二人のセッション
// SIGHUPシグナルは、プロセスグループのメンバーが以下をキャプチャしない限り、終了させます。
// またはSIGHUPシグナルを無視します。SIGCONTシグナルは、以下のようなプロセスの実行を継続します。
// は、SIGHUP信号では終了しません。しかし、ほとんどの場合、いずれかのプロセスが
// グループ内の//が停止状態の場合、グループ内のすべてのプロセスが停止状態になる可能性があります。
//
// プロセスグループが孤児であるかどうかを調べます。孤児でない場合は0を、孤児の場合は1を返します。コード
// ループ
// タスク配列をスキャンします。タスク項目が空である場合、あるいはプロセスグループIDが
// 指定されたものであるか、プロセスがすでにゾンビ状態であるか、プロセスの親が
// initプロセスの場合、スキャンされたプロセスがグループのメンバーでないか、リクエストが
// を満たしていないので、スキップします。それ以外の場合、そのプロセスはグループのメンバーであり、その親
// は
// init処理ではありません。このとき、親のグループIDが、グループの
// id pgrpですが、親のセッションとプロセスのセッションが同じであるということは
// 同じセッションに属しています。したがって、指定された pgrp グループは、確かに孤児ではありません。
//グループ、それ以外...ため息が出ます。
```

232 int is_orphaned_pgrp(int pgrp)

233 {

234 struct task_struct **p;

235

236 for (p = & LAST_TASK ; p > & FIRST_TASK ; --p) {.

237 if (!(*p) ||)

238 ((*p)->pgrp !=pgrp) ||

239 ((*p)-> state == TASK_ZOMBIE) ||。

240 ((*p)->p_pptr->pid == 1))

241 を続けています。

242 if (((*p)->p_pptr->pgrp != pgrp) &&)

243 ((*p)->p_pptr->session == (*p)->session))

244 0を返す。

245 }

246 return(1); /* (ため息をつく) “よくあることだ!” */

247 }

248

// 停止状態にあるジョブ（プロセスグループ）が含まれているかどうかを確認します。

// ない場合は1を、ある場合は0を返します。検索方法は、タスク全体をスキャンする

// 配列を作成し、グループ pgrp に属するプロセスが停止状態にあるかどうかを確認します。

249 static int has_stopped_jobs(int pgrp)

250 {.

251 struct task_struct ** p;

[252](#)

```

253     for (p = & LAST\_TASK; p > & FIRST\_TASK; --p)
254     {
255         if ((*p)->pgrp != pgrp)
256             を続けてい
257             ます。
258         if ((*p)-> state == TASK\_STOPPED)
259             return(1)
260             です。
261     }
262     return(0) です。
263 }
264 }
265
266 // プログラム終了処理関数です。下記365行目のsyscall sys_exit()によって呼び出されます。
267 // この関数は、現在のプロセス自体の特性に応じて処理を行います。
268 // そして、現在のプロセスの状態をTASK_ZOMBIEに設定し、最後にschedule()関数を呼び出します。
269 // 他のプロセスを実行するために、戻りません。
270 volatile void do\_exit(long code)
271 {
272     struct task\_struct *p;
273     int i;
274
275     // 最初に、現在のプロセスコードとデータセグメントによって占有されているメモリページを解放します。
276     // 関数 free_page_tables() の第一引数 (get_base() の戻り値) が示すのは
277     // CPUのリニアアドレス空間における開始ベースアドレス、2番目の (get_limit() return
278     // 値) は、解放するバイト長を示します。get_base()マクロの current->ldt[1] は
279     // プロセスコードセグメント記述子の位置を示し、current->ldt[2]は
280     // データセグメント記述子の位置。get_limit()の0x0fはコードのセクタです。
281     // セグメント、0x17はデータセグメントのセクタとなります。つまり、セグメントのベースアドレスが
282     // を取ると、セグメント記述子のアドレスがパラメータとして使用され、セグメントの
283     // の制限(長さ)がかかっている場合、セグメントセクターはパラメータとして使用されます(指定された
284     // セクタを介してセグメントリミットを得るために使用できるLSL命令。))
285     // free_page_tables()は、mm/memory.cファイルの69行目の冒頭にあります。
286     // get_base()マクロとget_limit()マクロはinclude/linux/sched.hの265行目にあります。
287     free\_page\_tables(get\_base(current->ldt[1]), get\_limit(0x0f));
288     free\_page\_tables(get\_base(current->ldt[2]), get\_limit(0x17));
289
290     // その後、カレントプロセスが開いていたファイルをすべて閉じ、作業ディレクトリpwdを
291     // ルートディレクトリ、実行ファイルのi-node、ライブラリファイルが同期されます。
292     // とi-nodeが戻され、それぞれブランク(解放)されます。そして、セット状態の
293     // プロセスをゾンビ状態(TASK_ZOMBIE)にし、プロセスの終了コードを設定
294     // します。
295     for (i=0; i< NR\_OPEN; i++)
296         if (current->filp[i])
297             sys\_close(i);
298     iput(current->pwd) です。
299     current->pwd = NULL。
300     iput(current->root)。
301     current->root = NULL。
302     iput(current->executable)。
303     current->executable = NULLとなります。
304     iput(current->library) です。
305     current->library = NULL。
306     current-> state = TASK\_ZOMBIE;
307     current->exit_code = code;
308     /*

```

- [283](#) プロセスグループが孤児になっていないかどうかを確認します。
- [284](#) 私たちが退出した結果、彼らは何かを止めているのか？


```

285      * ジョブには、SIGUP と                                SIGCONT を送ります。      (posix 3.2.2.2)
286      *
287      * Case i:父が私たちとは別のグループにいる場合
288      *で、外での接続は私たちだけだったので、私たちのpgrpは
289      孤児になりかけている。
290      */
// POSIX 3.2.2.2(1991年版)はexit()関数の記述です。もしプロセスグループが
// 親がいる//は現在のプロセスとは異なりますが、いずれも
// 同じセッションで、現在のプロセスが属するプロセスグループが
// 孤児であり、現在のプロセスグループに停止状態のジョブが含まれている場合、2つのシグナル
// をグループに送信する必要があります。SIGHUPとSIGCONTです。
291      if ((current->p_pptr->pgrp != current->pgrp) &&)
292          (current->p_pptr->session == current->session) &&
293          is_orphaned_pgrp(current->pgrp) &&
294          has_stopped_jobs(current->pgrp)) {
295          kill_pg(current->pgrp, SIGHUP, 1)。
296          kill_pg(current->pgrp, SIGCONT, 1)。
297      }
298      /* 死んだことを父に知らせる */
299      current->p_pptr->signal |= (1<<(SIGCHLD-1));
300
301      /*
302      * このループは2つのことをしています。
303      *
304      *   A. initにすべての子プロセスを 継承させる
305      *   B. プロセスグループが孤児になっていないかどうかの 確認
306      私たちが退出した結果、彼らが何かを止めていたら
307      *jonsは、SIGUPを送ってから                                SIGCONTを送り
308      ます。 (posix 3.2.2.2)
309      */
// 現在のプロセスに子プロセスがある場合（その p_cpтр が直近に作成された
//子）の場合、プロセス1（initプロセス）は、そのすべての子プロセスの親となります。もし、プロセス1が
// 子プロセスが既にゾンビ状態になっている場合、子プロセス終了信号SIGCHLDを
// initプロセス（親）のことです。
309      if (p = current->p_cpтр) {...
310          while (1) {
311              p->p_pptr = task[1];
312              if (p->state ==
313                  TASK_ZOMBIE)
314                  task[1]->signal |= (1<<(SIGCHLD-1));
315              /*
316              * プロセスグループオーファンチェック
317              ケースII: うちの子は別のpgrpにいたのですが
318              私たちよりもはるかに多くの人が、それが唯一のつながりでした。
319              *外にあるため、子のpgrpは孤児になっています。
320              */
// その子が現在のプロセスと同じグループに属しておらず、同じセッションに属している場合。
//と、現在のプロセスがあるプロセスグループが孤児になってしまうことと
// このグループに停止状態のジョブ（プロセス）がある場合、2つのシグナルを送信する必要があります。
//をこのグループに入れていきます。SIGHUPとSIGCONTです。子プロセスに兄弟プロセスがある場合は、引き続き
// は、これらの兄弟のプロセスをループします。
320      if ((p->pgrp != current->pgrp) &&)
321          (p->session == current->session) &&
322          is_orphaned_pgrp(p->pgrp) &&  です。
323          has_stopped_jobs(p->pgrp)) {

```

```

324         kill_pg(p->pgrp, SIGHUP, 1)。
325         kill_pg(p-
        >pgrp, SIGCONT, 1)。
326     }
327     if (p->p_osptr) {
328         p = p->p_osptr;
329         を続けています。
330     }
331     /*
332     全てをinitの子供たちに結びつける。
333     *と離れる
334     */
    // 以上の処理により、子プロセスの兄弟プロセスがすべて
    // 処理されます。この時点でpは、子プロセスの最も古い兄弟を指しています。つまり、これらすべての
    // 兄弟は、initプロセスの子プロセスのダブルリンクリストヘッダーに追加されます。
    // 結合後、initプロセスのp_cptrは、現在の子プロセスの末っ子を指します。
    // プロセス、そして最も古い兄弟子プロセス p_osptr は最も若い子プロセスを指します。
    // 一方、最も若いプロセスの p_ysptr は、最も古い兄弟のサブプロセスを指します。最後に
    // 現在のプロセスのポインタp_cptrをnullに設定し、ループを終了します。
335     p->p_osptr = task[1]->p_cptr;
336     task[1]->p_cptr->p_ysptr = p;
337     task[1]->p_cptr = current->p_cptr;
338     current->p_cptr = 0;
339     ブレークします。
340 }
341 }
    // 現在のプロセスがセッション・リーダーであり、制御端末を持っている場合、最初に
    // コントロールターミナルを使用しているグループに // SIGHUPという信号を送り、ターミナルを解放する。
    // そして、タスク配列をスキャンして、セッション内のプロセスの端末を空にする（キャンセルする）。
342     if (current->leader) {
343         struct task_struct **p;
344         struct tty_struct *tty;
345
346         if (current->tty >= 0) {...
347             tty = TTY_TABLE(current->tty);
348             if (tty->pgrp > 0)
349                 kill_pg(tty->pgrp, SIGHUP, 1)。
350             tty->pgrp = 0;
351             tty->session = 0;
352         }
353         for (p = & LAST_TASK ; p > & FIRST_TASK ; --p)
354             if ((*p)->session == current->session)
355                 (*p)->tty = -1;
356     }
    // 現在のプロセスが前回コプロセッサを使用した場合は、NULLを設定して、コプロセッサを破棄します。
    // のメッセージを表示します。また、デバッグシンボルが定義されている場合は、監査プロセスツリー機能が呼び出
    // されます。
    // 最後に、スケジューラが呼び出され、実行するプロセスを再スケジューリングして、親プロセスの
    // は、ゾンビプロセスの他の余波を処理することができます。
357     if (last_task_used_math == current)
358         last_task_used_math = NULL;
359 #ifdef debug_proc_tree
360     audit_ptree()で
        す。

```

361 #endif

362 [schedule](#)() で
す。

```

363 }
364
// システムコールのexit()で、プロセスを終了します。パラメータ error_code は、終了ステータスです。
// ユーザープログラムが提供する情報では、下位バイトのみが有効となります。エラーコードのシフト
// を8ビット左に寄せることは、wait()またはwaitpid()関数の要件です。ローバイト
// は、wait()の状態を保存するために使用されます。例えば、プロセスがサスペンド状態の場合は
// (TASK_STOPPED)の場合、その下位バイトは0x7fになります。sys/wait.hファイルの13-19行目を参照してくださ
// い。wait()
// や waitpid() は、これらのマクロを使って、子プロセスの終了ステータスコードや
// 子の終了の理由（シグナル）。
365 int sys_exit(int error_code)
366 {
367     do_exit((error_code&0xff)<<8);
368 }.
369

// システムコールのwaitpid()です。pid で指定された子が終了するまで、現在のプロセスを一時停止する
// （終了）、またはプロセスの終了を要求する信号を受け取った場合、または
// シグナルハンドラです。pid が指す子プロセスが既に終了している場合（これは
// いわゆるゾンビプロセス）になると、このシスコールはすぐに戻ります。が使用するすべてのリソースは
// 子供が解放される。
// pid > 0 の場合、プロセス ID が pid と等しい子を待ちます。
// pid = 0の場合、現在のプロセスのグループと同じグループを持つ子を待ちます。
// pid < -1の場合、子を待っており、そのプロセスグループはpidの絶対値に等しい。
// pid = -1の場合、子プロセスを待っていることを示す。
// オプションがWUNTRACEDの場合は、チャイルドが停止してもすぐに復帰することを意味します。
// options = WNOHANG の場合は、子が終了しない場合は、直ちに再起動することを意味します。
// リターンステートポインタ stat_addr が空でない場合、ステート情報はそこに保存されます。
// パラメータ
// pid-プロセスID; *stat_addr-状態情報へのポインタ; options-waitpidオプション。
370 int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
371 { {...
372     int flag; // 選択された子の準備完了状態またはスリープ状態。
373     struct task_struct *p;
374     unsigned long oldblocked;
375
// まず、ステータス情報を格納するのに十分なメモリ容量があることを確認します。次にリセット
// というフラグを立てます。子プロセスの兄弟姉妹のリストは、最年少の子から順にスキャンされます。
// 現在のプロセスの
376     verify_area(stat_addr, 4);
377     を繰り返し
// 返して
// いま
// す。
378     flag=0です。
379     for (p = current->p_cptr ; p ; p = p->p_osptr)
// {。
// 待機中の子プロセスのpidが0で、かつスキャンされた子プロセスのpidと一致しない場合
// プロセスpは、現在のプロセスの別の子プロセスであることを示し、次にスキップします。
// プロセスをスキャンしてから、次のプロセスをスキャンします。そうでない場合は、待機中の子プロセスが
// pidが見つかったので、390行目にジャンプして実行を続けます。
380     if (pid>0) {
381         if (p->pid != pid)
382             を続けています。

```

```
// それ以外の場合は、待機中のプロセスにpid=0を指定すると、待機中であることを  
// プロセスグループIDが現在のプロセスグループIDと等しいすべての子プロセスのために、//。  
// スキャンされたプロセスpのプロセスグループIDが、現在のプロセスpのグループIDと等しくない場合  
// プロセスである場合はスキップされます。それ以外の場合は、プロセスグループIDが
```

```

// 現在のプロセスグループIDと等しいものが見つかり、390行目にジャンプして続行します。
// 実行します。
383     } else if (!pid) {
384         if (p->pgrp != current->pgrp)
385             を続けています。
// そうでない場合、指定されたpid < -1であれば、プロセスグループidが等しい子は
// からpidの絶対値までが待機しています。スキャンされたプロセスpのグループidが
// pidの絶対値と等しい場合はスキップされます。そうでない場合は、その子の
// プロセスグループIDがpidの絶対値に等しいものが見つかり、390行目にジャンプする
// 実行を続けるために
386     } else if (pid != -1) {...
387         if (p->pgrp != -pid)
388             を続けています。
389     }
// 最初の3つがpidと一致しない場合、現在のプロセスが待機していることを意味します。
// その子プロセスのいずれか（今回はpid = -1）。
//
// この時点で、選択されたプロセスpは、そのプロセスidが指定されたpidと等しいかどうか。
// または、現在のプロセスグループ内の子プロセス、または、プロセスIDが
// pidの絶対値、または任意の子プロセス（pidは-1に等しい）。次に、処理される
// この選択されたプロセスの状態に応じて p.
//
// プロセスpが停止状態のとき、このときにWUNTRACEDオプションが設定されていない場合。
// プログラムがすぐに戻る必要がないことを意味するか、子の終了コードが
// プロセスが0になった場合、スキャンは他の子プロセスの処理を続けます。もし、WUNTRACED
// が設定され、子の終了コードが0でない場合、終了コードを上位バイトに移動させる、OR
// ステータスメッセージ0x7fが*stat_addrに格納され、子のpidが直ちに返される
// 子の終了コードをリセットした後。ここでは、戻り値のステータスが0x7fであるため、WIFSTOPPED() マクロの
// ファイルinclude/sys/wait.hの14行目を参照してください。
390     switch (p-> state) {
391         case TASK\_STOPPED:
392             if (!(オプション & WUNTRACED) ||)
393                 !p->exit_code)
394                     を続けています。
395                     put\_fs\_long((p->exit_code << 8) | 0x7f,
396                         stat_addr)を使用しています。)
397                     p->exit_code = 0;
398                     return p->pid;
// 子プロセスpが死んだ状態の場合、まずユーザーで実行した時間を蓄積します。
// モードとカーネルの状態を現在のプロセス(親プロセス)に入れる。その後、pidを取り出し
// と子プロセスの終了コード、終了コードをリターンステータスの位置に入れる stat_addr
// とし、子プロセスを解放します。最後に、子プロセスの終了コードとpidを返します。
// デバッグ用のプロセスツリーシンボルが定義されている場合は、プロセスツリーの監査機能が呼び出されます。
399         case TASK\_ZOMBIE:
400             current->cutime += p-> utime;
401             current->cstime += p-> stime;
402             flag = p->pid;
403             put\_fs\_long(p->exit_code, stat_addr);
404             release(p)です。
405 #ifdef DEBUG\_PROC\_TREE
406             audit\_ptree()です。
407 #endif
408             リターンフラグ。

```

```

// この子pの状態が停止状態でもゾンビ状態でもない場合、set flag = 1とする。
// これは、要件を満たす子プロセスが見つかったが、それが
// ランニング状態またはスリープ状態の
409         のデフォルトです。
410         flag=1です。
411         を続けています。
412     }
413 }
// タスク配列のスキャンが終了した後、フラグがセットされていれば、その子の
// 待機条件を満たしているプロセスは、終了状態やゾンビ状態ではありません。このとき、もし
// WNOHANGオプションが設定されている場合は、直ちに0を返して終了します。そうでなければ、現在の
// プロセスが割込み可能な待機状態になり、カレントプロセスの信号がブロックするビットマップ
// が保存され、SIGCHLD信号を受信できるように修正された後、スケジューラーを実行します。
414     も (フラグ) {
415         if (options & WNOHANG)
416             0を返す。
417         current->state=TASK_INTERRUPTIBLEとな
418         ります。
419         oldblocked = current->blocked;
420         current->blocked &= ~(1<<(SIGCHLD-1));
         schedule() です。

// システムがこのプロセスの実行を再び開始する際に、プロセスがマスクされていない
// SIGCHLD以外のシグナルであれば、終了コード "Restart System Call "で戻ります。それ以外の場合
// 関数の先頭にあるrepeatラベルにジャンプして、処理手順を繰り返します。
421         current->blocked = oldblockedとなります。
422         if (current->signal & ~(current->blocked | (1<<(SIGCHLD-1))))
423             return -ERESTARTSYS;
424         その他
425             gotoリピート。
426     }
// flag = 0 の場合は、要件を満たすサブプロセスが見つからないことを意味し、エラーとなります。
// のコードが返されます (子プロセスが存在しない)。
427     リターン -
428     ECHILD; 428 }。
429

```

8.9 fork.c

8.9.1 機能説明

子プロセスの生成には、システムコール `fork()` を使用します。Linuxのすべてのプロセスは、プロセス0 (タスク0) の子プロセスです。fork.cプログラムには、`sys_fork()`の補助処理関数群が含まれています (kernel/sys_call.sの222行目から始まります)。このプログラムは、`sys_fork()`システムコールで使われる2つのC関数、`find_empty_process()`と`copy_process()`を提供します。また、プロセスのメモリ領域の検証とメモリの割り当てを行う関数`verify_area()`と`copy_mem()`も含まれています。

`copy_process()`は、プロセスと環境のコードセグメントとデータセグメントを作成し、コピーす

るために使用します。プロセスの複製の手順では、主にプロセスのデータ構造に情報を設定する作業を行います。システムはまず、新しいプロセスのためのページを主記憶領域に要求して、その

タスク構造の情報を取得し、現在のプロセスタスク構造のすべての内容を新しいプロセスタスク構造のテンプレートとしてコピーします。

その後、コードはコピーされたタスク構造体の内容を変更します。まず、コードは現在のプロセスを新しいプロセスの親として設定し、シグナルビットマップをクリアし、新しいプロセスの統計情報をリセットします。次に、現在のプロセス環境に応じて、新しいプロセスのタスクステータスセグメント（TSS）のレジスタを設定します。新プロセスの戻り値は0であるべきなので、`tss.eax = 0`を設定する必要があります。新プロセスのカーネル状態スタックポインタ`tss.esp0`は、新プロセスのタスク構造体が配置されているメモリページの先頭に設定され、スタックセグメント`tss.ss0`は、カーネルデータセグメントセクタに設定されます。`Tss.ldt`には、GDT内のLDT記述子のインデックス値が設定されます。現在のプロセスがコプロセッサを使用している場合は、コプロセッサの完全な状態を新しいプロセスの`tss.i387`構造体に保存する必要があります。

その後、システムは新しいタスクコードセグメントとデータセグメントのベースアドレスとリミットを設定し、現在のプロセスのページング管理の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コピーします。親プロセスで開いているファイルがあれば、子プロセスの対応するファイルも開いているので、対応するファイルを開く回数を1回増やす必要があります。続いて、新しいタスクのTSSとLDT記述子のエントリをGDTに設定し、ベースアドレス情報が新しいプロセスのタスク構造の`tss`と`ldt`を指すようにします。最後に、新しいタスクを実行可能な状態に設定し、新しいプロセスIDを現在のプロセスに返します。

図8-13は、メモリ検証関数`verify_area()`において、開始位置と範囲を検証するための位置調整図である。メモリ書き込み検証関数`write_verify()`は、メモリページ（4096バイト）単位で動作するため、`write_verify()`を呼び出す前に、検証の開始位置をページの開始位置に合わせ、それに対応して検証範囲を調整する必要があります。

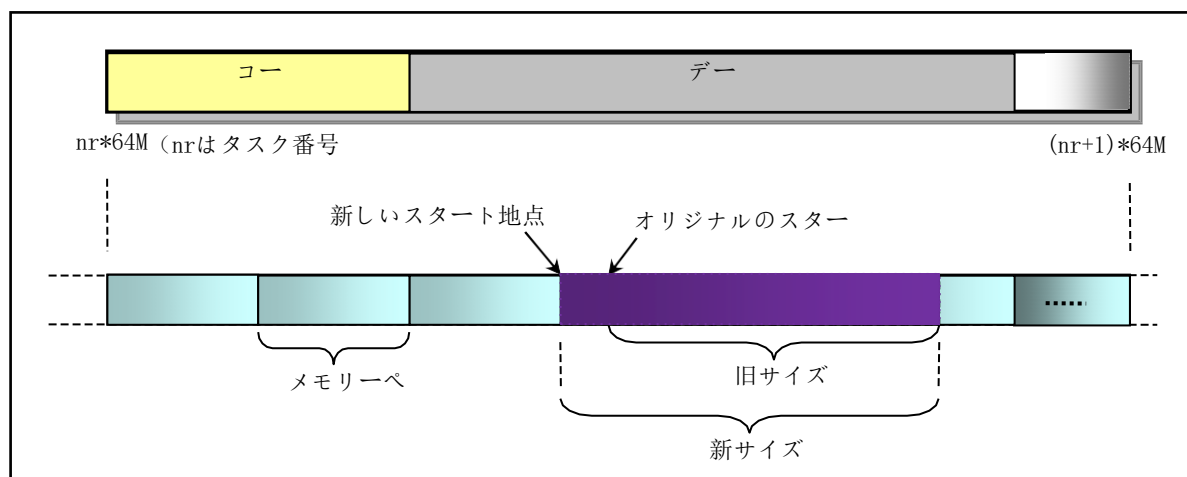


図8-13 メモリ検証範囲と開始位置の調整

以上、`fork.c`プログラムの各関数の目的に基づいて、`fork()`の役割を簡単に説明しました。ここではそれをもう少し詳しく説明します。一般的に`fork()`は、まず親プロセスのタスクデータ構造（別名：プロセスコントロールブロック、PCB）情報をコピーするために、新しいプロセス用のメモリページを申請し、コピーされたタスクデータ構造を新しいプロセス用に修正します。システムコール割り込みが発生したときにスタックに徐々に押し込まれるレジスタ（`copy_process()`のパラメータ）を利

用して、タスク構造体のTSS構造体のレジスタをリセットし、新しいプロセスの状態が割り込みに入る前の親プロセスの状態を維持するようにします。その後、プログラムは新しいプロセスのリニアアドレス空間における開始位置 ($\text{nr} * 64\text{MB}$) を決定します。CPUのセグメンテーション機構の場合は

Linux 0.12のコードセグメントとデータセグメントは、リニアアドレス空間では全く同じものです。そして、親プロセスの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が新しいプロセス用にコピーされます。Linuxの場合

0.12カーネルでは、すべてのプログラムが物理メモリの先頭のページディレクトリテーブルを共有しており、新しいプロセスのページテーブルは別のメモリページを申請する必要があります。

fork()の実行中、カーネルは新しいプロセスのためにコードとデータのメモリページをすぐには割り当てません。新しいプロセスは、親プロセスと協力して、親プロセスにすでにあるコードおよびデータのメモリページを使用します。いずれかのプロセスが書き込みモードでメモリにアクセスしたときにアクセスされるメモリページのみ、書き込み操作の前に新たに要求されたメモリページにコピーされます。

8.9.2 コードコメント

プログラム 8-8 linux/kernel/fork.c

```

1  /*
2      *linux/kernel/fork.c
3      *
4      *(C) 1991 Linus Torvalds
5      */
6
7  /*
8      * 'fork.c' には、'fork' システムコールのヘルプルーチンが含まれています。
9      * (system_call.s も参照) といくつかの雑多な関数 ('verify_area')。
10     * フォークはコツをつかめば割と簡単ですが、メモリは
11     * 管理は面倒です。mm/mm.c'を参照してください。'copy_page_tables()'
12     */
13     // <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。
14     // <linux/sched.h> スケジューラーのヘッダファイルでは、タスク構造体task_struct、データ
15     // 初期のタスク0 の、そしていくつかの組み込みアセンブリ関数のマクロのステートメント。
16     // デスクリプタのパラメータ設定と取得
17     // <linux/kernel.h> カーネルのヘッダファイルです。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
18     // が含まれています。
19     // カーネル の機能を利用します。
20     // <asm/segment.h> セグメント操作のヘッダファイルです。埋め込みアセンブリ関数が定義されています。
21     // セグメントのレジスタ操作。
22     // <asm/system.h> システムのヘッダファイルです。を定義または変更する埋め込みアセンブリマクロです。
23     // 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。
24
25 #include <errno.h>
26
27 #include <linux/sched.h>
28 #include <linux/kernel.h> (日本語)
29 #include <asm/segment.h>
30 #include <asm/system.h>
31
32 // 書き込みページの検証。ページが書き込み可能でない場合、そのページをコピーする。mm/memory.c, L274 を参
33 // 照してください。
34
35 extern void write_verify(unsigned long address);
36
37 long last_pid = 0; // get_empty_process() で生成された最新のプロセス

```

IDです。[23](#)

// プロセス空間の書き換え前の検証機能です。

// 80386 CPUの場合、特権レベル0のコードは、そのページがどのようなものかに関係なく実行されます。

ユーザスペースの//はページプロテクトされています。そのため、ユーザー空間のデータページ保護フラグは

カーネルコードが実行されると、Copy-on-Writeメカニズムが機能しなくなり
 // の効果が得られます。この問題を解決するためにverify_area()関数が使われます。ただし、80486の場合
 //以降のCPUでは、コントロールレジスタCR0に書き込み保護フラグWP（ビット16）があります。があります。
 // カーネルが特権レベル0のコードを無効にして、ユーザースペースの読み取り専用ページにデータを書き込むこ
 とができる
 フラグを設定することで、//を実現しています。したがって、80486以上のCPUでは、同じ目的を達成するために
 このフラグを設定することで、この機能を利用することができます。
 //
 // 論理アドレスの範囲で、書き込み前の検出操作を行う。
 // from addr to (addr + size). 検出はページごとに行われるので
 // プログラムはまず、'addr' が配置されているページの 'start' アドレスを見つける必要があります。
 // そして、'start' にプロセスデータセグメントのベースアドレスを加え、この 'start' が変更されるようにします
 // をCPU 4Gの線形空間のアドレスに変換します。最後に、write_verify()がサイクリックに呼び出されます。
 // 指定されたサイズのメモリ空間に対して、書き込み前の検証を行います。もし、そのページが
 //がリードオンリーの場合は、シェアチェックとコピーページ操作（コピーオンライト）が行われます。

```

24 void verify_area(void * addr,int size)
25 {。
26     unsigned long start;
27
28     // まず、開始アドレスの 'start' をページの開始位置に合わせ、サイズの
29     // それに伴い、検証領域の//が調整されます。次の文のスタート&0xffffは
30     // は、ページ内のオフセットを取得するために使用されます。元の検証範囲の 'size' にこのオフセットを加えた
31     // のものが
32     // は、ページの先頭から始まる範囲の値に展開されます。したがって、それは
33     // また、検証開始位置「start」をページ境界に合わせる必要があります。参照
34     // 上の図8-13。
35     start = (unsigned long) addr;
36     size += start & 0xffff;
37     start &= 0xfffff000; // これでstartが論理アドレス になりました。
38
39     // 次に、プロセスデータセグメントのベースアドレスを追加し、'start' をアドレスに
40     // をシステムの線形空間に配置します。その後、ページ検証を書き込むためにループします。もし、そのページが
41     // 書き込み可能なので、ページをコピーします(mm/memory.c, line 274)。
42     start += get_base(current->ldt[2]); // include/linux/sched.h, line 277
43     while (size>0) {
44         size -= 4096;
45         write_verify(start)です。
46         start += 4096;
47     }
48
49     // メモリのページテーブルをコピーします。
50     // パラメータ nr は新しいタスクの番号、p は新しいタスクのデータ構造のポインタです。この関数は
51     // コードセグメントとデータセグメントのベースアドレス、リミットを設定し、ページテーブルをコピーします。
52     // リニアアドレス空間の新しいタスク。Linuxシステムはコピーオンライト技術を使用しているので
53     // 新しいプロセスのために、新しい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のみが設定されます。
54     // となり、実際の物理メモリページは新しいプロセスに割り当てられません。この時点で
55     // 新しいプロセスは、親プロセスとすべてのメモリページを共有します。成功すれば 0 を返します。
56     // それ以外の場合は、エラーコードを返します。
57     int copy_mem(int nr,struct task_struct * p)
58 {。
59     このような場合には、次のようにします。
60     unsigned long old_code_base,new_code_base,code_limit;
61

```

// まず、現在のプロセスLDTのコードとデータセグメントの記述子の制限が取られます。

```

// 0x0fはコードセグメントセレクタ、0x17はデータセグメントセレクタです。次に、ベースとなる
// リニアアドレス空間における現在のプロセスのコード&データセグメントのアドレス。以降
Linux 0.12のカーネルはコードとデータの分離をサポートしていないため、コードとデータの分離が必要になります。
コード・セグメントとデータ・セグメントのベース・アドレスが同じかどうかを確認するために、 // を使用します。
データセグメントの長さは、少なくともコードの長さ以上でなければなりません。
//セグメント（図5-12参照）にアクセスしないと、カーネルはエラーメッセージを表示し、実行を停止します。
// get_limit()およびget_base()は、include/linux/sched.hファイルの277,279行目で定義されています。
44     code_limit=get_limit(0x0f)となります。
45     data_limit=get_limit(0x17)となります。
46     old_code_base = get_base(current->ldt[1]);
47     old_data_base = get_base(current->ldt[2]);
48     if (old_data_base != old_code_base)
49         panic("We don't support separate I&D")と表示されます。
50     if (data_limit < code_limit)
51         panic("Bad data_limit")となります。
// そして、リニアアドレス空間における新しいプロセスのベースアドレスを次のように設定します。
この値を使って、セグメントディスクリプターのベースアドレスを
// 新しいプロセスのLDTを設定します。そして、新しいプロセスのページディレクトリエントリとページテーブル
エントリを設定します。
// つまり、現在のプロセス（親プロセス）の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コピーしま
す。
//プロセス）を作成します。）この時点で、子は親プロセスのメモリページを共有します。通常は
// copy_page_tables()は0を返します。それ以外の場合は、エラーを示し、そのページのエントリが
// 適用されたばかりのものがリリースされます。
52     new_data_base = new_code_base = nr * TASK_SIZE;
53     p->start_code = new_code_base;
54     set_base(p->ldt[1], new_code_base);
55     set_base(p->ldt[2], new_data_base);
56     if (copy_page_tables(old_data_base, new_data_base, data_limit)) {...
57         free_page_tables(new_data_base, data_limit);
58     return -ENOMEM; 59     }
60     0を返す;
61 }。
62
63 /*
64  *Ok, これはメインのフォークルーティンです。システムプロセスをコピーします。
65  *情報(task[nr])を入力し、必要なレジスタを設定します。それは
66  *また、データセグメントを丸ごとコピーします。
67  */
// プロセス情報をコピーします。
// この関数のパラメータは、システムコール割り込みを入力したハンドラから始まる
// INT 0x80で、この関数が呼び出されるまで(sys_call.s 231行目)、これらのレジスタは
// 徐々にプロセスのカーネルースタックに押し込まれていきます。の値（パラメータ）は
// はsys_call.sファイルのincludeでスタックにプッシュされます。
// 1) 実行時にプッシュされたユーザースタックss、esp、eflag、およびリターンアドレスcs、eip
INT命令 です。
2) ds, es, fs, and edx, ecx, ebx が入力直後の85-91行目でスタックにプッシュされる。
// 3) 97行目でsys_call_tableのsys_fork()が呼ばれたときにプッシュされたリターンアドレス
//（誰にも代表されない）。
// 4) 226-230行目で、gs, esi, edi, ebp, eax(nr)がcopy_process()を呼び出す前にプッシュされる。
// その中で、nrはfind_empty_process()を呼び出した際に割り当てられたタスク配列のアイテムインデックスです
68 int copy_process(int nr, long ebp, long edi, long esi, long gs, long none),

```

[69](#)`long ebx, long ecx, long edx, long orig_eax,`[70](#)`長いfs、長いes、長いds。`


```

71         long eip, long cs, long eflags, long esp, long ss)
72 {
73     struct task\_struct *p;
74     int i;
75     struct file *f;
76
77     // コードはまず、新しいタスク構造体のためにメモリを割り当てます（割り当てに失敗した場合は、次のように返
78     // します）。
79     // エラーコードを表示して終了します。）そして、新しいタスク構造体のポインタを
80     // タスクの配列。ここで nr は、前の find_empty_process() によって返されたタスク番号です。
81     // そして、現在のプロセスのタスク構造の内容をコピーします。
82     // メモリページPを適用したところ。
83     p = (struct task\_struct *) get\_free\_page();
84     if (!p)
85         return -EAGAIN;
86     task[nr] = p;
87     *p = *current; /* 注意! これはスーパーバイザーのスタックをコピー
88     していません。
89
90     // そして、コピーされたプロセス構造の内容にいくつかの修正を加えて、タスク
91     // 新プロセスの構造。新しいプロセスの状態は、まず、無停止の
92     // カーネルが実行をスケジューリングするのを防ぐために // 待機状態にします。その後、プロセスIDのpidを設定
93     // プロセスのランタイムスライス値を優先度に合わせて初期化します。
94     // の値を設定します（通常は15ティック）。その後、シグナルビットマップ、アラームタイマー、セッションリー
95     // ダーをリセットします。
96     // フラグ、カーネルおよびユーザーモードでの実行時間の統計、システム時間の start_time
97     // プロセスの実行を開始したときの//。
98     p->state = TASK\_UNINTERRUPTIBLE;
99     p->pid = last\_pid ; // find_empty_process() で取得した新しいpid
100    p->counter = p->priority; // ランタイムのスライス値（ティック 数）。
101    p->signal = 0; // シグナルのビットマップです。
102    p->alarm = 0; // アラームタイマー。
103    p->leader = 0 ; /* プロセス・リーダーシップは継承されない */。
104    p->utime = p->stime = 0; // ユーザ状態とコア状態の実行時間。
105    p->cutime = p->cstime = 0; // childsのユーザー状態とコア状態の実行時間。
106    p->start_time = jiffies ; // プロセスの開始時刻（現在の時刻の刻み）です。
107
108    // ここで、タスクステータスセクションTSSの内容を変更します（プログラムリストの後の説明を参照）。
109    // システムはタスク構造体pに1ページ分のメモリを割り当てているので、esp0 = (PAGE_SIZE + )
110    ss0:esp0はスタックとして使用されます。
111    プログラムがカーネルモードで実行されるためには、 // 。
112    // さらに、第5章ですでに知っているように、各タスクは2つのセグメント記述子を
113    // GDTテーブル： 1つはタスクのTSSセグメント記述子、もう1つはタスクのLDTテーブル
114    // セグメントディスクリプター。110行目のステートメントは、LDTセグメントのセレクタを格納するためのもので
115    // す。
116    // このタスクの記述子をTSSセグメントに格納します。タスクスイッチを行う際、CPUは
117    // TSSのLDTセグメントセレクタを自動的にLDTRレジスタにロードします。
118    p->tss.back_link = 0;
119    p->tss.esp0 = PAGE\_SIZE + (long) p; // タスクカーネル状態のスタックポインタ。
120    p->tss.ss0 = 0x10; // カーネルのステートスタック のセレクタ。
121    p->tss.eip = eip;
122    p->tss.eflags = eflags;
123    p->tss.eax = 0; // このため、新しいプロセスでは0 が返されます。
124    p->tss.ecx = ecx;

```

```
98      p->tss.edx = edxとなります。  
99      p->tss.ebx = ebxとなります。  
100     p->tss.esp = esp;
```

```

101     p->tss.ebp = ebpとなります。
102     p->tss.esi = esi;
103     p->tss.edi = ediとなります。
104     p->tss.es = es & 0xffff; // セグメントレジスタは16ビット しかありません。
105     p->tss.cs = cs & 0xffff;
106     p->tss.ss = ss & 0xffff;
107     p->tss.ds = ds & 0xffff;
108     p->tss.fs = fs & 0xffff;
109     p->tss.gs = gs & 0xffff;
110     p->tss.ldt = LDT (nr); // タスクのLDT記述子のセクタ（GDT の場合）です。
111     p->tss.trace_bitmap = 0x80000000; // （上位16ビットが有効）。

```

// 現在のタスクがコプロセッサを使用している場合、そのコンテキストが保存されます。CLTSという命令が使われます。

// コントロールレジスタCR0のタスク交換フラグTSをクリアする。CPUは以下の場合にこのフラグを設定します。
// タスクスイッチが発生します。このフラグは、数学コプロセッサを管理するために使用されます：このフラグが設定されている場合。

// の場合、各ESC命令がキャッチされます（例外7）。もし、コプロセッサの存在フラグ
// MPもセットされているので、WAIT命令もキャプチャされます。そのため、タスクスイッチが発生した場合
ESC命令が実行開始された後に、コプロセッサの内容を変更する必要がある場合があります。

// 新しいESC命令を実行する前に保存されます。キャプチャハンドラは、その内容を保存する
コプロセッサの // を削除し、TS フラグをリセットします。FNSAVE命令は、すべての状態を保存するために使用されます。

// コプロセッサの、デスティネーションオペランドで指定されたメモリ領域への書き込み（tss.i387

```

112     if (last_task_used_math == current)
113         asm ("clts ; fnsave %0 ; frstor %0": "m" (p->tss.i387));

```

// 次に、プロセスページテーブルがコピーされます。つまり、ベースアドレスとリミットを新しい
タスクコードとデータセグメントのディスクリプターが設定され、ページテーブルがコピーされます。エラーが発生した場合

// が発生した（戻り値が0ではない）場合、タスク配列の対応するエントリがリセットされて

// 新しいタスク構造に割り当てられたメモリページが解放されます。

```

114     if (copy_mem(nr, p)) { // returnが0でない場合は、エラー を示しています。
115         task[nr] = NULL.
116         free_page((long) p)です。
117         return -EAGAIN;
118     }

```

// 新しく作成された子プロセスは、開いているファイルを親プロセスと共有するため、もし

// 親でファイルが開かれた場合、対応するファイルが開かれた回数が必要となる

// を1つ増やす必要があります。同じ理由で、参照数を増やす必要があります。

// 現在のプロセス（親プロセス）のi個のノードのうち、pwd、root、executableについては1ずつ。

```

119     for (i=0; i< NR_OPEN; i++)
120         if (f=p->filp[i])
121             f->f_count++となります。
122     if (current->pwd)
123         current->pwd->i_count++です。
124     if (current->root)
125         current->root->i_count++となります。
126     if (current->executable)
127         current->executable->i_count++となります。
128     if (current->library)
129         current->library->i_count++です。

```

// 続いて、新しいタスクTSSとLDTのセグメント記述子のエントリがGDTに設定されます。のです。

// 両セグメントのリミットは104バイトに設定されています（include/asm/system.h, line 52-66参照）。すると

```
// プロセス間のリレーションシップリストのポインタを設定する、つまり新しいプロセスを挿入する
// を、現在のプロセスの子プロセスのリンクリストに入れる。つまり、親プロセスである
// 新しいプロセスを現在のプロセスに設定し、最新の子プロセスポインタ p_cptr
```

// と、新しいプロセスの若い兄弟プロセスポインタp_ysptrが空になるように設定されます。すると
 // 新しいプロセスのバディプロセスポインタp_osptrを親の最新の子と同じに設定させる
 // ポインタになります。現在のプロセスが他の子プロセスを持っている場合、若い兄弟のポインタを
 // 隣接するプロセスの p_ysptr が新しいプロセスを指し、そのプロセスの子ポインタを
 // 現在のプロセスのポインタをこの新しいプロセスに移します。その後、新しいプロセスを準備完了状態にして
 set_tss_desc()およびset_ldt_desc()は、ファイル
 // include/asm/system.h, 52-66. "gt+(nr<<1)+FIRST_TSS_ENTRY "は、TSSのアドレスです。
 // グローバルテーブルのタスクnrの記述子。各タスクはGDTの中で2つのアイテムを占めるので
 テーブルの場合は、上の式に「(nr<<1)」を入れる必要があります。なお、タスクレジスタTR
 は、タスク切り替え時にCPUが自動的にロードします。

```

130     set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY, &(p->tss));
131     set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY, &(p->ldt));
132     p->p_pptr = current ; // 親ポインタ。
133     p->p_cptr = 0;
134     p->p_ysptr = 0;
135     p->p_osptr = current ->p_cptr; // 古い兄妹。
136     if (p->p_osptr) // 古い兄弟が存在する場合、その若い兄弟
137         p->p_osptr->p_ysptr = p; // siblingはこの新しいプロセスを指しています。
138     current->p_cptr = p; // 私は現在の新しい子供です。
139     p->state = TASK_RUNNING; /* 念のため、最後にこれを行います */。
140     return last_pid;
141 }。
142

```

// 新しいプロセスのための固有のプロセス番号 last_pid を取得します。この関数は、タスク
 // タスク配列の番号（配列項目）です。
 // まず新しいプロセスIDを取得します。グローバル変数 last_pid を 1 ずつ増加させたものが外にある場合は
 // 表現範囲は、1からのpid番号を再利用します。そして、先ほど設定したpidを検索します。
 タスク配列の中の // がすでに使用されているかどうかを確認します。もしそうであれば、タスク配列の
 // pid番号を再取得するための関数です。それ以外の場合は、一意のpidを見つけたことを意味し、それが
 // が last_pid となります。次に、タスク配列の中から新しいタスクの空きエントリを探し、その中から
 // アイテム番号。なお、last_pidはグローバル変数なので、返す必要はありません。で
 // さらに、現時点でタスク配列の64個のアイテムが完全に占有されている場合は、エラーコード
 を返します。

```

143 int find_empty_process(void)
144 {
145     int i;
146
147     を繰り返しています。
148     if ((++last_pid)<0) last_pid=1;
149     for(i=0 ; i< NR_TASKS ; i++)
150         if (task[i] && ((task[i]->pid == last_pid)
151                         ||
152                         (task[i]->pgrp == last_pid)))
153             goto リビート。
154     for(i=1 ; i< NR_TASKS ; i++) // タスク0は除外。
155         if (! task[i])
156             i.を返します。
157     return -EAGAIN;
158 }

```

8.9.3 インフォメーション

8.9.3.1 タスクステータスセグメント (TSS)

以下の図8-

14は、タスクステータスセグメント (TSS) の内容を示しています。各タスクのTSSは、タスクデータ構造task_structに保存されています。その詳細な説明は第4章を参照してください。

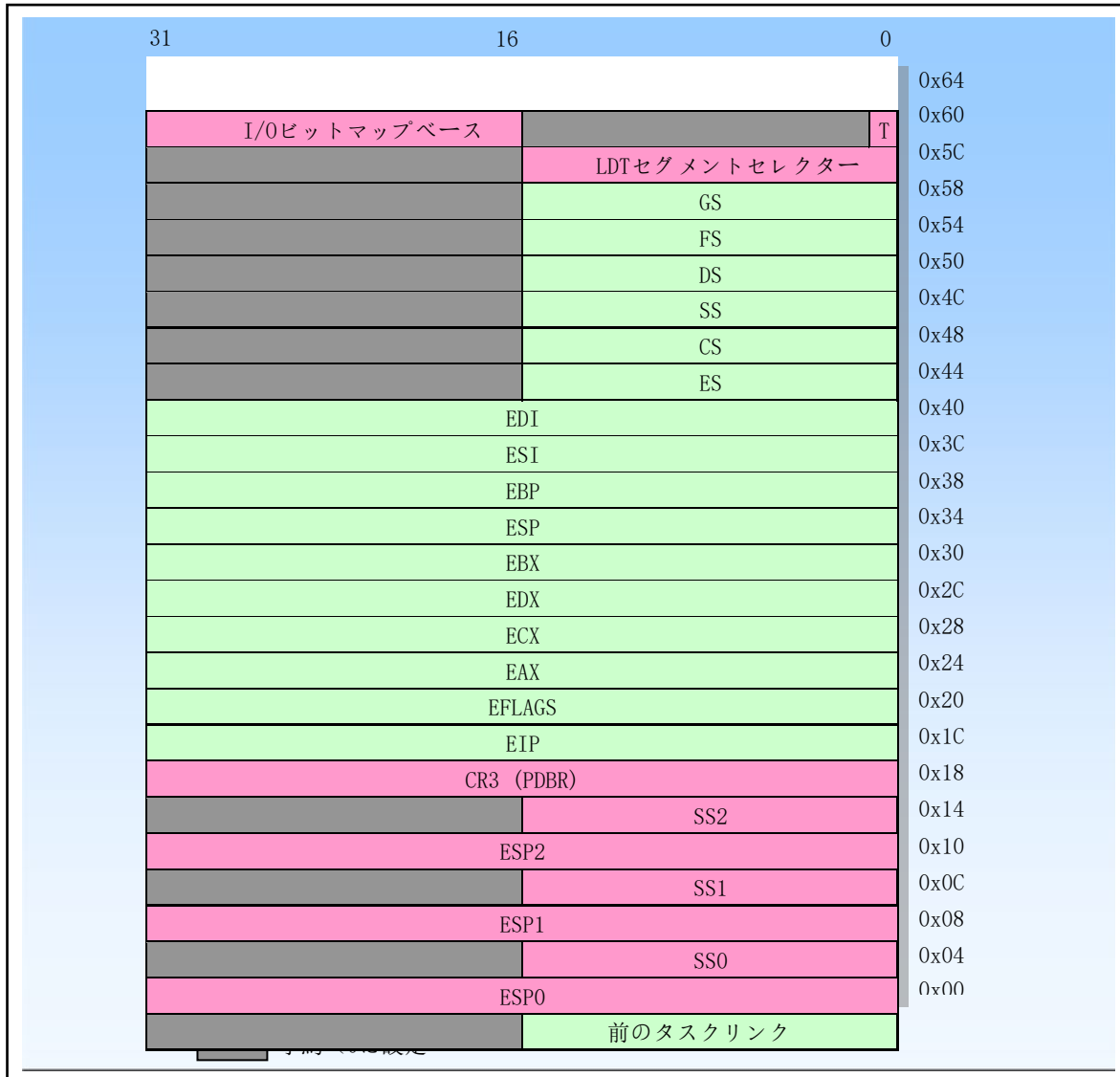


図8-14 タスクステータスセグメントTSSの情報

CPU管理タスクに必要なすべての情報は、タスクステータスセグメント (TSS) という特殊なタイプのセグメントに格納されています。図は、80386タスクを実行するためのTSSフォーマットです。

TSSのフィールドは2つのカテゴリーに分けられる。第1部は、CPUがタスクスイッチを行う際に動的に更新される情報群です。これらのフィールドは、汎用レジスタ (EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI)、セグメントレジスタ (ES、CS、SS、DS、FS、GS)、フラグレジスタ (EFLAGS)、命令ポインタ (EIP)、タスクを実行した前のTSSのセレクタ (復帰時のみ更新) などです。2番目のタイプのフィールドは、CPUが読み取るが変更されない静的な情報のセットです。

これらのフィールドは、タスクのLDTセクタ、タスクページのディレクトリのベースアドレスを含むレジスタ

(PDBR)、特権レベル0～2のスタックポインタ、タスク切り替え時にCPUにデバッグ例外を発生させるTビット、I/Oビットのビットマップベースアドレス(長さの上限はTSS記述子に記載された長さの上限)です。

タスクステータスセグメントは、線形空間のどこにでも格納することができます。他の種類のセグメントと同様に、TSSも記述子によって定義されます。現在実行中のタスクのTSSは、タスクレジスタ (TR) で示されます。LTR命令とSTR命令は、タスクレジスタ (タスクレジスタの可視部分) のセレクタを変更したり読み出したりするために使用されます。

I/Oビットマップの各ビットは、1つのI/Oポートに対応しています。例えば、ポート41のビットは、I/Oビットマップのベースアドレス+5で、ビットオフセットは1です。プロテクトモードでは、I/O命令 (IN、INS、OUT、OUTS) が発生すると、CPUはまず、現在の特権レベルがフラグレジスタのIOPLよりも小さいかどうかをチェックします。この条件が満たされていれば、I/O操作が実行されます。そうでない場合、CPUはTSSのI/Oビットマップを確認します。対応するビットがセットされていれば、一般保護例外が発生し、そうでなければI/O操作が実行されます。

I/OビットマップのベースアドレスがTSSセグメントの限界長以上に設定されている場合、TSSセグメントにI/O許可ビットマップがないことを意味し、現在の特権層CPL>IOPLのすべてのI/O命令が例外保護になります。デフォルトでは、Linux 0.12カーネルは、I/Oビットマップのベースアドレスを0x8000に設定していますが、これはTSSセグメントの制限長である104バイトよりも明らかに大きいので、Linux 0.12カーネルにはI/O許可ビットマップが存在しません。

Linux 0.12では、図中のSS0:ESP0は、カーネルモードで動作しているタスクのスタックポインタを格納するために使用されます。SS1:ESP1とSS2:ESP2は、それぞれ特権レベル1と2を実行しているときに使用されるスタックポインタに対応しています。この2つの特権レベルは、Linuxでは使用されていません。タスクがユーザーモードで動作しているときは、スタックポインタはSS:ESPレジスタに格納されます。上記からわかるように、タスクがカーネル状態になるたびに、カーネル状態のスタックポインタの初期位置は変更されず、タスクのデータ構造があるページの先頭位置になります。

8.10 sys.c

8.10.1 機能説明

sys.cプログラムには、システムコールのための多くの実装関数が含まれています。その中で、戻り値が-ENOSYSしかない関数は、このバージョンのLinuxカーネルがまだこの関数を実装していないことを意味していますので、現在のカーネルコードを参照してその実装を理解することができます。すべてのシステムコール関数の説明については、ヘッダファイルinclude/linux/sys.hを参照してください。

このプログラムには、プロセスID (pid)、プロセスグループID (pgrpまたはpgid)、ユーザーID (uid)、ユーザーグループID (gid)、実際のユーザーID (ruid)、有効なユーザーID (euid)、およびセッションIDに関する多くの機能が含まれています。(session)などの操作機能があります。以下に、これらのIDについて簡単に説明します。

ユーザーは、ユーザーID (uid) とユーザーグループID (gid) を持つ。この2つのIDは、passwd

ファイルでユーザーに設定されたIDであり、リアルユーザーID（ruids）、リアルグループID（rgids）と呼ばれることが多い。また、各ファイルのi-node情報には、ファイルの所有者と所属するユーザグループを示すホストユーザIDとグループIDが保存されており、主にファイルへのアクセスや実行時の権限判別操作に利用される。また、プロセスのタスクデータ構造には、表8-6に示すように、機能別に3つのユーザIDとグループIDが保存されている。

表8-6 プロセスに関連するユーザーIDとグループID

タイプ	ユーザーID	グループID
プロセス	gid - ユーザーIDで、ファイルを所有するユーザーを示します。 の処理を行います。	gid - ユーザーグループを示すグループIDです。 は、プロセスを所有しています。
エフィシ エンシー	euid - アクセスを示す効率的なユーザーIDです。 の権利を取得しています。	egid - 効率的なグループIDです。の許可を示す。 ファイルにアクセスします。
保存	suid - 保存されたユーザーIDです。実行ファイルのset-user-IDフラグが設定されている場合は、実行ファイルのsuidが保存されます。 それ以外の場合はsuid は、プロセスのeuidと同じです。	sgid - 保存されているグループIDです。実行ファイルのset-group-IDフラグが設定されている場合は、実行ファイルのgidがsgidに保存されます。それ以外の場合は、sgidは次のようになります。 プロセスのegidです。

プロセスのuidとgidは、プロセスのオーナーのユーザーIDとグループID、つまりプロセスのリアルユーザーID (ruid) とリアルグループID (rgid) です。スーパーユーザは、関数set_uid()およびset_gid()を使用してこれらを変更できます。プロセスがファイルにアクセスする際のパーミッション判定には、実効ユーザIDと実効グループIDが使用されます。

保存されたユーザID(suid)と保存されたグループID(sgid)は、プロセスがset-user-IDまたはset-group-IDフラグが設定されたファイルにアクセスする際に使用されます。プログラムを実行する際、プロセスのeuidは通常、実際のユーザーIDであり、egidは通常、実際のグループIDです。そのため、プロセスは、プロセスの実ユーザ、実ユーザ・グループで指定されたファイル、またはアクセスが許可されたその他のファイルにしかアクセスできません。ただし、ファイルのset-user-IDフラグが設定されている場合には、プロセスの実効ユーザIDがファイル所有者のユーザIDに設定されるため、このフラグが設定されている制限付きファイルにアクセスすることができます。ファイル所有者のユーザIDはsuidに保存されます。同様に、ファイルのset-group-IDフラグにも同様の効果があり、同じように扱われます。

例えば、プログラムファイルの所有者がスーパーユーザーであるにもかかわらず、プログラムがset-user-IDフラグを設定していた場合、そのプログラムがプロセスによって実行されると、プロセスの実効ユーザーID (euid) はスーパーユーザーのID (0) に設定されます。つまり、このプロセスはスーパーユーザーの権限を持つことになります。実例としては、Linuxのpasswdコマンドがあります。このコマンドは、set-user-Idを設定することで、ユーザーが自分のパスワードを変更できるようにするプログラムです。このプログラムは、ユーザの新しいパスワードを/etc/passwdファイルに書き込む必要があります、このファイルへの書き込み権限を持つのはスーパーユーザだけであるため、passwdプログラムはset-user-IDフラグを使用する必要があります。

さらに、プロセスは、自身の属性を識別するプロセスID (pid)、所有するプロセスグループのプロセスグループID (pgrpまたはpgid)、所有するセッションのセッションID (session) も持っています。これらの3つのIDは、ユーザーIDやグループIDとは関係なく、プロセスとの関係を示すために使用されます。

8.10.2 コードコメント

```
1 /*  
2  *linux/kernel/sys.c  
3  *  
4  *(C) 1991Linus Torvalds  
5  */  
6  
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。  
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ  
// 初期のタスク0の、そしていくつかの組み込みアセンブリ関数のマクロのステートメント。  
// デスクリプタのパラメータ設定と取得
```

```

// <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。
// コミュニケーション
// <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 でよく使われる機能
// <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。
// <asm/segment.h> セグメント操作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。
// セグメントのレジスタ操作。
// <sys/times.h> ランニングタイム構造体tmsと、関数プロトタイプtimes()を定義しています。
プロセス を
// <sys/utsname.h> システム名構造体のヘッダファイルです。
// <sys/param.h> パラメータファイルです。いくつかのハードウェア関連のパラメータ値が与えられています。
// <sys/resource.h> リソースファイル。システムの限界と利用に関する情報が含まれています。
// プロセスが使用するリソース
// <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。 7 #include <errno.h>
8
9 #include <linux/sched.h>
10 #include <linux/tty.h>
11 #include <linux/kernel.h> (日本語)
12 #include <linux/config.h>
13 #include <asm/segment.h>
14 #include <sys/times.h>
15 #include <sys/utsname.h>.
16 #include <sys/param.h>
17 #include <sys/resource.h>.
18 #include <string.h>
19
20 /*
21 * ローカルシステムが 置かれているタイムゾーンです。一部の企業 ではデフ
ォルトで使用されています。
22 * gettimeofdayを使ってこの値を取得するプログラム。
23 */
// タイムゾーン構造の第1フィールド(tz_minuteswest)は西への分数を表します。
2番目のフィールド(tz_dsttime)は、夏時間(DST)調整タイプです。この
// 構造体はinclude/sys/time.hで定義されています。
24 struct timezone sys\_tz = { 0, 0};
25
// プロセスグループID pgrpに従ってプロセスグループのセッションIDを取得します。
この関数は、ファイルkernel/exit.cの161行目に実装されています。
26 extern int session of pgrp(int pgrp);
27
// 日付と時刻を取得します ( ftime - fetch time )。
// 戻り値が-ENOSYSのシステムコールは、それが実装されていないことを示します。
28 int sys\_ftime()
29 {
30     リターン -
ENOSYS; 31 }。
32
33 int sys\_break()
34 {
35     return -ENOSYS;
36 }。
37

```

```

// 現在のプロセスが子プロセスをデバッグするために使用します。
38 int sys_ptrace()
39 {
40     return -ENOSYS;
41 }。
42
// 端末の回線設定を変更して印刷する。
43 int sys_stty()
44 {
45     リターン -
ENOSYS; 46 }。
47
// 端末の回線設定情報を取得します。
48 int sys_gtty()
49 {
50     return -ENOSYS;
51 }。
52
// ファイル名を変更する。
53 int sys_rename()
54 {
55     return -ENOSYS;
56 }
57
58 int sys_prof()
59 {
60     return -ENOSYS;
61 }
62
63 /*
64  * これは BSD スタイルで行われ、保存された gid は考慮されません。
65  * 有効なgidを設定すると、保存されたgid も設定されます。   これ
   は
66  * これにより、setgidプログラムがその特権を完全に放棄することができます。
67  * これは、セキュリティ対策を行う際に有用なアサーションです。
68  * プログラムに対する監査。
69  *
70  * 一般的な考え方としては、単にsetregid()を使用するプログラムは
71  * BSDと100%の互換性があります。setgid()だけを使ったプログラ
   ムは
72  * POSIXと100%互換性があり、IDも保存されています。
73  */
// 現在のタスクの現在および / または有効なグループ ID (gid) を設定する。タスクが以下を持って
// いない場合
// スーパーユーザー権限を持っている場合は、その本当のグループIDと有効なグループIDのみを入れ
// 替えることができます。
// タスクがスーパーユーザー権限を持っている場合は、実効グループIDと実グループIDを任意に設定で
// きます。
// で、保存したgid (sgid) を有効なgid (egid) に設定します。実質的なグループID (rgid) は、参
// 照する
// をプロセスの現在のgidに変換します。
74 int sys_setregid(int rgid, int egid)
75 {
76     if (rgid>0) {
77         if ((current->gid == rgid) ||)

```

```
78         suser()  
79         current->gid = rgid;  
80     その他  
81         return(-EPERM)となります。
```

```

82     }
83     if (egid>0) {...
84         if ((current->gid == egid) ||)
85             (current->egid == egid)
86                 ||。
87                 suser()) {
88                     current->egid = egid;
89                     current->sgid = egid;
90             } else
91                 return(-EPERM)となりま
92                 す。
93     }
94     0を返す。
95 }
96 /*
97  * setgid\(\) は SysV のように SAVED_IDS を使って実装されています。
98  */
99 // プロセスグループID(gid)を設定します。タスクがスーパーユーザの権限を持っていない場合には
100 // setgid\(\) で、実効 gid を保存した gid (sgid) または実 gid (rgid) に設定します。もし
101 // タスクにスーパーユーザ権限があり、rgid、egid、sgidのすべてに指定されたgidが設定されます。
102 // パラメータで
103 int sys\_setgid(int gid)
104 {
105     if (suser())
106         current->gid = current->egid = current->sgid = gid;
107     else if ((gid == current->gid) || (gid == current->sgid))
108         current->egid = gid;
109     その他
110         return -EPERM;
111     return 0;
112 }。
113
114 // プロセスの課金をオンまたはオフにする。
115 int sys\_acct()
116 {
117     return -ENOSYS;
118 }。
119
120 // 任意の物理メモリをプロセスの仮想アドレス空間にマッピングします。
121 int sys\_phys()
122 {
123     return -ENOSYS;
124 }。
125
126 int sys\_lock()
127 {
128     return -ENOSYS;
129 }。
130
131 int sys\_mpx()
132 {
133     return -ENOSYS;
134 }。

```

128


```

129 int sys_ulimit()
130 {
131     return -ENOSYS;
132 }。
133
// 1970年1月1日のGMT 00:00:00からの時間（秒単位）を返します。
// パラメータtlocがnullでない場合は、時刻の値もそこに格納されます。場所は
// パラメータが指すものがユーザ空間にある場合は、関数 put_fs_long()
// を使用して、ユーザスペースに時刻の値を保存します。カーネル内で実行される場合、セグメントレジスタ
fs
は、デフォルトでは現在のユーザデータ空間を指します。そのため、この関数では、fsセグメント
// ユーザー空間の値にアクセスするためのレジスタです。
134 int sys_time(long * tloc)
135 {。
136     int i;
137
138     i = CURRENT_TIME;
139     if (tloc) {
140         verify_area(tloc, 4); // メモリの容量が十分かどうかを確認する（4バイト）。
141         put_fs_long(i, (unsigned long *)tloc);
142     }
143     return i;
144 }。
145
146 /*
147  * 非特権ユーザーは、実際のユーザーIDを有効なuidに変更することができます。
148  * またはその逆です。(BSD-style)
149  *
150  * effective uidを設定すると、saved uidも設定されます。これは
151  * これにより、setuidされたプログラムがその特権を完全に放棄することが可能になります。
152  * これは、セキュリティ対策を行う際に有用なアサーションです。
153  * プログラムに対する監査。
154  *
155  * 一般的な考え方としては、setreuid()だけを使用したプログラムでは
156  * BSDと100%の互換性があります。setuid()だけを使ったプログラムは
157  * POSIXと100%互換性があり、IDも保存されています。
158  */
// タスクの実際の、あるいは有効なユーザーID (uid) を設定します。タスクがスーパーユーザー
// の権限を持っている場合、そのリアルuid (ruid) とエフェクティブuid (euid) のみを交換することができます
// もし、その人の
// タスクがスーパーユーザー権限を持っていれば、実効ユーザーIDと実ユーザーIDを任意に設定できます。保存さ
れた
// uid(suid)はeuidと同じ値が設定されています。
159 int sys_setreuid(int ruid, int euid)
160 {
161     int old_ruid = current->uid;
162
163     if (ruid>0) {
164         if ((current->euid==ruid) ||)
165             (old_ruid == ruid) ||。
166             suser()
167             current->uid = ruid;
168     その他
169     return(-EPERM) となり

```

ます。

[170](#)

}

[171](#)

if (euid>0) {

```

172         if ((old_ruid == euid) ||)
173             (current->euid == euid) ||)
174             suser()) {
175                 current->euid = euid;
176                 current->suid = euid;
177             } else {
178                 current->uid = old_ruid;
179                 return(-EPERM) となります。
180             }
181     }
182     0を返す。
183 }
184
185 /*
186  * setuid\(\) は SysV w/ SAVED\_IDS のように実装されています。
187  *
188  * SAVED\_IDの欠点は、setuidされたルートプログラムが
189  * 例えばsendmailのように、uidを通常のものに設定することはできません。
190  * rootであれば、setuid\(\)が
191  * 保存されたuid も。これが気に入らなければ、聡明な人々を非難して
192  *   POSIX committeeおよび/または      USGでは、以下ようになっていま
193  *   す。                                なお、BSDスタイルのsetreuid\(\)は
194  *   *は、ルートプログラムが一時的に権限を落として
195  *   実際のuidと効果的なuidを入れ替えることで、それらを取り戻すことができま
196  *   す。
197  */
198 // タスクのユーザーID(uid)を設定します。タスクがスーパーユーザーの権限を持っていない場合は、 setuid\(\)
199 // を使用して、実効 uid (euid) を保存 uid (suid) または実 uid (ruid) に設定します。タスクが
200 // がスーパーユーザー権限を持っている場合、ruid、euid、suidには
201 // パラメータ。
202 int sys\_setuid(int uid)
203 {
204     if (suser())
205         current->uid = current->euid = current->suid = uid;
206     else if ((uid == current->uid) || (uid == current->suid))
207         current->euid = uid;
208     その他
209         return -EPERM;
210     return(0);
211 }。
212
213 // システムの起動時間を設定します。パラメータtptrは、カウントされる時間値（秒単位）です。
214 // 1970年1月1日のGMT 00:00:00から。
215 // 呼び出したプロセスにはスーパーユーザーの権限が必要です。HZ=100は動作周波数
216 // カーネルシステムの // になります。パラメータポインタの位置はユーザ空間にあるため、次のようなものが必要
217 // です。
218 // で、関数 get\_fs\_long\(\) を使って値にアクセスします。カーネル内で実行される場合、セグメント
219 // レジスタfsは、デフォルトでは、現在のユーザデータ空間を指します。そのため、この関数では
220 // fs でユーザ空間の値にアクセスする。関数パラメータで提供される現在時刻の値
221 // システムが稼働していた時間の秒数値（ジフティ / HZ）を引いたものがブートタイム
222 // 秒単位です。
223 int sys\_stime(long * tptr)
224 {。
225     if (! suser())

```

```
210         return -EPERM;  
211     startup\_time = get\_fs\_long((unsigned long *)tptr) - jiffies/HZ;
```

```

212     jiffies_offset = 0;
213     0を返す。
214 }
215
216     ///// 現在のタスクのランタイム統計を取得します。
217     // 指定されたユーザーデータスペースのtms構造のタスクランタイム統計を返す
218     // tbufによるものです。tms 構造体には、プロセスユーザランタイム、カーネルランタイム、チャイ
219     // ルドユーザ
220     // ランタイム、およびチャイルドカーネルランタイムに対応しています。この関数の戻り値は、その
221     // 時点での
222     // システムは現在の時刻に合わせて実行されます。
223 int sys times(struct tms * tbuf)
224 {
225     if (tbuf) {
226         verify_area(tbuf, sizeof *tbuf) です。
227         put_fs_long(current-> utime, (unsigned long *)&tbuf->tms_utime);
228         put_fs_long(current-> stime, (unsigned long *)&tbuf->tms_stime);
229         put_fs_long(current->cutime, (unsigned long *)&tbuf->tms_cutime);
230         put_fs_long(current->cstime, (unsigned long *)&tbuf->tms_cstime);
231     }
232     return jiffies;
233 }
234
235     ///// プログラムの終了位置をメモリーに設定します。
236     // パラメータend_data_segの値が妥当で、システムに十分な量の
237     // メモリを使用し、プロセスが最大データセグメントサイズを超えていない場合、この関数は
238     // データセグメントの最後にend_data_segで指定された値を設定します。この値は、より大きな値で
239     // なければなりません。
240     // コードの終わりよりも、スタックの終わりの16KBよりも小さい。戻り値は
241     // データセグメントの新しい終了値（戻り値が要求されたものと異なる場合は
242     // の値であれば、エラーが発生したことになります。）この関数はユーザーから直接呼び出される
243     // ことはありませんが、ラッパーとして
244     // libcのライブラリ関数によって、戻り値は同じではありません。
245 int sys brk(unsigned long end_data_seg)
246 {
247     // パラメータがコードの終端よりも大きく、(スタック-16KB)よりも小さい場合、セット
248     // 新しいデータセグメントの終了値です。
249     if (end_data_seg >= current->end_code &&)
250         end_data_seg < current->start_stack - 16384)
251         current-> brk = end_data_seg;
252     return current-> brk ; // 現在のデータセグメントの終了値 を返し
253     ます。234 }
254
255 /*
256 * これは、いくつかのヘーブチェックを必要とする...
257 * 私はそのための胃袋を得ることができませんでした。また、私は完全には
258 * session/pgprなどを理解しています。誰かがそれを説明してくれるでしょう。
259 *
260 * OK, I think I have the protection semantics right.... this is really
261 * マルチユーザーシステムでは、1人のユーザーが
262 * *は、
263     他の人が所有しているプロセスにシグナルを送る
264     -TYT, 12/12/91
265 *は、
266 *は、
267 *は、
268 *は、
269 *は、
270 *は、
271 *は、
272 *は、
273 *は、
274 *は、
275 *は、
276 *は、
277 *は、
278 *は、
279 *は、
280 *は、
281 *は、
282 *は、
283 *は、
284 *は、
285 *は、
286 *は、
287 *は、
288 *は、
289 *は、
290 *は、
291 *は、
292 *は、
293 *は、
294 *は、
295 *は、
296 *は、
297 *は、
298 *は、
299 *は、
300 *は、
301 *は、
302 *は、
303 *は、
304 *は、
305 *は、
306 *は、
307 *は、
308 *は、
309 *は、
310 *は、
311 *は、
312 *は、
313 *は、
314 *は、
315 *は、
316 *は、
317 *は、
318 *は、
319 *は、
320 *は、
321 *は、
322 *は、
323 *は、
324 *は、
325 *は、
326 *は、
327 *は、
328 *は、
329 *は、
330 *は、
331 *は、
332 *は、
333 *は、
334 *は、
335 *は、
336 *は、
337 *は、
338 *は、
339 *は、
340 *は、
341 *は、
342 *は、
343 *は、
344 *は、
345 *は、
346 *は、
347 *は、
348 *は、
349 *は、
350 *は、
351 *は、
352 *は、
353 *は、
354 *は、
355 *は、
356 *は、
357 *は、
358 *は、
359 *は、
360 *は、
361 *は、
362 *は、
363 *は、
364 *は、
365 *は、
366 *は、
367 *は、
368 *は、
369 *は、
370 *は、
371 *は、
372 *は、
373 *は、
374 *は、
375 *は、
376 *は、
377 *は、
378 *は、
379 *は、
380 *は、
381 *は、
382 *は、
383 *は、
384 *は、
385 *は、
386 *は、
387 *は、
388 *は、
389 *は、
390 *は、
391 *は、
392 *は、
393 *は、
394 *は、
395 *は、
396 *は、
397 *は、
398 *は、
399 *は、
400 *は、
401 *は、
402 *は、
403 *は、
404 *は、
405 *は、
406 *は、
407 *は、
408 *は、
409 *は、
410 *は、
411 *は、
412 *は、
413 *は、
414 *は、
415 *は、
416 *は、
417 *は、
418 *は、
419 *は、
420 *は、
421 *は、
422 *は、
423 *は、
424 *は、
425 *は、
426 *は、
427 *は、
428 *は、
429 *は、
430 *は、
431 *は、
432 *は、
433 *は、
434 *は、
435 *は、
436 *は、
437 *は、
438 *は、
439 *は、
440 *は、
441 *は、
442 *は、
443 *は、
444 *は、
445 *は、
446 *は、
447 *は、
448 *は、
449 *は、
450 *は、
451 *は、
452 *は、
453 *は、
454 *は、
455 *は、
456 *は、
457 *は、
458 *は、
459 *は、
460 *は、
461 *は、
462 *は、
463 *は、
464 *は、
465 *は、
466 *は、
467 *は、
468 *は、
469 *は、
470 *は、
471 *は、
472 *は、
473 *は、
474 *は、
475 *は、
476 *は、
477 *は、
478 *は、
479 *は、
480 *は、
481 *は、
482 *は、
483 *は、
484 *は、
485 *は、
486 *は、
487 *は、
488 *は、
489 *は、
490 *は、
491 *は、
492 *は、
493 *は、
494 *は、
495 *は、
496 *は、
497 *は、
498 *は、
499 *は、
500 *は、
501 *は、
502 *は、
503 *は、
504 *は、
505 *は、
506 *は、
507 *は、
508 *は、
509 *は、
510 *は、
511 *は、
512 *は、
513 *は、
514 *は、
515 *は、
516 *は、
517 *は、
518 *は、
519 *は、
520 *は、
521 *は、
522 *は、
523 *は、
524 *は、
525 *は、
526 *は、
527 *は、
528 *は、
529 *は、
530 *は、
531 *は、
532 *は、
533 *は、
534 *は、
535 *は、
536 *は、
537 *は、
538 *は、
539 *は、
540 *は、
541 *は、
542 *は、
543 *は、
544 *は、
545 *は、
546 *は、
547 *は、
548 *は、
549 *は、
550 *は、
551 *は、
552 *は、
553 *は、
554 *は、
555 *は、
556 *は、
557 *は、
558 *は、
559 *は、
560 *は、
561 *は、
562 *は、
563 *は、
564 *は、
565 *は、
566 *は、
567 *は、
568 *は、
569 *は、
570 *は、
571 *は、
572 *は、
573 *は、
574 *は、
575 *は、
576 *は、
577 *は、
578 *は、
579 *は、
580 *は、
581 *は、
582 *は、
583 *は、
584 *は、
585 *は、
586 *は、
587 *は、
588 *は、
589 *は、
590 *は、
591 *は、
592 *は、
593 *は、
594 *は、
595 *は、
596 *は、
597 *は、
598 *は、
599 *は、
600 *は、
601 *は、
602 *は、
603 *は、
604 *は、
605 *は、
606 *は、
607 *は、
608 *は、
609 *は、
610 *は、
611 *は、
612 *は、
613 *は、
614 *は、
615 *は、
616 *は、
617 *は、
618 *は、
619 *は、
620 *は、
621 *は、
622 *は、
623 *は、
624 *は、
625 *は、
626 *は、
627 *は、
628 *は、
629 *は、
630 *は、
631 *は、
632 *は、
633 *は、
634 *は、
635 *は、
636 *は、
637 *は、
638 *は、
639 *は、
640 *は、
641 *は、
642 *は、
643 *は、
644 *は、
645 *は、
646 *は、
647 *は、
648 *は、
649 *は、
650 *は、
651 *は、
652 *は、
653 *は、
654 *は、
655 *は、
656 *は、
657 *は、
658 *は、
659 *は、
660 *は、
661 *は、
662 *は、
663 *は、
664 *は、
665 *は、
666 *は、
667 *は、
668 *は、
669 *は、
670 *は、
671 *は、
672 *は、
673 *は、
674 *は、
675 *は、
676 *は、
677 *は、
678 *は、
679 *は、
680 *は、
681 *は、
682 *は、
683 *は、
684 *は、
685 *は、
686 *は、
687 *は、
688 *は、
689 *は、
690 *は、
691 *は、
692 *は、
693 *は、
694 *は、
695 *は、
696 *は、
697 *は、
698 *は、
699 *は、
700 *は、
701 *は、
702 *は、
703 *は、
704 *は、
705 *は、
706 *は、
707 *は、
708 *は、
709 *は、
710 *は、
711 *は、
712 *は、
713 *は、
714 *は、
715 *は、
716 *は、
717 *は、
718 *は、
719 *は、
720 *は、
721 *は、
722 *は、
723 *は、
724 *は、
725 *は、
726 *は、
727 *は、
728 *は、
729 *は、
730 *は、
731 *は、
732 *は、
733 *は、
734 *は、
735 *は、
736 *は、
737 *は、
738 *は、
739 *は、
740 *は、
741 *は、
742 *は、
743 *は、
744 *は、
745 *は、
746 *は、
747 *は、
748 *は、
749 *は、
750 *は、
751 *は、
752 *は、
753 *は、
754 *は、
755 *は、
756 *は、
757 *は、
758 *は、
759 *は、
760 *は、
761 *は、
762 *は、
763 *は、
764 *は、
765 *は、
766 *は、
767 *は、
768 *は、
769 *は、
770 *は、
771 *は、
772 *は、
773 *は、
774 *は、
775 *は、
776 *は、
777 *は、
778 *は、
779 *は、
780 *は、
781 *は、
782 *は、
783 *は、
784 *は、
785 *は、
786 *は、
787 *は、
788 *は、
789 *は、
790 *は、
791 *は、
792 *は、
793 *は、
794 *は、
795 *は、
796 *は、
797 *は、
798 *は、
799 *は、
800 *は、
801 *は、
802 *は、
803 *は、
804 *は、
805 *は、
806 *は、
807 *は、
808 *は、
809 *は、
810 *は、
811 *は、
812 *は、
813 *は、
814 *は、
815 *は、
816 *は、
817 *は、
818 *は、
819 *は、
820 *は、
821 *は、
822 *は、
823 *は、
824 *は、
825 *は、
826 *は、
827 *は、
828 *は、
829 *は、
830 *は、
831 *は、
832 *は、
833 *は、
834 *は、
835 *は、
836 *は、
837 *は、
838 *は、
839 *は、
840 *は、
841 *は、
842 *は、
843 *は、
844 *は、
845 *は、
846 *は、
847 *は、
848 *は、
849 *は、
850 *は、
851 *は、
852 *は、
853 *は、
854 *は、
855 *は、
856 *は、
857 *は、
858 *は、
859 *は、
860 *は、
861 *は、
862 *は、
863 *は、
864 *は、
865 *は、
866 *は、
867 *は、
868 *は、
869 *は、
870 *は、
871 *は、
872 *は、
873 *は、
874 *は、
875 *は、
876 *は、
877 *は、
878 *は、
879 *は、
880 *は、
881 *は、
882 *は、
883 *は、
884 *は、
885 *は、
886 *は、
887 *は、
888 *は、
889 *は、
890 *は、
891 *は、
892 *は、
893 *は、
894 *は、
895 *は、
896 *は、
897 *は、
898 *は、
899 *は、
900 *は、
901 *は、
902 *は、
903 *は、
904 *は、
905 *は、
906 *は、
907 *は、
908 *は、
909 *は、
910 *は、
911 *は、
912 *は、
913 *は、
914 *は、
915 *は、
916 *は、
917 *は、
918 *は、
919 *は、
920 *は、
921 *は、
922 *は、
923 *は、
924 *は、
925 *は、
926 *は、
927 *は、
928 *は、
929 *は、
930 *は、
931 *は、
932 *は、
933 *は、
934 *は、
935 *は、
936 *は、
937 *は、
938 *は、
939 *は、
940 *は、
941 *は、
942 *は、
943 *は、
944 *は、
945 *は、
946 *は、
947 *は、
948 *は、
949 *は、
950 *は、
951 *は、
952 *は、
953 *は、
954 *は、
955 *は、
956 *は、
957 *は、
958 *は、
959 *は、
960 *は、
961 *は、
962 *は、
963 *は、
964 *は、
965 *は、
966 *は、
967 *は、
968 *は、
969 *は、
970 *は、
971 *は、
972 *は、
973 *は、
974 *は、
975 *は、
976 *は、
977 *は、
978 *は、
979 *は、
980 *は、
981 *は、
982 *は、
983 *は、
984 *は、
985 *は、
986 *は、
987 *は、
988 *は、
989 *は、
990 *は、
991 *は、
992 *は、
993 *は、
994 *は、
995 *は、
996 *は、
997 *は、
998 *は、
999 *は、
1000 *は、

```

```
//// 指定したプロセスのpidのプロセスグループIDをpgidに設定します。  
// パラメータpidは、プロセスIDです。パラメータ pid が 0 の場合は、この pid を等しくします。  
// を現在のプロセスのpidに変換します。パラメータ pgid は、プロセスグループ ID を指定します。もしそれが  
// 0の場合は、プロセスpidのプロセスグループidと同じにします。もし、この関数が  
あるプロセスグループから別のプロセスグループにプロセスを移動させるには、2つのプロセスグループが
```

```

// 同一のセッションです。この場合、パラメータ pgid は、既存のプロセスグループ ID を指定し
て
245 // 参加し、グループのセッションIDは参加するプロセスと同じでなければならない (L263)。 int
    sys_setpgid(int pid, int pgid)
246 {
247     int i;
248
// パラメータ pid が 0 の場合、pid は現在のプロセスの pid に設定されます。パラメーターが
// pgidが0であれば、pgidは現在のプロセスのpidでもあります。pgidが0より小さい場合は、無効な
// エラーコードが返されます。
249     if (!pid)
250         pid = current->pid;
251     if (!pgid)
252         pgid = current->pid;
253     if (pgid < 0)
254         return -EINVAL;
// タスク配列をスキャンして、指定されたプロセスpidを持つタスクを探します。指定された
// プロセスIDがpidであることがわかり、その親プロセスが現在のプロセスであることがわかる
// またはプロセスが現在のプロセスである場合、タスクがすでにセッションリーダーである場合は
// のエラーが返されます。タスクのセッション ID が現在のプロセスと異なる場合、または
// 指定されたプロセスグループIDのpgidがpidと異なり、セッションIDの
// pgidプロセスグループが現在のプロセスのセッションIDと異なる場合、エラーが
// を返しました。それ以外の場合は、見つかったプロセスの pgrp フィールドに pgid を設定し、0 を返します。
// 見つかったプロセスの
// 指定されたpidを持つプロセスが見つからない場合、リターンプロセスにはエラーコードがありません。
255     for (i=0 ; i< NR_TASKS ; i++)
256         if (task[i] && (task[i]->pid == pid) &&
257             ((task[i]->p_pptr == current) ||
258              (task[i] == current))) {
259             if (task[i]->leader)
260                 return -EPERM;
261             if ((task[i]->session != current->session)
262                 ||
263                 ((pgid != pid) &&
264                  (session of pgrp(pgid) != current->session)))
265                 return -EPERM;
266             task[i]->pgrp = pgid;
267             0を返す。
268         }
269     return -ESRCH;
270 }
// 現在のプロセスのプロセスグループIDを返します。getpgid(0)と同等です。
271 int sys_getpgrp(void)
272 {
273     return current->pgrp;
274 }
275
// セッションを作成し（つまりリーダーを1に設定）、そのセッションID=グループID=プロセスを
// 設定する
// IDです。現在のプロセスが既にセッションリーダーであり、スーパーユーザでない場合、エラー
// を返します。それ以外の場合は、現在のプロセスを新しいセッションリーダーに設定します (leader
// = 1)。このようにして
276 // セッションとグループID pgrpがプロセスpidと同じに設定され、現在のプロセスが
// 制御端末はありません。最後のシステムコールはセッションIDを返しま

```

```

279         return -EPERM;
280     current->leader = 1;
281     current->session = current->pgrp = current->pid;
282     current->tty = -1; // 現在のプロセスには制御端子がありません。
283     return current->pgrp;
284 }
285
286 /*
287 * 補足的なグループID
288 */
289 // 現在のプロセスの他の補助ユーザーグループIDを取得します。
290 // タスク構造体のgroups[]配列には、プロセスが属する複数のユーザーグループのIDが格納されています。
291 // が属しています。配列には、合計でNGROUPS個の項目があります。ある項目の値がNOGROUPの場合（その
292 // が、-1）の場合は、すべてのアイテムの開始後にアイドルになることを意味します。そうでなければ、ユーザ
293 // グループIDは配列アイテムに保存されます。
294 // パラメータ gidsetsize は、ユーザーグループ ID を保存できる最大数です。
295 // ユーザーキャッシュ、つまり、グループリストの最大アイテム数です。グループリストは
296 // これらのユーザーグループ番号を保存するユーザースペースキャッシュ。
297 int sys\_getgroups(int gidsetsize, gid\_t *grouplist)
298 {
299     インティ;
300
301     // まず、グループリストが指すユーザーキャッシュスペースが十分であることを確認してから
302     // 現在のプロセス構造体のgroups[]配列から、ユーザグループIDを1つずつ取得する
303     // とし、ユーザーキャッシュにコピーします。コピー処理中に、groups[] のアイテム数が減少すると
304     // が、与えられたパラメーターgidsetsizeで指定された数よりも大きい場合は
305     // 与えられたキャッシュは、現在のプロセスのすべてのグループを収容するには小さすぎます。操作が
306     // はエラーコードを返します。コピー操作に問題がなければ、この関数は最終的に
307     // コピーされたユーザーグループIDの数を返します。
308     if (gidsetsize)
309         verify\_area(grouplist, sizeof(gid\_t) * gidsetsize);
310
311     for (i = 0; (i < NGROUPS) && (current->groups[i] != NOGROUP);
312          i++, grouplist++) {
313         if (gidsetsize) {
314             if (i >= gidsetsize)
315                 return -EINVAL;
316             put\_fs\_word(current->groups[i], (short *) grouplist);
317         }
318     }
319     return(i); // ユーザーグループ ID の数
320     を返します。 305 }
321
322 // 現在のプロセスが所属する他のセカンダリユーザーグループのIDを設定します。
323 // パラメータ gidsetsize は、設定するユーザーグループ ID の数で、grouplist は
324 // ユーザグループのIDを含むユーザ空間のキャッシュ。
325 int sys\_setgroups(int gidsetsize, gid\_t *grouplist)
326 {
327     inti; 310
328     // まず、パーミッションとパラメータの有効性を確認します。スーパーユーザーのみが修正できる
329     // または、現在のプロセスのセカンダリユーザーグループのIDを設定し、アイテムの数ができない
330     // グループ[NGROUPS]配列の容量を超えています。次に、ユーザグループIDを1つずつコピーします

```



```

// ユーザーバッファから配列へ。gidsetsizeの合計がコピーされます。コピーの数が
// がgroups[]に記入しない場合は、次の項目に-1 (NOGROUP) の値を記入します。最後に
// 関数は0を返します。
311     if (! suser())
312         return -EPERM;
313     if (gidsetsize > NGROUPS)
314         return -EINVAL;
315     for (i = 0; i < gidsetsize; i++, grouplist++) {...
316         current->groups[i] = get fs word((unsigned short *) grouplist);
317     }
318     if (i < NGROUPS)
319         current->groups[i] = NOGROUP;
320     return 0;
321 }。
322
// 現在のプロセスがユーザグループgrpに属しているかどうかをチェックします。Yesであれば1を、そうでなければ0を返します。
323 int in_group_p(gid_t grp)
324 {。
325     intiです。
326
// 現在のプロセスの有効なグループ ID (egid) が grp の場合、そのプロセスは
// grp グループをスキャンした場合、この関数は 1 を返します。それ以外の場合は、プロセスのセカンダリユーザグループである
// grpのグループIDを表す配列です。そうであれば、この関数も1を返します。値を持つアイテムが
// NOGROUPがスキャンされると、有効なアイテムがすべてスキャンされ、一致するグループがないことを意味します。
// idが見つかったので、この関数は0を返します。
327     if (grp == current->egid)
328         を返します。
329
330     for (i = 0; i < NGROUPS; i++) {。
331         if (current->groups[i] == NOGROUP)
332             ブレークします。
333         if (current->groups[i] == grp)
334             を返します。
335     }
336     return 0;
337 }。
338
// utsname構造体は、システムの名前を保持するいくつかの文字列フィールドを含んでいます。これには
// 現在のオペレーティングシステム名、ネットワークノード名 (ホスト
// 名)、現在のオペレーティングシステムのリリースレベル、オペレーティングシステムのバージョン番号、および
// システムが動作しているハードウェアタイプ名。この構造が定義されているのは
// include/sys/utsname.h ファイルを参照してください。ここでは、定数
// インクルード/linux/config.hファイル内の//シンボルです。それらは"Linux", "(none)", "0", "0.12", "i386".
339 static struct utsname thisname = {。
340     uts_sysname, uts_nodename, uts_release, uts_version, uts_machine
341 };
342
// システム名の情報を取得します。
343 int sys_uname(struct utsname * name)
344 {。

```

```
345         int i;  
346  
347         if (!name) return -ERROR;
```

```

348     verify\_area(name, sizeof *name);
349     for(i=0; i<sizeof *name; i++)
350         put\_fs\_byte((char *) & thisname)[i], i+(char *)
            name);
351     0を返す。
352 }
353
354 /*
355 * sethostname; gethostname だけは uname\(\) を呼ぶことで実装できます。
356 */
    // システムのホスト名（ネットワークノード名）を設定する。
    // パラメータ名は、ユーザーデータのホスト名文字列を含むバッファを指します。
    lenはホスト名の文字列の長さです。
357 int sys\_sethostname(char *name, int len)
358 {
359     インティ;
360
    // システムのホスト名はスーパーユーザーのみが設定・変更可能であり、ホスト名の長さは
    // は、最大長のMAXHOSTNAMELENを超えることはできません。
361     if (! suser\(\))
362         return -EPERM;
363     if (len > MAXHOSTNAMELEN)
364         return -EINVAL;
365     for (i=0; i < len; i++) {
366         if ((thisname.nodename[i] = get\_fs\_byte(name+i)) == 0)
367             ブレークします。
368     }
    // コピーが完了した後、ユーザーが指定した文字列にNULLが含まれていなければ
    // 文字、コピーしたホスト名の長さがMAXHOSTNAMELENを超えていない場合は、NULL
    // は、ホスト名の文字列の後に追加されます。MAXHOSTNAMELENの文字数が埋まってしまった場合は、
    // 次のように変更します。
    // 最後の文字をNULLにします。
369     if (thisname.nodename[i]) {...
370         thisname.nodename[i> MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] =
            0;
371     }
372     0を返す。
373 }
374
    // 現在のプロセスのリソース制限を取得します。
    // タスクの構造体に配列rlim[RLIM_NLIMITS]が定義されていて、その境界を制御する。
    // システムがシステムリソースを使用する際に使用されます。配列の各項目は、2つの "rlimit" 構造体です。
    // フィールドがあります。1つは指定されたリソースの現在の制限（ソフトリミット）を指定し、もう1つは
    // は、指定されたリソースに対するシステムの最大制限（ハードリミット）を示します。の各項目は
    // rlim[]配列は、現在のプロセスにおけるリソースの制限情報に対応しています。
    // Linux 0.12システムでは、RLIM_NLIMITS=6という6つのリソースに対する制限があります。を参照してください
    // ファイルinclude/sys/resource.hの41~46行目。「resource」はリソースの名前を指定しています。
    // を参照しています。実際にはタスク構造体の rlim[] 配列のインデックスである。
    // は、取得したリソースを格納するための rlimit 構造体へのユーザバッファポインタです。
    // リミット情報
375 int sys\_getrlimit(int resource, struct rlimit *rlim)
376 {
    // 照会されるリソースは、実際にはプロセス内のrlim[]配列のインデックス値である
    // タスクの構造です。インデックスの値は、もちろんアイテムの最大数よりも大きくてはいけません。
    // を配列RLIM_NLIMITSに設定します。ユーザーバッファが十分であることを確認した後、リソース

```

```
// 構造体の情報をコピーして、0を返す。
```

```

377     if (resource >= RLIM_NLIMITS)
378         return -EINVAL;
379     verify_area(rlim, sizeof *rlim);
380     put_fs_long(current->rlim[resource].rlim_cur,           // 現在の（ソフト）リミッ
                                                                ト
381                 (unsigned long *) rlim)になります。
382     put_fs_long(current->rlim[resource].rlim_max,           // システム（ハード）の限
                                                                界。
383                 ((unsigned long *) rlim)+1)となりま
                                                                す。
384     0を返す。
385 }
386
// 現在のプロセスのリソース制限を設定します。
// パラメータ resource は、制限を設定するリソース名を指定します。これは実際には
// タスク構造体のrlim[]配列のインデックスです。パラメータ rlim は、ユーザバッファ
// カーネルが新しいリソース制限を読み込むための rlimit 構造体へのポインタ。
387 int sys_setrlimit(int resource, struct rlimit *rlim)
388 {
389     struct rlimit new, *old;
390
// まず、パラメータリソース（タスク構造体rlim[]インデックス）の有効性を判断します。
// そして、rlimit構造のポインタ'old'に、現在のrlimit構造である
// 指定されたリソースをユーザーから提供されたリソースの制限情報は、次に
// 一時的なrlimit構造の「new」。このとき、ソフトリミットの値やハードリミットの値が
// 「新しい」構造体のリミット値が元のリミット値よりも大きく、現在の
// がスーパーユーザーでない場合は、パーミッションエラーが返されます。それ以外の場合は、情報
// 新』の//が妥当であるか、プロセスがスーパーユーザーである場合には、で指定された情報を
// 元の処理を「新しい」構造体の情報に変更し、0を返します。
391     if (resource >= RLIM_NLIMITS)
392         return -EINVAL;
393     old = current->rlim + resource; // old = current->rlim[resource].
394     new.rlim_cur = get_fs_long((unsigned long *) rlim);
395     new.rlim_max = get_fs_long((unsigned long *) rlim)+1);
396     if (((new.rlim_cur > old->rlim_max) ||
397         (new.rlim_max > old->rlim_max)) &&
398         ! Suser())
399         return -EPERM;
400     *old = new;
401     0を返す。
402 }
403
404 /*
405  * struct rusuageをtask_structに入れることに意味があると思います。
406  * ただし、そうするとtask_structが非常に大きくなって
    しまいます。その後
407  * task_structがmallocされたメモリに移動されると
    このようにすることに意味があります。残りの情
408 報を移動させることになり ます。
409  * もっとシンプルに !(今はやっていません が、なぜなら私たちは
410 まだ測定していません) 。)
411 */
// 指定されたプロセスのリソース使用情報を取得します。
// このシスコールは、現在のプロセスやその終了した、あるいは待機中の子プロセスのリソース使用量を提供しま
    します。

```

```
// パラメータ'who'がRUSAGE_SELFと等しい場合、現在のリソース使用情報の  
// プロセスが返されます。who' が RUSAGE_CHILDREN の場合、終了した、または待っている子供を返す。  
// 現在のプロセスのリソース使用情報シンボル定数 RUSAGE_SELF および  
RUSAGE_CHILDRENやrusage構造体は、すべてinclude/sys/resource.hで定義されています。
```

```

412 int sys_getrusage(int who, struct rusage *ru)
413 {。
414     構造体ルサージュR
415     符号化されていない     ロング*lp,
*lp, *dest; 416
    // まず、パラメータ'who'で指定したプロセスの有効性をチェックします。もし 'who' がどちらでもない場合
    // RUSAGE_SELF (現在のプロセスを指定)、 RUSAGE_CHILDREN (子を指定) のいずれか。
    // 無効なパラメータコードで返されます。そうでなければ、ユーザバッファを確認した後
    // ポインタruで指定された領域では、一時的なルサージュ構造の領域'r'がクリアされます。
417     if (who != RUSAGE_SELF && who != RUSAGE_CHILDREN)
418         return -EINVAL;
419     verify_area(ru, sizeof *ru);
420     memset((char *) &r, 0, sizeof(r)); // include/strings.h の最後にあります。

    // パラメータ who が RUSAGE_SELF の場合、現在のプロセスのリソース使用情報がコピーされる
    // を r 構造体に格納する。指定されたプロセスの who が RUSAGE_CHILDREN である場合、終了した、または
    // 現在のプロセスの待機中の子リソース使用量が一時的なタスクにコピーされる
    // 構造体r. マクロCT_TO_SECSおよびCT_TO_USECSは、現在のシステムを変換するために
    // ティックを秒とマイクロ秒に分けます。これらはinclude/linux/sched.hというファイルで定義されています。
    // jiffies_offsetは、システムパラメータの誤差調整です。
421     if (who == RUSAGE_SELF) {...
422         r.ru_utime.tv_sec = CT_TO_SECS(current-> utime);
423         r.ru_utime.tv_usec = CT_TO_USECS(current-> utime);
424         r.ru_stime.tv_sec = CT_TO_SECS(current-> stime);
425         r.ru_stime.tv_usec = CT_TO_USECS(current-> stime);
426     } else {
427         r.ru_utime.tv_sec = CT_TO_SECS(current->cutime);
428         r.ru_utime.tv_usec = CT_TO_USECS(current->cutime);
429         r.ru_stime.tv_sec = CT_TO_SECS(current->cstime);
430         r.ru_stime.tv_usec = CT_TO_USECS(current->cstime);
431     }
    // そして、lpポインタはr構造体を指し、lpendはr構造体の終わりを指し、そして
    // ユーザー空間のru構造体へのdestポインタ。最後に、rの情報を
    // ユーザースペースの台詞を入力して0を返す。
432     lp = (unsigned long *) &r;
433     lpend = (unsigned long *) (&r+1);
434     dest = (unsigned long *) ru;
435     for (; lp < lpend; lp++, dest++)
436         put_fs_long(*lp, dest)です。
437     return(0);
438 }。
439
    // システムの現在時刻を取得し、指定されたフォーマットで返します。
    // timeval構造体には、秒とマイクロ秒の2つのフィールド (tv_secとtv_usec) があります。
    // timezone構造体には2つのフィールドがあり、グリニッジ標準時から西に何分離れているかを示します。
    // 時間 (tz_minuteswest) とサマータイム (dst) の調整タイプ (tz_dsttime) です。両方とも
    // 構造体はinclude/sys/time.hファイルで定義されています。
440 int sys_gettimeofday(struct timeval *tv, struct timezone *tz)
441 {。
    // timeval構造体のポインタが空でなければ、現在の時刻 (秒とマイクロ秒) を表す
    // が構造体で返されます。与えられたユーザーのタイムゾーン構造体のポインタが
    // データ空間が空でない場合は、構造体も返されます。コード中のstartup_timeは
    // システムの起動時間 (秒) です。マクロのCT_TO_SECSとCT_TO_USECSは、次のように変換するために使用されま
    // す。

```

```

442     if (tv) {
443         verify\_area(tv, sizeof *tv);
444         put\_fs\_long(startup\_time + CT\_TO\_SECS(jiffies+jiffies\_offset),
445             (unsigned long *)tv)となります。
446         put\_fs\_long(CT\_TO\_USECS(jiffies+jiffies\_offset),
447             ((unsigned long *) tv)+1)となります。
448     }
449     も (tz) {
450         し
451         verify\_area(tz, sizeof *tz);
452         put\_fs\_long(sys\_tz.tz\_minuteswest, (unsigned long *) tz);
453         put\_fs\_long(sys\_tz.tz\_dsttime, ((unsigned long *) tz)+1);
454     }
455     0を返す。
456 }
457 /*
458  * 最初にタイムゾーンを設定する際には、以下のように時計をワープさせま
459  * す。
460  * ローカル                    タイムではなく、GMTタ
461  * イムを刻んでいます。        推定ですが
462  * 誰かがタイムゾーンを設定している場合、私たちはそのタイムゾーンで動
463  * 作しています。
464  * プログラムがタイムゾーンを理解する環境。
465  * これは、ブート時に/etc/rcスクリプトで、次のように行う必要があります。
466  * 一刻も早く、時計を                正しい位置に戻
467  * すことができるように。            そうしないと。
468  * 時計が狂うと様々なプログラムが混乱します。
469  */
470 // システムの現在時刻を設定します。
471 // パラメータ tv は、ユーザデータ領域の timeval 構造体へのポインタ tz は、ポインタ
472 // をユーザデータ領域のtimezone構造体に追加します。この操作には、スーパーユーザーの権限が必要です。
473 // 両方とも空の場合は何もせず、0を返します。
474 int sys\_settimeofday(struct timeval *tv, struct timezone *tz)
475 {
476     static          intfirsttime = 1;
477     voidadjust\_clock();
478     // システムの現在時刻を設定するには、スーパーユーザーの権限が必要です。tzポインタが
479     // が空でない場合は、システムのタイムゾーン情報を設定し、ユーザーのタイムゾーン構造をコピーする
480     // の情報をシステム内のsys_tz構造体に格納します(24行目参照)。もし、システムコールが
481     // が初めてで、パラメータtvのポインタが空でない場合、システムクロックの値を調整します。
482     もし (! サザエさん())
483         return -EPERM;
484     もし (tz) {
485         sys\_tz.tz\_minuteswest = get\_fs\_long((unsigned long *) tz);
486         sys\_tz.tz\_dsttime = get\_fs\_long((unsigned long *) tz)+1);
487         if (firsttime) {
488             firsttime = 0;
489             if (!tv)
490                 adjust\_clock()
491                 です。
492         }
493     }
494 }

```



```
481     }  
    // パラメータの timeval 構造体ポインタ tv が空でなければ、システムクロックが設定される  
    構造体の情報を持つ//。まず、2番目の値（秒）で表されるシステム時間  
    //に加えて、tvで示された位置からマイクロ秒の値（usec）を取得して
```

```

// システム起動時間のグローバル変数startup_timeが第2の値で変更される。
// システムエラー値jiffies_offsetはマイクロ秒の値で設定されます。
482     if (tv) {
483         int sec, usec;
484
485         sec = get\_fs\_long((unsigned long *)tv)です。
486         usec = get\_fs\_long((unsigned long *)tv)+1);
487
488         startup\_time = sec - jiffies/HZ;
489         jiffies\_offset = usec * HZ / 1000000 - jiffies%HZ;
490     }
491     0を返す;
492 }
493
494 /*
495  * CMOSから取得した時間をGMT時間に調整する。
496  * ローカルタイム
497  *
498  これは醜いことですが、                                他の方法よりも望ましいことです。                それ以外
    の場合は
499  * は、/etc/rc でそれを行うためのプログラムを書く必要があるでしょう（そして、そのリスクは
500  プログラムが複数回実行されると、混乱を招く恐れがあります。
501  プログラムが時計を正確にn                                時間ワープさせるのは難しい。
502  *                                カーネルにタイムゾーン情報をコンパイルしま
    す。                                悪い、悪い....
503  *
504  * XXX 現在、                                サマータイムの調整は行っておりません。し
    ない                                場合があります。
505  *は、BIOSの賢さ(馬鹿さ?)に応じて、何かをする必要があります。
506  *     です。Blast it all.... CMOSに依存しないことが一番です。
507  * 時計は一切使用せず、NTPで時間を取得するか、または他の機器を使用している場合は
508  *     ネットワーク.... - TYT, 1/1/92
509  */
    // システムの起動時間をGMTを基準とした時間に合わせる。
    // startup_timeの単位は秒なので、タイムゾーンの分を60倍にする必要があります。
510 void adjust\_clock()
511 {
512     startup\_time += sys\_tz.tz_minuteswest*60;
513 }。
514
    // 現在のプロセスの作成ファイル属性マスクをmask & 0777に設定し、返す
    // オリジナルのマスク。
515 int sys\_umask(int mask)
516 {。
517     int old = current-> umask;
518
519     current-> umask = mask & 0777;
520     return (old);
521 }。
522

```

8.11 vsprintf.c

8.11.1 機能説明

このプログラムでは、主にvsprintf()関数を使って、パラメータを整形してバッファに出力しています。この関数は、Cライブラリの標準的な関数なので、カーネルの動作原理に関わる内容は基本的にはないので、読み飛ばすことができます。カーネルでの応用を理解するためには、コードの後にある関数の説明を直接読むとよいでしょう。また、vsprintf()関数の使用方法については、Cライブラリ関数のマニュアルを参照してください。

8.11.2 コードコメント

プログラム 8-10 linux/kernel/vsprintf.c

```

1  /*
2      *linux/kernel/vsprintf.c
3      *
4      *(C)      1991Linus Torvalds
5      */
6
7  /* vsprintf.c -- Lars Wirzenius & Linus Torvalds. */
8  /*
9      * Wirzeniusはこれを移植可能に書いたが、Torvaldsはそれを台無しにした :-)
10 */
11 // Lars WirzeniusはLinusの友人で、ヘルシンキ大学のオフィスで働いていました。ときに
12 // 1991年の夏にLinuxを開発したリーナスは、当時のC言語にはあまり馴染みがありませんでした。
13 // また、変数パラメータのリスト機能にも精通していませんでした。そこでラース
14 // Wirzeniusは、カーネルがメッセージを表示するためにこのコードを書きました。彼は後に（1998年）、次のよ
15 // うに認めています。
16 // しかし、このコードには1994年になって初めて発見され、修正されたバグがあった。このバグは
17 // は * が出力フィールドの幅として使用されている場合、コードはポインタをインクリメントすることを忘れて
18 // ています。
19 // でアスタリスクをスキップします。このバグはこのコード(130行目)にまだ残っています。彼の個人的なホーム
20 // ページ
21 // は http://liw.iki.fi/liw/
22
23 // #include <stdarg.h> 標準的なパラメータファイルです。で変数パラメータのリストを定義します。
24 // 形式のマクロのことです。主に型(va_list)と3つのマクロ(va_start, va_arg
25 // vsprintf, vprintf, vfprintf関数      では、//and va_end)を使用します。
26 // #include <string.h> 文字列のヘッダファイルです。主に、文字列に関するいくつかの組み込み関数を定義して
27 // います。
28 // 文字列の操作。
29
30 #include <stdarg.h>
31 #include <string.h>
32
33 ctypeライブラリがなくても大丈夫なように、 /* これを使う */。
34 #define is_digit      (c)((c) >= '0' && (c) <= '9')// デジタル文字であるかどうか
35 // をチェックします。
36 // 文字列を整数に変換します。入力、数値の文字列へのポインタです。
37 // ポインタを表示し、結果の値を返します。さらに、ポインタは前方に移動します。
38 static int skip_atoi(const char **s)

```

```
19 {...  
20     int i=0;  
21  
22     while (is\_digit(**s))
```

```

23         i = i*10 + *((*s)++) - '0';
24     iを返す;
25 }。
26
    // ここでは、さまざまな変換タイプのシンボル定数を定義しています。
27 #define ZEROPAD 1/* ゼロで埋める */
28 #define SIGN 2/* 符号付き/符号付きロング */
29 #define PLUS 4/* show plus */
30 #define SPACE 8/* プラスの場合のスペース */
31 #define LEFT 16/* 左揃え */
32 #define SPECIAL 32/* 0x */
33 #define SMALL 64/* 'ABCDEF'の代わりに'abcdef'を使用します */ (注)
34
    // 割り算の演算。入力: nは配当、baseは除数、結果: nは商。
    // となり、この関数は余りを返すことになります。埋め込みアセンブリについては、3.3.2 を参照してください。
35 #define do_div(n,base)
    ({ 36 int res; __asm__ (
37     "divl %4\" : \"%=d\" (n), \"%=d\" (res): \"%0\" (n), \"%1\" (0), \"%r\" (base));
38     res; })
39
    // 整数を指定された基数の文字列に変換します。
    // 入力: num - 整数; base - 基数; size - 文字列の長さ。
    // precision - 数値の長さ (精度)、type - 型のオプション。
    // 出力します。変換された文字列は、バッファ内の str ポインタに格納されます。戻り値は
    // は、数値が文字列に変換された後の文字列の末尾へのポインタです。
40 static char * number(char * str, int num, int base, int size, int precision)
41     , int type)
42 {
43     char c, sign, tmp[36]。
44     const char *digits="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"。
45     int i;
46
    // も 'type' が小文字を示す場合は、小文字のセットが定義されます。もし、'type'
    // が左に調整したいことを示すと、'type' のゼロフィルフラグがマスクされます。もし、ベース
    // が2より小さいか、36より大きい場合は、プログラムを終了します。つまり、このプログラムが処理できるのは
    // 2-32の間のベースを持つ数字。
47     if (type& SMALL) digits="0123456789abcdefghijklmnopqrstuvwxyz "となります。
48     if (type& LEFT) type &= ~ZEROPAD;
49     if (base<2 || base>36)
50         0を返す。
    // 'type' がゼロフィルを示す場合、変数c='0'を設定し、そうでない場合はcをスペースcharと同じに設定します。
    // 'type' が符号付き数値を示し、値numが0より小さい場合は、負の符号
    // が設定され、numは絶対値として扱われます。そうでない場合は、'type' が示す
    // がプラス記号の場合は、sign=plusとし、そうでない場合は、'type' にスペース記号がある場合は、sign=space
    // とします。
    // そうでなければ、符号を0に設定します。
51     c = (タイプ & ZEROPAD) ? '0' : ' ' ;
52     if (タイプ& SIGN && num<0)
    { 53         sign='- 'とします。
54             num = -numです。
55         } else
56             sign=(type& PLUS) ? '+' : ((type& SPACE) ? ' ' : 0);
    // 符号付きの場合、幅の値は1だけデクリメントされます。 タイプが特殊な変換を示す場合

```

続いて、16進数の幅をさらに2（0xの場合）、8進数の幅を1（の場合）減らします。

```

// 結果の前にゼロを置く。)
57     if (sign) size--。
58     if (type&SPECIAL)
59         if (base==16) size -= 2;
60         else if (base==8) size--。
// numが0の場合、一時的な文字列は'0'となり、そうでない場合はnumが文字列に変換されます。
// 与えられた基数に応じてもし、桁数が精度よりも大きい場合は
// 精度は桁数に応じて拡張されます。幅のサイズから数値の数を引いたものが
// 文字が格納されます。
61     i=0;
62     if (num==0)
63         tmp[i++]='0';
64     else while (num!=0)
65         tmp[i++]=digits[do_div(num, base)];
66     if (i>precision) precision=i;
67     size -= precision;

// これ以降、変換結果が徐々に形成され、一時的に配置される
// を文字列strで埋める。type'にゼロフィルと左寄せのフラグがない場合、残りの幅で示される
// Spacesがまずstrに埋められる。符号が必要な場合は、その中に符号記号を格納する。
//
68     if (!(タイプ&(ZEROPAD+LEFT)))
69         while(size-->0)
70             *str++ = ' ';
71     if (sign)
72         *str++ = sign;
// タイプ'が特別な変換を示している場合は、最初の位置に「0」が置かれます。
// 8進法の結果、16進法の場合は「0x」が格納されます。
73     if (type&SPECIAL)
74         if (ベース==8)
75             *str++ = '0';
76         else if (base==16) {
77             *str++ = '0';
78             *str++ = digits[33]; // 'X' 或は 'x'
79         }
// 「type」に左調整フラグがない場合は、c('0'またはスペース)が残りの
// の幅は、51行目を参照してください。
80     if (!(type& LEFT))
81         while(size-->0)
82             *str++ = c;
// この時、iはnumの桁数を保持しています。もし、その桁数が
// 精度、(精度-i)'0'をstr.に入れる。すると、変換された数値文字も
// 記入したストiの合計です。
83     一方(i<precision)
84         *str++ = '0';
85     while(i-->0)
86         *str++ = tmp[i];
// 幅の値がまだゼロより大きい場合は、左に調整があることを意味します。
// 'type' フラグを立てます。そして、残りの幅にはスペースを入れる。
87     while(size-->0)
88         *str++ = ' ';
89     return str ; // 変換された文字列 の末尾へのポインタを返します。90 }

```

```

91 // 以下の関数は、フォーマットされた出力を文字列バッファに送信します。パラメータ fmt
// はフォーマット文字列、args はパラメータリストへのポインタ、buf は出力文字列バッファです。
92 int vsprintf(char *buf, const char *fmt, va_list args)
93 {。
94     int len;
95     int i;
96     char *str; // 変換時に文字列を保持するために使用します。
97     char *s;
98     int *ip;
99
100     int flags; /* フラグからnumber() */
101
102     int field_width; /* 出力フィールドの幅 */
103     intの精度です。 /* min. 整数の桁数、最大
104                     文字列から取得する文字列数 */
105     intの修飾語です。 /* 整数フィールドの場合は、'h'、'l'、'L'のい
                        ずれかになります。
106
107 // まず、文字ポインタがbufに向けられ、次にフォーマット文字列がスキャンされます。
108 // となり、各フォーマット変換命令はそれに応じて処理されます。
109 // フォーマット変換文字列は '%' で始まります。ここでは、fmtフォーマット文字列から '%' をスキャンします。
110 // フォーマット変換文字列の先頭を探すために // を使用します。フォーマットではない通常の文字
111 // ディレクティブはstrに順番に格納されます。
112     for (str=buf; *fmt; ++fmt)
113     { 108         if (*fmt != '%') { 108
114             *str++ = *fmt;
115             を続けています。
116         }
117     }
118 // フォーマット文字列のフラグフィールドを取得し、変数 flags に格納します。
119     プロセスフラグ /* プロセスフラグ
120     flags = 0です。
121     を繰り返しています。
122     ++fmt ; /* これも最初の '%' をスキップします */。
123     スイッチ (*fmt) {
124         case '-': flags |= LEFT; goto repeat;
125         case '+': flags |= PLUS; goto repeat;
126         case ' ': flags |= SPACE; goto repeat;
127         case '#': flags |= SPECIAL; goto repeat;
128         case '0': flags |= ZEROPAD; goto repeat;
129     }
130
131 // 現在のパラメータ幅のフィールド値を field_width 変数に取り込みます。もし、幅が
132 // フィールドが数字の場合は、そのまま幅の値として扱われます。幅のフィールドが文字の場合
133 // '*' は、次が幅を指定していることを意味するので、va_argが呼ばれて幅の値を取ります。
134 // 幅の値が0より小さい場合、負の数はフラグフィールドを持っていることを示す
135 // '-' (左寄せ) なので、フラグ変数に追加してフィールドを取る必要があります。
136 // 絶対値としての幅の値。
137     /* フィールドの幅を取得する */
138     field_width = -1;
139     if (is_digit(*fmt))
140         field_width = skip_atoi(&fmt);
141     else if (*fmt == '*') {...
```



```

130         次の引数です *///ここでのバグ、"++fmt;"を追加すべきです。
131         field_width = va_arg(args, int);
132         if (field_width < 0) {
133             field_width = -field_width;
134             flags |= LEFT;
135         }
136     }
137
138     // 以下は、formatのprecisionフィールドを取り出して、precision変数に入れます。
139     // 精度の高いフィールドの開始を示すフラグは '.' です。処理は幅と同様に
140     // 上記の//フィールドを使用します。精度フィールドが数値の場合は、それがそのまま精度の値として扱われます。
141     // 精度欄が文字 '*' であれば、次のパラメータで
142     // 精度です。そこでva_argを呼んで精度の値を取ります。幅の値が0より小さい場合は
143     // の場合、フィールド精度の値は0となります。
144     /* 精度の取得 */
145     精度=-1です。
146     if (*fmt == '.') {
147         ++fmt;
148         if (is_digit(*fmt))
149             precision = skip_atoi(&fmt)です。
150         else if (*fmt == '*') {...
151             /* 次の引数です */// は ++fmt を追加してください。
152             precision = va_arg(args, int)です。
153         }
154         if (precision < 0)
155             精度=0となり
156             ます。
157     }
158
159     // このコードは、長さ修飾子を解析し、それをqualifier変数に格納します。意味については
160     // h、l、Lの//は、リストの後の説明を参照してください。
161     /* 変換修飾子の取得 */
162     qualifier = -1となります。
163     if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
164         qualifier = *fmt;
165         ++fmt;
166     }
167
168     // 変換フォーマットインジケータは、以下のように分析されます。
169     // インジケータが 'c' の場合は、対応するパラメータが文字であることを意味します。
170     // このとき、フラグフィールドが左寄せではないことを示している場合は、フィールドの前に
171     // を '幅 - 1' 個のスペースで配置してから、パラメータ文字を配置します。もし、幅のフィールドが
172     // それでも0より大きい場合は、左寄せになっていることを意味するので、「幅 - 1」のスペースを後に追加しま
173     // す。
174     // パラメータの文字です。
175     スイッチ (*fmt) {
176     ケース 'c':
177         if (!(flags & LEFT))
178             while (--field_width > 0)
179                 *str++ = ' ';
180         *str++ = (unsigned char) va_arg(args, int);
181         while (--field_width > 0)
182             *str++ = ' ';
183         ブレークします。

```



```
// インジケータが's'であれば、対応するパラメータが文字列であることを意味します。その場合は
// パラメータ文字列の長さが最初に計算され、それが精密度フィールドの値を超えると
// 文字列の長さに等しい拡張精度フィールドを設定します。フラグが示す場合は
// 左寄せではなく、フィールドの前に「幅-文字列の長さ」のスペースがあり、その中に文字列
// が配置されます。それでも幅が0より大きい場合は、左揃えであることを意味しますので、その場合は
// '幅 - 文字列の後に「文字列の長さ」のスペースを入れる。
```

```
169         case 's':
170             s = va_arg(args, char *);
171             len = strlen(s);
172             if (precision < 0)
173                 precision = len;
174             else if (len > precision)
175                 len = precision;
176
177             if (!(flags & LEFT))
178                 while (len < field_width--)
179                     *str++ = ' ';
180             for (i = 0; i < len; ++i)
181                 *str++ = s[i];
182             while (len < field_width--)
183                 *str++ = ' ';
184             ブレークします。
185
186 // の場合 フォーマット文字が'o'の場合は、対応するパラメータが
187 // は
188 // 8進数の文字列に変換されます。number()関数を呼び出して処理します。
```

```
186         case 'o':
187             str = number(str, va_arg(args, unsigned long), 8,
188                 field_width, precision, flags);
189             ブレークします。
```

```
190 // フォーマットコンバータが'p'の場合は、対応するパラメータがポインタ型であることを意味します。
191 // この時、パラメータで幅のフィールドが設定されていない場合、デフォルトの幅は8であり
192 // ゼロを追加する必要があります。その後、処理のためにnumber()関数を呼び出します。
```

```
191     ケース 'p':
192         if (field_width == -1) {...
193             field_width = 8;
194             flags |= ZEROPAD;
195         }
196         str = number(str,
197             (unsigned long) va_arg(args, void *), 16,
198             field_width, precision, flags);
199         ブレークします。
```

```
200 // フォーマットコンバータが'x'または'X'の場合、対応するパラメータが必要であることを意味します。
201 // を16進数で出力します。'x' は小文字を意味します。
```

```
201     case 'x':
202         flags |= SMALL;
203     case 'X':
204         str = number(str, va_arg(args, unsigned long), 16,
205             field_width, precision, flags);
206         ブレークします。
```

```
207 // フォーマット文字が'd'、'i'、'u'の場合、パラメータは整数となります。'd'、'i'のスタンド
```

記号付き整数の場合は、// 符号付きフラグを追加する必要があります。'u'は符号なし整数を表します。

```

208     ケース 'd':
209     ケース 'i':
210         フラグ |=
211             SIGN;
212     ケース 'u':
213         str = number(str, va_arg(args, unsigned long), 10,
214             field_width, precision, flags)。)
215         ブレークします。

// フォーマットインジケータが'n'の場合は、これまでに変換された文字数が
// 対応するパラメータポインタで指定された位置に保存されます。va_arg()を最初に使用します。
// パラメータへのポインタを取得し、変換済みの文字数を格納する
// ポインタが指す位置に。
216     ケース 'n':
217         ip = va_arg(args, int *)。
218         *ip = (str - buf) です。
219         ブレークします。
220

// フォーマットコンバータが'%'でない場合は、'%'が出力文字列に直接書き込まれます。もし
// フォーマット変換の文字が残っている場合、その文字も直接書き込まれる
// を出力文字列に追加し、ループはフォーマット文字列の処理を続けます。それ以外の場合は
// を意味しま がフォーマット文字列の最後まで処理された後、ループを終了します。
221     す。 のデフォルトです。
222         if (*fmt != '%')
223             *str++ = '%';
224         if (*fmt)
225             *str++ = *fmt;
226         その他
227             --fmt;
228         ブレークします。
229     }
230 }
231 *str = '\0'; // 出力文字列の最後にNULLを追加します。
232 return str - buf; // 文字列の長さを返します。
233 }
234

```

8.11.3 インフォメーション

8.11.3.1 vsprintf()のフォーマット文字列

vsprintf()関数は、printf()シリーズのひとつです。これらの関数はすべてフォーマットされた出力を生成します。出力フォーマットを決定するフォーマット文字列fmtを受け取り、パラメータをフォーマット文字列でフォーマットして、フォーマットされた出力を生成します。関数の宣言形式は以下のとおりです。

```
int vsprintf(char *buf, const char *fmt, va_list args)
```

printf()シリーズの他の関数は、これと同様の形式を宣言します。cprintfも出力をコンソールに送ります。fprintfは出力をファイルに送るので、第1引数にはファイルハンドルを指定します。printfに

「v」文字を付けたもの（例えば、`vsprintf`）は、引数の

は、`va_arg` 配列の `va_list args` に由来する。`printf` に接頭辞 '`s`' が付いている場合は、フォーマット結果を文字列バッファ `buf` に出力し（`buf` には十分なスペースが必要）、`null` で終了することを意味する。以下、フォーマット文字列の使い方を詳しく説明します。

1. フォーマット文字列

`printf` ファミリーのフォーマット文字列は、関数の変換、フォーマット、パラメータの出力を制御するために使用されます。各フォーマットには、対応するパラメータが必要で、パラメータが多すぎると無視されます。1つは出力に直接コピーされる単純な文字で、もう1つは対応するパラメータのフォーマットに使用される変換指示文字列です。

2. フォーマット指示文字列

フォーマットインジケータ文字列のフォーマットは以下の通りです。

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

各変換指示文字列は、パーセント記号（%）で始まる必要があります。各部分の意味は以下の通りです。

- `[flags]` は任意のフラグ文字列です。
- `[width]` は任意の幅表示です。
- `[.prec]` はオプションの精度表示です。
- `[h|l|L]` 入力の長さを修飾するオプションです。
- `[type]` は変換型文字（または変換指示子）です。

● フラグは、出力のアライメント、数値シンボル、小数点、末尾のゼロ、2進数、8進数、16進数を制御するもので、上記27～33行目の注釈を参照してください。フラグ文字とその意味は以下のとおりです。

'#' 対応するパラメータを「特殊な形式」に変換する必要があることを示します。8進数（`o`）の場合、変換後の文字列の最初の文字は0でなければなりません。16進数（`x`または`X`）の場合、変換後の文字列は'`0x`'または'`0X`'で始まる必要があります。`e`、`E`、`f`、`F`、`g`、`G`については、たとえ小数点以下の桁がなくても、変換結果には必ず小数点があります。`g` または `G` の場合、末尾のゼロは削除されません。

'0' 変換結果はゼロが付くはずですが、`d`、`i`、`o`、`u`、`x`、`X`、`e`、`E`、`f`、`g`、`G`については、変換結果の左側は、スペースではなくゼロで埋められます。0フラグと-フラグの両方が存在する場合、0フラグは無視されます。数値変換では、精度フィールドが与えられていれば、0フラグも無視されます。

'-' 変換された結果は、対応するフィールドの境界内で左調整（`left`）されます。（デフォルトでは、右調整（`right`）されます）。`n`の変換は例外で、変換結果は右のスペースで埋められます。

'+' 符号付き変換の結果である正の結果の前には、スペースを確保する必要があります。

'+' 記号変換結果の前に、常に記号（+または-）が必要であることを示します。デフォルトのケースでは、負の数だけが負の記号を使用します。

- `width` は、出力文字列の幅を指定するもので、フィールドの最小幅の値を指定する。変換の結

果が指定された幅よりも小さい場合は、左側（または、右調整フラグが与えられている場合は右側）をスペースまたはゼロ（フラグによって決まる）などで埋める必要がある。幅のフィールドを指定するのに数字を使うだけでなく、フィールドの幅が次の整数パラメータで与えられることを示すために '*' を使うこともできる。変換値の幅が `width` で指定された幅よりも大きい場合。

は、いかなる状況でも結果を切り捨てることはありません。フィールドの幅が拡大され、結果が完全に表示されます。

● 精度は、出力の最小数を示す数値である。d、I、o、u、x、およびXの変換では、精度の値は最低の桁数を示します。e、E、f、Fについては、この値は、小数点以下の桁数を示す。gまたはGについては、最大の有効桁数を示す。sまたはS変換の場合、精度値は出力文字列の最大文字数を示す。

● 長さ修飾子の表示は、整数変換後の出力タイプの形態を表します。以下の説明では、「整数値変換」は、d、i、o、u、xまたはXの変換を表しています。

'hh' 後続の整数変換が符号付きまたは符号なしの文字パラメータに対応することを示す。

'h'後続の整数変換が、符号付き整数または符号なし短整数のパラメータ に対応することを示す。

'l'後続の整数変換が、長整数または符号なし長整数の引数 に対応することを示す。

'll'後の整数変換がlong long integerまたはunsigned long long integerの引数 に対応することを示す。

L'e、E、f、F、gまたはGの変換結果が、長い倍精度パラメータに対応することを示す。

● typeは、受け入れられる入力パラメータの種類と出力の形式です。各変換指示子の意味は以下の通りです。

'd,i' 整数のパラメータは符号付き整数に変換されます。精度がある場合は、出力する必要がある最小桁数が与えられます。変換される値の数が少ない場合は、左にゼロになります。デフォルトの精度値は1です。

'o,u,x,X' 符号なし整数は、符号なし8進数(o)、符号なし10進数(u)、符号なし16進数(xまたはX)の表現に変換されます。xは16進数を小文字(abcdef)で表現することを意味し、Xは16進数を大文字(ABCDEF)で表現することを意味します。精度欄がある場合は、出力する必要がある最小桁数を意味します。変換される値の数が少ない場合は、左にゼロになります。デフォルトの精度値は1です。

'e,E' これらの2つの変換文字は、引数を[-]d.dde+ddの形に丸めるために使用されます。小数点以下の桁数は精度と同じです。もし精度フィールドがなければ、デフォルト値の6を取る。もし精度が0であれば、小数は現れない。Eは、インデックスが大文字のEで表されることを意味し、インデックス部分は常に2桁で表されます。値が0の場合、インデックスは00になります。

'f,F' これらの2つの文字は、引数を丸めて[-]ddd.dddという形にするために使われます。小数点以下の桁数は精度と同じです。精度フィールドがない場合は、デフォルト値の6を取ります。精度が0の場合は、小数は表示されません。小数点がある場合は、後ろに少なくとも1桁の数字があります。

'g,G' この2つの文字は、引数をFまたはEの形式に変換します(Gの場合はFまたはEの形式)。精度の値は、整数の数を指定します。精密度フィールドがない場合、そのデフォルト値は6です。精密度が0の場合は1として扱われます。変換時にインデックスが-4より小さいか、精密度以上の場合、e形式が使用されます。小数部の後のゼロは削除されます。小数点が1つ以上ある場合のみ、小数点が表示されます。

'c' 引数を符号なし文字に変換し、その結果を表示することを示す

が出力されます。

's' 入力文字列を指すように要求され、その文字列がnullで終わることを示す。精度フィールドがある場合は、精度に必要な文字数のみが出力され、文字列はNULLで終わる必要はありません。

p] 16進数の数値をポインタとして出力することを示す。

'n' これまでに変換した文字数を、対応する入力ポインタで指定した位置に保存するために使用します。パラメータは変換されません。

'%' パーセント記号が出力され、変換は行われなことを示す。つまり、全体の変換表示が'%%'になったことを示す。

8.12 printk.c

8.12.1 機能説明

printk()は、カーネルが使用する印刷(表示)関数で、C標準ライブラリのprintf()と同じ機能を持っています。このような関数を書き換える理由は、ユーザーモード専用のfsセグメントレジスタをカーネルコードで直接使用することはできず、まず保存する必要があるからです。

fsを直接使用できない理由は、実際の画面表示関数であるtty_write()では、表示すべきメッセージがfsセグメントの指すデータセグメント、つまりユーザプログラムのデータセグメントから取得されるためです。printk()関数で表示すべきメッセージは、カーネルコードで実行した場合、dsレジスタが指すカーネルデータセグメントにある。そのため、printk()関数では、一時的にfsセグメントレジスタを使用する必要がある。

printk()関数は、まず、vsprintf()を用いてパラメータを整形し、次にtty_write()を呼び出して、fsセグメントレジスタの保存時に情報を印刷します。

8.12.2 コードコメント

プログラム 8-11 linux/kernel/printk.c

```

1 /*
2  *linux/kernel/printk.c
3  *
4  *(C)      1991Linus Torvalds
5  */
6
7 /*
8  カーネルモードでは、printfを使うことができません。
9  * 「面白い」ものを指します。fs-savingでprintfを作成して
10 すべてが順調です。
11 */
12 // <stdarg.h> 標準パラメータのヘッダファイルです。変数パラメータのリストを以下の形式で定義します。
    //のマクロです。主に1つの型(va_list)と3つのマクロ(va_start, va_arg and
    vsprintf, vprintf, vfprintf関数 では、//va_end)となっています。
13 // <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。
14 // <linux/kernel.h> カーネルのヘッダファイルです。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。
15 カーネル の機能を利用します。
16 #include <stdarg.h>

```

[13](#) #include <stddef.h>

[14](#)

```

15 #include <linux/kernel.h>.
16
17 static char buf [1024]; // 表示用の一時的なバッ
    ファです。18
19 extern int vsprintf(char * buf, const char * fmt, va list
args); 20
    // カーネルが使用する表示機能。
21 int printk(const char *fmt, ...)
22 {
23     va list args; //は、実際には文字ポインタ型です。
24     int i;
25
    // 最初にパラメータ処理開始関数を実行し、フォーマット文字列fmtで変換します。
    // パラメータリストargsを入力し、bufに出力します。戻り値の i は、パラメータリスト args の長さと同じです
    // 出力文字列を表示します。そして、パラメータ処理の終了関数を実行します。最後に、コンソールに
    // 呼び出されて関数を表示し、表示された文字数を返します。
26     va start(args, fmt)です。
27     i=vsprintf(buf, fmt, args)となります。
28     va end(args)です。
29     console print(buf) ); // chr_drv/console.c, line 995.
30     return i;
31 }。
32

```

8.13 panic.c

8.13.1 機能説明

panic()関数は、カーネルのエラーメッセージを表示したり、システムを無限にデッドループさせるために使用されます。カーネル内の多くの場所で、カーネルコードの実行中に重大なエラーが発生した場合にこの関数が呼び出されます。多くの場合、panic()関数を呼び出すことは、ストレートなアプローチです。このアプローチは、UNIXの "as concise as possible "の原則に従っています。

8.13.2 コードコメント

プログラム 8-12 linux/kernel/panic.c

```

1 /*
2  *linux/kernel/panic.c
3  *
4  *(C) 1991Linus Torvalds
5  */
6
7 /*
8 この関数は、カーネル全体で使用されます (mmやfsも含む)。
9 は、大きな問題であることを示しています。
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 の機能を利用します。

```

// <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、データ初期のタスク0 の、そしていくつかの組み込みアセンブリ関数のマクロのステートメント。

// デスクリプタのパラメータ設定と取得

11 #include <linux/kernel.h> (日本語)

12 #include <linux/sched.h>

13

14 void [sys_sync](#)(void); /* 本当はintです */ // fs/buffer.c, 44行目. 15

// この関数は、カーネルに表示される主要なエラーメッセージを表示して

// ファイルシステムの同期機能を実行してから、無限ループに入る - システムは

// クラッシュします。現在のプロセスがタスク 0 の場合は、スワッパータスクが

のエラーが発生し、ファイルシステムの同期機能がまだ実行されていません。揮発性キーワード

関数名の前の // は、コンパイラ gcc に、その関数がリターンしないことを伝えるために使われます。

// これにより、gccはより良いコードを生成することができ、さらに重要なことに、このキーワードを使用して、誤った

// 警告（初期化されていない変数）を表示します。これは、現在のgccの属性と同じです。

// 'void panic(const char *s) attribute ((noreturn));' 16

volatile void [panic](#)(const char * s)

17 {

18 [printk](#)("Kernel panic: %sn\r", s)。

19 if ([current](#) == [task](#)[0])

20 [printk](#)("In swapper task - not syncing\n\r")。

21 その他

22 [sys_sync](#)() です。

23 for(;;) となっています。

24 }

25

8.14 概要

本章では、linux/kernelディレクトリにある12個のソースファイルを中心に、システムコール、プロセススケジューリング、プロセス複製、プロセス終了処理など、カーネルの中で最も重要な仕組みの実装を紹介しています。

次の章からは、ハードディスク、フロッピーディスク、メモリ仮想ディスクの3つのブロックデバイスをLinuxカーネルがどのようにサポートしているかを学び始めました。まず、ブロックデバイスが使用するデバイス要求アイテムや要求キューなどの重要なデータ構造を説明し、次にブロックデバイスの具体的な動作モードを詳細に説明します。その後、ブロックデバイスのディレクトリにある5つのソースファイルを1つずつハックしていきます。

9Blockデバイスドライバ

オペレーティングシステムの主な機能の1つは、周辺のI/Oデバイスと通信し、統一されたインターフェイスでこれらの周辺デバイスを制御することである。OSのすべてのデバイスは、大きく分けて「ブロックデバイス」と「キャラクターデバイス」の2種類に分けられる。ブロックデバイスとは、ハードディスク装置やフロッピーディスク装置のように、固定サイズのデータブロックを単位としてアドレスやアクセスが可能なデバイスである。キャラクターデバイスとは、キャラクタストリームで動作するデバイスで、アドレス指定はできない。例えば、プリンターデバイス、ネットワークインターフェースデバイス、ターミナルデバイスなどがあります。オペレーティングシステムでは、管理やアクセスを容易にするために、これらのデバイスをデバイス番号で統一して区別している。Linux 0.12カーネルでは、デバイスは7つのクラスに分けられており、主要なデバイス番号は合計7つ（0～6）あります。各タイプのデバイスは、サブ（副、二次）デバイス番号に基づいてさらに区別されます。各機器番号に対応する機器タイプと関連機器を表9-1に示します。この表から、一部のデバイス（メモリ・デバイス）は、ブロック・デバイスとしてもキャラクタ・デバイスとしてもアクセスできることがわかります。本章では、主にブロック・デバイス・ドライバの実装原理と方法について説明します。キャラクター・デバイスについては、次の章で説明します。


表9-1 Linux 0.12カーネルの主要デバイス番号について

メジャーNo.	名前	デバイスタイプ	説明
0	なし	なし	なし
1	ラム	ブロック/チャー	ラムデバイス（仮想ディスク
2	fd	ブロック	フロッピーディスク
3	HD	ブロック	ハードディスク・デバイス
4	ttyx	シャル	デバイス（仮想端末またはシリアル端末
5	tty	シャル	ttyデバイス
6	lp	シャル	lpプリンター

Linux

0.12カーネルは、主にハードディスク、フロッピーディスク、メモリ仮想ディスクの3種類のブロックデバイスをサポートしています。ブロックデバイスは、主にファイルシステムやキャッシュに関係するものなので、本章を進める前に、ファイルシステムの章の内容をざっと見ておくとよいでしょう。本章で扱うソースコードファイルをリスト9-1に示します。

リスト 9-1 linux/kernel/blk_drv

ファイル名サイズ最終更新時刻（GMT） Desc.			
 Makefile	2759	バイト	1992-01-12 19:49:21

blk.h	3963 バイト	1991-12-26 20:02:50
floppy.c	11660 バイト	1992-01-10 03:45:33
hd.c	8331 バイト	1992-01-16 06:39:10
ll_rw_blk.c	4734 バイト	1991-12-19 21:26:20
ramdisk.c	2740 バイト	1991-12-06 03:08:06

この章のプログラムの目的は、2つに分けられます。1つは各デバイスに対応したドライバで、そのようなプログラムとしては、ハードディスクドライバ`hd.c`、フロッピーディスクドライバ`floppy.c`、メモリ仮想ディスクドライバ`ramdisk.c`があります。

もうひとつのクラスには、ブロックデバイス固有のヘッダーファイル`blk.h`があり、これら3つのブロックデバイスが`ll_rw_blk.c`プログラムとやりとりするために、統一された設定と同じデバイス要求開始手順を提供しています。

9.1 主な機能

ハードディスクやフロッピーのブロックデバイスに対するデータの読み書きは、割込みハンドラによって行われます。毎回カーネルが読み書きするデータ量は1論理ブロック（1024バイト）単位で、ブロックデバイスコントローラはセクタ（512バイト）単位でブロックデバイスにアクセスします。処理の際、カーネルはリード&ライトリクエストのエントリ待ち行列を使って、複数の論理ブロックの読み書き動作を順次バッファリングします。

プログラムがハードディスク上の論理ブロックを読み取る必要がある場合、バッファーマネジメントプログラムに申請し、プログラムのプロセスはスリープ待機状態に入ることになる。バッファマネージャは、まず、以前に読み込まれたことがあるかどうか、バッファの中を調べます。既にバッファに存在する場合は、対応するバッファブロックのヘッダポインタが直接プログラムに返され、待機中のプロセスが起こされる。必要なデータブロックがバッファ内に存在しない場合、バッファマネージャは本章の低レベルブロックリード/ライト関数`ll_rw_block()`を呼び出し、対応するブロックデバイスドライバにリードデータブロック操作要求を発行します。この関数は、そのための要求構造アイテムを作成し、要求キューに挿入します。ディスクの読み書きの効率を高め、ヘッドの移動距離を短くするために、カーネルコードはエレベータアルゴリズムを用いて、ヘッドの移動距離が最も小さくなるリクエストキューの位置にリクエストアイテムを挿入します。

この時、ブロックデバイスのリクエストキューが空であれば、そのブロックデバイスが今はビジーではないことを示しています。そして、カーネルは直ちにブロックデバイスのコントローラにリードデータコマンドを発行します。コントローラは、指定されたバッファブロックにデータを読み込むと、割り込み要求信号を発行し、対応するリードコマンド後処理関数を呼び出して、セクタの読み取りの継続処理や要求の終了処理を行います。例えば、対応するブロックデバイスの動作を終了させたり、バッファブロックのデータが更新されたことに関するフラグを設定したり、最後にブロックデータを待つプロセスを起こしたりします。

9.1.1 ブロックデバイスのリクエストとリクエストキュー

上記の説明によると、低レベルのリード/ライト関数`ll_rw_block()`は、リクエストアイテムを介して各種ブロックデバイスとの接続を確立し、リード/ライトのリクエスト操作を発行するものであることがわかる。様々なブロックデバイスに対して、カーネルはブロックデバイステーブル（配列）`blk_dev[]`を用いて管理を行います。各ブロック・デバイスは、ブロック・デバイス・テーブルの1項目を占有します。ブロックデバイステーブルの各ブロックデバイス項目の構造は以下の通りである（下記ファイル`blk.h`参照）。

```
struct blk\_dev\_struct {  
    void (*request_fn)(void); // リクエストの  
    関数ポインタ struct request * current_request; // 現在のリクエスト  
    構造体ポインタ。  
};  
extern struct blk\_dev\_struct blk_dev[NR_BLK_DEV]; // ブロックデバイステーブル (NR_BLK_DEV = 7)。
```

最初のフィールドは、対応するブロック・デバイスのリクエスト・アイテムを操作するために使用される関数ポインタです。例えば、ハードディスク・ドライブの場合は

`do_hd_request()`、フロッピー・デバイスの場合は `do_floppy_request()`

となります。第2フィールドは、現在のリクエスト・アイテム構造体ポインタで、ブロック・デバイスが現在処理しているリクエスト・アイテムを示します。テーブル内のすべての要求項目は、初期化時にNULLに設定されます。

ブロックデバイステーブルは、`init/main.c`プログラムの各デバイスの初期化機能の中で設定されます。リーナス氏は、拡張性を考慮して、ブロックデバイステーブルをメジャーデバイス番号でインデックス化した配列に構築した。Linux 0.12では、表9-2のように7つのメジャーデバイス番号がある。その中で、メジャーデバイス番号1、2、3は、ブロックデバイスである仮想ディスク、フロッピーディスク、ハードディスクに対応している。ブロックデバイス配列の他の項目は、デフォルトではNULLに設定されています。

表9-2 主なデバイスノーと関連する操作機能

主な開発番号	タイプ	説明	リクエスト機能
0	なし	なし	NULL
1	ブロック/ チャー	ラム、メモリーデバイス（ラムディスク など	<code>do_rd_request()</code>
2	ブロック	fd, フロッピーディスク装置	<code>do_fd_request()</code>
3	ブロック	HD、ハードウェア・ディスク・デバイス	<code>do_hd_request()</code>
4	シャル	ttyx デバイス（仮想端末、シリアル端末 など	NULL
5	シャル	ttyデバイス	NULL
6	シャル	lp印刷装置	NULL

`ll_rw_block()`関数は、カーネルがブロックデバイスのリード/ライトなどの操作要求を出すと、パラメータで指定されたコマンドとデータバッファブロックヘッダのデバイスNoに従って、操作関数`do_XX_request()`を使ってデバイス要求アイテムを作成し、エレベータアルゴリズムを使って要求キューに挿入します。関数名の「XX」は、メモリ、フロッピーディスク、ハードディスクのブロックデバイスを表す「rd」、「fd」、「hd」のいずれかです。要求項目キューは、要求項目テーブル（配列）の項目で構成されています。アイテムは全部で32個あります。各リクエストアイテムのデータ構造は以下の通りです。

```

struct request {
    int                dev; // 使用されていないデバイス（-1は空）。
    int                cmd; // コマンド（READまたはWRITE）。
    int                errors; // 動作中に発生したエラーの数。
    unsigned long      sector; // 開始セクタ。（1ブロック=2セクタ）
    unsigned long      nr_sectors; // 読み書き可能なセクタ番号。
    char               *buffer; // データバッファ。
    struct task_struct *waiting; // タスクが操作を待つ場所 struct
    buffer_head *      bh; // バッファヘッダ（include/linux/fs.h, 68）。
    struct request *    next; // 次のリクエストアイテム。
};

extern struct request request[NR_REQUEST]; // リクエスト項目の配列（NR_REQUEST = 32）。

```

各ブロックデバイスの現在の要求アイテムポインタは、要求配列のデバイスの要求リンクリストとともに、そのデバイスの要求キューを構成します。アイテム間では、次のポインタフィールドがリンクリストを形成するために使用されます。したがって、ブロックデバイスのアイテムとそれに関連するリクエストキューは、図9-

1のような構造になります。要求項目が配列とリンクリスト構造を採用している主な理由は、2つの目的を満たすためです。まず、リクエストアイテムの配列構造を利用して、検索時のループ処理を行うことができます。

のように、アイドル要求ブロックに対して検索アクセス時間の複雑さが一定であるため、プログラムを非常に簡潔に書くことができる。第二に、エレベータアルゴリズムの挿入要求項目操作を満足させるために、Linked list構造を採用する必要もある。図9-1に示すように、ハードディスク装置には現在4つの要求項目があり、フロッピーディスク装置には1つの要求項目しかなく、仮想ディスク装置には現在読み書きの要求項目がありません。

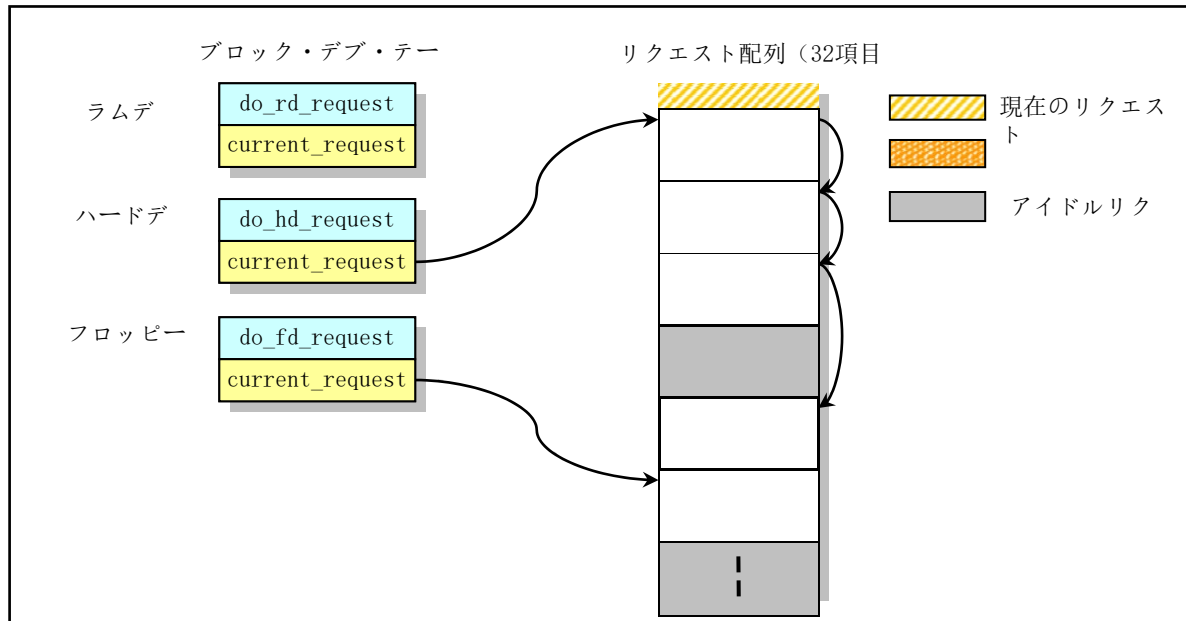


図9-1 デバイステーブルエントリーとリクエスト項目

現在アイドル状態のブロックデバイスに対して、`ll_rw_block()`関数が最初のリクエストアイテムを確立すると、現在のリクエストアイテムポインタ`current_request`は、新しく作成されたリクエストアイテムを直接指し、直ちにリクエスト関数を呼び出して、ブロックデバイスの読み書き操作を開始します。ブロックデバイスがすでに複数のリクエストアイテムのリンクリストを持っている場合、`ll_rw_block()`はエレベーターアルゴリズムを使って、最小ヘッド移動距離の原則に従って、新しく作成したリクエストをリンクリストの適切な位置に挿入します。

また、読み出し操作の優先順位を満たすために、新規要求項目を確立するための要求項目の配列を検索する際に、書き込み操作の空き項目の検索範囲を要求配列全体の最初の2/3に限定し、残りの1/3の要求項目を読み出し操作確立要求に特別に使用します。

9.1.2 ブロックデバイスのアクセススケジューリング

ハードディスクやフロッピーディスクなどのブロック内のデータにアクセスする操作は、メモリにアクセスする操作に比べて時間がかかり、システムのパフォーマンスに影響を与えます。ハードディスクのヘッドシーク動作（リード/ライトヘッドをあるトラックから別の指定トラックに移動させる動作）には時間がかかるため、ハードディスクコントローラに動作コマンドを送る前に、ディスクセクタのアクセス順序をソートする必要がある。すなわち、要求連結リストの各要求項目の順序をソートし、要求項目によってアクセスされるすべてのディスクセクタブロックが順に操作されるようにするのである。Linux 0.12カーネルでは、要求項目はエレベータアルゴリズムを用いてソートされています。動作原理はエレベータの動きに似ています -- 最後の「要求」がその方向の層を止めるまで、

ある方向に移動します。その後、逆方向の移動を行います。ディスクの場合は、ヘッドがディスクの中心に向かってずっと移動し、その逆も同様です。ハードの構造については、図2-11を参照してください。

ディスクを使用しています。

そのため、リクエストアイテムは、受信した順番に処理するためにブロックデバイスに直接送られることはありません。

が、まずリクエストアイテムの順序を最適化する必要があります。そのハンドラを通常I/Oスケジューラと呼んでいる。Linux 0.12のI/Oスケジューラは要求項目をソートするだけですが、現在の一般的なLinuxカーネル（2.6.xなど）のI/Oスケジューラでは、隣接するディスクセクタへの2回のアクセスや複数の要求項目がマージされることもあります。

9.1.3 ブロックデバイスの操作方法

システム（カーネル）とハードディスクの間でIOアクセス操作を行う場合、3つのオブジェクトの相互作用を考慮する必要があります。図9-2に示すように、システム、コントローラ、ドライブ（ハードドライブやフロッピードライブなど）です。システムは、コントローラに直接コマンドを送信するか、コントローラが割り込み要求を発行するのを待つことができます。コマンドを受信した後、コントローラはドライブを制御して、関連する操作やデータの読み取り/書き込み、その他の操作を実行します。したがって、ここではコントローラから送られてくる割り込み信号を3つの間の同期動作信号とみなすことができ、動作手順は以下のようになります。

- まず、コマンド実行による割り込み時にコントローラが起動すべきCファンクションをシステムが指示し、ブロックデバイスコントローラにリード、ライト、リセットなどの操作コマンドを送信します。
- コントローラが指定されたコマンドを完了すると、割り込み要求信号が発行され、システムはブロックデバイスの割り込み処理を実行し、指定されたC関数を呼び出して、これらのコマンドが終了した後、リード/ライトなどの後処理を行います。

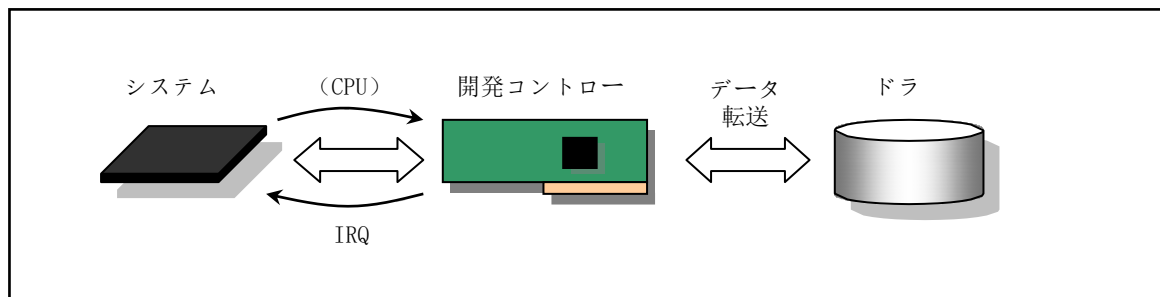


図9-2 システム、ブロックデバイスコントローラ、およびドライブ

ディスクの書き込み操作では、`(hd_out())`を使用して書き込みコマンドを発行した後、システムは、データをコントローラに書き込めるようにするために、コントローラからの応答を待つ必要があります。DRQがセットされると、システムはデータのセクタをコントローラ・バッファに送ることができます。

すべてのデータがドライブに書き込まれたとき（またはエラーが発生したとき）、コントローラは割り込み要求信号も生成し、割り込み処理中にあらかじめ設定されたC関数（`write_intr()`）を実行します。この関数は、まだ書き込むべきデータがあるかどうかをチェックします。もしあれば、1セクタ分のデータをコントローラのバッファに転送し、コントローラが発生させた割り込みを待って、再びデータをドライブに書き込む、これを繰り返していきます。この時、全てのデータがドライブに

書き込まれていれば、この書き込みの終了後にCファンクションが仕上げ作業を行います。要求項目データを待っている関連プロセスのウェイクアップ、要求項目を待っているプロセスのウェイクアップ、現在の要求項目の解放、リンクされたリストからの要求項目の削除、ロックされたバッファの解放。最後に、要求項目操作関数を呼び出して、次の読み書きディスク要求を実行する

の項目（もしあれば）があります。

ディスクの読み出し動作では、セクタの開始位置やセクタ数などの情報を含むコマンドをコントローラに送信した後、コントローラが割り込み信号を発生するのを待ちます。コントローラは、リードコマンドの要求に応じて、指定されたセクタデータをドライブから自分のバッファに渡すと、割り込み要求を発行します。これにより、先にリード操作のために設定されたC関数（`read_intr()`）が実行される。この関数は、まずコントローラのバッファにある1セクタ分のデータをシステムのバッファに入れ、その後、読み取るべきセクタ数をデクリメントしていきます。まだ読むべきデータがある（デクリメントの結果値が0ではない）場合は、引き続きコントローラから次の割り込み信号が送られるのを待ちます。この時点で必要なセクタがすべてシステムバッファに読み込まれていれば、上記の書き込み動作と同じ終了動作を行います。

仮想ディスク装置の場合、その読み書き動作は外部装置との同期動作を伴わないため、上述の割り込み処理はありません。仮想デバイスに対する現在のリクエストアイテムのリード・ライト操作は、`do_rd_request()`で完全に実装されています。

注意点としては、ハードディスクやフロッピーディスクのコントローラに読み書きなどのコマンドを送った後、コマンドを送った関数は、発行されたコマンドの実行を待たずに、すぐに呼び出したプログラムに戻ります。そして最後にブロックデバイス関数`ll_rw_block()`を呼び出した他のプログラムに戻り、ブロックデバイスのIOの完了を待ちます。例えば、キャッシュマネージャのリードブロック関数`bread()`（`fs/buffer.c` 267行目）は、`ll_rw_block()`を呼び出した後、`wait`関数`wait_on_buffer()`を呼び出して、プロセスをすぐにスリープ状態にします。関連するブロックデバイスのIOが終了するまで、`end_request()`関数で状態を覚醒させる。

9.2 blk.h

9.2.1 機能説明

ハードディスクなどのブロックデバイスのパラメータ用のヘッダファイルです。ブロックデバイスでのみ使用されるため、ブロックデバイスのソースファイルと同じ場所に置かれます。主に要求キュー内の要求項目のデータ構造要求を定義し、エレベータ検索アルゴリズムはマクロ文で定義します。仮想ディスク、フロッピーディスク、ハードディスクについては、それぞれのメジャーデバイス番号に応じて、対応する定数値が定義されています。

9.2.2 コードアノテーション

シ ョ ン

プログラム 9-1 linux/kernel/blk_drv/blk.h

```

1 #ifndef BLK_H
2 #define BLK_H
3
4 #define NR_BLK_DEV 7// デバイスタイプ
   の数 5/*
6 * NR_REQUEST は、リクエスト・キューのエントリー数です。
7 * 書き込みは、これらのうち低い方の2/3のみを使用できるように注意してください。
```

8 *が優先されます。

9 *

10 32は妥当な数字だと思います: ある程度の利益を得るには十分です


```

11 エレベーター・メカニズムから、多くの
12 バッファがキューに入っているとき。64は多すぎるようです（簡単に
13 * 重い書き込み/同期が行われているときに、読むのに長い休止が発生する）
14 */
15 #define NR_REQUEST 32
16
17 /*
18 * OK、これは拡張フォームなので、同じ
19 * ページングリクエストが実装された際のリクエストです。で
20 * ページング、'bh'はNULL、'waiting'は待ち時間に使用されます。
21 読み込み/書き込み完了。
22 */
// 以下は、リクエストキュー内のリクエストアイテムの構造です。フィールド dev = -1 の場合は
// は、キュー内のアイテムが使用されていないことを意味します。フィールドcmdには、READ(0)の定数を指定でき
// ます。
// または WRITE(1) (include/linux/fs.h で定義されています)を使用します。また、カーネルでは、待機中の
// ポインタを使用します。代わりに、カーネルはバッファブロックの待ち行列を使用します。なぜなら
// バッファブロックは、リクエストの完了を待つことと同じです。
23 struct request {
24     int dev ; /* -1 if no request */ // device requested.
25     int cmd ; /* READ or WRITE */ (英語)
26     int errors; // エラー の数。
27     unsigned long sector ; // 開始セクター番号
28     unsigned long nr_sectors; // 必要な nr個のセクタ。
29     char * buffer; // データバッファ。
30     struct task_struct * waiting; // タスク の待ち行列。
31     struct buffer_head * bh; // バッファヘッダ (include/linux/fs.h, 73)。
32     struct request * next; // 次のリクエスト を
指します。33 };
34
35 /*
36 * これは、エレベータのアルゴリズムで使用されます。に注意してください。
37 読み込みは常に書き込みよりも先に行われます。これは自然なことです。
38 書き込みよりも*の方がはるかにタイムクリティカルである。
39 */
// 以下のマクロのパラメータs1とs2の値は、リクエスト構造体へのポインターです。
// このマクロは、2つのリクエストアイテムs1とs2の順序を決定するために、リクエストの
// の情報（コマンドcmd（READまたはWRITE）、デバイス番号dev、およびそれらの情報に基づいて、//キュー
// セクター番号 操作されるセクター）。この順番は、次のような順番として使われます。
// ブロックデバイスへのアクセス時に、// 要求項目が実行されます。このマクロは
// function add_request() (blk_drv/ll_rw_blk.c, line 96). このマクロでは、部分的に
// I/Oスケジューリング機能。リクエストアイテムのソート機能を実装している（他の
// はリクエストアイテムのマジ）。)
40 #define IN_ORDER(s1,s2) \
41 ((s1->cmd<(s2->cmd) || (s1->cmd==(s2->cmd) && „^w^„)
42 ((s1->dev < (s2->dev) || ((s1->dev == (s2->dev) && \
43 (s1-> sector < (s2-> sector)))
44
45 // ブロックデバイスの構造
46 struct blk_dev_struct {
47     void (*request_fn)(void); // リクエスト処理機能。
48     struct request * current_request; // 現在
の リクエスト項目。48 };
49

```

```

// ブロックデバイステーブル（配列）。各ブロックデバイスは1項目を占め、合計7項目となります。があります。
// 配列のインデックスは、メジャーデバイスの番号です。次は、リクエストアイテム構造体の配列です。
wait_for_request は、アイドルのリクエストを待つプロセスキューのヘッダーです。
50 extern struct blk_dev_struct blk_dev[NR_BLK_DEV];
51 extern struct request request[NR_REQUEST];
52 extern struct task_struct * wait_for_request;
53
// デバイスデータブロックの総数を示すポインタの配列。各ポインタのエントリは
// 指定されたメジャーの総ブロック数の配列hd_sizes[] (blk_drv/hd.c、62行目) に
// のデバイスです。配列の各項目は、1つのデバイスが所有するデータブロックの総数に対応します。
// マイナーデバイス (1ブロックサイズ=1KB) です。
54 extern int * blk_size[NR_BLK_DEV];
55
// このヘッダーファイルを含むブロックデバイスドライバ (hd.cなど) では、まず、次のように定義します。
// 処理されているデバイスのメジャーデバイス番号です。したがって、正しいマクロ定義は
// 以下の63~90行目で、ドライバーに//をつけることができます。
56 #ifdef MAJOR_NR // 現在のメジャー番号が使用さ
れます。 57
58 /*
59 * 必要に応じてエントリを追加します。現在、ブロックデバイスは
60 ハードディスクとフロッピーに対応しています。
61 */
62
// デバイスがRAMディスクの場合、以下の定数とマクロが使用されます。
63 #if (MAJOR_NR == 1)
64 ラムディスク*1
65 #define DEVICE_NAME "ramdisk"
66 #define DEVICE_REQUEST do_rd_request // リクエストハンドラ。
67 #define DEVICE_NR(device) ((device) & 7) // サブデバイスのnr(0 - 7)。
68 #define DEVICE_ON (device) // オンに
する (ラムディスクには必要ありません)。 69 #define DEVICE_OFF(device)
70
// デバイスがフロッピードライバの場合、以下の定数とマクロが使用されます。
71 #elif (MAJOR_NR == 2)
72 フロッピーディスク*1
73 #define DEVICE_NAME "floppy"
74 #define DEVICE_INTR
do_floppy//デバイスの割り込みハンドラです。 75 #define DEVICE_REQUEST
do_fd_request // リクエストハンドラ。
76 #define DEVICE_NR(device) ((device) & 3) // サブデバイスのnr(0 -
3)。 77 #define DEVICE_ON(device) floppy_on(DEVICE_NR(device))
78 #define DEVICE_OFF(device) floppy_off(DEVICE_NR(device))
79
// デバイスがハードディスクの場合、以下の定数とマクロが使用されます。
80 #elif (MAJOR_NR == 3)
81 ハードディスク*1
82 #define DEVICE_NAME "harddisk"
83 #define DEVICE_INTR
do_hd//デバイスの割り込みハンドラです。 84 #define DEVICE_TIMEOUT
hd_timeout // デバイスのタイムアウト。
85 #define DEVICE_REQUEST do_hd_request // リクエストハンドラ。
86 #define DEVICE_NR(device) (MINOR (device)/5) // ハード
ディスクの番号(0,1)を指定します。
87 #define DEVICE_ON (device) // 常に起動しているハー

```

ドディスクです。[88](#) #define [DEVICE_OFF](#) (デバイス)
[89](#)

```

90 #elif
91 /* unknown blk device */
92 #error "unknown blk
device" 93
94 #endif
95
// プログラミングを容易にするために、ここでは2つのマクロが定義されています。CURRENT は、現在のリクエス
ト項目
// ポインタであり、CURRENT_DEVは現在のリクエスト項目CURRENTのデバイス番号です。
96 #define CURRENT (blk_dev[MAJOR_NR].current_request)
97 #define CURRENT_DEV_DEVICE_NR(CURRENT->dev)
98
// デバイスの割り込み処理シンボルが定義されている場合は、関数ポインタとして宣言され
// デフォルトはNULL
です。99 #ifdef
DEVICE_INTR
100 void (*DEVICE_INTR)(void) = NULL;
101 #endif
// デバイスのタイムアウトシンボルが定義されている場合は、同じ変数に
// 0で、SET_INTR() マクロが定義されています。
102 #ifdef DEVICE_TIMEOUT
103 int DEVICE_TIMEOUT = 0;
104 #define SET_INTR(x) (DEVICE_INTR = (x), DEVICE_TIMEOUT = 200)
105 #else
106 #define SET_INTR(x) (DEVICE_INTR = (x))
107 #endif
// シンボルのDEVICE_REQUESTは、引数なし、戻り値なしのスタティクな関数ポインタであることを宣言していま
す。
108 static void (DEVICE_REQUEST)(void);
109
// 指定されたバッファブロックのロックを解除します。指定されたバッファブロックbhがロックされていない場合
警告
// のメッセージが表示されます。それ以外の場合は、バッファブロックのロックが解除され、待機中のプロセスが
// バッファブロックが起こされます。これはインライン関数ですが、「マクロ」として使用されます。引数の
// は、バッファブロックのヘッダポインタです。
110 extern inline void unlock_buffer(struct buffer_head * bh)
111 {。
112     if (!bh->b_lock)
113         printk(DEVICE_NAME ": free buffer being unlocked\n");
114     bh->b_lock=0となります。
115     wake_up(&bh->b_wait);
116 }。
117
// リクエスト処理の "マクロ "を終了します。パラメータuptodateは、更新フラグです。
// まず、指定されたブロックデバイスをオフにし、その後、リード/ライトバッファが有効かどうかをチェックし
ます。
// 有効であれば、パラメータ値に応じてバッファデータ更新フラグが設定され、バッファの
// のロックが解除されます。パラメータのuptodateの値が0の場合は、その動作が
// リクエストアイテムが失敗したので、関連するブロックデバイスのIOエラーメッセージが表示されます。最後に
// リクエストアイテムを待っているプロセスと、アイドルのリクエストを待っているプロセスの
// アイテムが覚醒し、リクエストアイテムが解除され、リクエストリストから削除されて
// 現在のリクエストアイテムのポインタが次のリクエストに向けられます。
118 extern inline void end_request(int uptodate)
119 {。

```

```
120     DEVICE\_OFF(CURRENT                                ->dev); // デバイス をオフにします。  
121     if (CURRENT->bh                                ) { //現在のリクエスト項目のポインタ。  
122         CURRENT->bh->b_uptodate = uptodate; // フラグを設定します。  
123         unlock\_buffer(CURRENT->bh) です。
```

```

124     }
125     if (!uptodate
126         printk(DEVICE_NAME " I/O error\n|r").
127         printk("dev %04x, block %d\n|r", CURRENT->dev,
128             CURRENT->bh->b_blocknr)となります。
129     }
130     wake_up(& CURRENT
131         アップします。
132     wake_up(& wait_for_request
133         ); // アイドルのリクエスト を待っているプロセスを起こしま
134         す。
135     CURRENT->dev
136     = -1; // リクエストアイテム を解除します。
137     CURRENT = CURRENT
138     ->next; // 次のリクエスト項目
139     を指します。 134 }
140
141 // デバイスタイムアウトシンボルDEVICE_TIMEOUTが定義されている場合、CLEAR_DEVICE_TIMEOUTシンボルは
142 // "DEVICE_TIMEOUT = 0 "と定義されています。それ以外の場合は、CLEAR_DEVICE_TIMEOUTのみを定義します。
143 #ifdef DEVICE_TIMEOUT
144 #define CLEAR_DEVICE_TIMEOUT DEVICE_TIMEOUT = 0;
145 #else
146 #define CLEAR_DEVICE_TIMEOUT
147 #endif
148
149 // デバイス割り込みシンボルDEVICE_INTRが定義されている場合は、CLEAR_DEVICE_INTRシンボルを定義する
150 // "DEVICE_INTR = 0 "として定義され、それ以外は空として定義されます。
151 #ifdef DEVICE_INTR
152 #define CLEAR_DEVICE_INTR DEVICE_INTR = 0;
153 #else
154 #define CLEAR_DEVICE_INTR
155 #endif
156
157 // リクエストアイテムの初期化を定義するマクロです。の初期化が行われないので
158 // ブロックデバイスドライバの最初のリクエスト項目も同様で、一律の初期化を行います。
159 // マクロが定義されています。このマクロは、現在の
160 // リクエストアイテムです。作業内容は以下の通りです。
161 // デバイスの現在の要求項目が空（NULL）の場合は、デバイスには
162 // リクエストアイテムがまだ処理されていない。その後、その作業はスキップされ、対応する関数
163 // を終了します。それ以外の場合は、現在のリクエストアイテムのメジャーデバイス番号が
164 // ドライバで定義されているメジャー番号に // すると、リクエストアイテムのキューが文字化けしてしまい、カー
165 // ネル
166 // エラーメッセージを表示して停止します。それ以外の場合は、リクエストアイテムで使されたバッファブロッ
167 // クが
168 // がロックされていない場合は、カーネルコードに問題があることも示しているので、エラー
169 // のメッセージが表示され、機械も停止します。
170 #define INIT_REQUEST
171
172 if (! CURRENT
173     ) {~~~ no more requests need to be processed.
174     clear_device_intr があります。
175     clear_device_timeout
176     Return; eldest
177 } \
178 if (MAJOR(CURRENT->dev) != MAJOR_NR
179     ) ~~~~~
180     panic(DEVICE_NAME ": request list destroyed"); ~~~~~
181 if (CURRENT->bh) { Яeal
182     if (! CURRENT->bh->b_lock
183         ) ~~~~~ バッファエラー

```

```
159                                panic(DEVICE NAME " : block not locked"); ^w^_ )  
160                                }  
161
```

```

162 #endif
163
164 #endif
165

```

9.3 hd.c

9.3.1 機能説明

hd.cプログラムは、ハードディスク・コントローラー・ドライバです。ハードディスク・コントローラやブロック・デバイスに対する読み書きや、ハードディスクの初期化处理などを行います。プログラム内のすべての機能は、その機能によって5つのカテゴリーに分けられます。

- `sys_setup()`や`hd_init()`など、ハードディスクを初期化したり、ハードディスクが使用するデータ構造情報を設定する関数です。
- ハードディスク・コントローラにコマンドを送信する関数`hd_out()`。
- ハードディスクの現在の要求項目を処理する関数`do_hd_request()`です。
- `read_intr()`、`write_intr()`、`bad_rw_intr()`、`recal_intr()`など、ハードディスクの割り込み処理の際に呼び出されるC関数です。`do_hd_request()`関数は、`read_intr()`や`write_intr()`でも呼び出されます。
- ハードディスクコントローラは、`controller_ready()`、`drive_busy()`、`win_result()`、`hd_out()`、`reset_controler()`などの補助的な関数を動作させます。

`sys_setup()`関数は、`boot/setup.s`から提供された情報を用いて、システムに含まれるハードディスクのパラメータを設定します。その後、ハードディスクのパーティションテーブルを読み込み、起動ディスクのルートfsイメージをメモリ仮想ディスクにコピーすることを試みる。成功した場合は、仮想ディスク内のルートファイルシステムがロードされ、そうでない場合は、通常のルートファイルシステムのロード操作が継続されます。

`hd_init()`関数は、カーネルの初期化時にハードディスク・コントローラの割り込みディスクリプターを設定し、ハードディスク・コントローラの割り込みマスクをリセットして、ハードディスク・コントローラが割り込み要求信号を送れるようにします。

`Hd_out()`は、ハードディスク・コントローラの操作コマンド送信関数です。この関数は、割り込み時に呼び出されるC関数ポインタのパラメータを受け取ります。コントローラにコマンドを送信する前に、まずこのパラメータを使って、割り込み時に呼び出される関数ポインタ（`do_hd`、`read_intr()`など）を事前に設定します。その後、コマンド・パラメータ・ブロックをハードディスク・コントローラ（ポート0x1f0～0x1f7）に所定の方法で送信します。この関数はすぐに戻り、ハードディスク・コントローラがリード/ライト・コマンドを実行するのを待ちません。ハードディスク・コントローラは、コントローラの診断（`WIN_DIAGNOSE`）とドライブ・パラメータの確立（`WIN_SPECIFY`）に加えて、その他のコマンドを受信し、コマンドを実行した後、CPUに割り込み要求信号を送信します。これにより、システムはハードディスクの割り込み処理を行います（`system_call.s`の235行目）。

`do_hd_request()`は、ハードディスクの要求項目の操作関数である。操作の流れは以下の通りです。

- (1) まず、現在の要求アイテムが存在するかどうかを判断します。現在の要求ポインタが空の場合

合は、現在のハードディスクブロック装置に保留中の要求項目がないことを示しているので、直ちにプログラムを終了します。これは、マクロINIT_REQUESTで実行されるステートメントである。そうでなければ、現在の要求項目の処理を続ける。

(2) 現在の要求項目で指定されているデバイス番号の妥当性を検証することと

要求されたディスクの開始セクタ番号

- (3) 現在の要求項目から得られた情報に基づいて、要求データのディスクトラック番号、ヘッド番号、シリンダ番号を計算するステップと
- (4) リセットフラグが設定されている場合は、ハードディスクの再校正フラグも設定され、ハードディスクがリセットされ、「ドライブパラメータの作成」コマンド (WIN_SPECIFY) がコントローラに再送されます。このコマンドでは、ハードディスクの割り込みは発生しません。
- (5) 再校正フラグが設定されている場合、ハードディスク再校正コマンド (WIN_RESTORE) をコントローラに送信し、そのコマンドによる割り込みで実行する必要のあるC関数 (recal_intr()) をあらかじめ設定しておき、ドロップアウトします。recal_intr() の主な機能は、コントローラがコマンドを終了して割り込みを発生させたときに、この関数を再実行することです。
- (6) 現在の要求項目に書き込み操作が指定されている場合、まずC関数に write_intr() を設定して、書き込み操作のコマンドパラメータブロックをコントローラに送信します。コントローラのステータスレジスタは、リクエストサービスフラグ (DRQ) が設定されているかどうかを周期的に問い合わせます。フラグがセットされていれば、コントローラがデータの受信に「同意」したことを示し、リクエストアイテムが指すバッファ内のデータがコントローラのデータバッファに書き込まれます。ループクエリがタイムアウトした後もフラグが設定されていない場合は、操作が失敗したことになります。その後、bad_rw_intr() 関数が呼び出され、現在の要求アイテムを放棄するか、エラーの発生回数に応じてリセットフラグを設定して現在の要求アイテムの再処理を継続するかが決定される。
- (7) 現在の要求項目が読み出し操作の場合、C関数を read_intr() に設定し、コントローラに読み出し操作コマンドを送信します。

write_intr() は、現在の要求項目が書き込み操作の場合、割り込み時に呼び出されるように設定されたC関数です。コントローラが書き込みコマンドを完了すると、直ちにCPUに割り込み要求信号を送信するため、この関数はコントローラの書き込み操作が完了した直後に呼び出されます。

この関数はまず win_result() を呼び出し、コントローラのステータス・レジスタを読み込んでエラーが発生したかどうかを判断する。書き込み動作中にエラーが発生した場合は、bad_rw_intr() が呼び出され、現在の要求の処理中に発生したエラーの数に応じて、現在の要求の処理を継続するために中止するか、現在の要求項目の再処理を継続するためにリセット・フラグを設定する必要があるかが判断される。エラーが発生していない場合は、現在の要求項目に示されている書き込まれるべきセクタの総数に従って、必要なデータがすべてディスクに書き込まれたかどうか判断される。まだディスクに書き込むデータがある場合は、port_write() 関数を使用して1セクタ分のデータをコントローラバッファにコピーします。データがすべて書き込まれた場合は、現在のリクエストの終了を処理するために、end_request() が呼び出されます。リクエストの完了を待っているプロセスをウェイクアップし、(もしあれば) アイドルのリクエストアイテムを待っているプロセスをウェイクアップし、現在のリクエストアイテムによるバッファデータ更新フラグをセットし、現在のリクエストをリリースする (ブロックデバイスリストからアイテムを削除する)。最後に、do_hd_request() 関数の呼び出しを続けて、ハードディスク・デバイス上の他のリクエスト・アイテムの処理を継続する。

read_intr() は、現在の要求項目が読み取り操作の場合、割り込み時に呼び出されるように設定されたC関数です。コントローラは、ハードディスクドライブから指定されたセクタのデータを自身のバッファに読み込んだ後、直ちに割り込み要求信号を送信します。この関数の主な目的は、コントロ

ーラ内のデータをカレントリクエストで指定されたバッファにコピーすることです。

`write_intr()`の起動時と同じように、まず`win_result()`を呼び出してコントローラのステータスレジスタを読み込み、エラーが発生していないかどうかを判断する。ディスクの読み込み中にエラーが発生した場合は、`write_intr()`と同じ処理を行う。エラーが発生していなければ、`port_read()`関数を使って、コントローラのバッファからリクエストで指定されたバッファに1セクタ分のデータをコピーする。その後、セクタの総数に応じて

現在の要求項目に示されている読み出すべきデータがすべて読み出されたかどうかをチェックします。まだ読み取るべきデータがある場合は、次の割り込みが来るのを待つために終了します。データが取得できていれば、`end_request()`関数を呼び出して、現在の要求項目の終了処理を行います：現在の要求項目の完了を待っているプロセスのウェイクアップ、（もしあれば）アイドル要求項目を待っているプロセスのウェイクアップ、バッファデータ更新フラグの設定、現在の要求項目のリリース（ブロックデバイスリストから項目を削除）。最後に、`do_hd_request()`の呼び出しを続けて、ハードディスク・デバイス上の他のリクエスト・アイテムの処理を継続する。

ハードディスクの読み書きの処理をよりわかりやすくするために、これらの機能と割り込み処理、ハードディスクコントローラの実行タイミングの関係を図9-3、図9-4のように示します。

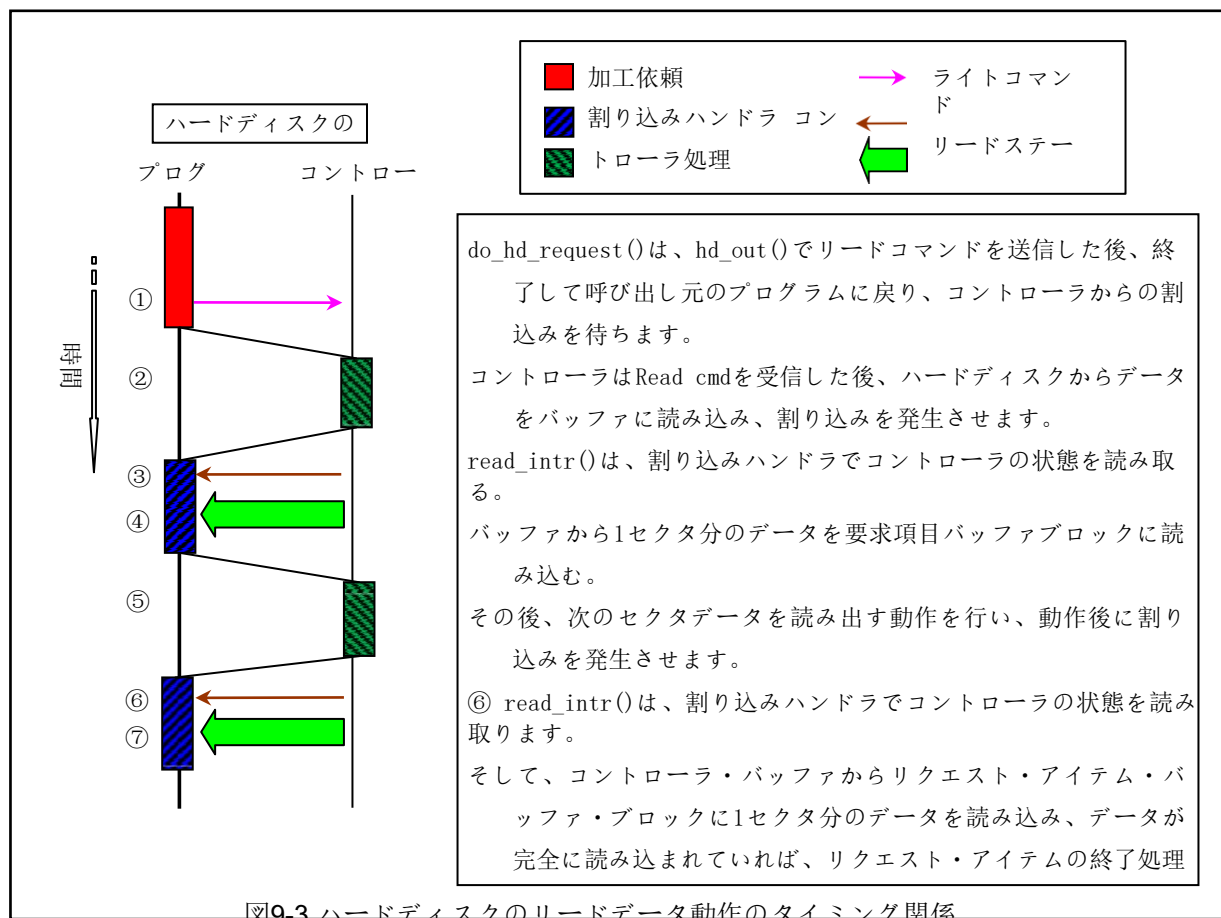


図9-3 ハードディスクのリードデータ動作のタイミング関係

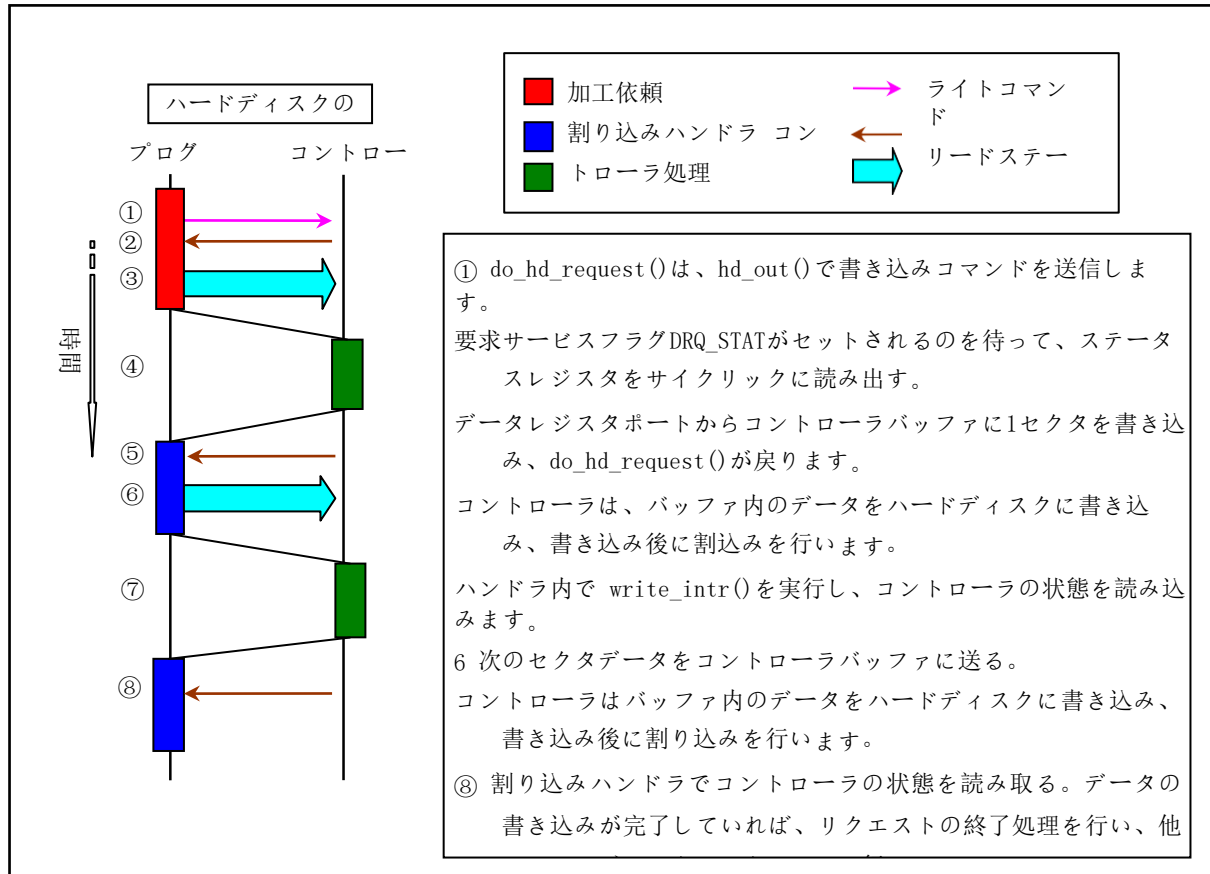


図9-4 ハードディスクの書き込みデータ操作のタイミング関係

以上の分析結果からわかるように、このプログラムで最も重要な4つの関数は、hd_out()、do_hd_request()、read_intr()、write_intr()です。この4つの関数の役割を理解し、ハードディスクドライブの動作プロセスを理解してください。

hd_out()を使ってハードディスクコントローラに読み書きなどのコマンドを送った後、hd_out()関数は発行されたコマンドの実行を待たずに、すぐに呼び出したプログラム、例えばdo_hd_request()に戻るということは、改めて注目すべきことです。do_hd_request()関数は、すぐに呼び出した関数(add_request())に戻り、最後にブロックデバイスの読み書き関数ll_rw_block()を呼び出した他のプログラム(例えば、fs/buffer.cのbread()関数)に戻り、ブロックデバイスのIOの完了を待つ。

9.3.2 コードアノテーション

プログラム 9-2 linux/kernel/blk_drv/hd.c

```

1 /*
2  *linux/kernel/hd.c
3  *
4  *(C)      1991Linus Torvalds
5  */
6
7 /*
8  これは、低レベルのhd割り込みサポートです。これは
9 割り込みを利用して、機能間のジャンプを行います。また
10 すべての機能が割り込みの中で呼び出されていることから、私たちは

```


// ハードディスクの読み込みと書き込みの失敗を処理する関数です。

```

// この要求項目の処理を終了するか、リセットフラグを設定してハードディスクの
// コントローラーのリセット操作をしてから、もう一度やり直してください（242行）。
38 static void bad_rw_intr(void);
39
// Recalibrationフラグ。このフラグがセットされていると、プログラム内で recal_intr() が呼び出され、移動
// ヘッドを0番のシリンダーに
40 static int recalibrate = 0;
// リセットフラグ。このフラグは、読み取りエラーまたは書き込みエラーが発生したときに設定され、関連するリ
// セット関数の
// ハードディスクとコントローラーをリセットするために // が呼び出されます。
41 static int reset = 0;
42
43 /*
44  *この構造体は、HDとそのタイプ を定義します。
45 */
// ハードディスクの情報構造体です。
// フィールドは、ヘッドの数、トラックあたりのセクタの数、シリンダーの数、セクタの数です。
// 書く前のシリンダーnr、頭のシリンダーランディングnr、そして
// 制御バイト。その意味については、プログラムリストに続く命令を参照してください。
46 struct hd_i_struct {
47     ヘッド、セクター、シリンダー、ウェブコム、Lゾーン、CTL。
48 };

// シンボル定数HD_TYPEがinclude/linux/config.hファイルで定義されている場合は
// ここで定義されたパラメータは、ハードディスク情報配列hd_info[]のデータとして扱われる。
// そうでない場合は、まずデフォルト値が0に設定され、setup()関数でリセットされます。
49 #ifdef HD_TYPE
50 struct hd_i_struct hd_info[] = { HD_TYPE };
51 #define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct          )))// count hd
nr. 52 #else
53 struct hd_i_struct hd_info[] = { {0,0,0,0,0}, {0,0,0,0,0}.}; 54
static int NR_HD = 0;
55 #endif
56
// ハードディスクのパーティション構造を定義します。物理的な開始セクタの番号と合計の
// ハードディスクの0トラックから、各パーティションのセクタ数を指定します。の項目があります。
// hd[0]やhd[5]などの5の倍数は、ハードディスク全体のパラメータを表します。
57 static struct hd_struct {
58     long start_sect; // ハードディスク のパーティション開始セクタです。
59     long nr_sects; // パーティション 内のセクタ
// の合計数。60 } hd[5*MAX_HD]={{0,0},};
61
// ハードディスクの各パーティションに存在するデータブロックの総数を表す配列です。
62 static int hd_sizes[5*MAX_HD] = {0, };
63
// ポートを読むインラインアセンブリマクロ。ポートを読んで、nrワードを読んで、bufに保存する。
64 #define port_read(port, buf, nr) \ (^o^ )
65 asm ("cld;rep;insw":: "d" (port), "D" (buf), "c" (nr): "cx",
"di") 66
// 書き込みポートのインラインマクロ。ポートの書き込み、nrワードの書き込み、bufからのデータ取得。
67 #define port_write(port, buf, nr) ♪ ♪ ♪
68 asm ("cld;rep;outsw":: "d" (port), "S" (buf), "c" (nr): "cx",
"si") 69

```



```

70 extern void hd\_interrupt(void); // ハードディスクの割り込みハンドラ (sys_call.s, 235行目)。
71 extern void rd\_load (void); // ラムディスクのロード機能 (ramdisk.c, 71
行目). 72
73 /* これは一度しか使用できません。
// システムセットアップのシスコール関数です。
// 機能パラメータBIOSは、初期化プログラムのinitサブルーチンで設定される
// init/main.cでハードディスクのパラメータテーブルを指定する。ハードディスクのパラメータテーブル構造
// 2つのハードディスクのパラメータテーブル（合計32バイト）の内容がコピーされています。
//をメモリ0x90080から取得しています。0x90080の情報は、setup.sプログラムが
// ROM BIOS機能です。ハードディスクのパラメータテーブルの説明については、表6-4
セクション6.3.3の // を参照してください。この機能の主な目的は、CMOSハードディスクの情報を読み取ることで
す。
ハードディスクのパーティション構造を設定し、RAMの仮想ディスクをロードするために使用されます。
// ルートファイルシステムです。
74 int sys\_setup(void * BIOS)
75 {
76     static int callable = 1; // 呼び出しを一度だけに制限するために使用されます。
77     int i, drive;
78     unsigned char cmos_disks;
79     struct partition *p;
80     struct buffer\_head * bh;
81
// 最初に callable フラグを設定して、この関数が一度しか呼び出されないようにします。次に、ハード
// ディスク情報配列 hd\_info[], シンボルHD_TYPEが既に定義されている場合は
// include/linux/config.hファイルでは、上記49行目でhd\_info[]配列が設定されていることを意味しています。
それ以外の場合は、メモリ0x90080にあるハードディスクのパラメータテーブルを読み込む必要があります。セット
アップ.sの
// プログラムは、1つまたは2つのハードディスクのパラメータテーブルをここのメモリに格納します。
82     if (!callable)
83         1を返す。
84     callable = 0;
85 #ifndef HD_TYPE // HD_TYPEが定義されて いない場合に読みます。
86     for (drive=0 ; drive<2 ; drive++) {。
87         hd\_info[drive].cyl = *(unsigned short *) BIOS; // cylinders.
88         hd\_info[drive]. head = *(unsigned char *) (2+BIOS); // headers.
89         hd\_info[drive].wpcom = *(unsigned short *) (5+BIOS); // write pre-com.
90         hd\_info[drive].ctl = *(unsigned char *) (8+BIOS); // 制御バイト。
91         hd\_info[drive].lzone = *(unsigned short *) (12+BIOS); // ランディングゾーン。
92         hd\_info[drive].sect = *(unsigned char *) (14+BIOS); // セクタ/トラック。
93     BIOS += 16; // 各hdパラメータテーブルは16バイト長
        です。 94 }
// setup.sプログラムがBIOSのハードディスクのパラメータテーブル情報を取得する際に、もしそれが
システムにハードディスクが1台しかない場合、2台目のハードディスクに対応する16バイトは
をクリアします。そのため、2つ目のハードディスクのシリンダーの番号があるかどうかを確認すると
// 0であれば、2台目のハードディスクがあるかどうかわかります。
95     if (hd\_info[1].cyl)
96         NR\_HD =2; // ハードディスクのnrが2に 設定されています。
97     その他
98         NR\_HD=1となります。
99 #endif

// この時点で、ハードディスク情報配列hd\_info[]が設定されており
// ハードディスクのNR_HDが決まります。ここで、ハードディスクのパーティション構造を設定します 配列
hd[], Items

```

配列の0と5は、2つのハードディスクの全体的なパラメータを表し、項目
// 1-4と6-9は、2台のハードディスクの4つのパーティションのパラメータを表しています。

ハードの全体的な情報を示す2つの項目（項目0と5）のみが、それぞれ
// ディスクはここにセットされます。

```

100     for (i=0 ; i< NR\_HD ; i++) {...
101         hd[i*5].start_sect = 0; // 開始セクター番号。
102         hd[i*5].nr_sects = hd\_info[i].head*
103             hd\_info[i].sect*hd\_info [i].cyl; // のセクター
                トータルnr                になります。
104     }
105
106     /*
107         ハードディスクについてはCMOSに問い合わせてみましたが、も
            しかしたら
108         SCSI/ESDI/etcのコントローラを持っていて、それがBIOS
109         ST-506との互換性があるため、当社でも表示されます。
110         BIOSテーブルではなく、レジスタコンパティブルではないため
111         は、CMOSでは存在しません。
112
113         さらに、ST-506のドライブを想定しています。
114         <ある場合>は、システムのプライマリ・ドライブであり
115         ドライブ1や2として反映されるもの
116
117         最初のドライブは、CMOSのハイニブルに格納されます。
118         バイト0x12、下位                ニブルの2番目です。    これは、次のようにな
            ります。
119         4 ビットのドライブタイプまたは使用を示す 0xf のいずれか バイト 0x19
120         8ビットタイプの場合、CMOSではドライブ1、ドライブ2は0x1aとなります。
121
122         言うまでもなく、ゼロでない値は、我々が
123         そのドライブに対応するATコントローラーのハードディスクを
124
125     */
126
127     上記の原則に基づき、ここでは、ハードディスクが
    // ATコントローラーのことです。CMOS情報の説明は7.1.3項を参照してください。ここでは、ハード
    CMOSオフセットアドレス0x12からディスクタイプバイトを読み出し、下位ニブルの値（2番目の
    ハードディスクの種類）が0でない場合は、システムに2つのハードディスクがあることを意味し、そうでない場合
    はシステムの
    //には1つのハードディスクしかありません。0x12から読み取った値が0であれば、AT互換機がないことを意味しま
    す。
    // システムのハードディスク。
128     if ((cmos\_disks = CMOS\_READ(0x12)) & 0xf0)
129         if (cmos\_disks & 0x0f)
130             NR\_HD=2となります。
131         その他
132             NR\_HD = 1となります。
133     その他
134         NR\_HD = 0です。
    // NR\_HD = 0の場合、2つのハードディスクはATコントローラとの互換性がなく、データの
    // 2つのハードディスクの構造がすべてクリアされます。NR\_HD = 1の場合、2つ目のハードディスクのパラメータ
    は
    // ハードディスクがクリアされます。
135     for (i = NR\_HD ; i < 2 ; i++) {。
136         hd[i*5].start_sect = 0;

```

```
137         hd[i\*5].nr_sects = 0;  
138     }
```

// OK、ここまででシステムに含まれるハードディスクの数NR_HDがはっきりとわかりました。

// では、各ハードディスクの第1セクターにあるパーティションテーブル情報を読み込んでみましょう。
パーティション構造の中で、ハードディスクの各パーティションの情報を設定するためのものです。

// 配列 hd[], まず、ハードディスクの最初のデータブロック (fs/buffer.c, 267行目)を読み込んで最初のパラメータ (0x300, 0x305) はデバイス番号です。
 2つのハードディスクのうちの // を、2番目のパラメータ (0) を、読み取るべきブロック番号とします。もし読み込み操作が成功すると、データはバッファブロックのデータ領域に格納されます。
 // bhです。バッファブロックヘッドポインタbhが0の場合、読み込み操作が失敗したことを意味し、その場合は// エラーメッセージが表示され、機械が停止します。それ以外の場合は、その有効性を判断して第1セクタの最後の2バイトが0xAA55であるかどうかによって、// データを表示することができます。
 セクター内のオフセット0x1BEの先頭にあるパーティションテーブルが、// be known
 // 有効です。有効であれば、ハードディスクのパーティションテーブル情報を
 // パーティション構造の配列hd[]。最後にbhバッファを解放します。

```

139     for (drive=0 ; drive<NR_HD ; drive++) {...
140         if (! (bh = bread(0x300 + drive*5, 0)))           { // 0x300, 0x305はdev no.
141             printk("Unable to read partition table of drive %d\n|r")となりま
                す。
142                 ドライ
                    ブ) を
                    搭載し
                    ていま
                    す。)
143             panic("")です。
144         }
145         if (bh->b_data[510] != 0x55 || (unsigned char))
146             bh->b_data[511] !=0xAA                      ) { // チェックフラグ 0xAA55
147             printk("Bad partition table on drive %d\n|r", drive)。
148             panic("")です。
149         }
150         p = 0x1BE + (void                                *)bh->b_data; // パーティションテーブル
            は0x1BE                                         にあります。
151         for (i=1; i<5; i++, p++) {。
152             hd[i+5*drive].start_sect = p->start_sect;
153             hd[i+5*drive].nr_sects = p->nr_sects;
154         }
155         brelse                                           (bh); // バッファを解放する。
156     }
// ここで、各パーティションのデータブロックの総数を数え、ハードディスクに保存する
// 総データブロック配列を分割 hd_sizes[], そして、ブロックのデバイス項目 (3) に、ブロックの
// サイズ配列は、この配列を指します。
157     for (i=0 ; i<5*MAX_HD ; i++)
158         hd_sizes[i] = hd[i].nr_sects>>1 ;
159     blk_size[MAJOR_NR] = hd_sizes                      ; // MAJOR_NR = 3

```

// これでようやく、ハードディスクのパーティション構造の配列を設定する作業が完了しました。
 // hd[], ハードディスクが存在し、その情報がパーティションテーブルに読み込まれている場合。
 // OKメッセージが表示されます。次に、ルートfsイメージのロードを試みます (blk_drv/ramdisk.c, line
 // 71) をメモリ・ラム・ディスクに入れています。つまり、システムに仮想ラムディスクが用意されている場合で
 す。
 起動ディスクにルートfsのイメージデータが含まれているかどうかを判断します。もし
 //です (この時、起動ディスクは統合ディスクと呼ばれています)、ロードして保存するようにしてください。
 // イメージをラムディスクに格納し、ルートfsのデバイス番号ROOT_DEVを、デバイス
 // ラムディスクの番号。その後、スイッチングデバイスが初期化されます。最後に、ルートをインストールします
 // ファイルシステムです。

```

160     if (NR_HD)
161         printk("Partition table%s ok. |n|r", (NR_HD>1)? "s": "");
162     rd_load                                             (); // blk_drv/ramdisk.c, line
71.

```



```

// ハードディスクコントローラのステータスレジスタポートHD_STATUS (0x1f7)を読み、ループで検出する。
// ドライブレディビット (ビット6) がセットされ、コントローラビジービット (ビット7) がリセットされた場合。
// もし、リターン
// retriesの値が0の場合、コントローラのアイドル待ち時間がタイムアウトしたことを意味します。
// 返却値が0でない場合、コントローラはアイドル状態に戻ります。
// 待機 (ループ) 時間帯に、OK!
// 実際には、ステータス・レジスタのビジー・ビット (ビット7) が1であるかどうかを確認するだけで、判断する
// ことができます。
// コントローラがビジー状態の場合は、//。ドライブの準備ができているかどうか (すなわち、ビット6が1であるか
// どうか) は、以下とは無関係です。
// コントローラの状態です。そこで、172行目の記述を次のように書き換えることができます。
// while (--retries && (inb_p(HD_STATUS)&0x80));
// さらに、現在のPCのスピードは非常に速いので、待ち時間を増やすことで
// サイクルは、例えば10倍!
168 static int controller_ready(void)
169 {。
170     int retries = 100000;
171
172     while (--retries && (inb_p(HD_STATUS)&0xc0) != 0x40);
173     return (retries);
174 }。
175
// コマンド実行後にハードディスクの状態を確認する (winはWinchester hdの略)。
// コマンド実行結果のステータスをステータスレジスタに読み出す。0を返すことは
// 正常; 1はエラーを意味します。実行コマンドが間違っている場合は、エラーレジスタを読み取る必要があります
// HD_ERROR (0x1f1) です。
176 static int win_result(void)
177 {。
178     int i=inb_p(HD_STATUS) ); // ステータ
// スを取得する。179
180     if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT))
181         == (ready_stat | seek_stat))
182         return(0); /* ok */
183     if (i&1) i=inb(HD_ERROR) ); // ERR_STATが設定されている場合は、HD_ERROR を読み込
// む。
184     return (1);
185 }。
186
//// ハードディスク・コントローラにコマンド・ブロックを送信します。
nsect - 読み書き可能なセクタの数。
// sect - 開始セクター、head - ヘッド番号、cyl - シリンダー番号。
// cmd - コマンドコード (コントローラのcmdリストを参照); intr_addr() - C関数ポインタ。
// ハードディスクコントローラの準備が整った後、この関数はグローバル関数ポインタ変数
// ハードディスクの割り込みハンドラで呼び出されるCハンドラを指すように // do_hd を設定し
// その後、ハードディスクのコントロールバイトと7バイトのパラメータコマンドブロックを送信します。ハードディ
// スクの
// 割り込みハンドラは、ファイルkernel/sys_call.sの235行目にあります。
// 191行目では、レジスタ変数resを定義しています。この変数は、レジスタに保存されて
// クイックアクセスです。レジスタ (eaxなど) を指定したい場合は、次のような文章を書きます。
// "register char res asm("ax");" のようになります。
187 static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect.)
188     符号化されていない整数のヘッド、符号化されていない整数のシリンダー、符号化されていない
// 整数のコマンド。
189     void (*intr_addr)(void))

```

[190](#) {。

[191](#) register int port asm("dx "); // レジスタ変数を定義しま

す。 [192](#)

// まず、パラメータの有効性を確認します。ドライブ番号が1より大きい場合（0のみ。

// 1が有効）、または頭番号が15より大きい場合は、プログラムがサポートしていないため、停止します。それ以外の場合

// チェックして、ドライブの準備が整うのを待つループです。を待っても準備ができていない場合は
と表示されて停止してしまいます。

```
193     if (drive>1 || head>15)
194         panic ("Trying to write bad sector")が発生します。
195     if (! controller\_ready())
196         panic ("HD controller not ready")と表示されます。
```

// 次に、ハードディスクの割り込み時に呼び出されるC関数ポインタdo_hdを設定します。
// が発生します（関数ポインタはファイルblk.hの56～109行目で定義されています、特に注意してください）。
// 83-100行目に注目。）その後、ハードディスクコントローラのcmdポートに制御バイトを送る
// (0x3f6)を使って、指定されたドライブの制御モードを確立することができます。この制御バイトは、ctl
ハードディスクの情報構造体の配列の中の//フィールドです。その後、7バイトのパラメータコマンド
// ブロックはコントローラポート0x1f1-0x1f7に送信されます。

```
197     SET\_INTR                                (intr_addr); // do_hd = intr_addr
198     outb\_p(hd\_info[drive].ctl,HD\_CMD          ); // 制御バイト   を出力。
199     port=HD\_DATA; // (0x1f0)
200     outb\_p(hd\_info[drive].wpcom>>2,++port); // Param: write pre-comp. (4で割る)
201     outb\_p                                (nsect,++port); // Param: r/wセクタ      の合計nr。
202     outb\_p                                (sect,++port); // Param: 開始セクター。
203     outb\_p                                (cyl,++port); // Param: cylinder nrの下位8ビット
        です。
204     outb\_p                                (cyl>>8,++port); // Param: cylinder nr high 8 bits.
205     outb\_p(0xA0|(drive<<4)|head              ,++port); // Param: drive no + head no.
206     outb                                (cmd,++port); // Param: Hard
disk command. 207 }
```

```
208
    ///// ドライブの準備が整うのを待ちます。
    // この関数は、メイン・ステータス・レジスタのビジー・フラグがリセットされるのを待つためにループします。
    もし、レディ
    // またはシーク終了フラグがセットされていれば、ハードディスクの準備が整い、成功していれば
    // 0. しばらくしてもビジー状態であれば1を返します。
```

```
209 static int drive\_busy(void)
```

```
210 {。
```

```
211     符号なし整数 i;
```

```
212     unsigned char c;
```

```
213
```

// コントローラのマスター・ステータス・レジスタHD_STATUSは、サイクリックに読み込まれます。
レディフラグがセットされ、ビジービットがリセットされます。その後、ビジービット、レディビットを検出しま
す。
//とシークエンドビットを設定します。準備完了フラグまたはシークエンドフラグのみがセットされている場合は
ハード
// ディスクの準備ができたので0を返します。そうでなければ、最後にタイムアウトが切れたことを意味する
ループの // のため、警告メッセージが表示され、1が返されます。

```
214     for (i = 0; i < 50000; i++) {。
215         c = inb\_p(HD\_STATUS                    ); // メインのステータスバイトを取得します。
216         c &= (BUSY\_STAT | READY\_STAT | SEEK\_STAT);
217         if (c == (READY\_STAT | SEEK\_STAT))
218             0を返す。
219     }
220     printk ("HD controller times out\n|r").
221     return(1);
222 }。
```

```
223
```

///// ハードディスク・コントローラーの診断リセット。

// イネーブルリセット（4）の制御バイトは、まずコントロールレジスタポート（0x3f6）に送られて
その後、コントローラがリセット操作を行うまでの時間を待ちます。通常の
// 制御バイト（再試行、再読込許可）をポートに送信し、ハードディスクを待つ

//の準備ができました。ハードディスク準備完了タイムアウトまで待つと、ビジー警告メッセージが表示されます
 // その後、エラーレジスタの内容を読み取ります。それが1になっていない場合（1はエラーがないことを意味する）。

// ハードディスクコントローラのリセット失敗のメッセージが表示されます。

```

224 static void reset_controller(void)
225 {...
226     inti; 227
228     outb(4, HD_CMD); //4はリセットバイト。
229     for(i = 0; i < 1000; i++) nop (); // しばらく 待つ。
230     outb(hd_info[0].ctl & 0x0f, HD_CMD); // 通常の制御バイト（再試行、再読  を許可する）。
231     if (drive_busy())
232         printk("HD-controller still busy|\r").
233     if ((i = inb(HD_ERROR)) != 1)
234         printk("HD-controller reset
failed: %02x\n\r", i); 235 }.
236
237 // ハードディスクのリセット操作。
238 // 最初にハードディスクコントローラをリセットしてから、ハードディスクコントローラのコマンド「Create
239 // ドライブのパラメータ」です。この関数は、引き起こされたハードディスクの割り込みハンドラで再び呼び出さ
240 // れます。
241 // をこのコマンドで実行します。この時点で、この関数はエラー処理を行うかどうかを決定します。
242 // を実行した結果に基づいて、リクエストアイテムの処理を継続するか、あるいは
243 // コマンド。
244 static void reset_hd(void)
245 {。
246     static int i;
247
248     // リセットフラグが設定されている場合は、ハードディスクコントローラのリセット操作を行った後に
249     // リセットフラグをクリアします。その後、コントローラに「ドライブパラメータ作成」コマンドを送信します。
250     i番目のハードディスクのための//。コントローラがこのコマンドを実行すると、ハードディスクの
251     // 割り込み信号を表示します。この時点で、この関数は割り込みハンドラによって再び実行されます。
252     // この時点でリセットフラグがリセットされているので、246行目からのステートメントでは
253     コマンドの実行がOKかどうかを判断するために、最初に // が実行されます。それでもエラーが発生した場合は
254     // bad_rw_intr()関数が呼び出され、エラーの数をカウントして、リセットされたかどうかを判断します。
255     // フラグはエラー回数に応じて再度設定されます。リセットフラグが再度設定された場合は、ジャンプする
256     この機能を再実行するには、ラベル repeat に // を入力します。リセット操作がOKであれば、「Create
257     次のハードディスクに対して「//Drive Parameter」コマンドを送信し、上記と同様の処理を行います。
258     // 上記が実行されます。システム内のすべてのハードディスクが、送られてきたコマンドを正常に実行した場合。
259     // do_hd_request()関数が呼び出されます。 を再度実行し、リクエストアイテムの処理を開始
260     します。
261     を繰り返しています。
262     if (reset) {
263         reset = 0
264         となりま
265         す。
266         i = -1; // 現在のHD番号を初期化します。
267         reset_controller()です。
268     } else if (win_result()) {。
269         bad_rw_intr()です。
270         if (reset)
271             goto リビート。
272     }
273     i++; // 次のHDを処理し
274     if (i < NR_HD) {...
  
```

```
253         hd_out(i, hd_info[i].sect, hd_info[i].sect, hd_info[i].head-1,  
254             hd_info[i].cyl, WIN SPECIFY, & reset_hd) 。 )  
255     } else
```

```

256         do\_hd\_request                                ( ); // リクエスト項目の処理。
257     }
258     ///// 予期せぬハードディスクの割り込みで呼び出されるデフォルトの関数です。
    // この関数は、ハードディスクの割り込みハンドラの中で以下の場合に呼び出されるデフォルトのC
    関数です。
    // 予期せぬハードディスクの割り込みが発生します。この関数は、呼び出された関数が
    // ポインタはNULLです。kernel/sys_call.s の 256 行目を参照してください。この関数は、リセッ
    トフラグが
    // 警告メッセージを表示した後、引き続きリクエストアイテム関数を呼び出す
    // go_hd_request()を行い、その中でリセット処理動作を行う。
259 void unexpected\_hd\_interrupt(void)
260 {
261     printk("Unexpected HD interruptn\r").
262     reset = 1となります。
263     do\_hd\_request()を行います。
264 }
265
    ///// ハードディスクのリード/ライト障害を処理する機能です。
    // 読み取りセクタ操作でのエラー数が7以上の場合、現在の
    // リクエストアイテムが終了し、リクエストを待っていたプロセスが覚醒して
    // 対応するバッファ更新フラグがリセットされ、データが更新されないことを示します。もしこのフ
    ラグが
    // セクターの読み書きのエラー回数が3回以上になったことがある。
    // は、コントローラのリセット操作（リセットフラグの設定）を行うために必要です。
266 static void bad\_rw\_intr(void)
267 {
268     if (++CURRENT->errors >= MAX\_ERRORS)
269         end\_request(0)です。
270     if (CURRENT->errors > MAX\_ERRORS/2)
271         reset = 1となります。
272 }
273
    ///// 割り込みで呼び出されたリードセクタ機能。
    // この関数は、終了時に発生するハードディスクの割り込み時に呼び出されます。
    ハードディスクのリードコマンドの//。の後、コントローラは割り込み要求信号を生成します。
    // リードコマンドが実行され、割り込みハンドラの実行がトリガーされます。このとき
    割り込みハンドラのC関数ポインタdo_hdがread_intr()を指していたので、 // ポイント
    セクターの読み込みが完了した後（またはエラーが発生した後）に、 // 関数が実行されます。
274 static void read\_intr(void)
275 {。
    この機能は、まずリードコマンドの操作がエラーになっているかどうかを判断します。もしコントローラが
    コマンド終了後も // がビジー状態であったり、コマンド実行エラーが発生した場合は、ハード
    ハードディスクの動作不良の問題を処理し、再度ハードディスクに
    // 処理をリセットし、他のリクエスト項目の実行を継続して、リターンします。
    //
    // 関数bad_rw_intr()では、読み取り操作のエラーが発生するたびに、エラーの数を累積する
    // 現在のリクエストアイテムでエラーの数が最大数の半分以上の場合は
    許可されている場合は、まずハードディスクのリセット操作が行われ、その後、リクエストの
    //処理が実行されます。エラーの数が上記の値以上であれば
    // 最大許容エラー数 MAX_ERRORS (7回)、このリクエスト項目の処理は
    //が終了し、キューの中の次のリクエストアイテムが処理されます。
    // do_hd_request()関数では、リセットやキャリブレーションなどが必要かどうかを判断します。
    その時の特定のフラグの状態に応じて実行されるように、 // 次の要求
    は継続または処理されます。

```

276if (win_result

()) {//エラーがあったら...

```

277         bad\_rw\_intr\(\) です。 // r/w 失敗の処理。
278         do\_hd\_request\(\) を行いま // リクエストアイテムの処理を継続し
279         す。 // ます。
280         を返すことができます。
    }

    // 読み込み操作にエラーがなければ、1セクタ分のデータをデータ
    // レジスタポートをリクエストのバッファに入れ、読み取るべきセクタ番号をデクリメントする。
    // デクリメントしても0にならない場合は、これに読み込まれるデータがあることを意味します。
    // 要求があるので、割り込みC関数のポインタは再び ead\_intr\(\) を指すように設定され、戻ります。
    // 直接、別のセクタデータを読み込んだ後、ハードディスクが再び割り込みをかけるのを待ちます。
    注1: 281行目の256番は、512バイトのメモリワードを意味します。
    // 注2: 262行目のステートメントでは、再びdo\_hdポインタをread\_intr\(\)に設定しています。
    // ハードディスクの割り込みハンドラは、 do\_hd が呼ばれるたびに関数ポインタを NULL に設定する。
kernel/sys_call.sファイルの251~253行目を参照してください。
281         port\_read(HD\_DATA, CURRENT ->buffer, 256); // バッファ に入れる。
282         CURRENT->errors = 0; // エラー の数をクリアします。
283         CURRENT->buffer += 512; // バッファポインタ を調整します。
284         CURRENT-> sector ++; // 読み込んだ セクターを増やす。
285         if (--CURRENT->nr_sectors ) { // まだ読む べきデータがある場合
286             SET\_INTR(& read\_intr ); // 再び read\_intr\(\)を指します。
287             を返すことができます。
288         }

    // ここで実行すると、このリクエストアイテムのすべてのセクタデータが読み込まれたことを示します。のです。
    // その後、 // end\_request\(\) 関数が呼び出され、リクエストの終了を処理します。最後に
    // do\_hd\_request\(\)を再度行い、他のハードディスクの要求を処理します。
289         end\_request (1); // データ更新フラグ を設定します。
290         do\_hd\_request (0); // その他のリクエスト操作を行い
    ます。 291 }
292
    // 割り込みで呼び出された書き込みセクタ機能。
    // この関数は、終了時に発生するハードディスクの割り込み時に呼び出されます。
    // ハードディスクの書き込みコマンドの //。この関数はread\_intr\(\)と同様に扱われます。その後
    // 書き込みコマンドが実行されると、ハードディスクの割り込み信号が発生し、ハードディスクの
    // 割り込みハンドラが実行されます。この時点で、C関数ポインタdo\_hdで呼び出された
    // 割り込みハンドラはすでに write\_intr\(\) を指定しているので、この関数が実行されるのは
    // セクターの書き込み操作が完了した（またはエラーになった）場合。
293 static void write\_intr(void)
294 {
    この機能は、まず、書き込みコマンド操作がエラーになっているかどうかを判断します。もし、コントローラが
    コマンド終了後も // がビジー状態であったり、コマンド実行エラーが発生した場合は、ハード
    ハードディスクの動作不良の問題を処理し、再度ハードディスクに
    // 処理をリセットし、他のリクエスト項目の実行を継続して、リターンします。
295     if (win\_result ) { // コントローラがエラーメッセージ を返す場合。
296         bad\_rw\_intr (); // r/wエラー処理。
297         do\_hd\_request (); // 引き続きリクエストアイテムを処理します。
298         を返すことができます。
299     }

    // この時点では、1セクタの書き込み操作が成功したことを示しているので、数
    // 書き込まれるべきセクタの//を1つデクリメントします。これが0でない場合は、まだ
    // 書くための // セクターなので、現在のリクエスト開始セクター番号は +1、リクエストデータは
    // バッファポインタは、次に書き込まれるデータを指すように調整されます。次に
    // ハードディスクの割り込みハンドラのC関数ポインタdo\_hdがこの関数を指すようにする。この関数は
    // 512バイトのデータがコントローラのデータポートに書き込まれた後、待機状態に戻ります。
    // 演算終了後に発生する割り込みのための//。

```

```

300     も (--CURRENT->nr_sectors) {,                // まだ書き込むべきセクタがあれば...
301     し
302         CURRENT-> sector++です。
303         CURRENT->buffer += 512;
304         SET_INTR(& write_intr)で                // do_hdが再びwrite_intr()を指す。
305         ず。
306         port_write(HD_DATA, CURRENT->buffer, 256)。
307         を返すことができます。
308     }
309 // このリクエストアイテムのすべてのセクターデータが書き込まれた場合、end_request()関数は
310 // が呼び出され、リクエストアイテムの終了処理が行われます。最後に、do_hd_request()を再度呼び出して
311 // 他のリクエストを処理します。
312 end_request                (1); // データ更新フラグ    を設定します。
313 do_hd_request                (); // その他のリクエスト操作
314     を行います。309 }
315
316 // 割込みで呼び出されたハードディスクの再較正（リセット）機能。
317 // ハードディスク・コントローラがエラーメッセージを返した場合、この関数はまずリード / ライトの
318 // の故障処理を行い、それに対応した（リセット）処理をハードディスクに要求しています。
319 // do_hd_request()関数では、リセットやキャリブレーションなどが必要かどうかを判断します。
320 // その時の特定のフラグの状態に応じて実行されるように、 // 次の要求
321 // が継続または処理されます。
322 static void recal_intr(void)
323 {
324     if (win_result                ())// if error invoke bad_rw_intr()
325         bad_rw_intr()です。
326     do_hd_request();
327 }
328
329 // ハードディスクのタイムアウト処理機能。
330 // この関数は do_timer() (kernel/sched.c, line 340)で呼び出されます。コマンドの送信後
331 // の後、コントローラが割込み信号を発行していない場合は、ハードディスク・コントローラに
332 // hd_timeout ticks, コントローラ（またはハードディスク）の動作がタイムアウトした。この時点で
333 // do_timer()は、この関数を呼び出してリセットフラグを設定し、do_hd_request()を呼び出して実行します。
334 // リセット処理を行います。ハードディスクコントローラがハードディスク割込み要求信号を発行した場合
335 // 所定の時間（200ティック）以内に、ハードディスクの割込みハンドラの実行を開始します。
336 // ハンドラの中で ht_timeout の値が 0 に設定されます。この時点でdo_timer()はこれをスキップします。
337 // 機能です。
338 void hd_times_out(void)
339 {
340     // 現在、処理すべきリクエストアイテムがない場合（リクエストアイテムポインタがNULLの場合）は
341     // はタイムアウトがないので、直接返すことができます。そうでない場合は、警告メッセージが表示されます
342     // まず最初に、実行中に発生したエラーの数が多いかどうかを判断します。
343     // 現在のリクエストアイテムの // が、値 MAX_ERRORS (7) よりも大きい場合。はいの場合、処理は
344     // このリクエストアイテムの // が失敗形式（データ更新フラグが設定されていない）で終了した場合は
345     // C関数ポインタdo_hdがNULLに設定され、リセットフラグが設定されます。リセット操作は
346     // は、リクエストアイテムハンドラdo_hd_request()で実行されます。
347     if (! CURRENT)
348         を返すことができます。
349     printk("HD timeout")となります。
350     if (++CURRENT->errors >= MAX_ERRORS)
351         end_request(0)です。
352     SET_INTR(NULL                ); // do_hd = NULL, time_out = 200 とします。
353     reset = 1となります。

```


327 [do_hd_request\(\)](#)を行います。

```

328 }
329
330 // ハードディスクの読み取り/書き込み要求操作を行う。
331 // この関数は、まず、ハードディスクのシリンダー番号、セクタ番号、および
332 // デバイス番号と開始位置に応じて、現在のトラック、ヘッド番号などを表示します。
333 // 現在の要求項目にある // セクタ番号情報を、コマンドに応じて
334 // リクエスト項目 (READ/ WRITE) の//が、ハードディスクにリードまたはライトのコマンドを送信します。もし、そ
335 // のような
336 // コントローラのリセットフラグまたは再校正フラグが設定されている場合、リセットまたは再校正の
337 // の操作が最初に行われます。
338 // リクエストアイテムがブロックデバイスの最初のものである場合 (デバイスがもともとアイドルである場合)。
339 // 現在のリクエストアイテムのポインタは、直接リクエストアイテムを指すようになります (ll_rw_blk.c参照)
340 // 84行目) となり、その関数が呼び出されて読み書きの操作が実行されます。それ以外の場合は
341 // Read/Writeの完了によるハードディスクの割り込みの処理で、 //があった場合。
342 // 処理すべき要求項目が残っている場合、この関数は割り込み時にも呼ばれます。
343 // 処理を行います。kernel/sys_call.s の 235 行目を参照してください。
344 void do_hd_request(void)
345 {
346     int i,r;
347     unsigned int block,dev;
348     署名された秒数、頭数、円筒数
349     unsigned int nsect;
350
351     // この関数は、まずリクエストアイテムの有効性をチェックします。リクエストがない場合は終了します
352     // をリクエストキューに入れます (blk.hの148行目を参照)。次に、サブデバイスの番号をデバイスの
353     // 現在のリクエスト・アイテムにおける // 番号と開始セクタです。サブデバイスの番号は
354     // をハードディスクの各パーティションに設定します。サブデバイスの番号が存在しない場合や、開始の
355     // セクターがパーティション・セクター番号-2よりも大きい場合、リクエストは終了し、ジャンプ
356     // をラベルリピート (blk.hの149行目のマクロINIT_REQUESTで定義)。の1つのブロックがあるので
357     // データ (2セクタ、つまり1024バイト) を一度に読み書きする必要がある場合、要求されたセクタは
358     // 番号は、パーティション内の最後のペナルティー・セクター番号よりも大きくすることはできません。すると
359     // サブデバイス番号に対応するパーティションの開始セクタ番号を加算することで、 // サブデバイス番号に対応す
360     // るパーティションの開始セクタ番号を加算します。
361     // 読み書きされるブロックは、ハードディスク全体の絶対的なセクタ番号にマッピングされます。
362     // サブデバイスの番号を5で割ると、対応するハードディスクの番号になります。については
363     // ハードディスクのデバイス番号の説明は、6.2.3項の表6-1を参照してください。
364     INIT_REQUESTです。
365     dev = MINOR(CURRENT->dev);
366     block = CURRENT->sector ; // 開始セクター。
367     if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
368         end_request(0)です。
369         goto repeat; // blk.hの149
370                             行目にあります。
371     }
372     block += hd[dev].start_sect;
373     dev /= 5; // devは現在のHDの番号 (0, また
374             は1) です。
375
376     // そして、得られた絶対セクタ番号「block」とハードディスク番号「dev」をもとに
377     // セクターナンバー (sec)、シリンダーナンバー (cyl)、ヘッドナンバー (head) を計算します。
378     // ハードディスクの中の //(ヘッド)です。これらのデータを計算するために、以下のインラインアセンブリコードが
379     // 使用されています。
380     // トラックあたりのセクタ数とハードディスクのヘッド数に応じて、以下のようになります。
381     // の情報構造を持っています。算出方法は

```

// 346行目のステートメントは、EAXがセクタ番号「block」、EDXが「0」であることを示しています。
セクタ番号EDX:EAXをトラックごとのセクタ数で割るDIVL命令
// (hd_info[dev].sect)のように、結果的に商はEAXに、余剰はEDXになります。中でも
// それらの中で、EAXは指定された位置（すべてのヘッドフェイス）までのトラックの総数、EDXは

は、現在のトラックのセクタ番号です。

// 348行目のステートメントは、EAXが計算されたトラックの総数であることを示しており
 // EDXは0に設定されます。DIVL命令は、EDX:EAXのトラック総数を、EDX:EAXのトラック数で割ります。
 // 総ヘッド数 (hd_info[dev].head)。EAXで得られる整数分割値は
 // シリンダー番号 (cyl)、EDXで得られた余りをヘッド番号 (head) としています。

```
346     asm ("divl %4": "=a" (block), "=d" (sec): "0" (block), "1" (0),
347         "r" (hd_info[dev].sect)) である。
348     asm ("divl %4": "=a" (cyl), "=d" (head): "0" (block), "1" (0),
349         "r" (hd_info[dev].head)  のようになります。)
350     sec++; // 現在のトラックのセクター番号が調整されます。
351     nsect = CURRENT          ->nr_sectors; // 読み書き するセクタの数です。
```

// ここで、開始セクター「block」に対応するシリンダー番号 (cyl) が次のようになります。

// 読み書き、現在のトラックのセクタ番号 (sec)、ヘッド番号 (head)、そして
 // 読み書きされる総セクタ数 (nsect)。そして、I/O操作コマンドを送ることができます
 この情報に基づいて、ハードディスクコントローラに//を送信します。しかし、送信する前にも
 // コントローラの状態をリセットして、ハードディスクを再調整するフラグがあるかどうかを確認します。それ

は
 通常、リセット操作の後には、ハードディスクのヘッド位置を再調整する必要があります。もし
 これらのフラグが設定されている場合は、前回のハードディスクの動作に問題があったことを示している可能性が
 あります。

または、システムの最初のハードディスクの読み取りと書き込みの操作になっているので、リセットする必要があります。

ハードディスクやコントローラを交換して、再校正してください。

//

// この時にリセットフラグが設定されていると、リセット操作が必要になります。その後、ハードをリセット
 ハードディスクの再調整が必要であることを示すフラグを設定して返す。

// 関数reset_hd()は、まず、ハードディスクにリセット（再校正）コマンドを送信します。

// コントローラを起動し、「ドライブパラメータの作成」コマンドを送信します。

```
352     if (reset) {
353         recalibrate =          1; // 再校正   が必要です。
354         reset_hd()  です。
355         返すことができます。
356     }
```

// この時点でrecalibrateフラグが設定されている場合は、まずフラグをリセットし、次にrecalibrate

// コマンドがコントローラに送信されます。このコマンドは、トラックのシーク操作を行って

// どこからでもいいから0番のシリンダーにヘッドを

```
357     if (recalibrate) {
358         recalibrate = 0;
359         hd_out(dev, hd_info[CURRENT_DEV].sect, 0, 0, 0,
360             WIN RESTORE, & recal_intr) となりました。)
361         返すことができます。
362     }
```

// 上記の2つのフラグがいずれも設定されていない場合、実際のデータのリード/ライトの送信を開始することがで
 きます。

ハードディスク・コントローラへの // 操作です。現在の要求が、セクタの書き込み操作の場合。

// 書き込みコマンドが送られ、ステータスレジスタの情報が周期的に読み込まれて

// 要求サービスフラグDRQ_STATが設定されているかどうか判断されます。DRQ_STATは、リクエスト

ハードディスク・ステータス・レジスタの//サービス・ビットで、ドライブが転送の準備ができていることを示す

// ホストとデータポートの間に1ワードまたは1バイトのデータを入れる。ループを終了するのは、リクエストが

// サービスDRQが設定されます。ループ終了後もセットされていない場合は、リクエストの

ハードディスクへの書き込みコマンドが失敗したので、問題に対処するためにジャンプするか、実行を継続する。

// 次のハードディスクの要求に対応します。それ以外の場合は、ハードディスクコントローラに1セクタ分のデー
 タを書き込むことができます

```
// データレジスタポートHD_DATA。
363     if (CURRENT->cmd == WRITE) {...
364         hd_out(dev, nsect, sec, head, cyl, WIN_WRITE, & write_intr)で
           す。
365         for(i=0 ; i<10000 && !(r=inb_p(HD_STATUS)& DRQ_STAT) ;
           i++)
```

```

366          /* nothing */ ;
367      if (!r) {
368          bad_rw_intr() です。
369          goto リピート。 //ラベルは blk.h の149 行
                           目にあります。
370      }
371      port_write(HD_DATA, CURRENT->buffer, 256)。

// 現在の要求がハードディスクのデータを読むことであれば、リード・セクタ・コマンドが
// コントローラ。コマンドが無効な場合は停止します。
372      } else if (CURRENT->cmd == READ) {...
373          hd_out(dev, nsect, sec, head, cyl, WIN_READ, & read_intr) となります。
374      } else
375          panic("unknown hd-command");
376 }。
377

// ハードディスクシステムの初期化。
// この関数は、ハードディスクの割り込み記述子を設定し、コントローラが
// 割り込み要求信号です。ハードディスクのリクエストハンドラを do_hd_request() に設定します。
hd_interrupt は、そのゲート記述子のアドレスです。
// 割り込みハンドラ (kernel/sys_call.s, 235行目) です。ハードディスクの割り込みは INT 0x2E (46)
// です。
// 8259A チップの割り込み要求信号 IRQ14 に対応しています。のマスクビットは
// マスターチップ 8259A の INT2 がリセットされ、割り込み要求信号の発行が可能になる
// スレーブチップからのその後、ハードディスク（スレーブ側）の割り込みマスクビットをリセットすることで
// ハードディスクコントローラに割り込み要求信号を送信します。のマクロ set_intr_gate() を使用します。
// IDT の割り込みゲートディスクリプターは include/asm/system.h にあります。
378 void hd_init(void)
379 {
380     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST ; // do_hd_request()。
381     set_intr_gate(0x2E, & hd_interrupt) ); // 割り込みハンドラ の設定
382     outb_p(inb_p (0x21)&0xfb, 0x21); // マスターチップのマスクビット 2
        をリセットする。
383     outb(inb_p (0xA1)&0xbf, 0xA1); // スレーブチッ
        プのマスクビット 6 をリセットします。 384 }
385

```

9.3.3 インフォメーション

9.3.3.1 ATハードディスク・インターフェース・レジスタ

ATハードディスクコントローラのプログラミングレジスタポートの記述を表9-3に示す。また、include/linux/hdreg.hヘッダーファイルの記述も参照してください。

表9-3 ATハードディスクコントローラのレジスタポートとその用途

ポ ー ト	名前	読み取り操作	書き込み操作
0x1f0	HD_DATA	データレジスタ - セクターデータ (読み出し、書き込み、フォーマット)	
0x1f1	hd_error, hd_precomp	エラーレジスタ (HD_ERROR)	ライトプリコンプレジスタ (HD_PRECOMP)
0x1f2	HD_NSECTOR	総セクタ数レジスタ - 総セクタ数 (読み出し、書き込み、チェック、フォーマット)	

0x1f3	HD_SECTOR	セクター番号レジスター - スタートセクター番号 (読み出し、書き込み、チェック)	
0x1f4	HD_LCYL	気筒数レジスタ - 気筒数の下位バイト (読み出し、書き込み、チェック、フォーマット)	
0x1f5	HD_HCYL	気筒数レジスタ - 気筒数上位バイト (読み出し、書き込み、チェック、フォーマット)	
0x1f6	HD_CURRENT	Drive/Head Register - Drive/Head No.(101dhhhh, d=Drive No, h=Head No)	
0x1f7	hd_status,hd_command	メインステータスレジスタ (HD_STATUS)	コマンドレジスタ (HD_COMMAND)

0x3f6	HD_CMD	---	ハードディスク・コントロール・レジスタ (HD_CMD)
0x3f7		デジタル入力レジスタ (1.2MB Floppy)	---

各ポートレジスタの詳細は以下の通りです。

◆ データレジスタ (HD_DATA, 0x1f0)

セクターの読み書きやトラックのフォーマット操作を行う16ビットの高速PIOデータ送信機のペアです。CPUは、データレジスタを介してハードディスクに書き込みを行ったり、ハードディスクから1セクタ分のデータを読み出したりする。つまり、コマンド「rep outsw」または「rep insw」を用いて、cx=256ワードの読み出し/書き込みを繰り返す。

◆ エラーレジスタ(リード)/書き込み前補正レジスタ(ライト) (HD_ERROR, 0x1f1)

読み出し時、このレジスタは8ビットのエラーステータスを保持します。ただし、本レジスタのデータは、メイン・ステータス・レジスタ (HD_STATUS, 0x1f7) のビット0がセットされている場合のみ有効です。コントローラ診断コマンド実行時の意味は、他のコマンドとは異なります。表9-4を参照してください。

このレジスタは、書き込み動作中の書き込み事前補正レジスタとして動作する。書き込み事前補正開始シリンダー番号を記録します。ハードディスクの基本パラメータテーブルの0x05のワードに対応しており、これを4で割って出力しています。しかし、現在のハードディスクのほとんどは、このパラメータを無視しています。ハードディスクのパラメータテーブルの説明は、6.3.3項の表6-4を参照してください。

書面による事前補償とは？

初期のハードディスクは、1トラックあたりのセクタ数が決まっており、各セクタは512バイトと決まっているため、各セクタが占める物理的なトラック長は、ディスクの中心に近づくほど短くなる。磁気メディアにデータを記憶させる能力が低下してしまうのだ。そのため、ハードディスクのヘッドには、1セクタ分のデータを比較的小さなセクタに比較的高密度に入れるための一定の工夫が必要となるのである。一般的に使われているのは、「Write Precompensation」という手法です。つまり、ディスクの端から、ディスク中央のトラック（シリンダー）付近の位置までを起点に、ヘッドの書き込み電流を何らかの方法で調整するのである。

具体的な調整方法は、ディスク上の2値データ0、1の表現を磁気記録・符号化方式（例えば、FM、MFMなど）で記録する。隣接する記録ビットを磁化して2回反転させると、磁界の重なりが生じることがあります。そのため、この時にデータを読み出すと、対応する電気波形のピーク値がドリフトしてしまう。記録密度を上げると、ピークドリフトの度合いが大きくなり、時にはデータビットの分離・認識ができず、読み取りデータのエラーとなる。この問題を解決する方法として、書き込み前の補正や読み出し後の補正技術がある。事前書き込み補償とは、書き込みデータをドライバーに送る前に、読み出しのピークドリフトとは逆方向のパルス補償をあらかじめ書き込んでおくことです。読み取り時に信号のピーク値が前方にドリフトしている場合は、信号の書き込みを遅らせ、読み取り時に信号が後方にドリフトしている場合は、信号の書き込みを先行させます。そのため、読み出し時にはピークの位置を正常な位置に近づけることができます。

表9-4 ハードディスク・コントローラ・エラー・レジスタ

価値	ディアノスティック・コマンド	その他のコマンド
0x01	エラーなし	データフラグがない
0x02	コントローラエラー	トラック0エラー
0x03	セクターバッファエラー	

0x04	ECCパーツエラー	コマンドの放棄
0x05	コントロール・プロセス・エラー	
0x10		IDが見つかりません
0x40		ECCエラー
0x80		バッドセクター

◆セクター番号レジスタ (HD_NSECTOR, 0x1f2)

このレジスタは、リード、ライト、ベリファイ、フォーマットの各コマンドで指定されたセクタ数を保持します。マルチセクタ動作に使用されている場合、このレジスタは、セクタ動作が完了するたびに、0になるまで自動的に1ずつデクリメントされます。初期値が0の場合は、最大セクタ数が256であることを意味します。

◆Begin Sector No Register (HD_SECTOR, 0x1f3)

本レジスタは、リード、ライト、ベリファイの各動作コマンドで指定されたセクタ開始番号を保持します。マルチセクタ動作時には、開始セクタ番号が格納され、セクタ動作ごとに自動的に1ずつ増加していきます。

◆シリンダ番号レジスタ (HD_LCYL, HD_HCYL, 0x1f4, 0x1f5)

2つのシリンダーレジスタには、それぞれシリンダー番号の下位8ビットと上位2ビットが格納されています。

◆ドライバ/ヘッドレジスタ (HD_CURRENT, 0x1f6)

このレジスタは、リード、ライト、ベリファイ、シーク、フォーマットの各コマンドで指定されたドライブ番号とヘッド番号を保持します。ビットフォーマットは101dhhhhです。101はECCチェックコードを使用し、1セクタ512バイトであることを示し、dは選択されたドライブ（0または1）を示し、hhhは選択されたヘッドを示します（表9-5参照）。

表9-5 ドライバ/ヘッドレジスタ

ビット	名前	説明
0	HS0	ヘッドナンバー最下位ビット
1	HS1	
2	HS2	
3	HS3	ヘッドナンバー最上位ビット
4	DRV	ドライブの選択、0 - ドライブ0の選択、1 - ドライブ1の選択
5	予約	常に1
6	予約済み	常に0
7	予約済み	常に1

◆メインステータスレジスタ（リード）／コマンドレジスタ（ライト） (HD_STATUS/HD_COMMAND, 0x1f7)

読み出し時は8ビットのメインステータスレジスタに対応しています。コマンド実行前後のハードディスク・コントローラの動作状態を反映しています。本レジスタの各ビットの意味を表9-6に示します。

表9-6 8ビットメインステータスレジスタ

ビット	名前	マスク	説明
0	ERR_STAT	0x01	コマンド実行エラー。このビットがセットされていると、前のコマンドが終了時のエラーが発生します。この時点で、エラーレジスタとステータスレジスタのビットには、エラーの原因となった情報が含まれています。
1	INDEX_STAT	0x02	インデックスを受信しました。このビットは、ディスクの処理中にインデックスフラグが発生した場合にセットされます。 が回っています。

2	ECC_STAT	0x04	ECC チェックサムエラー。このビットは、回復可能なデータエラーが発生したときにセットされ が修正されました。このような状況でも、マルチセクタの読み出し動作が中断されることはありません。
3	DRQ_STAT	0x08	データ要求サービス。このビットがセットされていると、ドライブの準備ができていることを示し ホストとデータポートの間で、1ワードまたは1バイトのデータを転送することができます。
4	SEEK_STAT	0x10	ドライブのシークが終了します。このビットがセットされていると、シーク操作が終了したことを示します。

			<p>が完了し、ヘッドが指定されたトラックに停止したことを示します。このビットは、エラーが発生しても変化しません。このビットは、再び現在の完了状態を示す</p> <p>ホストがステータスレジスタを読み込んだ後にシークします。</p>
5	WRERR_STAT	0x20	<p>ドライブの故障（書き込みエラー）。このビットは、エラーが発生しても変化しません。このビット</p> <p>は、ホストがステータスレジスタを読み込んだ後に、現在の書き込み操作のエラー状態を再び示します。</p>
6	READY_STAT	0x40	<p>ドライブの準備ができています。ドライブがコマンドを受信する準備ができていることを示します。このビットは、エラーが発生しても変化しません。このビットは、ホストがステータス・レジスタを読み込んだ後、再び現在のドライブ・レディの状態を示します。電源投入時には、このビットは</p> <p>ドライブの速度が正常になり、コマンドを受信できるようになるまで、リセットしてください。</p>
7	BUSY_STAT	0x80	<p>コントローラがビジー状態です。このビットは、ドライブのコントローラが操作しているときにセットされます。この時点では、ホストはコマンドブロックを送信できません。いずれかのコマンド・レジスタのリードは、ステータス・レジスタの値を返します。このビットは以下の条件で設定されます。</p> <p>(*) マシンリセット信号RESETが負になるか、デバイスコントロールレジスタのSRSTがセットされてから400ナノ秒以内です。本ビットのセット状態は、マシンリセット後30秒以内であることが必要です。</p> <p>(*) ホストは、再校正、リード、リードバッファ、ドライブパラメータの初期化、診断の実行などのコマンドを書き込む際に、400ナノ秒以内になっています。</p> <p>(*) 書き込み時に512バイトのデータが転送されてから5マイクロ秒以内。ライトバッファ、またはフォーマットトラックコマンド。</p>

書き込み操作を行うと、このポートはコマンドレジスタに対応する。CPUからのハードディスク制御コマンドを受け付けます。表9-

7に示すように、8つのコマンドがあります。最後の列は、対応するコマンドの終了後にコントローラが行う動作（割り込みを発生させる、または何もしない）を記述するために使用されます。

表9-7 ATハードディスクコントローラコマンド一覧

コマンド名		コマンドコード バイト		デフォルト 値	コマンドエンド フォーム
		上位4ビット	D3 D2 D1 D0		
WIN_RESTORE	DRIVE RECALIZATION (リセット)	0x1	R R R R	0x10	インタラプト
WIN_READ	セクターを読む	0x2	0 0 L T	0x20	インタラプト
WIN_WRITE	ライトセクター	0x3	0 0 L T	0x30	インタラプト
WIN_VERIFY	セクターチェック	0x4	0 0 0 T	0x40	インタラプト
WIN_FORMAT	フォーマットトラック	0x5	0 0 0 0	0x50	インタラプト
WIN_INIT	コントローラの初期化	0x6	0 0 0 0	0x60	インタラプト

WIN_SEEK	トラックを探す	0x7	R R R R	0x70	インタラプト
WIN_DIAGNOSE	コントローラディagnosis ティック	0x9	0 0 0 0	0x90	インタラプトま たはアイドル
WIN_SPECIFY	ドライブパラメータの構築	0x9	0 0 0 1	0x91	インタラプト

表中のコマンドコードバイトの下位4ビットは、追加パラメータということになります。

Rは、ステップレートです。R=0であればステップレートは35us、R=1であれば0.5msとなり、この分だけ増加していきます。

プログラムではデフォルトのR=0となっています。

Lはデータモードです。L=0は、読み書き可能なセクターが512バイトであることを示し、L=1は、読み書き可能なセクターが512に4バイトのECCコードを加えたものであることを示します。プログラムの初期値はL=0です。

Tはリトライモードです。T=0はリトライを許可することを示し、T=1はリトライを禁止することを示す。カーネルプログラムでT=0を指定してください。

これらのコマンドの詳細は以下の通りです。

(1) 0x1X -- (WIN_RESTORE), ドライブ再校正コマンド

このコマンドは、リード/ライト・ヘッドをディスク上の任意の位置から0シリンダーに移動させます。このコマンドを受信すると、ドライブは BUSY_STAT フラグをセットし、0円筒シークコマンドを発行します。その後、シーク動作の終了を待って、状態を更新し、BUSY_STATフラグをリセットし、割り込みを発生させます。

(2) 0x20 -- (WIN_READ) 再試行可能な読み取りセクター; 0x21 -- 再試行不可能な読み取りセクター。

リード・セクタ・コマンドは、指定されたセクタを起点に1～256セクタの読み出しが可能です。指定されたコマンド・ブロック（表9-9参照）のセクタ数が0の場合は、256セクタの読み出しを意味します。ドライブがコマンドを受け付けると、BUSY_STATフラグがセットされ、コマンドの実行が開始されます。シングル・セクタの読み出し動作では、ヘッドのトラック位置が正しくない場合、ドライブは暗黙のうちにシーク動作を行います。ヘッドが正しいトラックに位置すると、ドライブ・ヘッドはトラック・アドレス・フィールドの対応するIDフィールドに位置決めされます。

リトライなしのセクター・リード・コマンドの場合、2つのインデックス・パルスが発生する前に指定されたIDフィールドを正しく読み取れないと、ドライブはエラー・レジスタにIDが見つからないというエラーメッセージを表示します。リトライ可能なセクター・リード・コマンドの場合、ドライブはリードしたIDフィールドに問題が発生した場合、複数回リトライします。再試行の回数は、ドライブ・メーカーが設定します。

ドライブがIDフィールドを正しく読み取った場合、指定されたバイト数でデータ・アドレス・マークを特定する必要があり、そうでない場合はデータ・アドレス・マークが見つからなかったというエラーが報告されます。ヘッドがデータ・アドレス・マークを見つけると、ドライブはデータ・フィールドのデータをセクタ・バッファに読み込みます。エラーが発生した場合、ドライブはエラー・ビットをセットし、DRQ_STATをセットし、割り込みを発生させます。エラーが発生した場合は、エラービットの設定、DRQ_STATの設定、割り込みの発生を行います。コマンドが完了すると、コマンド・ブロック・レジスタには、最後に読み込んだセクタのシリンダ番号、ヘッド番号、セクタ番号が格納されます。

マルチ・セクタ・リード・オペレーションでは、ドライブがホストにデータのセクタを送信する準備ができるたびに、DRQ_STATが設定され、BUSY_STATフラグがクリアされ、割り込みが生成されます。セクター・データの転送が終了すると、ドライブはDRQ_STATとBUSY_STATフラグをリセットしますが、最後のセクターの転送が完了した後にBUSY_STATフラグを設定します。コマンドの終了時には、コマンド・ブロック・レジスタには、最後に読み込んだセクタのシリンダ番号、ヘッド番号、セクタ番号が格納されます。

マルチセクタの読み出し動作で訂正不可能なエラーが発生した場合、読み出し動作はエラー

が発生したセクタで終了します。同様に、コマンド・ブロック・レジスタには、エラーが発生したセクタのシリンダ番号、ヘッド番号、セクタ番号が格納されます。エラーが訂正できるかどうかにかかわらず、ドライブはデータをセクタ・バッファに入れます。

(3) 0x30 -- (WIN_WRITE) 再試行可能な書き込みセクター; 0x31 -- 再試行不可能な書き込みセクター。

Write Sector コマンドは、指定したセクタを起点に1～256セクタの書き込みが可能です。指定されたコマンド・ブロック（表9-9参照）のセクタ数が0の場合は、256セクタを書き込むことを意味します。ドライブがコマンドを受け取ると、DRQ_STATを設定し、セクタ・バッファがデータで満たされるのを待ちます。最初にセクタ・バッファにデータを追加し始めるときには、中断はありません。データが一杯になると、ドライブはDRQをリセットし、BUSY_STATフラグを設定して、コマンドの実行を開始します。

1つのセクタにデータを書き込む操作のために、ドライブはコマンドを受信したときに DRQ_STATを設定し、ホストがセクタ・バッファを埋めるために待機します。データが転送されると、ドライブはBUSY_STATを設定し、DRQ_STATをリセットします。ヘッドのトラック位置が正しくない場合、リード・セクタ・オペレーションと同様に、ドライブは暗黙のうちにシーク・オペレーションを実行します。ヘッドが正しいトラックに配置されると、ドライブ・ヘッドはトラック・アドレス・フィールドの対応するIDフィールドに配置されます。

IDフィールドが正しく読み込まれると、ECCバイトを含むセクタバッファのデータがディスクに書き込まれます。ドライブがセクタを処理すると、BUSY_STATフラグがクリアされ、割り込みが発生します。この時点で、ホストはステータスレジスタを読むことができます。コマンドの終了時には、コマンドブロックレジスタに最後に書き込まれたセクタのシリンダ番号、ヘッド番号、セクタ番号が格納されます。

マルチ・セクタ・ライト動作時には、最初のセクタの動作に加えて、ドライブがホストから1セクタ分のデータを受信する準備ができると、DRQ_STATがセットされ、BUSY_STATフラグがクリアされ、割り込みが発生します。セクタが転送されると、ドライブはDRQをリセットし、BUSYフラグを設定します。最後のセクタがディスクに書き込まれると、ドライブはBUSY_STATフラグをクリアし、割り込みを発生させます（この時点でDRQ_STATはリセットされています）。書き込みコマンドの終了時には、コマンド・ブロック・レジスタに、最後に書き込まれたセクタのシリンダ番号、ヘッド番号、セクタ番号が格納されます。

マルチセクタの書き込み操作でエラーが発生した場合、書き込み操作はエラーが発生したセクタで終了します。同様に、コマンドブロックレジスタには、エラーが発生したセクタのシリンダ番号、ヘッド番号、セクタ番号が格納されます。

(4) 0x40 -- (WIN_VERIFY) リトライ可能なセクターリード検証; 0x41 -- リトライしないセクター検証。

このコマンドの実行は、リード・セクタの操作と同じですが、このコマンドでは、ドライブはDRQ_STATを設定せず、ホストへのデータ転送も行いません。読み取り検証コマンドを受信すると、ドライブは BUSY_STAT フラグを設定します。指定されたセクタが検証されると、ドライブは BUSY_STAT フラグをリセットし、割り込みを発生させます。コマンドの最後に、コマンド・ブロック・レジスタには、最後に検証されたセクタのシリンダ番号、ヘッド番号、セクタ番号が格納されます。

マルチセクタ検証動作でエラーが発生した場合、検証動作はエラーが発生したセクタで終了します。同様に、コマンドブロックレジスタには、エラーが発生したセクタのシリンダ番号、ヘッド番号、セクタ番号が格納されます。

(5) 0x50 -- (WIN_FORMAT) トラックコマンドのフォーマット。

トラック・アドレスは、セクタ・カウント・レジスタで指定されます。ドライブがコマンドを受信すると、DRQ_STATビットを設定し、ホストがセクタ・バッファを埋めるのを待ちます。バッファがいっぱいになると、ドライブはDRQ_STATをクリアし、BUSY_STATフラグを設定して、コマンドの実行を開始します。

(6) 0x60 -- (WIN_INIT) コントローラの初期化。

(7) 0x7X -- (WIN_SEEK) シーク操作。

シーク動作コマンドは、コマンドブロックレジスタで選択されたヘッドを、指定されたトラックに移動させます。ホストがシークコマンドを発行すると、ドライブは BUSY フラグを設定

し、割り込みを発生させます。ドライブは、シーク動作が完了するまで **SEEK_STAT** (DSC - シーク完了) を設定しません。ドライブが割り込みを発生させる前に、シーク動作が完了していない可能性があります。シーク動作中にホストがドライブに新しいコマンドを発行した場合、**BUSY_STAT**はシークが終了するまで設定されます。その後、ドライブは新しいコマンドの実行を開始します。

(8) **0x90 -- (WIN_DIAGNOSE)** ドライブ診断コマンド。

このコマンドは、ドライブ内部に実装されている診断テストプロセスを実行します。ドライブ0は、コマンドから400ns以内に**BUSY_STAT**ビットを設定します。

システムに第2のドライブであるドライブ1が搭載されている場合は、両方のドライブが診断操作を行います。ドライブ0は、ドライブ1が診断操作を行うのを5秒間待ちます。ドライブ1の診断が失敗した場合、ドライブ0

は、その診断状態に0x80を付加します。ホストは、ドライブ0の状態を読み込んでいるときに、ドライブ1の診断動作が失敗したことを検出すると、ドライブ/ヘッド・レジスタ（0x1f6）のドライブ・セレクト・ビット（ビット4）を設定してから、ドライブ1の状態を読み込みます。ドライブ1が診断テストに合格した場合、またはドライブ1が存在しない場合、ドライブ0は自身の診断ステータスをエラー・レジスタに直接ロードします。ドライブ1が存在しない場合は、ドライブ0は自身の診断結果のみを報告し、BUSY_STATビットをリセットした後に割り込みを発生させます。

(9) 0x91 -- (WIN_SPECIFY) ドライブパラメータコマンドを作成します。

このコマンドは、ホストがマルチ・セクタ・オペレーションのヘッド・スワップとセクタ・カウント・ループの値を設定するために使用されます。このコマンドを受信すると、ドライブはBUSY_STATビットを設定し、割り込みを発生させます。このコマンドは、2つのレジスタの値のみを使用します。1つはセクタ数を指定するためのセクタ・カウント・レジスタ、もう1つはヘッド数を指定するためのドライブ/ヘッド・レジスタで、具体的に選択されたドライブに応じてドライブ・セレクト・ビット（ビット4）が設定されます。

このコマンドは、選択されたセクタ・カウント値とヘッド数を検証しません。これらの値が無効な場合、他のコマンドがこれらの値を使用してアクセス・エラーを無効にするまで、ドライブはエラーを報告しません。

◆ ハードディスク・コントロール・レジスタ（ライト）（HD_CMD, 0x3f6）

このレジスタは書き込み専用で、ハードディスクの制御バイトの格納とリセット動作の制御に使用されます。その定義は、表9-8に示すように、ハードディスク基本パラメータテーブルのシフト0x08のバイト記述と同じです。

表9-8 ハードディスク制御バイトの意味

オフセット	ビット	コントロールバイト 説明（ドライブステップ選択
0x08	0	未使用
	1	予約済み(0) (Close IRQ)
	2	リセットの許可
	3	ヘッドの数が8より大きい場合に設定
	4	未使用 (0)
	5	気筒+1でメーカーのパッドエリアマップがある場合は1を設定
	6	ECCリトライの禁止
	7	アクセスリトライの禁止

9.3.32 ATハードディスクコントローラのプログラミング

ハードディスクコントローラを操作する際には、パラメータとコマンドを同時に送信する必要があります。コマンドのフォーマットは表9-

9のとおりです。まず、6バイトのパラメータを送信し、最後に1バイトのコマンドコードを発行しなければなりません。どのようなコマンドであっても、7バイトのコマンドブロックを完全に出力して、ポート0x1f1～0x1f7に順番に書き込む必要があります。コマンドブロックレジスタがロードされる

と、コマンドの実行が始まります。

表9-9 ハードディスク・コントローラのコマンドフォーマット

ポート	説明
0x1f1	補正前開始シリンダー番号の書き込み
0x1f2	セクター数

0x1f3	開始セクター番号
0x1f4	シリンダー番号の下位バイト
0x1f5	シリンダー番号上位バイト
0x1f6	ドライブ番号 / ヘッド番号
0x1f7	コマンドコード

まず、CPUはコントロールレジスタポート(HD_CMD, 0x3f6)にコントロールバイトを出力し、対応するハードディスクの制御モードを確立します。モードが確立された後、上記の順序でパラメータやコマンドを送ることができます。その手順は

1. コントローラのアイドル状態を検出します。CPUはメイン・ステータス・レジスタを読み込みます。ビット7 (BUSY_STAT) が0の場合、コントローラはアイドル状態です。指定された時間内にコントローラが常にビジー状態の場合は、タイムアウトエラーと判断する。hd.cの168行目のcontroller_ready()関数を参照してください。
2. ドライブの準備ができているかどうかを確認します。CPUは、メイン・ステータス・レジスタのビット6 (READY_STAT) が1であるかどうかで、ドライブの準備ができているかどうかを判断します。1の場合、CPUはパラメータやコマンドを出力することができます。hd.cの209行目のdrive_busy()関数を参照してください。
3. 出力コマンドブロック。パラメータやコマンドを順次、対応するポートに出力する。hd.cの187行目から始まるhd_out()関数をご覧ください。
4. CPUは割り込みの発生を待ちます。コマンド実行後、ハードディスクコントローラは、動作の終了やセクタ転送 (マルチセクタリード/ライト) の要求を示す割り込み要求信号 (IRQ14 - 対応するint46) を発生させるか、コントローラの状態をアイドル状態にします。割り込み処理の際に、プログラムhd.cで呼び出される関数をコード237--293に示します。ハードディスクリセット、予期せぬ割り込み、不良読み書き割り込み、読み出し割り込み、書き込み割り込みの5つのケースに対応した5つの関数があります。
5. 検出動作の結果です。CPUはメイン・ステータス・レジスタを再度読み込みます。ビット0が0であれば、コマンドの実行は成功し、そうでなければ失敗します。失敗した場合は、さらにエラーレジスタ (HD_ERROR) を照会して、エラーコードを取得します。hd.cの176行目のwin_result()関数を参照してください。

9.3.33 sphere of land area (submarine)

PCがハードディスクからOSを起動した場合、ROM BIOSプログラムは、マシンセルフテスト診断プログラムの実行後、最初のセクタをメモリ0x7c00に読み込み、そのセクタ上のコードに実行制御を与えて実行を継続します。この特定のセクタをマスターブートレコード(MBR)と呼び、その内容構成を表9-10に示す。

表9-10 ハードディスクのマスターブートセクタの構造 MBR

オフセット	名前	サイズ (バイト)	説明
0x000	MBRコード	446	ブートプログラムのコードとデータ。
0x1BE	パーティションテーブルエントリー1	16	パーティションテーブルの最初のエントリで、合計16バイトです。

0x1CE	パーティションテーブルエントリ2	16	2つ目のパーティションテーブルエントリ、16バイト。
0x1DE	パーティションテーブルエントリ3	16	パーティションテーブルの3番目のエントリ、16バイト。
0x1EE	パーティションテーブルエントリ4	16	4つ目のパーティションテーブルエントリ、16バイト。
0x1FE	ブートフラグ	2	有効なブートセクタフラグです。0x55, 0xAA

MBRには、446バイトの初期ブート実行コードに加えて、合計4つのエントリを持つハードディスクのパーティションテーブルが格納されています。パーティションテーブルは、ハードディスクの第1セクタの0x1BE--0x1FDのオフセット位置に格納されています。

は、ハードディスクの0番のシリンダーです。複数のOSがハードディスクの資源を共有できるように、ハードディスクはすべてのセクタを論理的に1-4のパーティションに分割することができます。各パーティション間のセクタ番号は連続しています。パーティションテーブルの各エントリは16バイトで、パーティションの特性を表すのに使われます。表9-11に示すように、パーティションのサイズ、シリンダ番号、トラック番号、開始と終了のセクタ番号が格納されています。

表9-11 ハードディスク・パーティション・テーブルのエントリ構造

オフセット	名前	サイズ	説明
0x00	boot_ind	1バイト	ブートインデックス。4つのパーティションのうち、一度に起動できるのは1つのパーティションのみです。 0x00 - このパーティションから起動しない、0x80 - このパーティションから起動する。
0x01	ヘッド	1バイト	パーティション開始時のヘッド番号です。ヘッドナンバーの範囲は0～255です。
0x02	セクター	1バイト	現在のシリンダー内のセクター番号（ビット0～5）と上位2ビット（ビット6-7）のシリンダー番号をパーティションの最初に表示します。
0x03	円筒	1バイト	パーティションの先頭にあるシリンダー番号の下位8ビット。
0x04	sys_ind	1バイト	パーティションタイプ。0x0b - DOS、0x80 - オールドミニックス、0x83 - Linux
0x05	エンド・ヘッド	1バイト	パーティションの最後にあるヘッド番号。0～255の範囲で設定できます。
0x06	end_sector	1バイト	現在のシリンダー内のセクター番号（ビット0～5）と上位2ビット（ビット6-7）のシリンダー番号をパーティションの最後にシリンダー番号の6-7を表示します。
0x07	end_cyl	1バイト	パーティション終了時のシリンダー番号の下位8ビット。
0x08-0x0b	スタート_セクタ	4バイト	パーティションの先頭にある物理セクター番号。0から数えます。 を、ハードディスク全体のセクタ番号順に表示します。
0x0c-0x0f	nr_sects	4バイト	パーティションが占有しているセクタの数。

表中のフィールド「head」、「sector」、「cyl」は、それぞれパーティションの先頭のヘッド番号、セクター番号、シリンダー番号を表しています。head」は、0～255の範囲で設定できます。sector」バイトフィールドの下位6ビットは、現在のシリンダーでカウントされているセクター番号を表しています。セクター番号のカウント範囲は1～63です。sector'フィールドの上位2ビットは、'cyl'フィールドで10ビットのシリンダー番号を形成し、その範囲は0から-1023までです。同様に、テーブルの「end_head」、「end_sector」、「end_cyl」フィールドは、それぞれパーティションの最後のヘッド番号、セクター番号、シリンダー番号を示す。したがって、シリンダ番号をC、ヘッド番号をH、セクタ番号をSとすると、パーティションの開始CHS値は次のように表すことができる。

H＝ヘッド。

S = sector & 0x3f;

```
C = (sector & 0xc0) << 2) + cyl;
```

テーブルの「start_sect」フィールドは、4バイトのパーティション開始物理セクター番号です。0からコンパイルされたハードディスク全体のセクター番号を表しています。エンコード方法は、0シリンダー、0ヘッド、1セクター（0、0、1）のCHSから始まり、まず現在のシリンダーのセクターを順番にエンコードし、次にヘッドを0から最大ヘッド番号までエンコードします。最後に、シリンダーをカウントします。ハードディスクの総ヘッド数がMAX_HDで、トラックあたりの総セクタ数がMAX_SECTの場合、CHS値に対応するハードディスクの物理セクタ番号phy_sectorは。

```
phy_sector = (C * MAX_HEAD + H) * MAX_SECT + S - 1
```

ハードディスクの第1セクター（0シリンダー0ヘッダー1セクター）は、1つのパーティションテーブルを除いて、フロッピーディスクの第1セクター（ブートセクター）と同じ目的を持っています。そのコードだけは、実行中に自分自身を0x7c00から0x6000に移動させて0x7c00のスペースを空け、パーティションテーブルの情報をもとにアクティブなパーティションがどれなのかを調べます。そして、アクティブなパーティションの第1セクターを0x7c00にロードして実行します。パーティションテーブルには、ハードディスクのどのシリンダー、ヘッド、セクターからのパーティションかが記録されています。したがって、パーティションテーブルを見れば、アクティブなパーティションの第1セクター（つまり、そのパーティションのブートセクター）がハードディスクのどこにあるかを知ることができます。

9.3.34 絶対的なセクターと現在のシリンダー、セクター、ヘッドの関係。

ハードディスクの1トラックあたりのセクタ数を `track_secs`、ヘッドの総数を `dev_heads`、トラックの総数を `tracks` とします。指定されたシーケンシャルセクタ番号のセクタに対して、対応する現在のシリンダ番号を `cyl`、現在のトラックのセクタ番号を `sec`、現在のヘッド番号を `head` とします。次に、指定された連番のセクター番号から、対応する現在のシリンダー番号、現在のトラックのセクター番号、現在のヘッド番号に変換したい場合は、以下の手順で行います。

- `sector / track_secs` = 商が `track`、余りが `sec` です。
- `tracks / dev_heads` = 商が `cyl` で、余りが `head` です。
- 現在のトラックでは、セクター番号は1から始まるので、`sec` を1だけ増やす必要があります。

指定された `current cyl, sec, head` をハードディスクのシーケンシャルセクタ番号に変換したい場合は、全く逆の手順になります。変換式は上記のものと全く同じです。

```
sector = (cyl * dev_heads + head) * track_secs + sec - 1
```

9.4 ll_rw_blk.c

9.4.1 機能説明

このプログラムは、主にブロックデバイスの低レベルの読み取り/書き込み操作を行うために使用されます。本章のすべてのブロックデバイス（ハードドライブ、フロッピードライブ、仮想ラムディスク）とシステムの他の部分との間のインターフェースプログラムです。このプログラムのリード/ライト関数 `ll_rw_block()` を呼び出すことで、システム内の他のプログラムはブロックデバイスから非同期にデータを読み書きすることができます。この関数の主な目的は、他のプログラムのためにブロックデバイスのリード/ライト要求アイテムを作成し、指定されたブロックデバイスの要求キューに挿入することです。実際の読み書き操作は、デバイスのリクエスト処理関数 `request_fn()` が行います。ハードディスク操作の場合は `do_hd_request()`、フロッピー操作の場合は `do_fd_request()`、仮想ディスクの場

合はdo_rd_request()となります。

ll_rw_block()がブロックデバイスの要求項目を構築し、ブロックデバイスの現在の要求項目ポインタがNULLであることを確認してデバイスがアイドル状態であると判断した場合、新たに作成された要求項目が現在の要求として設定され、request_fn()が直接呼び出されます。それ以外の場合は、エレベータリクエストアルゴリズムが使用されます

を使用して、新しく作成されたリクエスト・アイテムをデバイスのリクエスト・リンクリスト・キューに挿入して処理します。request_fn()が処理を終了すると、リクエスト・アイテムはリンクリストから削除されます。

request_fn()はリクエストアイテムの処理を完了したので、割り込みコールバックC関数（主にread_intr()やwrite_intr()）を通じてrequest_fn()自身を再度呼び出し、リンクリスト内の残りのリクエストアイテムを処理します。したがって、リンクリスト（またはキューと呼ばれる）に未処理のリクエストアイテムがある限り、デバイスのリクエストアイテムのリンクリストが空になるまで、次々と処理されていきます。リンクされたリクエスト・アイテムのリストが空になると、request_fn()はドライブ・コントローラにコマンドを送らなくなり、直ちに終了します。したがって、図9-5に示すように、request_fn()関数のループ呼び出しは終了します。

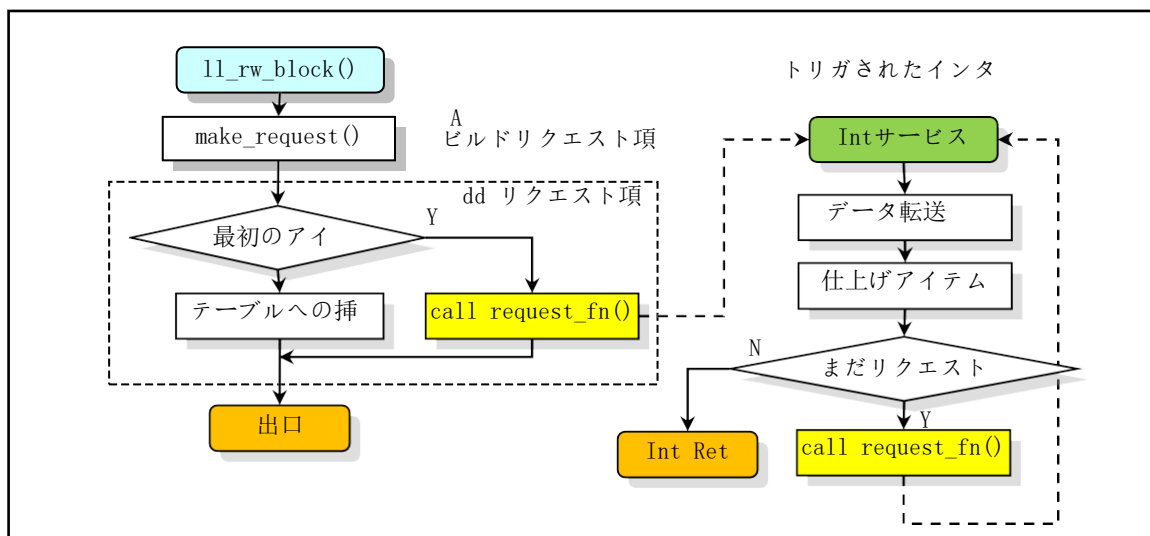


図9-5 ll_rw_blockのコールシーケンス

仮想ディスク装置については、その読み書き動作が、上記のような外部ハードウェア装置との同期動作を伴わないため、上述の割り込み処理はありません。仮想デバイスに対する現在のリクエストアイテムのリード・ライト操作は、do_rd_request()で完全に実装されています。

9.4.2 コードアノテーション

プログラム 9-3 linux/kernel/blk_drv/ll_rw_blk.c

```

1  /*
2      *linux/kernel/blk_dev/ll_rw.c
3      *
4      * (C) 1991 Linus Torvalds
5      */
6
7  /*
8      ブロックデバイスに対するすべてのリード/ライト要求を処理します。
9      */
10 // <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。
11 // <linux/sched.h> スケジューラーのヘッダファイルでは、タスク構造体task_struct、データ
12 初期のタスク0 の、そしていくつかの組み込みアセンブリ関数のマクロのステートメント。

```

//デスクリプタのパラメータ設定と取得
// <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。
// ディスクリプター/割り込みゲートなどが定義されています。

```

10 #include <errno.h>
11 #include <linux/sched.h>
12 #include <linux/kernel.h> (日本語)
13 #include <asm/system.h>
14
15 #include "blk.h"           "// リクエスト構造、リンクリストキュー。
16
17 /*
18  * 必要なデータをすべて含んだリクエスト構造
19  * nr個のセクタをメモリにロードする
20 */
21 // リクエスト項目の配列 queue.NR_REQUEST =
32 struct request request[NR_REQUEST];
22
23 /*
24  * 空いているリクエストがないときに待機するために使用される
25 */
26 struct task_struct * wait_for_request = NULL;
27
28 /* blk_dev_structは。
29  *do_request-address
30  *next-request
31 */
32 // ブロックデバイスの配列です。この配列では、メジャーデバイスの番号をインデックスとして使用します。実際の
33 // の
34 // 内容は、各デバイスドライバの初期化時に記入されます。例えば、次のような場合
35 // ハードディスク・ドライバが初期化されると(hd.c、378行目)、最初のステートメントでは
36 // blk_dev[3]の内容を表示します。ファイルblk_drv/blk.hの45行目、50行目を参照してください。
37 struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
38     { NULL, NULL }, /* no_dev */
39     { NULL, NULL }, /* dev mem */
40     { NULL, NULL }, /* dev fd */
41     { NULL, NULL }, /* dev hd */
42     { NULL, NULL }, /* dev ttyx */
43     { NULL, NULL }, /* dev tty */
44     { NULL, NULL }, /* dev lp */
45 };
46
47 /*
48  * blk_size は、すべてのブロックデバイスのサイズを含みます。
49  *
50  * blk_size[MAJOR][MINOR].
51  *
52  * if (!blk_size[MAJOR])であれば、マイナーサイズのチェックは行われません。
53  */
54 // デバイスデータブロックの総数を示すポインタの配列。各ポインタの項目は
55 // 指定されたメジャーデバイス番号の総ブロック数を配列にしたもの。総
56 // 所有するデータブロックの総数に相当します。
57 // サブデバイス番号で決まるサブデバイス(1ブロックサイズ=1KB)。
58 int * blk_size[NR_BLK_DEV] = { NULL, NULL, ..., }; // ブロックデバイスの数,
NR_BLK_DEV = 7.

```

```

// 指定されたバッファブロックをロックします。
// 指定されたバッファブロックが他のタスクによってロックされていた場合、スリープ自体を（中断することなく
// ロック解除を行うタスクが明示的にウェイクアップするまでの間、 // 待機する）。
51 static inline void lock_buffer(struct buffer_head * bh)
52 {。
53     cli                                (); // int を無効にする。
54     while                               (bh->b_lock)// ロックされていれば、バッファのロックが解除
        されるまでスリープします。
55         sleep_on(&bh->b_wait)となります。
56     bh->b_lock=1;// すぐに                バッファをロックしました。
57     sti                                (); // int を
有効にする。58 }
59
// ロックされたバッファのロックを解除します。
// この関数は、blk.hファイルにある同名の関数と同じですが
// blk.hの実装がマクロとして使用されます。
60 static inline void unlock_buffer(struct buffer_head * bh)
61 {。
62     if                                  (!bh->b_lock)// ロックされていなければ・・・
63         printk("ll_rw_block.c: buffer not locked\n");
64     bh->b_lock =                          0;// ロックフラグ をリセットします。
65     wake_up                             (&bh->b_wait); // このバッファ を待ってい
るタスクを起こします。66 }
67
68 /*
69 * add-request は、リンクリストにリクエストを追加します。
70 * 割り込みを無効にしていますので、その間には何もできません。
71 * request-listを平和に。
72 *
73 * スワップ要求は常に他の要求よりも優先されることに注意してください。
74 * 登場した順に行われます。
75 */
////// リンクリストにリクエストアイテムを追加します。
// パラメータdevは、指定されたブロックデバイス構造体(blk.h、45行目)へのポインタです。
// リクエスト関数のポインタと現在のリクエストアイテムのポインタを持っている。
// コンテンツセットを持つアイテム構造ポインタ。
// この関数は、すでに設定されているリクエストアイテムreqを、リンクされたリストの
// 指定されたデバイス。デバイスの現在のリクエストポインタがNULLの場合は、reqには
// 現在のリクエストアイテムと、デバイスのリクエストアイテムハンドラをすぐに呼び出すことができます。
// そうでない場合は、reqリクエストアイテムのリンクリストに挿入されます。
76 static void add_request(struct blk_dev_struct * dev, struct request * req)
77 {。
78     struct request * tmp;
79
// まず、パラメータで提供されたリクエスト項目のポインタとフラグをさらに
// セットされます。リクエスト内の次のリクエスト項目のポインタにはNULLが設定されます。割り込みの無効化
// そして、リクエスト関連のバッファダーティフラグをクリアします。
80     req->next = NULL;
81     cli()です。
82     if (req->bh)
83         req->bh->b_dirt = 0;// バッファのダーティフラグをクリアします。
// 次に、指定されたデバイスが現在の要求アイテムを持っているかどうかをチェックする、つまり、デバイスが
// がビジー状態であることを示しています。指定されたデバイスdevの現在のリクエスト項目(current_request)
// フィールドが

```

//がNULLの場合、デバイスには現在、リクエストアイテムがないことを意味し、今回は最初の

//の要求項目であり、唯一のものである。そのため、ブロックデバイスの現在のリクエストポインタは
 // リクエスト項目を直接指定して、対応するデバイスのリクエスト機能を
 // が直ちに実行されます。

```

84     if (!(tmp = dev->current_request)) {...
85         dev->current_request = req;
86         sti(); // int を有効にする。
87         (dev->request_fn)(); // リクエスト関数、すなわち do_hd_request() を実行します。
88         を返すことができます。
89     }
```

// デバイスが現在、処理中のリクエストアイテムを持っている場合、エレベータアルゴリズムは最初に
 // を使用して最適な挿入位置を検索し、リクエストアイテムを
 // リクエストリストを表示します。検索コースの途中で、バッファブロックポインタが
 挿入されるべき // が NULL である場合、つまりバッファブロックが存在しない場合、アイテムを見つける必要が
 あります。

// のように、すでにバッファブロックが利用可能な状態になっています。そのため、もしフリーエントリのバッファ
 ブロックのヘッダが

// 現在の挿入位置 (tmpの後) のポインタが空でなければ、この位置が選択される。
 // 次にループを抜けて、ここにリクエストアイテムを挿入します。最後に割込みを有効にして終了
 // 機能のことです。の移動距離を最小化することがエレベータアルゴリズムの役割です。
 ディスクヘッドの回転を抑制し、ハードディスクのアクセス時間を短縮します。

//
 // ループ内の次のステートメントは、reqで参照されるリクエストアイテムを比較するために使用されます。
 リクエストキューにある既存のリクエストアイテムで // 正しいポジションの順番を見つけるために
 // の中で、reqがキューに挿入されます。その後、ループを解除し、reqを
 // キューの正しい位置。

```

90     for ( ; tmp->next ; tmp=tmp->next) {...
91         if (!req->bh)
92             if (tmp->next->bh)
93                 ブレークします。
94             その他
95                 を続けています。
96         if ((IN_ORDER(tmp, req) < IN_ORDER(tmp, tmp->next)) || //blk.h, line 40.
97             !IN_ORDER(tmp, tmp->next)) &&
98             IN_ORDER(req, tmp->next))
99             ブレークします。
```

```

100     }
```

```

101     req->next=tmp->nextとなります。
```

```

102     tmp->next=req;
```

```

103     sti();
```

```

104 }。
```

```

105
```

//// リクエストを作成し、リクエストキューに挿入します。

// パラメータ major はメジャーデバイス番号、rw は指定されたコマンド、bh はバッファ

// データを格納するためのヘッダポインタです。

```

106 static void make_request(int major,int rw, struct buffer_head * bh)
```

```

107 {。
```

```

108     struct request * req;
```

```

109     int rw_ahead;
```

```

110
```

```

111 /* WRITEA/READAは特殊なケースで、実際には必要ありませんので、もし */
```

```

112 /* バッファがロックされている場合は、そのことを忘れ、そうでない場合
    は、通常の読み込みを行います。
```

ここでは、「READ」と「WRITE」の後の接尾語「A」の文字が、「Ahead」という単語を表しています。

// データの読み取り/書き込み前のブロックです。この関数は、まず

// READA/WRITEAコマンドを使用します。これらの2つのコマンドは、ケースのリード/ライト要求を破棄します。
// 指定されたバッファが使用中で、ロックされている場合。それ以外の場合は、通常の


```

// READ/WRITEコマンドです。また、パラメータで指定されたコマンドがREADでもなく
// WRITEの場合は、カーネルプログラムの不具合を意味し、エラーメッセージを表示して停止します。
// カーネルになります。フラグ rw_ahead がパラメータに設定されていることに注意してください。
// コマンドを修正する前に、プリフェッチ/ライトコマンドを
113     if (rw_ahead = (rw == READA || rw == WRITEA)) {...
114         if (bh->b_lock)
115             を返すことができます。
116         if (rw == READA)
117             rw=READ。
118         その他
119             rw = WRITE。
120     }
121     if (rw!=READ && rw!=WRITE)
122         panic("Bad block dev command, must be R/W/RA/WA")と表示され
            ます。
123     lock\_buffer(bh)です。
124     if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate))
        {...
125         unlock\_buffer(bh)です。
126         を返すことができます。
127     }
128     を繰り返しています。
129     /* 書き込み要求がキューを完全に満たしてしまうことは許されません。
130     読み込みに余裕を持たせたい場合は、読み込みが優先されます。最後の3分の1は
131     依頼のうち、読み込みのみの依頼もあります。
132     */
// OK、次はこの関数に読み書き可能なリクエストアイテムを生成して追加しなければなりません。まず、私たちは
// 新しいリクエストアイテムを格納するために、リクエスト配列の中で空いているアイテム（スロット）を探す必
// 要があります。検索は
// 処理はリクエスト配列の最後から始まります。上記の要件によると
// リードコマンドのリクエストでは、キューの最後から直接検索を開始します。
// の頭にあるキュー2/3から空のエントリを埋めることしかできません。
// 順番に並んでいます。なので、後ろから探し始めます。リクエストのデバイスフィールドデブが
// 構造体の要求が-1の場合は、そのアイテムがアイドルであることを意味します。どのアイテムも空いていない場
// 合、その
// つまり、リクエストアイテムの配列がヘッダーを越えて検索されたかどうかをチェックして、リクエストの
// が事前に読み書きされているかどうか（READAまたはWRITEA）を確認し、そうであれば要求操作を破棄する。
// そうでなければ、リクエスト操作を最初にスリープさせます（キューが空になるのを待つため
// アイテム）を使って、しばらくしてからリクエストキューを検索しています。
133     if (rw == READ)
134         req = request+NR\_REQUEST ; // 読み込みの場合、最後 から検索
            します。
135     その他
136         req = request+((NR\_REQUEST *2)/3); // 書き込みの場合は、2/3から後
            りに向かっ て検索します。
137     /* 空のリクエストを見つける */
138     while (--req >= request)
139         if (req->dev<0)
140             ブレークします。
141     /* 見つからなかった場合、新しいリクエストではスリープする: rw_aheadをチェックする
        */。
142     if (req < request ) { // no free item ...
143         if (rw_ahead ) { //read/write ahead であれば終了し
            ます。
144             unlock\_buffer(bh)です。

```

```
145                 を返すことができます。
146             }
147             sleep\_on(& wait for request)です。
148             goto repeat; //128行目。
149         }
150 /* request-infoを埋めて、キューに追加する */。
```

// OK、アイドル状態のリクエストが見つかりました。そこで、新しいリクエストを設定した後、add_request()を呼び出します。
 // でリクエストキューに追加して、すぐに終了することができます。リクエストについてはblk_drv/blk.hを参照してください。

//構造体の23行目です。ここで、req->sectorは、読み取り/書き込みの開始セクタ番号です。

//操作を行い、req->bufferはリクエストアイテムがデータを格納するバッファです。

```

151     req->dev =                bh->b_dev; // デバイスNo.
152     req->cmd =                rw; // コマンド (READ/WRITE)。
153     req->errors=0; // エラーの数。
154     req-> sector =            bh->b_blocknr<<1; // 開始セクタ、(1ブロック=2セクタ)です。
155     req->nr_sectors =        2; // 読み書きされた セクタの数。
156     req->buffer =            bh->b_data; // データバッファの位置。
157     req->waiting = NULL      ; // 操作 を待つリスト。
158     req->bh =                bh; // バッファのヘッダ。
159     req->next = NULL         ; // 次のリクエスト を指します。
160     add_request(major+blk_dev , req); //

```

```

add_request (blk_dev[major], req)。161 }

```

```

162

```

//// 低レベルページの読み書き機能 (Low-Level Read Write Page)。

ブロックデバイスのデータは、ページ単位 (4K) でアクセスされ、8セクタが読み書きされます。

// 毎回です。以下のll_rw_blk()関数を参照してください。

```

163 void ll_rw_page(int rw, int dev, int page, char * buffer)

```

```

164 {。

```

```

165     struct request * req;
166     unsigned int major = MAJOR(dev);
167

```

// まず、機能パラメータの正当性を確認します。もし、デバイスのメジャー番号が

// が存在するか、デバイスのリクエスト処理機能が存在しない場合は、エラーメッセージが表示されます。

// と返されます。パラメータで与えられたコマンドがREADでもWRITEでもない場合、それは

// カーネルプログラムに不具合があった場合、エラーメッセージを表示し、カーネルを停止します。

```

168     if (major >= NR_BLK_DEV || !(blk_dev[major].request_fn)) {...
169         printk("Trying to read nonexistent block-device\n|r");
170         を返すことができます。

```

```

171     }

```

```

172     if (rw!=READ && rw!=WRITE)

```

```

173         panic("Bad block dev command, must be R/W");

```

// パラメータのチェックが完了したら、今度はこの操作のためのリクエストを作成する必要があります。

// まず、新しいリクエストアイテムを格納するために、リクエスト配列の中で空いているアイテム (スロット) を見つける必要があります。

// 検索アクションは、リクエスト配列の最後から始まります。から検索を開始したわけです。

//を返します。リクエスト構造体のデバイスフィールドのdevが0より小さい場合、それは

// のアイテムがアイドルであることを示します。どの項目もアイドルではない場合は、リクエスト操作を最初にスリープさせます (次の項目を待つため)。

// アイドルアイテムのために)、しばらくしてからリクエストキューを検索します。

```

174 を繰り返しています。

```

```

175     req = request+NR_REQUEST ; // 最後を指します。

```

```

176     while (--req >= request)

```

```

177         if (req->dev<0)

```

```

178             ブレークします。

```

```

179     if (req < request) {。

```

```

180         sleep_on(& wait_for_request ); //wait_for_requestを          スリープさせま
            す。

```

```

181         gotoリピート。

```

```

182     }

```

[183](#) */* request-infoを埋めて、キューに追加する */*。

// OK、アイドルのリクエストを見つけました。そこで、新しいリクエストアイテムを設定し、現在のプロセスを
// 遮断されないスリープ状態にしてから、add_request()を呼び出してリクエストキューに追加します。
// そして、直接スケジューラーを呼び出して、現在のプロセスをスリープさせ、ページの読み取りを待ちます。

スイッチングデバイスからの //。のように直接関数を終了するのではなく、ここでは
 // make_request(), schedule() がここで呼ばれます。これは、make_request() が読み取るのは
 // 2セクタですが、ここではスイッチングデバイスへの読み書きに8セクタを要するため
 // 長い時間が必要です。そのため、現在のプロセスは間違いなく待機してスリープする必要があります。そこで、
 プロセスに

このような判断は、プログラムの他の部分で行う必要はありません。

```

184     req->dev = dev; // デバイスNo.
185     req->cmd = rw; // コマンド。
186     req->errors = 0; // エラーの数。
187     req->sector = page<<3; // 開始セクター。
188     req->nr_sectors = 8; // 読み書きされる セクタの数。
189     req->buffer = buffer; // データバッファ。
190     req->waiting = current; // 待機中のキューです。
191     req->bh = NULL; // バッファヘッダ。
192     req->next = NULL; // 次のリクエスト を指します。
193     current->state = TASK_UNINTERRUPTIBLE;
194     add_request(major+blk_dev, req); // リクエストキュー に追加します。
195     schedule();
196 }。
197
```

//// ロウレベルデータブロックのリード&ライト機能 (Low Level Read Write Block)。
 // この関数は、ブロックデバイスドライバとシステムの他の部分との間のインターフェースです。
 通常、fs/buffer.c プログラムで呼び出される。その主な目的は、ブロックデバイスの読み込みを行うことである
 // と書き込み要求のアイテムを、指定されたブロックデバイスの要求キューに挿入します。その際には
 // 実際の読み書きの操作は、デバイスのrequest_fn() 関数で行われます。
 // ハードディスクの場合は do_hd_request(); フロッピーの場合は do_fd_request(); のようになります。
 // 仮想ディスクは do_rd_request() です。さらに、この関数を呼び出す前に、呼び出し元の
 // 最初にバッファブロックヘッダに読み書き可能なブロックデバイスの情報を保存する必要がある
 デバイス番号やブロック番号などの//構造体のことです。パラメータ: rw - コマンド READ です。
 // READA, WRITE, WRITEA; bh - データバッファブロックのヘッダポインタ。

```

198 void ll_rw_block(int rw, struct buffer_head * bh)
199 {...
200     unsigned int major; // メジャーデバイス番号 (ハード
ディスクは3)。201
    // デバイスのメジャー番号が存在しない場合や、デバイスのリクエストアクション関数が
    // 存在しない場合は、エラーメッセージを表示して返します。それ以外の場合は、リクエストを作成して
    // をリクエストキューに入れることができます。
202     if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||)
203         !(blk_dev[major].request_fn) {。
204         printk("Trying to read nonexistent block-device\n|r");
205         を返すことができます。
206     }
207     make_request(major, rw, bh) です。
208 }
209
```

//// 初期化プログラムmain.cから呼び出されるブロックデバイス初期化関数です。
 // リクエスト配列を初期化し、すべてのリクエストアイテムをフリーアイテム (dev = -1) に設定し
 ます。そこに
 //は32項目 (NR_REQUEST = 32) 。

```

210 void blk_dev_init(void)
211 {
212     int i;
213
214     for (i=0 ; i< NR_REQUEST ; i++) {...
```

[215](#) [request](#)[i].dev = -1;

```

216         request[i].next = NULL;
217     }
218 }
219

```

9.5 ramdisk.c

9.5.1 機能

このファイルは、Theodore

Ts'o氏が作成した仮想ラムディスクドライバです。ラム・ディスク・デバイスとは、物理メモリを使って実際のディスク・ストレージ・データをシミュレートする方法です。その目的は、主に「ディスク」データの読み書きの速度を向上させることです。貴重なメモリ資源を消費するだけでなく、システムがシャットダウンしたりクラッシュしたりすると、仮想ディスク内のすべてのデータが消えてしまうという主な欠点がある。そのため、仮想ディスクには重要な入力文書ではなく、一般的なツールプログラムやシステムコマンドなどの一時的なデータしか保存されていないのが普通である。

linux/MakefileにシンボルRAMDISKが定義されていると、カーネルのイニシャライザは、仮想ディスクのデータ用に、指定されたサイズのメモリ領域をメモリ上に描画します。仮想ラムディスクの容量は、RAMDISKの値(KB)に等しい。RAMDISK=512の場合、仮想ディスクのサイズは512KBとなります。物理メモリ内の仮想ディスクの具体的な位置は、カーネルの初期化フェーズ（init/main.c、150行目）で決定され、カーネルキャッシュとメインメモリ領域の間に位置します。実行中のマシンに16MBの物理メモリがある場合、カーネルコードは仮想ラムディスク領域を4MBのメモリの手前に設定します。このとき、メモリの割り当ては図9-6のようになります。

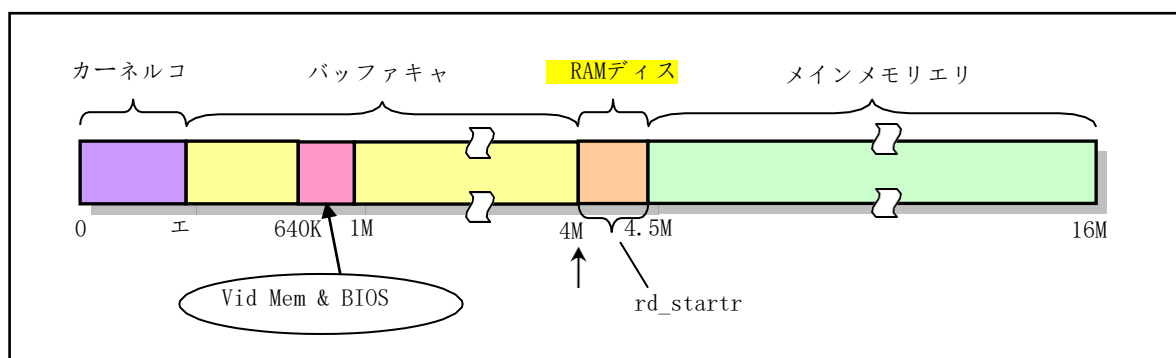


図9-6 16MBメモリシステムにおけるラムディスクの具体的な位置について

ラムディスクに対する読み書きのアクセス操作は、原理的には通常のディスクと同じであり、ブロックデバイスのアクセスモードに応じて操作する必要があります。外部のコントローラやデバイスとの同期をとらないので、実装は比較的簡単です。システムとデバイス間のデータ転送では、インメモリーのブロックコピー操作を行うだけでよい。

ramdisk.cファイルには3つの関数が含まれています。rd_init()関数は、システムの初期化時にinit/main.cプログラムから呼び出され、物理メモリ上の仮想ディスクの具体的な位置とサイズを決定します。do_rd_request()は、仮想ディスクのデータアクセス操作を実行するための仮想ディスクデバイ

スのリクエスト関数です。`rd_load()`は、仮想ディスクのルートファイルシステムのロード関数です。`rd_load()`は仮想ディスクのルートファイルシステムをロードする関数で、システムの初期化時に、指定されたディスクブロックの位置から仮想ディスクにルートファイルシステムをロードしようとするものです。

を起動ディスクに追加します。本機能では、この起動ディスクの開始ブロック位置を256に設定しています。もちろん、この値で指定されたディスク容量にカーネルイメージファイルが収まるのであれば、必要に応じてこの値を変更することも可能です。このように、カーネルブートイメージファイル（Bootimage）とルートファイルシステムイメージファイル（Rootimage）を組み合わせた「ツースインワン」ディスクは、DOSシステムディスクのようにLinuxシステムを起動することができます。このような組み合わせのディスク（統合ディスク）を実験的に作成したことは、第17章にあります。

通常の方法でルートファイルシステムイメージをディスクから読み込む前に、システムはまずrd_load()関数を実行し、ディスクのブロック257からルートファイルシステムのスーパーブロックの読み込みを試みます。成功すると、ルートファイルイメージファイルがメモリ仮想ディスクに読み込まれ、ルートファイルシステムデバイスフラグROOT_DEVが仮想ディスクデバイス(0x0101)に設定される。それ以外の場合はrd_load()を終了し、通常の方法で他のデバイスからのルートファイルシステムのロードを続ける。図 9-7にラムディスクへのルートファイルシステムのロードの動作プロセスを示す。

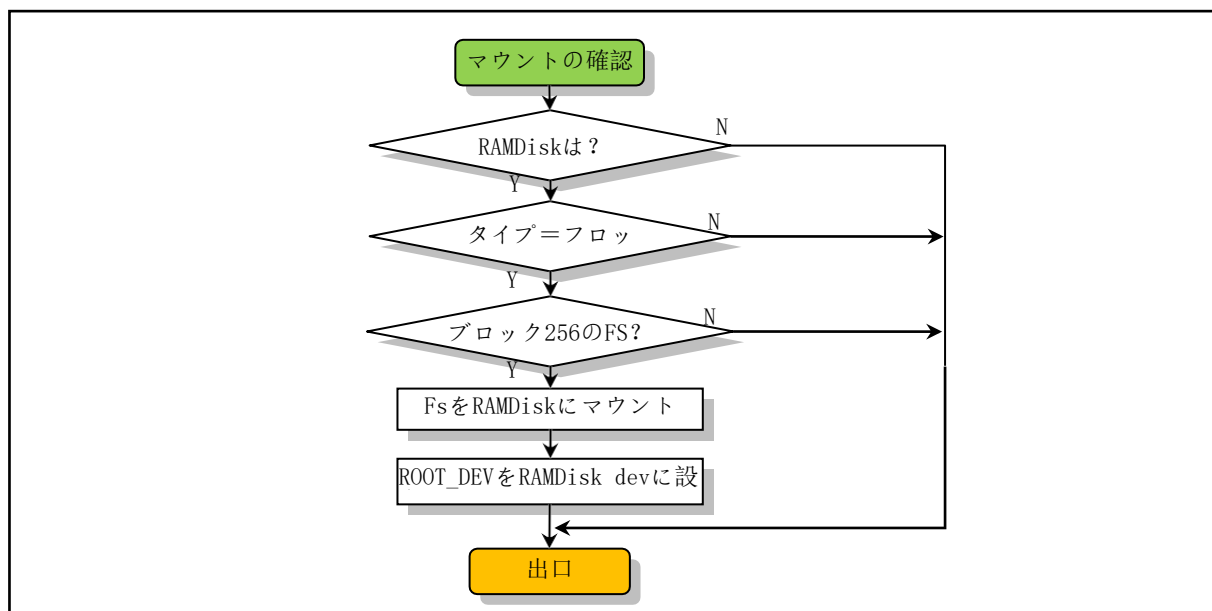


図9-7 ラムディスクにルートファイルシステムをロードするためのフローチャート

Linux

0.12カーネルのソースコードをコンパイルする際に、linux/Makefile設定ファイルにシンボルRAMDISKとそのサイズが定義されている場合、起動してRAMDISK領域を初期化した後、まずディスクの256番目のディスクブロックにある位置をチェックしようとします、Is there a root file system?検出方法は、257番目のディスクブロックに有効なファイルシステムのスーパーブロックがあるかどうかを判断します。もしあれば、そのファイルシステムはメモリ内のRAMDISK領域にロードされ、ルートファイルシステムとして使用されます。そのため、ルートファイルシステムを統合した起動ディスクを使用して、システムをシェルコマンドプロンプトに起動することができます。起動ディスクの指定されたディスクブロックの位置（256番目のディスクブロック）に有効なルートファイルシステムが格納されていない場合、カーネルはルートファイルシステムのディスクを挿入するように促します。ユーザーがEnterキーを押して確認すると、カーネルは別のディスク上のルートファイ

ルシステムを仮想ディスク領域に読み込んで実行します。

1.44MBのカーネルブート起動ディスクに、基本的なルートファイルシステムをディスク上の256番目のブロックの先頭に配置し、統合ディスクを形成するために組み合わせることができ、そのレイアウトを図9-8に示します。

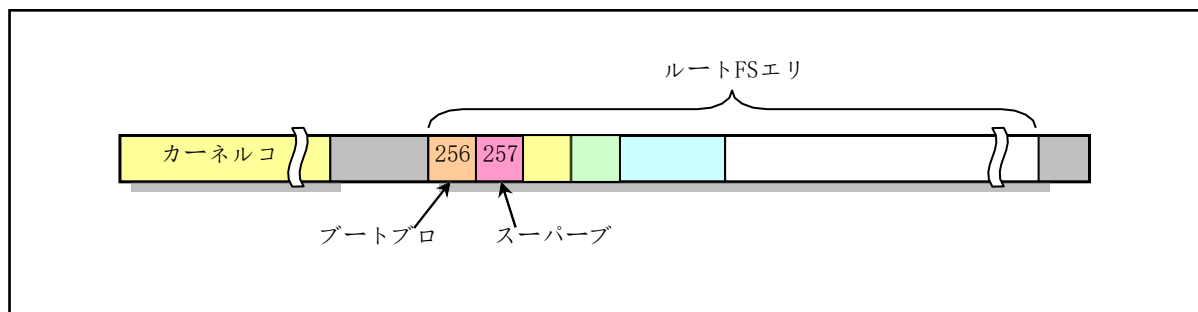


図9-8 集積ディスクのデータブロックレイアウト

9.5.2 コードアノテーション

プログラム 9-4 linux/kernel/blk_drv/ramdisk.c

```

1  /*
2      *linux/kernel/blk_drv/ramdisk.c
3      *
4      セオドア・ツオ著、12/2/91
5      */
// Theodore Ts'o (テッド・ツオ) は、Linuxコミュニティでは有名な人物です。Linuxの人気は
// また、世界には彼の大きな貢献があります。Linuxオペレーティングシステムが登場して間もない頃
// アウト、Linuxの開発にメイリストサービスを熱心に提供してくれたし
// 北米にLinuxのftpサーバーサイト(tsx-11.mit.edu)を開設しました。彼の最大の功績のひとつは
// のLinuxへの貢献は、ext2ファイルシステムの提案と実装でした。このファイルシステム
// は、Linuxの世界では事実上のファイルシステムの標準となっています。最近では、彼が導入した
// ext3およびext4ファイルシステムを採用したことで、安定性、復元性、信頼性が大幅に向上しました。
// ファイルシステムのアクセス効率を高める。彼に敬意を表して、Linux Journalの第97号では
// 2002年5月)に取材され、カバーパーソンとして起用されました。現在、彼はスタッフエンジニアとして
// Googleでは、現在もファイルシステムとストレージの研究を行っています。彼のホームページ:thunk.org/tytso/
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。
// <linux/config.h> カーネル設定用のヘッダファイルです。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/sched.h> スケジューラーのヘッダファイルでは、タスク構造体task_struct、データ
// 初期のタスク0 の、そしていくつかの組み込みアセンブリ関数のマクロのステートメント。
// デスクリプタのパラメータ設定と取得
// <linux/fs.h> ファイルシステムのヘッダファイル。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 の機能を利用します。
// <asm/system.h> システムのヘッダファイルです。を定義または変更する埋め込みアセンブリマクロです。
// ディスクリプター/割り込みゲートなどが定義 されています。
// <asm/segment.h> セグメント操作のヘッダファイルです。埋め込みアセンブリ関数が定義されています。
// セグメントのレジスタ操作。
// <asm/memory.h> メモリコピーのヘッダファイルです。memcpy() の組み込みアセンブリマクロ関数が含まれてい
// ます。
7 #include <string.h>
8
9 #include <linux/config.h>
10 #include <linux/sched.h>
11 #include <linux/fs.h>

```

[12](#) #include <linux/kernel.h> (日本語)

```

13 #include <asm/system.h>
14 #include <asm/segment.h>
15 #include <asm/memory.h>.
16
// RAMディスクのメジャー番号シンボル定数を定義します。メジャーデバイス番号を定義する必要があります
blk.hファイルがインクルードされる前のドライバでは、このシンボリックな定数値が使用されているため、//。
// blk.hファイルの中で、他の定数記号やマクロの範囲を決定します。
17 #define MAJOR_NR 1
18 #include "blk.h"
19
// 初期化時に決定される、メモリ上の仮想ディスクの開始位置
52行目の関数rd_init()を参照してください。カーネルの初期化プログラム init/main.c, line 151 を参照してく
ださい。
20 char*rd_start ; // メモリ上のラムディスクの開始アドレス。21
intrd_length = 0; // ラムディスク が占有するメモリサイズ(バイト
数)22
// ラムディスクのカレントリクエスト操作機能です。
// この関数の構造は、ハードディスクドライバのdo_hd_request()に似ています。
// (hd.cの330行目参照)。低レベルのブロックデバイスインターフェース関数ll_rw_block()の後に
// ラムディスク(rd)のリクエストアイテムを確立し、rdのリンクリストに追加する、この
// 関数は、rdの現在のリクエストアイテムを処理するために呼び出されます。この関数は、まず
// 指定された開始セクタのラムディスクに対応するメモリの開始アドレ
// 現在のリクエストアイテムの中で、必要な数に対応するバイト長len
セクタの//を入力し、要求項目のコマンドに従って動作します。もしコマンドが
// が WRITE の場合、リクエストで指し示されたバッファのデータが直接メモリにコピーされる
// location addr.読み取り操作であれば、その逆も同様です。データがコピーされた後
// 直接 end_request() を呼び出して、リクエストを終了させることができます。その後、先頭にジャンプして
// 関数を実行してから、次のリクエスト項目を処理します。リクエストが残っていない場合は終了します。
23 void do_rd_request(void)
24 {
25     int len
26     char *addr;
27
// 最初にリクエストアイテムの正当性をチェックし、リクエストアイテムがない場合は終了します (blk.hを参
照)。
// 148行目)を行います。)続いて、仮想の開始セクタに対応するアドレスaddrを計算します。
// ディスクと占有メモリのバイト長len.を取得するために、次のような文章があります。
// リクエストの開始セクタに対応するメモリ開始位置とメモリ長
セクター << 9 はセクター * 512 を表す」というように、バイト単位に変換されます。CURRENT
// はblk.hでは「(blk_dev[MAJOR_NR].current_request)」と定義されています。
28 INIT_REQUESTです。
29 addr = rd_start + (CURRENT->sector << 9);
30 len = CURRENT->nr_sectors << 9;
// 現在のリクエストに含まれるマイナーデバイス番号が1でない場合、または対応するメモリスタート
// の位置がラムディスクの終端よりも大きい場合は、リクエストアイテムを終了し、ジャンプ
次の仮想ディスク要求項目を処理するためにリピートへの // が実行されます。ラベルの「リピート」は
はマクロINIT_REQUESTで定義されています (blk.hファイルの149行目を参照)。
31 if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd_start+rd_length)) {...
32     end_request(0)です。
33     gotoリピート。
34 }
// その後、実際の読み取りと書き込みの操作を行います。書き込みコマンド (WRITE) であれば
// リクエストのバッファの内容がアドレス「addr」にコピーされ、長さが
// 'len' バイトです。リードコマンド (READ) の場合は、'addr' で始まるメモリの内容がコピーされます。

```

// をリクエスト項目のバッファに格納し、その長さは「len」バイトです。そうでなければ、次のように表示されます。

```

// のコマンドが存在せず、クラッシュしてしま
// います。
35     if (CURRENT->cmd == WRITE) {...
36         (void) memcpy(addr,
37             CURRENT->buffer.
38             len)を使用してい
// ます。
39     } else if (CURRENT->cmd == READ) {...
40         (void) memcpy(CURRENT->buffer,
41             addr,
42             len)を使用してい
// ます。
43     } else
44         panic("unknown ramdisk-command")となります。
// そして、リクエストアイテムが正常に処理されると、アップデートフラグが設定され、次の
// デバイスのリクエスト項目を処理します。
45     end_request(1)です。
46     goto repeat;
47 }。
48
49 /*
50 * 確保しなければならないメモリ量を返します。
51 */
// 仮想RAMディスクの初期化機能。
// この関数は、まず、仮想ディスクのリクエストハンドラポインタを
// do_rd_request() で、仮想ディスクの開始アドレス、バイト長を決定します。
// 物理メモリをクリアし、仮想エクステンツ全体をクリアします。最後にラムの長さを返します。
// ディスクです。linux/MakefileでRAMDISKの値が0以外に設定されている場合、それは
// 仮想ラムディスク装置がシステムに作成されます。この場合、カーネルの初期化
// プロセスがこの関数を呼び出します (init/main.c, line 151)。2番目のパラメータ「length」には
// をRAMDISK * 1024にバイトで表示します。
52 long rd_init(long mem_start, int length)
53 {。
54     inti
55     char    *cp;
56
57     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST ; // do_rd_request()
58     rd_start = (char *) mem_start; // 16MBのマシン      では4MB。
59     rd_length = length; // ラムディスク      のサイズ。
60     cp = rd_start;
61     for (i=0; i < length; i++)// クリ
// しました。62      *cp++ =
// '\0';
63     return(length);
64 }。
65
66 /*
67 ルートデバイスがラムディスクの場合、それをロードしてみてください。
68 * これを行うために、ルートデバイスにはもともと
69 フロッピーディスクをラムディスクに変更しました。
70 */
///// ラムディスクにルートファイルシステムをロードしてみてください。
// この関数は、カーネルのセットアップ関数setup()で呼び出されます (hd.c、162行目)。この関数は
// 75行目の変数'block=256'は、ルートファイルシステムのイメージファイルがあることを示しています。
// ブートディスクの256番目のディスクブロックの先頭にある。(1ディスクブロック=1024バイト)。

```

```
71 void rd_load(void)
72 {
```



```

73     構造体 buffer head *bh;           // キャッシュバッファのヘッドポインタ。
74     構造体 スーパーブロック s;
75     int          block = 256で      /* ブロック256からスタート */
76     int          i = 1;
77     int          nblocksです。      // ファイルシステムのディスクブロック数
78     チャー       *cp;              ポインタの移動
79
// まずラムディスクの有効性と整合性をチェックします。ラムディスクの長さがゼロの場合。
// であれば終了します。それ以外の場合は、ラムディスクのサイズとメモリの開始位置を
// を表示しました。このとき、ルートファイルデバイスがフロッピーデバイスでない場合も終了し
80     ます。
81     if (! rd\_length)
82         を返すことができます。
83     printk("Ram disk: %d bytes, starting at 0x%x\n", rd\_length.").
84         (int) rd\_start) になります。
85     if (MAJOR(ROOT\_DEV) != 2)
86         を返すことができます。
// 次に、ルートファイルシステムの基本パラメータを読み取る、つまりフロッピーディスクのブロックを読み取る
// 256+1、256、256+2です。ここでblock+1は、ルートファイルシステムのスーパーブロックを
// 仮想ディスクです。関数 breada() は、フロッピーディスクから指定されたデータブロックを読み取るために使用
// されます。
// ディスクに、まだ読む必要のあるブロックをマークして、そのブロックを含むバッファポインタを返します。
// データブロックを作成します(fs/buffer.c, 322行目)。次に、バッファ内のディスクスーパーブロックを
// s変数(d_super_blockはスーパーブロック構造体)を設定し、バッファを解放します。そして、次のように開始
// します。
// でスーパーブロックの有効性を確認します。スーパーブロックの fs マジックナンバーが
// 正しくない場合は、読み込まれたデータブロックがMINIXファイルシステムではないことを意味するので、終了
// します。
// MINIXのスーパーブロックの構造については、「ファイルシステム」の章を参照してください。
86     bh = breada(ROOT\_DEV, block+1, block, block+2, -1);
87     if (!bh) {
88         printk("Disk error while looking for ramdisk!\n").
89         を返すことができます。
90     }
91     *((struct d super block *) &s) = *((struct d super block *) bh->b_data) となりま
92     す。
93     ブレルス (bh) です。
94     if (s.s_magic != SUPER\_MAGIC)
95         ラム・ディスク・イメージが存在しないため、通常のフロッピー
96         ・ブートを想定しています。
97         を返すことができます。
// そして、ルートファイルシステム全体を、メモリ仮想ディスクのエクステンツに読み込もうとしま
// す。については
// ファイルシステムでは、論理ブロックの量（またはゾーンの数）は、s_nzones
// スーパーブロック構造の//フィールド。1つの論理ブロックに含まれるデータブロックの数
// はフィールドs_log_zone_sizeで指定されます。したがって、データブロックの総数 nblocks
// ファイルシステム内の // は、(論理ブロックの量 * 2（各ブロックの累乗）) と同じです。
// つまり、nblocks = (s_nzones * 2s_log_zone_size) となります。ファイルのデータブロックの量が
// システムのブロック数が、ラムディスクが保持できるブロック数よりも多い場合、ロード操作を行います。
// は実行できず、エラーメッセージだけが表示されて返されます。
96     nblocks = s.s_nzones << s.s_log_zone_size;
97     if (nblocks > (rd\_length >> BLOCK\_SIZE\_BITS)) {...
98         printk("Ram ディスクイメージが大きすぎる!(%d blocks, %d avail)"]と表示されます。

```

```
99         nblocks, rd\_length >> BLOCK\_SIZE\_BITS)。)  
100         を返すことができます。  
101     }
```

```
// そうでなければ、仮想ディスクがファイルシステムブロックの合計数を保持できる場合、次のように表示します  
// ロードブロック情報を指定し、'cp' がメモリ上の仮想ディスクの先頭を指すようにします。  
// その後、フロッピーディスク上のルートファイルシステムのイメージファイルをロードするためのループ操作を  
開始する  
//をラムディスクに転送します。操作の過程で、必要なディスクブロックの数が増えれば
```

一度に読み込まれる量が2よりも大きい場合は、高度な先読み機能breada()を使用します。
 // そうでない場合は、シングルフロックの読み込みに bread() 関数を使用します。読み込み中にI/Oエラーが発生した場合
 // ディスクの読み込み処理を放棄して戻ることしかできません。読み込まれたディスクは
 // を使用して、キャッシュからメモリ仮想ディスクの対応する場所にコピーされます。
 memcpy() 関数でロードされたブロックの数を表示します。8進数の「^w^」は
 // で は、タブが表示されていることを示す表示文字列です。

```

102     printk("Read %d bytes into ram disk....0000k",
103           nbblocks << BLOCK_SIZE_BITS) となります。)
104     cp = rd_start;
105     while (nbblocks) {
106         if (nbblocks > 2) // 先読みが必要?
107             bh = breada(ROOT_DEV, block, block+1, block+2, -1);
108             else // 毎回 1ブロックずつ読む。
109                 bh = bread(ROOT_DEV, block) です。
110         if (!bh) {
111             printk("I/O error on block %d, aborting load\n",
112                   ブロック) を使用しています。)
113             を返すことができます。
114         }
115         (void) memcpy(cp, bh->b_data, BLOCK_SIZE); // ロケーションにコピー cp.
116         ブレルス (bh) です。
117         printk("\010\010\010\010\010%4dk", i); // ロードされたブロックの数
118         cp += BLOCK_SIZE; // 次のディスクブロック。
119         block++;
120         nbblocks--;
121         i++;
122     }
123     // 起動ディスクの256ディスクブロックから始まるルートファイルシステム全体が読み込まれるとき。
124     // "done" と表示され、現在のルートfsのデバイス番号が仮想ディスクに変更されます。
125     // デバイス番号0x0101を、最後に返しました。
126     printk("\010\010\010\010\010done\n");
127     ROOT_DEV=0x0101;
128 }

```

9.6 floppy.c

9.6.1 機能説明

本プログラムは、フロッピーディスクコントローラドライバです。他のブロックデバイスドライバと同様に、このプログラムも要求項目操作関数（フロッピーディスクドライブの場合はdo_fd_request()）を使用して、フロッピーディスクの読み取りと書き込みの操作を行います。ハードディスク・ドライバとの大きな違いは、フロッピーディスク・ドライバの方がより多くのタイミング関数や演算を使用することです。

フロッピーディスクドライブが動作していない時には通常回転しないことを考えると、実際にフロッピーディスクの読み書きができるようになるまでには、ドライブのモーターが起動して通常の動

作速度になるのを待つ必要がある。この時間はコンピューターの速度に比べると非常に長く、通常0.5秒程度かかる。また、ディスクへの読み書きが完了したら、ヘッドのディスク面への摩擦を減らすために、ドライブモーターを停止させる必要もある。しかし、ディスクの読み書きが終了した後では、ディスクの読み取りが必要な場合があるため、停止させることはできない。

とすぐに書き込まれます。そのため、ドライブが操作されていない後、一定時間ドライブモーターをアイドル状態にして、読み取りや書き込みが可能になるのを待つ必要があります。長時間ドライブが動作しない場合、プログラムはドライブの回転を停止させます。回転を維持する時間は約3秒に設定できます。さらに、ディスクの読み書き操作に失敗するなどして、ドライブのモーターがオフにならない場合、一定時間後に自動的にオフになるようにすることも必要です。Linuxカーネルでは、この遅延値を100秒に設定しています。

フロッピーディスクドライブの動作には、多くの遅延（タイマー）動作が使われているので、ドライバにはより多くのタイミング処理関数に関わっていることがわかります。また、kernel/sched.c（215～281行目）には、タイマーと密接に関連する関数がいくつか配置されています。これがフロッピーディスクのドライバーとハードディスクのドライバーの最大の違いであり、フロッピーディスクのドライバーがハードディスクのドライバーよりも複雑である理由でもあります。

プログラムはより複雑になっていますが、フロッピーディスクの読み書き操作の動作原理は他のブロックデバイスと同じです。また、プログラムはリクエストアイテムとリクエストのリンクリスト構造を使って、フロッピーディスクへのすべての読み書き操作を処理するので、リクエストアイテム関数do_fd_request()は今でもプログラムの重要な関数のひとつです。この関数は、読みながら本線として展開できる。また、フロッピーディスクコントローラのプログラミングと操作は、多くのコントローラの実行状態やフラグを含む複雑なものです。そのため、プログラムの背後にある説明書や、ヘッダファイルinclude/linux/fdreg.hを参照する必要があります。このヘッダファイルには、すべてのフロッピーディスクコントローラのパラメータ定数が定義されており、これらの定数の意味が説明されています。

9.6.2 コードアノテーション

プログラム 9-5 linux/kernel/blk_drv/floppy.c

```

1  /*
2      *linux/kernel/floppy.c
3      *
4      *(C)      1991Linus Torvalds
5      */
6
7  /*
8      02. 12. 91 - リセットの必要性を示すために、静的変数に変更しました。
9      と再調整します。これにより、いくつかのことが簡単になります (output_byte reset
10     また、エラー時の割込みジャンプも少なくなります。
11     *なので、コードが理解しやすくなっていることを期待しています。
12     */
13
14  /*
15     * このファイルは確かに混乱しています。頑張って動作するようにしました。
16     しかし、私はプログラミングのフロッピーが好きではないし、とにかく1枚しか持っていない。
17     * Urgel. もっと多くのエラーをチェックして、もっと優雅なエラーをするべきです。
18     復旧しました。いくつかのドライブに問題があるようです。私が試したのは
19     訂正します。約束はしない。
20     */
21
22  /*
23     hd.cと同様に、このファイル内のすべてのルーチンは、次のように呼び出すことができます（そして、そう

```

なるでしょう)。

[24](#) 割り込みには細心の注意を払う必要があります。ハードウェア割り込み

[25](#) ハンドラがスリープしなかったり、カーネルパニックが発生したりします。したがって、私は

[26](#) `floppy-on` を直接呼び出しますが、特別なタイマー割り込みを設定する必要があります。

[27](#) *など

[28](#) *

[29](#) * また、1枚以上のフロッピーで動作するかどうかは定かではありません。バグの可能性

[30](#) * abundance

[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、組み込みのアセンブリ関数マクロ文
// ディスクリプタのパラメータ設定と取得 について。
// <linux/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。のプロトタイプ定義が含まれています。
// カーネル でよく使われる機能
// <linux/fdreg.h> フロッピーディスクのファイルです。フロッピーディスクコントローラの定義が含まれていま
//す。
//パラメータ
// <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/割込みゲートなどの変更が定義 されています。
// <asm/io.h> Io のヘッダーファイルです。の io ポート进行操作する関数を定義します。
//マクロの組み込みアセンブラ の形式。
// <asm/segment.h> セグメント操作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の操作 のために
```

[33](#) #include <linux/sched.h>

[34](#) #include <linux/fs.h>

[35](#) #include <linux/kernel.h> (日本語)

[36](#) #include <linux/fdreg.h> (日本語)

[37](#) #include <asm/system.h>

[38](#) #include <asm/io.h>

[39](#) #include <asm/segment.h>

[40](#)

```
// フロッピードライブのメジャーデバイス番号記号を定義する。ドライバでは、メジャーデバイス番号の
//このシンボリック定数が使用されるため、 // blk.hファイルをインクルードする前に定義する必要があります。
//他の関連するシンボル定数やマクロを決定するために、blk.hファイルの // を参照してください。
```

[41](#) #define MAJOR NR 2//フロッピードライブのメジャー番号。

[42](#) #include "blk.h" // ブロックデバイスのヘッダーファイル。 [43](#)

[44](#) static int recalibrate = 0;// flag: 頭部の位置を再校正する（ゼロ に戻

す）。 [45](#) static int reset = 0;// flag: リセット操作が必要です。

[46](#) static int seek = 0;// flag: シーク操作 を行う。

[47](#)

```
kernel/sched.cの223行目で定義されている、現在のデジタル出力レジスタ（DOR）、デフォルト
//値は0x0Cです。この変数には、フロッピードライブの動作からの重要なフラグが格納されています。
//フロッピーディスクドライブの選択、モーターの起動制御、フロッピーディスクコントローラーのリセット起動な
//どの機能があります。
```

```
// DMAや割り込み要求の有効/無効を設定します。の後のDORレジスタの説明を参照してください。
```

```
// 番組表を見る
```

[48](#) extern unsigned char current_DOR;

[49](#)

```
// バイトダイレクト出力（インラインアセンブリマクロ）。値'val'をポートに出力します。
```

[50](#) #define immoutb_p(val,port) \ (^o^)

[51](#) asm ("outb %0,%1|n": "a" ((char) (val)), "i" (port)) [52](#)

```
// この2つのマクロは、フロッピードライブのデバイス番号を計算するために定義されています。パラメータは
// xはマイナーデバイス番号です。マイナーデバイス番号=TYPE*4+DRIVEです。計算方法は
//は番組表の後に表示されます。
```

```

53 #define TYPE(x) ((x) >> 2) // フロッピードライブの種類 (2--1.2Mb, 7--1.44Mb).
54 #define DRIVE(x) ((x) & 0x03) // フロッピーディスクのドライブ番号 (0~3はA~D
    に対応)。55 /*
56 * MAX_ERRORS=8は、すべての不正な読み取りを再試行することを意味するものではないことに注意してくだ
    さい。
57 最大8回 - エラーの種類によっては、エラー数が2倍になります。
58 そのため、実際には5~6回程度のリトライで諦めてしまうかもしれません。
59 */
60 #define MAX_ERRORS 8
61
62 /*
63 * 「result()」で使用されるグローバル
64 */
    これらのステータスバイトのビットの意味については、 // include/linux/fdreg.hヘッダーファイルを参照してく
    ださい。
    // プログラムリストの最後にある説明も参照してください。
65 #define MAX_REPLIES 7 // FDCは、最大7バイトの結果を返します。
66 static unsigned char reply_buffer[MAX_REPLIES]; // レスポンスの結果を格納するのに使用
    します。67 #define ST0(reply_buffer) [0] // 結果のステータスバイト0です。
68 #define ST1(reply_buffer) [1] // 結果ステータスバイト1。
69 #define ST2(reply_buffer) [2] // 結果ステータスバイト2。
70 #define ST3(reply_buffer) [3] // 結果ステータスバ
    イト3。71
72 /*
73 * この構造体は、異なるフロッピータイプを定義します。minixとは異なり
74 * linuxには、コードのように「正しいタイプを検索する」タイプがありません。
75 それにしては、複雑で奇妙だ。で十分な問題を抱えています。
76 このドライバーは、そのままでいいんです。
77 *
78 ストレッチ」は、トラックが何らかの理由でバブルされる必要があるかどうかを示します。
79 タイプ（例：1.2MBのドライブに360kBのディスクなど）があります。その他は
80 * 自明のことである。
81 */
    // フロッピーdrvieのデータ構造を定義します。フロッピーディスクのパラメータは
    // セクター の数を示す。
        1トラック あたりのセクトセクタ数
    // 頭 の数を表します。
    // トラック の数
        トラックを特別に処理する 必要がある場合は、 // stretchflag を指定します。
    // gapSector ギャップの長さ（バイト）。
    // rateデータ転送速度。
    // specParameters (high 4-bit step rate, low 4-bit head unloading time).
82 static struct floppy_struct {
83     符号なしのint size, sect, head, track, stretch。
84     unsigned char gap, rate, spec1;
85 } floppy_type[] = {...
86     {0, 0, 0, 0, 0, 0x00, 0x00, 0x00 }となり /* テストはしません。
        ます。
87     { 720, 9, 2, 40, 0, 0x2A, 0x02, 0xDF }です。 /* 360kBのPCディスク
        */
88     { 2400, 15, 2, 80, 0, 0x1B, 0x00, 0xDF }です。 /* 1.2MBのAT-diskettes */
89     { 720, 9, 2, 40, 1, 0x2A, 0x02, 0xDF }です。 /* 360kB in 720kB ドライ
        ブ */
90     { 1440, 9, 2, 80, 0, 0x2A, 0x02, 0xDF }となり /* 3.5インチ720kBディスケ
        ット */
        ます。

```



```
91      { 720, 9, 2, 40, 1, 0x23, 0x01, 0xDF } です。      /* 1.2MBのドライブに360kB
92                                     */
93      { 1440, 9, 2, 80, 0, 0x23, 0x01, 0xDF } となっ      /* 1.2MBのドライブに720kB
94                                     を搭載 */
95      { 2880, 18, 2, 80, 0, 0x1B, 0x00, 0xCF } となり      /* 1.44MBのディスク
96                                     */
97  };
```

```

95
96 /*
97 レートは0が500kb/s、2が300kbps、1が250kbpsです。
98 Spec1は0xSHで、Sはステッピングレート (F=1ms、E=2ms、D=3msなど) です。
99 * Hはヘッドアンロード時間 (1=16ms、2=32msなど) 。
100 *
101 * Spec2は (HLD<<1 / ND) で、HLDはヘッドロードタイム (1=2ms、2=4msなど) です。
102 * およびNDが設定されている場合は、DMAを使用しないことを意味します。6にハードコードされています
    (HLD=6ms、DMAを使用) 。
103 */
104
    // floppy_interrupt は、カーネル/sys_call.s のフロッピードライブ割り込みハンドラのラベルです。
    // プログラムを作成します。これは、フロッピーディスクの初期化関数 floppy_init() (469行目) で使用されます
    // 割り込みトラップゲート記述子を初期化する。
105 extern void floppy_interrupt(void);
    // これは、boot/head.sの132行目で定義された一時的なフロッピーバッファです。
    // リクエストアイテムがメモリ上で1MBを超える場所にある場合は、DMAバッファを
    // 一時的なバッファ領域です。8237Aチップは、1MBのアドレス範囲内でしかアドレス指定できないので
    // の範囲です。
106 extern char tmp_floppy_area[1024];
107
108 /*
109 * これらはグローバル変数です。
110 * 割り込みへの情報です。それらは、現在使われているデータ
111 * リクエスト
112 */
    // これらのいわゆる "グローバル変数" は、C言語の関数が使用する変数を指します。
    // フロッピーディスクの割り込みハンドラの中の // です。もちろん、これらのC関数はすべてプログラムの中にあり
    // ます。
113 static int cur_spec1 = -1; // 現在の
spec1です。114 static int cur_rate = -1;
115 static struct floppy_struct * floppy = floppy_type ; // floppyはfloppy_type[] を指しま
す。116 static unsigned char current_drive = 0;
117 static unsigned char sector = 0;
118 static unsigned char head = 0;
119 static unsigned char track = 0
120 static unsigned char seek_track = 0;
121 static unsigned char current_track = 255;
122 static unsigned char command = 0; // read/writeコマンド。
123 unsigned char selected = 0; // ドライブ選択フラグ。
124 struct task_struct * wait_on_floppy_select = NULL ; // フロッピーキューを
待つ。125
    // フロッピードライブの選択を解除します。
    // ファンクションパラメーターで指定されたフロッピードライブnrが現在選択されていない場合、警告
    // のメッセージが表示されます。その後、フロッピードライブ選択済みフラグをリセットし、タスク待ちを解除し
    // ます。
    // でフロッピードライブを選択します。デジタル出力レジスタ (DOR) の下位2ビットが使用される
    // 選択したフロッピーディスクドライブ (0-3~A-D) を指定するための//。
126 void floppy_deselect(unsigned int nr)
127 {。
128     if (nr != (current_DOR & 3))
129         printk("floppy_deselect: drive not selected\n");
130     selected = 0;
131     wake_up(& wait_on_floppy_select);

```

[132](#) }。

[133](#)

```

134 /*
135  * floppy-change は割り込みから呼び出されることはないので、少しリラックスできます。
136  * ここで、スリープなど。なお、floppy-onでは、current_DORを
137  * 目的のドライブに移動させても、おそらくスリープには耐えられないでしょう。
138  * 複数のフロッピーを同時に使用するため、ループが発生します。
139 */
140 // 指定したフロッピードライブのフロッピーディスクの交換を確認してください。
141 // パラメータの 'nr' はフロッピーディスクのドライブ番号です。フロッピーディスクを交換した場合は1を返します。
142 // この関数は、まず指定されたフロッピーディスクドライブ 'nr' を選択し、次に
143 // フロッピーディスクが使用されているかどうか、コントローラのデジタル入力レジスタ (DIR) をテストします。
144 // ドライブが交換されました。この関数は、プログラムの check_disk_change() によって呼び出されます。
145 // fs/buffer.c (119行目)。
146 int floppy_change(unsigned int nr)
147 {
148     // まず、フロッピードライブ内のフロッピーディスクを回転させ、通常の動作速度にします。
149     // これには一定の時間がかかります。その方法は、フロッピーのタイマー機能を使って
150     // do_floppy_timer() (kernel / sched.c, line 264) で一定の遅延処理を行います。を実行します。
151     // floppy_on() 関数 (sched.c, line 251) を使って、遅延時間が終了したかどうかを判断します。
152     // (mon_timer[nr]==0?). そうでない場合は、現在のプロセスのスリープを継続させる。もし、遅延が
153     // 期限が切れると、do_floppy_timer() が現在のプロセスをウェイクアップします。
154     // を繰り返しています。
155     // floppy_on (nr); // 指定されたフロッピードライブ nr を起動して待つ。
156     // フロッピーディスクが起動 (回転) したら、現在選択されている
157     // フロッピーディスクドライブは、ファンクションパラメーターで指定されたドライブ 'nr' です。もし、現在選択
158     // されている
159     // フロッピードライブが指定されたフロッピードライブ nr ではなく、他のフロッピードライブが選択されている。
160     // そうすると、現在のタスクは中断されない待機状態になり、他のフロッピー
161     // 選択を解除するドライブは、上記の floppy_deselect() を参照してください。他のフロッピーディスクドライブ
162     // が現在
163     // 選択されている場合や、もう一方のフロッピーディスクドライブの選択が解除されていて、現在のフロッピーディ
164     // スクドライブがまだ
165     // 現在のタスクが起動したときに、指定されたフロッピー・ドライブの NR ではなく、最初にジャンプする
166     // 機能の // と再循環します。
167     while ((current_DOR & 3) != nr && selected)
168         sleep_on(& wait_on_floppy_select);
169     if ((current_DOR & 3) != nr)
170         goto リバート;
171     // これで、フロッピーコントローラは、指定したフロッピードライブ「nr」を選択しました。の値が変更されまし
172     // した。
173     // 続いて、デジタル入力レジスタ「DIR」を取得します。その最上位ビット (ビット7) がセットされていれば、それ
174     // は
175     // フロッピーディスクが交換されている場合は、モーターをオフにすることができ、1が終了します。それ以外の場合
176     // は
177     // ディスクが交換されていないことを示すために、モーターがオフになり、0が終了します。
178     // もし (inb(FD_DIR) & 0x80) {。
179     //     floppy_off(nr) です。
180     //     を返します。
181     // }
182     // floppy_off(nr) です。
183     // 0を返す。
184 }
185 // メモリアドレス「from」からアドレス「to」に1024バイトのデータをコピーしま

```

す。

```
156 #define copy\_buffer(from,to) \ (^o^)  
157     asm ("cld ; rep ; movsl")  
158         :: "c" (BLOCK\_SIZE/4), "S" ((long)(from)), "D" ((long)(to))\  
159         : "cx", "di", "si")  
160  
//// フロッピーディスクのDMAチャンネルの設定（初期化）を行います。
```

```

// フロッピーディスクのデータアクセス動作はDMAで行われます。そのためには
// フロッピーディスク専用のチャンネル2をDMAチップに設定します。
// DMAのプログラミング方法は、プログラムリストの後の情報を参照してください。
161 static void setup_DMA(void)
162 {。
163     long addr = (long) CURRENT                ->buffer; // 現在のリクエストバッ
// ファのアドレス。164
// まず、リクエストアイテムのバッファの位置を確認します。もしバッファが上記のどこかに
// 1MBのメモリを使用する場合は、一時的なバッファ領域 (tmp_floppy_area) にDMAバッファを設定する必要があ
// ります。
// 8237Aチップは、1MBのアドレス範囲内でしかアドレス指定できないからです。書き込みの場合
// ディスクコマンドを実行するには、リクエストアイテムバッファのデータを、一時的な
// エリアです。
165     cli() です。
166     if (addr >= 0x100000) {...
167         addr = (long) tmp_floppy_area;
168         if (command == FD_WRITE)
169             copy_buffer(CURRENT->buffer, tmp_floppy_area) です。
170     }
// 次にDMAチャンネル2の設定を始めますが、その前にチャンネルをマスクする必要があります
// 設定しています。シングルチャンネルマスクレジスタのポートは10です。ビット0~1でDMAチャンネルを指定
// (0-3)、ビット2:1はマスキングを示し、0はリクエストを許可することを示す。モードは
// その後、DMAコントローラポート12と11に//ワードが書き込まれます(読み出しは0x46、書き込みは0x4A)。
// バッファのアドレス「addr」と転送するバイト数0x3ff(0-1023)を書き込む。
// 最後に、DMAチャンネル2のマスクがリセットされ、DMA2から要求されたDREQ信号がオープンされる。
171 /* マスク DMA 2 */
172     immoutb_p                (4|2, 10); // ポート10
173 /* 出力コマンドバイト。理由はわかりませんが、みんな (minix, */
174 サンチェス&カントン) 最初に12、次に11と2回出力されます。
// 次のインラインアセンブリコードは、モードワードを "Clear Sequence Trigger "ポートに書き込みます。
// 12とDMAコントローラのモードレジスタポート11(ディスクの読み込み時は0x46、読み込み時は0x4A。
// ディスクに書き込まれる。))
各チャンネルのアドレスとカウントのレジスタは16ビットなので、操作には
// を設定する際には、ローバイトとハイバイトの2段階で設定します。どのバイトが実際に書き込まれるか
// は、トリガの状態によって決まります。トリガーが0の時は、下位バイトにアクセスする。
トリガーが1の時、上位バイトにアクセスします。トリガーの状態は、1回の
// 訪れる。ポート12に書き込むと、フリップフロップが0の状態になるので、16ビットの設定が
// レジスタは下位バイトから始まります。
175     asm ("outb %al, $12\n\tjmp 1f\nl: \t")
176         "outb %al, $11\n\tl: \t::。
177         "a" ((char) ((command == FD_READ)?dma_read:dma_write))であ
            る。)
178 /* addrの下位8ビット */
// ベース / カレントアドレスレジスタ (ポート4) をDMAチャンネル2に書き
// 込む。
179     immoutb_p(addr, 4) です。
180     addr >>= 8;
181 /* addrのビット8-15 */
182     immoutb_p(addr, 4) です。
183     addr >>= 8;
184 /* addrの16-19ビット目 */
// DMAは1MBのメモリにしかアドレスを取ることができず、その上位16-19ビットを
// ページレジスタ (ポート0x81)。
185     immoutb_p(addr, 0x81) です。

```

[186](#) `/* count-1の下位8ビット(1024-1=0x3ff) */ (注)`

`// ベース/カレントのバイトカウンタ値(ポート5)をDMAチャンネル2に書き込みます。`

```

187     immoutb\_p(0xff, 5);
188 /* count-1の上位8ビット */
(注)
    // 一度に合計1024バイト（2セクタ）を送信します。
189     immoutb\_p(3, 5)です。
190 /* DMA 2を起動 */
191     immoutb\_p(0|2, 10)です。
192     sti()です。
193 }
194
    ///// フロッピーディスクドライブコントローラにバイトコマンドまたはパラメータを出力します。
    // コントローラにバイトを送信する前に、コントローラがレディ状態になっている必要があります
    // データの転送方向はCPUからFDCに設定する必要があるので、この関数では
    // コントローラの状態情報を先に表示します。ここでは、適切な遅延のためにループクエリメソッド
    // を使用しています。
    // エラーが発生した場合は、リセットフラグが設定されます。
195 static void output\_byte(char byte)
196 {
197     int counter;
198     unsigned char status;
199
    // まず、メインステートコントローラFD_STATUS (0x3f4) の状態をサイクリックに読み取る。もし
    // リードステータスがSTATUS_READYで、ディレクションビットがSTATUS_DIR = 0 (CPU FDC) の場合、
    // 指定された
    // バイトがデータポートに出力されます。
200     if (reset)
201         返すことができます。
202     for(counter = 0 ; counter < 10000 ; counter++) {...
203         status = inb\_p(FD_STATUS) & (STATUS\_READY | STATUS\_DIR);
204         if (status == STATUS\_READY) {...
205             outb(byte, FD\_DATA)です。
206             返すことができます。
207         }
208     }
    // 10,000回のサイクルが終了しても送信できない場合は、リセットフラグを設定して
    // エラーメッセージが表示されます。
209     reset = 1となります。
210     printk("Unable to send byte to
    FDC\n|r"); \_}.
212
    ///// FDCの実行結果情報を読み取ることができます。
    // 結果のメッセージは最大7バイトで、配列reply_buffer[]に格納されます。
    // 読み込んだ結果のバイト数。戻り値=-1の場合は、エラーを示します。プログラムの
    // は、上記関数と同様の方法で処理されます。
213 static int result(void)
214 {
215     int i = 0, counter, status;
216
    // リセットフラグが設定されている場合は、直ちに終了して、後続の
    // プログラムを実行します。それ以外の場合は、メインステートコントローラFD_STATUS (0x3f4) の状態をサイク
    // リックに
    // 読み込みます。リードコントローラのスレータスがREADYで、データがないことを示している場合は
    // 読んだバイト数を返します。もし、コントローラのスレータスが方向フラグが
    // //セット (CPU <- FDC)、レディ、ビジーは、データが読めることを示す。の結果データは

```


続いて、コントローラが応答結果の配列に読み込まれます。読み込んだバイト数の最大値は
//はMAX_REPLIES(7)です。

[217](#) if ([reset](#))

```

218         戻る    -1;
219     for (counter = 0 ; counter < 10000 ; counter++) {。
220         ステータス = inb_p(FD_STATUS)&(STATUS_DIR|STATUS_READY|STATUS_BUSY)
                タス    となります。
221         if (status == STATUS_READY)
222             i.を返します。
223         if (status == (STATUS_DIR|STATUS_READY|STATUS_BUSY)) {...
224             if (i >= MAX_REPLIES)
225                 ブレークします。
226                 reply_buffer[i++] = inb_p(FD_DATA);
227         }
228     }
    // 10,000回のサイクルが終了しても読み取れない場合は、リセットフラグを設定して
    // エラーメッセージが表示されます。
229     reset = 1となります。
230     printk("Getstatus times outn\\r").
231     return -1;
232 }。
233
    /// フロッピーディスクのリード/ライトエラー処理機能。
    // この関数は、以下の数に基づいて、さらに実行する必要のあるアクションを決定します。
    // フロッピーディスクの読み書きエラー。もし、現在処理されているリクエストエラーの数が
    // 指定された最大エラー数 MAX_ERRORS (8回)よりも大きい場合は、それ以上の
    // 現在のリクエストに対して操作の試行が行われます。読み込み/書き込みエラーの数が
    // がMAX_ERRORS/2を超えた場合、フロッピードライブをリセットする必要があるため、リセットフラグを
    // を設定します。そうでない場合は、エラー数が最大値の半分以下の場合にのみ
    // 頭の位置を再校正する必要があるので、再校正フラグが設定されています。実際の
    // 後の番組では、リセットと再キャリブレーションの処理が行われます。
234 static void bad_flp_intr(void)
235 {。
    // まず、現在のリクエスト項目のエラーの数を1つ増やします。もし、現在のリクエストの
    // の項目で許容される最大値よりも多くのエラーが発生した場合、現在のフロッピードライブが選択解除されて
    // リクエストは終了します（バッファの内容は更新されません）。
236     CURRENT->errors++;
237     if (CURRENT->errors > MAX_ERRORS) {...
238         floppy_deselect(current_drive)です。
239         end_request(0)です。
240     }
    // 現在のリクエストアイテムのエラー数が最大数の半分よりも多い場合
    許可されたエラーの///の後に、リセットフラグを設定してフロッピードライブをリセットしてから、もう一度試し
    てみてください。
    そうでない場合は、フロッピードライブの再調整が必要となりますので、再度お試しください。
241     if (CURRENT->errors > MAX_ERRORS/2)
242         reset = 1となります。
243     その他
244         recalibrate = 1;
245 }。
246
247 /*
248  * Ok, この割り込みは、DMAのリード/ライトが成功した後に呼び出されます。
249  * そのため、結果を確認し、バッファがあればコピーします。
250  */
    /// 割り込みで呼び出されたフロッピーディスクのリード/ライト機能です。
    // の後に開始される割り込み処理の間に呼び出されます。

```

// フロッピーディスクドライブコントローラの動作が終了します。この関数は、まず

操作結果に問題があるかどうかを判断し、それに応じて

// がそれに応じて処理します。読み取り/書き込み操作が成功した場合、リクエストが
 // 読み取り操作で、そのバッファが1MB以上のメモリにある場合、データをコピーする必要があります。
 // フロッピーの一時的なバッファからリクエストのバッファへ。

251 static void [rw_interrupt](#)(void)

252 {...

// まず、FDC実行の結果情報を読みます。もし、返された結果の数が
 バイトが7になっていないか、ステータスバイト0、1、2にエラーフラグがある場合、書き込みの
 保護エラーが発生すると、エラーメッセージが表示され、現在のドライブが解放されて
 // 現在のリクエストが終了します。そうでない場合は、エラーカウントが行われ、その後
 // フロッピー要求項目の操作を継続します。の意味については、fdreg.hファイルを参照してください。
 // 以下の状態です。

// (0xf8 = ST0_INTR | ST0_SE | ST0_ECE | ST0_NR)

// (0xbf = ST1_EOC | ST1_CRC | ST1_OR | ST1_ND | ST1_WP | ST1_MAM), should be 0xb7

// (0x73 = ST2_CM | ST2_CRC | ST2_WC | ST2_BC | ST2_MAM)

253 if ([result](#)() != 7 || ([ST0](#) & 0xf8) || ([ST1](#) & 0xbf) || ([ST2](#) & 0x73)) {...

254 if ([ST1](#) & 0x02) { // 0x02 = ST1_WP - Write Protected.

255 [printf](#)("Drive %d is write
 protected\\r", [current drive](#)).

256 [floppy_deselect](#)([current drive](#)) です。

257 [end_request](#)(0) です。

258 } else

259 [bad_flp_intr](#)() です。

260 [do_fd_request](#)() です。

261 を返すことができます。

262 }

// 現在のリクエストアイテムのバッファが1MBアドレスを超えている場合、フロッピーの内容が
 // ディスクの読み取り操作は、まだ一時的なバッファに置かれており、コピーされる必要があります。

// 現在の要求項目のバッファ。最後に、現在のフロッピードライブを解放する（非選択

// そして現在のリクエストアイテムを実行する end processing: を待っているプロセスをウェイクアップします

// リクエスト・アイテム、アイドル・リクエスト・アイテムを待っているプロセス（もしあれば）をウェイクアッ
 プさせ、リクエスト・アイテムを削除します。

要求のリンクされたリストから // 要求アイテム .その後、他のフロッピーのリクエストを続けて行う
 //の操作を行います。

263 if ([command](#) == [FD_READ](#) && (unsigned long)([CURRENT](#)->buffer) >= 0x100000)

264 [copy_buffer](#)([tmp_floppy_area](#), [CURRENT](#)->buffer)。

265 [floppy_deselect](#)([current drive](#)) です。

266 [end_request](#)(1) です。

267 [do_fd_request](#)();

268 }.

269

//// DMAチャンネル2を設定し、フロッピーディスクコントローラにコマンドやパラメータを出力する。

// (1バイトのコマンド+0~7バイトのパラメータ)。

// リセットフラグが設定されていない場合は、フロッピーディスク割り込みが発生し、フロッピー

// ディスク割り込みハンドラは、関数が終了した後に実行され、フロッピーディスクコントローラの

// 対応する読み取り/書き込み操作を行います。

270 inline void [setup_rw_floppy](#)(void)

271 {...

272 [setup_DMA](#) (); // フロッピーディスクのDMAチャンネルを 初期化します。

273 [do_floppy](#) = [rw_interrupt](#) ; // int で呼ばれる関数を設定する。

274 [output_byte](#)([command](#)); // コマンドを送信します。

275 [output_byte](#)([head](#)<<2 | [current drive](#)); // param: head no + drive no.

276 [output_byte](#)([track](#)); // param: トラックNo.

277 [output_byte](#)([head](#)); // param: 頭の番号。

278 output_byte(sector); // param: 開始セクター番号

```

279     output_byte(2                ); /* セクターサイズ = 512 */ (英語)
280     output_byte(floppy->sect); // param: トラックあたりのセクタ数。
281     output_byte(floppy          ->gap); // param: セクタ間のギャップ。
282     output_byte(0xFF            ); /* セクターサイズ (n!=0の時は0xff ?) */。
// 上記のoutput_byte()処理のいずれかが失敗した場合、リセットフラグが設定されます。リセット
// 処理は
// do_fd_request()のコードはすぐに実行されます。
283     if (reset)
284         do_fd_request()です。
285 }
286
287 /*
288  * シーク (または再校正) 割り込みの後に呼び出されるルーチンです。
289  * フロッピーコントローラからの *。なお、「予期せぬ割り込み」ルーチンは
290  * また、再校正を行います、ここには来ません。
291  */
///// シーク操作後の割り込み処理で呼び出されるC関数。
// まず、検出割り込みのステータスコマンドを送信し、ステータス情報ST0と
// ヘッドのトラック情報を取得します。エラーが発生した場合は、エラーカウント検出
// 処理を実行するか、フロッピー操作要求項目をキャンセルします。それ以外の場合は
// ステータス情報に応じて現在のトラック変数を変更し、関数を呼び出す
// setup_rw_floppy()でDMAを設定し、読み書きのコマンドとパラメータを出力します。
292 static void seek_interrupt(void)
293 {
// シーク操作の結果を得るために、最初に割り込みステータスチェックコマンドを送信する
// 実行します。このコマンドは引数を取りません。返される結果は2バイトです。ST0と
// ヘッドの現在のトラック番号。そして、FDC実行の結果情報を読み取ります。
// 返された結果のバイト数が2に満たない場合、またはST0がシークの終わりでない場合、
// または、ヘッドが置かれているトラック (ST1) が、設定されているトラックと一致していない場合、
// エラーとなります。
// が発生しました。その後、エラーカウントの処理を行い、フロッピーディスクの実行の
// 要求項目またはリセット処理の実行を継続します。なお、センス
// 割り込みステータスコマンド(FD_SENSEI)は2つの結果バイトを返すべきで、それがresult()のリタ
// ーンです。
// の値が2になるようにします。
294 ドライブの状態を把握する */ /* sense drive status
295     output_byte(FD_SENSEI)です。
296     if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {...
297         bad_flp_intr()です。
298         do_fd_request()です。
299         を返すことができます。
300     }
// シーク操作が成功した場合、現在のリクエストのフロッピーディスク操作は
// 続けて、コマンドとパラメータがフロッピーディスクコントローラに送信されます。
301     current_track = ST1            ;// 現在のトラック を設定します。
302     setup_rw_floppy                (); // DMA、出力フロッピーのコマンドとパラメータ を設
// 定します。
303 }
304
305 /*
306  * このルーチンは、すべてが正しくセットアップされるべきときに呼び出されます。
307  * フロッピーモーターがオンになっていて、正しいフロッピーがセットされている状態)。
308  * 選択)。)
309  */
///// リード/ライトデータ転送機能。

```

```
310 static void transfer(void)  
311 {
```

```

// まず、現在のドライブパラメータが、指定されたドライブのものであるかどうかをチェックします。そうでない
// 場合は
// ドライブパラメータ設定コマンドと対応するパラメータ (param1: 上位4ビットステップ
// レート、低4ビットのヘッドアンロード時間、param2: ヘッドローディング時間)を設定します。)と判断されま
// す。
// 現在のデータ転送速度が指定されたドライブと一致しているかどうかを確認し、一致していない場合は
// 指定したフロッピー・ドライブのレートをデータ転送レート制御レジスタに
// (FD_DCR)となります。
312     if (cur_spec1 != floppy->spec1) { //現在のパラメータ をチェックします。
313         cur_spec1 = floppy->spec1;
314         output_byte(FD_SPECIFY) ); // set disk parametersコマンド を送信します。
315         output_byte(cur_spec1); /* 小屋など */。
316         output_byte(6) ); /* ヘッドのロード時間 =6ms, DMA */ (注)
317     }
318     if (cur_rate != floppy->rate) // 現在のレート を確認します。
319         outb_p(cur_rate = floppy->rate, FD_DCR)となります。
// 上記のoutput_byte() 操作のいずれかが失敗した場合、リセットフラグが設定されます。そこで、ここでは
// リセットフラグの確認が必要です。実際にリセットが設定されている場合、のリセット処理コードは
// do_fd_request()はすぐに実行されます。
320     if (reset) {
321         do_fd_request()です。
322         を返すことができます。
323     }
// この時点でシークフラグがゼロの場合 (つまり、シークの必要がない場合)、DMAが設定されて
// 対応する操作コマンドとパラメータがフロッピーディスクコントローラに送られて
// を返しました。それ以外の場合は、シーク処理が行われるので、フロッピーで呼び出された関数は
// 割り込みはまず、シークトラック機能に設定されます。開始トラック番号が等しくない場合
// を0にすると、ヘッドシークコマンドとパラメータが送信されます。使用されるパラメータは、グローバルな
// 112--121行目で設定された変数の値です。開始トラック番号seek_trackが0の場合は
// recalibrationコマンドを実行し、ヘッドをゼロトラックに戻します。
324     if (! seek) {
325         setup_rw_floppy (); // コマンド&パラメータブロックを送信します。
326         を返すことができます。
327     }
328     do_floppy = seek_interrupt ; // 呼び出された関数 を設定します。
329     if (seek_track) { //トラックを開始します。
330         output_byte(FD_SEEK) ); // シークコマンド を送信します。
331         output_byte(head<<2 | current_drive); // param: head + current drive.
332         output_byte(seek_track) ); // param: トラックNo.
333     } else {
334         output_byte(FD_RECALIBRATE) ); // recalibrateコマンド を送信します。
335         output_byte(head<<2 | current_drive); // param: head + current drive.
336     }
// 同様に、上記の output_byte() の操作のいずれかが失敗した場合、リセットフラグが設定されます。
// そして、do_fd_request()のリセット処理コードがすぐに実行されます。
337     if (reset)
338         do_fd_request();
339 }。
340
341 /*
342 * 特殊なケース - 予期せぬ割り込み (またはリセット) の後に使用されます。
343 */
//// 割り込みで呼び出されたフロッピードライブの再校正機能。
// 割り込みステータスチェックコマンド (パラメータなし) を最初に送信します。復帰結果が

```


//がエラーを示す場合はリセットフラグがセットされ、そうでない場合はrecalibrationフラグがクリアされます。
その後

```

// フロッピーディスク要求項目処理機能を実行して、対応する
// 操作を行います。なお、センス割り込みステータスコマンド (FD_SENSEI) は、2の結果を返します。
// バイト、つまりresult()の戻り値は2になるはずです。
344 static void recal_interrupt(void)
345 {
346     output_byte(FD_SENSEI                                ); // センス割り込みの状態を送る
        cmd.
347     if (result() != 2 || (STO & 0xE0) == 0x60) // エラー      があればリセット
        する。
348         reset = 1となります。
349     その他
350         recalibrate = 0;
351     do_fd_request() です。
352 }
353
//// フロッピー割り込みで呼び出された予期せぬ割り込み処理関数です。
// 割り込みステータスチェックコマンド (パラメータなし) を最初に送信します。復帰結果が
// エラーが発生した場合は、リセットフラグが設定され、そうでない場合は、再校正フラグが設定されます。
354 void unexpected_floppy_interrupt(void)
355 {
356     output_byte(FD_SENSEI                                ); // センス割り込みの状態を送る cmd.
357     if (result() != 2 || (STO & 0xE0) == 0x60) // エラーがあればリセットします。
358         reset = 1となります。
359     その他
360         recalibrate = 1;
361 }。
362
//// フロッピーディスクの再校正機能です。
// 最初に再校正フラグがリセットされ、再校正コマンドとそのパラメータが表示されます。
// をフロッピーディスクコントローラ(FDC)に送信します。コントローラが再校正を実行すると
// コマンドを実行すると、発生したフロッピーディスク割り込みの中で recal_interrupt() 関数を呼び出します。
363 static void recalibrate_floppy(void)
364 {。
365     recalibrate = 0;
366     current_track = 0;
367     do_floppy = recal_interrupt                        ; // recal関数   を指す。
368     output_byte(FD_RECALIBRATE                        ); // cmd: recalibrate.
369     output_byte(head<<2 | current_drive                ); // param: head no + drive no.
// 同様に、上記の output_byte() の操作のいずれかが失敗した場合、リセットフラグが設定されます。
// となって do_fd_request() のリセット処理コードがすぐに実行されます。
// います。
370     もし (リセット)
371         do_fd_request() です。
372 }
373
//// 割り込みで呼び出されたフロッピーディスクコントローラFDCのリセット処理機能。
// 最初にsense interrupt statusコマンド (パラメータなし) を送信し、戻ってきた結果を読む
// バイトです。その後、set floppy drive parameterコマンドとその関連パラメータを送信し、最後に
// リクエスト処理関数do_fd_request() を再度呼び出し、リクエスト項目を実行するか
// エラー処理操作。
374 static void reset_interrupt(void)
375 {。
376     output_byte(FD_SENSEI                                ); // センス割り込みの状態を送る cmd.
377     (void) result() です。
378     output_byte(FD_SPECIFY                                ); // ドライブパラメータ設定の送信 cmd.

```

379 output_byte(cur_spec1); /* 小屋など */。

```

380     output_byte(6)で                      /* ヘッドロードタイム =6ms, DMA */。
        す。
381     do_fd_request()
        です。
382 }
383
385 * リセットは、DORのビット2をしばらくの間Lowにすることで行われます。
386 */
    ///// フロッピーディスクコントローラーをリセットします。
    // この関数は、まずパラメータとフラグを設定し、リセットフラグをクリアして
    // フロッピー変数のcur_spec1とcur_rateが無効になっています。この2つのパラメータをリセットする必要がある
    リセット操作の後に //。その後、再校正フラグを設定し、C関数の
    // リセット後にFDCが起動するフロッピーディスク割込みで呼ばれるreset_interrupt()
    //の動作を行います。最後に、デジタル出力レジスタ（DOR）のビット2をしばらくの間0に設定して
    // フロッピードライブのリセット操作を行います。DORのビット2は、フロッピードライブのスタート/リセットビ
    ットです。
387 static void reset_floppy(void)
388 {...
389     int i;
390
391     reset = 0となります。
392     cur_spec1                      = -1;// 無効になりました。
393     cur_rate = -1;
394     recalibrate =                    1;// 再校正フラグ      を設
        定します。
395     printk("Reset-floppy calledn\\r").
396     cli()です。
397     do_floppy = reset_interrupt      ; // ポイントリセット機能。
398     outb_p(current_DOR & ~0x04, FD_DOR )// リセットコマンド を実
        行する。
399     for (i=0 ; i<100 ;                i++)// 暫く      待つ。
400         asm ("nop") です。
401     outb(current_DOR, FD_DOR        ); // コントローラを再び
        有効にする。
402     sti()です。
403 }
404
    ///// タイマー割込みで呼び出されたフロッピータイマー機能です。
    // do_fd_request()関数は、現在のリクエストに遅延タイマーを追加して、次のリクエストを待ちます。
    を実行する前に、指定したフロッピーディスクのモーターを通常の動作速度になるまで回転させます。
    // リクエストで要求された操作を行います。これは、タイマーが起動したときに呼び出される関数です。
    // の期限が切れます。まず、デジタル出力レジスタ（DOR）をチェックし、現在指定されているデジタル出力を選
    択します。
    // ドライブを起動し、読み書き可能な転送関数transfer()を呼び出します。
405 static void floppy_on_interrupt      (void)// floppy_on()
interrupt. 406 {
407 /* フロッピー選択はできません、スリープする可能性があるからです。ただ強制的に行うだけです */
    // 現在のドライブがDORと異なる場合には、DORを現在の
    // 指定されたドライブです。現在のDOR値をDORレジスタに出力した後、タイマーは
    // コマンドが実行されるまでの2ティックの遅延に使用され、その後、フロッピーディスクの読み書き
    // 転送機能 transfer() が呼び出されます。現在のドライブがDORと一致した場合、フロッピーの
    // ディスクの読み書き転送機能を直接呼び出すことができます。
408     selected =                    1;// ドライブ選択フラグ      の設定
409     if (current_drive != (current_DOR & 3)) {...
410         current_DOR &=                0xFC; // 選択されたドライブ      をクリアする。
411         current_DOR |= current_drive ; // 現在のドライブを設定する。

```

```
412         outb(current\_DOR, FD\_DOR) ; // 現在のDOR を送信します。  
413         add\_timer(2, & transfer) ; // タイマーと関連する関数 を追加する。  
414     } else
```

```

415         transfer\(\) です。
416     }
417
418     ///// フロッピーディスクの読み書き要求項目処理機能。
419     // これはフロッピードライバーのメイン関数です。主な用途は以下の通りです。(1) ケースの処理
420     // リセットフラグまたは再補正フラグが設定されている箇所; (2) パラメータブロックの取得
421     // リクエスト項目のデバイス番号を使用して、リクエスト項目によるフロッピードライブの //(3)
422     // カーネルタイマを使ってフロッピーディスクの読み書き動作を開始する。
423 void do fd request(void)
424 {
425     unsigned int block;
426
427     // まず、リセットフラグや再校正フラグがあるかどうかを確認します。どちらかが存在する場合は
428     // 関数は、該当するフラグを処理した直後に戻ります。
429     seek = 0; // シークフラグ をリセットします。
430     if (reset) {
431         reset floppy\(\) です。
432         // 返すことができます。
433     }
434     if (recalibrate) {
435         recalibrate floppy\(\) を行います。
436         // 返すことができます。
437     }
438     // この関数の重要な点は、ここから始まります。まず、INIT_REQUESTマクロを
439     // blk.hファイルでリクエストアイテムの有効性をチェックし、リクエストがない場合は終了します。その後
440     // リクエスト項目のデバイス番号を使って、指定したフロッピーのパラメータブロックを取得する
441     // ドライブになります。このパラメータブロックは、以下のようにグローバル変数のパラメータブロックを設定す
442     // るために使用されます。
443     // フロッピーディスクの操作で使用されます(112~122行目参照)。フロッピーディスクのタイプ
444     // リクエスト項目番号の(MINOR(CURRENT->dev)>>2)がディスクのインデックス値として使用されます。
445     // type array floppy_type[]で、指定したフロッピードライブのパラメータブロックを取得します。
446     INIT REQUESTです。
447     floppy = (MINOR(CURRENT->dev)>>2) + floppy type;
448
449     // 次のコードは112~122行目でグローバル変数のパラメータ値の設定を開始しています。もし
450     // 現在のドライブ 'current_drive' がリクエスト・アイテムで指定されたドライブではない場合、フラグ
451     // を実行する前に、ドライブがシーク処理を行う必要があることを示すために、 // seek が設定されます。
452     // 読み込み/書き込み操作を行います。そして、現在のドライブをリクエストで指定されたドライブに設定します
453     if (current drive != CURRENT DEV ) // リクエスト で指定されたドライブです。
454         seek = 1となっています。
455     current\_drive = CURRENT DEV;
456
457     // 次に、読み書き開始セクターブロックの設定を開始します。それぞれの読み書きはブロックにあるので
458     // 単位(1ブロックは2セクタ)の場合、開始セクタは少なくとも2セクタ小さくする必要があります。
459     // ディスクの総セクタ数よりも多い場合。それ以外の場合は、要求項目のパラメータは無効です。
460     // となり、フロッピーディスクの要求項目は次の要求項目を実行するため
461     // に終了します。その後
462     // セクター番号、ヘッド番号、トラック番号、シーク・トラック番号を計算します。
463     // フロッピードライブで異なるフォーマットのディスクを読み取ることができる。)
464     block = CURRENT-> sector;
465     if (block+2 > floppy->size) {...
466         end request(0)です。
467         goto リピート。
468     }

```

441 sector = block % floppy ->sect; // トラック のセクター番号。

```

442     block /= floppy          ->sect; // トラック番号。
443     head = block % floppy-> head    ; // 頭の番号。
444     track = block / floppy-> head    ; // トラックNo.
445     seek_track = track << floppy->stretch; // ドライブの種類に応じたシークトラック番号。

// その後、まだ最初にシーク操作を行う必要があるかどうかを確認します。シーク番号が異なる場合
// 現在のヘッドのトラック番号から // シーク操作が必要であり、シークフラグが
// が必要です。最後にfloppyコマンドが実行されるように設定します。
446     if (seek_track != current_track)
447         seek = 1となっています。
448     sector          ++; // セクターは1から数えます。
449     if (CURRENT->cmd == READ)
450         command = FD_READ;
451     else if (CURRENT->cmd == WRITE)
452         command = FD_WRITE;
453     その他
454     panic("do_fd_request: unknown command").

// 112-122行目ですべてのグローバル変数の値を設定した後、リクエストアイテムを開始します。
// の操作を行います。ここでは、タイマーを使って動作を開始します。を開始する必要があるからです。
// ドライブモーターが正常な動作速度に達すると、フロッピーディスクの読み取りと交換が可能になります。
// 書き込まれますが、これには一定の時間がかかります。そこで、ここではticks_to_floppy_on()
// を使って
// 開始遅延時間を計算し、この遅延時間を利用してタイマーを設定することができます。関数は
455 add_timer(ticks to floppy on(current drive), & floppy_on_interrupt);
456 }
457

// 様々な種類のフロッピーディスクに含まれるデータブロックの合計数です。
458 static int floppy_sizes          [] = { // 配列blk_size[]          の初期データです。
459     0, 0, 0, 0,
460     360, 360, 360, 360,
461     1200, 1200, 1200, 1200,
462     360, 360, 360, 360,
463     720, 720, 720, 720,
464     360, 360, 360, 360,
465     720, 720, 720, 720,
466     1440, 1440, 1440, 1440
467 };
468

///// フロッピーディスクシステムの初期化機能。
// フロッピーデバイス要求の処理関数do_fd_request()を設定し、フロッピー
// ディスク割り込みゲート (int 0x26、ハードウェア割り込み要求信号IRQ6に対応)。
// その後、割り込み信号のマスキングをリセットして、フロッピーディスクコントローラFDCの
// 割り込み要求信号を送信するための //。トラップゲートの設定マクロset_trap_gate()は
// 割り込みディスクリプターテーブルのディスクリプター IDTはヘッダーファイルで定義される
// include/asm/system.h.
469 void floppy_init(void)
470 {
    floppy_interrupt はその割り込みハンドラです。
    // kernel/sys_call.sの267行目を参照してください。割り込み番号はint 0x26 (38)で、以下に対応します。
    // 8259Aチップの割り込み要求信号IRQ6を表示します。
471     blk_size[MAJOR_NR] = floppy_sizes;
472     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST; // = do_fd_request().
473     set_trap_gate(0x26, & floppy_interrupt);

```

```

474         outb(inb_p                               (0x21)&~0x40, 0x21); // フロッ
ピーのintマスクビット                             をリセットします。 475 }
476

```

9.6.3 インフォメーション

9.6.3.1 フロッピーディスクドライブのデバイス番号

Linuxでは、フロッピードライブのメジャー番号は2で、マイナーデバイス番号は、フロッピードライブの種類とフロッピードライブのシーケンス番号によって決定されます。

FDマイナーNo. = TYPE * 4 + DRIVE

このうち、DRIVEは0～3で、フロッピードライブA、B、C、Dに対応し、TYPEはフロッピードライブの種類で、例えば、表9-

12に示すように、2は1.2Mフロッピードライブ、7は1.44Mフロッピードライブを意味します。つまり、floppy.cの85行目で定義されているフロッピータイプ配列 (floppy_type[]) のインデックス値です。

表9-12 フロッピーディスクの種類

タイプ	説明
0	使われていません。
1	360KB PCフロッピードライブ。
2	1.2MBのATフロッピードライブ。
3	720kBドライブで使用される360kBフロッピーディスク。
4	3.5インチ720kBフロッピードライブ。
5	1.2MBドライブで使した360kBフロッピーディスク。
6	1.2MBドライブで使した720kBフロッピーディスク。
7	1.44MB フロッピードライブ

例えば、タイプ7は1.44MBのドライブを示し、ドライブ番号0はAドライブを示します。7*4 + 0 = 28なので、(2, 28)は1.44MのAドライブを指し、デバイス番号は0x021Cで、対応するデバイスファイル名は /dev/fd0または/dev/PS0です。同様に、タイプ2は1.22MBのドライブを表し、2*4 + 0 = 8となるので、(2,8)は1.2MのドライブAを指し、デバイス番号は0x0208、対応するデバイスファイル名は/dev/at0となります。

9.6.3.2 フロッピーディスクコントローラ

フロッピーディスクの読み書きを行うためには、フロッピーディスクドライブを選択し、モーターが一定の速度に達するのを待ち、データブロックを転送する際にはDMAコントローラーによって実現する必要があるため、フロッピーディスクコントローラー (FDC) のプログラムは煩雑である。FDCをプログラムする際には、通常、フロッピーディスクコントローラの1つ以上のレジスタに対応する4つのポートにアクセスする必要があります。1.2Mフロッピーディスクコントローラの場合は、表9-13に示すようなポートがあります。

表9-13 フロッピーディスクコントローラのポート

I/O ポート	名前	リード/ライト	レジスタ名
0x3f2	FD_DOR	書き込みのみ	デジタル出力レジスタ (DOR)
0x3f4	FD_STATUS	読み取り専用	メイン・ステータス・レジスタ (STATUS)

0x3f5	FD_DATA	読み取り専用	結果レジスタ(RESULT)
		書き込みのみ	データレジスタ(DATA)
0x3f7	FD_DIR	読み取り専用	デジタル入力レジスタ (DIR)
	FD_DCR	書き込みのみ	ドライブコントロールレジスタ (DCR) (転送レート

デジタル出力レジスタ (DOR) ポートは、ドライバモータのターンオン、ドライブセレクト、FDCのスタート／リセット、DMAや割り込み要求の有効／無効を制御する8ビットのレジスタです。このレジスタの各ビットの意味を表9-14に示します。

表 9-14 デジタル出力レジスタ定義

ビット	名前	説明
7	MOT_EN3	ドライブDのモーター制御: 1-モーター起動、0-モーター停止。
6	MOT_EN2	ドライブCのモーター制御: 1-モーター起動、0-モーター停止。
5	MOT_EN1	ドライブBのモーター制御: 1-モーター起動、0-モーター停止。
4	MOT_EN0	ドライブAのモーター制御: 1-モーター起動、0-モーター停止。
3	DMA_INT	DMAとIRQのチャンネル。1-有効、0-無効。
2	RESET	コントローラのリセット。1-コントローラ有効、0-コントローラリセット。
1	DRV_SEL1	00～11はそれぞれフロッピードライブA～Dの選択に使用します。
0	DRV_SEL0	

FDCのメインステータスレジスタ (MSR) も8ビットのレジスタで、フロッピーディスクコントローラFDCとフロッピーディスクドライブFDDの動作の基本状態を反映しています。通常、メイン・ステータス・レジスタのステータス・ビットは、CPUがFDCにコマンドを送る前、あるいはFDCが動作結果を取得する前に読み込まれ、現在のFDCデータ・レジスタがレディであるかどうかを判断し、データ転送の方向を決定します。MSRの各ビットの定義を表9-15に示す。

表9-15 MSRの各ビットの定義

ビット	名前	説明
7	RQM	データポートの準備完了。FDCデータレジスタの準備完了
6	DIO	トランスファーの方向1- fdc -> cpu; 0 - cpu -> fdc
5	NDM	非DMAモード。1- コントローラがDMAでない、0 - DMAモード
4	CB	コントローラがビジー状態です。FDCがコマンドの実行中でビジー状態。

3	DDB	Dドライブはビジー状態です。
2	DCB	Cドライブはビジー状態です。
1	DBB	Bドライブは忙しい。
0	DAB	Aドライブは忙しい。

FDCのデータポートは、書き込み可能なコマンド・パラメータ・レジスタと読み出し可能なリザルト・レジスタという複数のレジスタに対応しています。データポート0x3f5には、一度に1つのレジスタしか表示できません。書き込み専用レジスタにアクセスする場合、メイン・ステータス・レジスタのディレクション・ビットDIOは0（CPU→FDC）でなければならず、読み取り専用レジスタにアクセスする場合はその逆となります。書き込み専用レジスタにアクセスしてコマンドおよびパラメータを送信する場合、コマンドは1バイト、関連するパラメータは1～8バイトです。読み出し専用レジスタにアクセスして結果を読み出す場合は、FDCがビジー状態でないときにのみ結果が読み出され、通常、結果データは最大7バイトです。

データ入力レジスタ (DIR) は、フロッピーディスクの場合は、ディスクの交換状態を示すビット7 (D7) のみが有効で、残りの7ビットはハードディスク・コントローラ・インターフェースに使用されます。

ライトオンリーのディスクコントロールレジスタ (DCR) は、ディスクが使用するデータ転送レートをドライブの種類によって選択するために使用されます。下位2ビット (D1D0) のみが使用され、00は500 kbps、01は300 kbps、10は250 kbpsを意味します。

Linux 0.12カーネルでは、ドライバとフロッピードライブ内のディスクとの間のデータ転送は、DMAコントローラによって実装されています。そのため、読み書きの操作を行う前に、まずDMAコントローラを初期化し、フロッピーディスクドライブコントローラをプログラムする必要があります。386互換のPCの場合、フロッピーディスクドライブコントローラは、ハードウェア割り込み要求信号IRQ6 (割り込みディスクリプター0x26に対応) を使用し、DMAコントローラのチャンネル2を使用します。DMA制御処理の詳細については、次のセクションを参照してください。

9.6.3.3 フロッピーディスクコントローラコマンド

フロッピーディスクコントローラは、合計15個のコマンドを受け付けることができ、それぞれのコマンドは、コマンドフェーズ、実行フェーズ、結果フェーズの3つのフェーズを経て実行されます。

コマンドフェーズとは、CPUがFDCにコマンドバイトとパラメータバイトを送信することです。最初のバイトは常にコマンドバイト (コマンドコード) です。これに続いて0~8バイトのパラメータが続く。これらのパラメータは通常、ドライブ番号、ヘッド番号、トラック番号、セクタ番号、読み取り/書き込みセクタの合計数です。

実行フェーズは、FDCコマンドで指定された動作です。実行フェーズでは、CPUはFDCに介入しません。通常、FDCはコマンド実行の終了をCPUに知らせるために、割り込み要求を発行する。CPUから送信されたFDCコマンドがデータを転送するものである場合、FDCは割り込みモードまたはDMAモードで動作することができます。割り込みモードは1バイト送信するたびに、DMAモードは一度に大量のデータを転送することができる。DMAコントローラの管理のもと、すべてのデータが転送されるまで、FDCとメモリの間でデータが転送される。この時、DMAコントローラはFDCに転送バイト数終了信号を通知し、最後にFDCは割り込み要求信号を発行してCPUに実行フェーズの終了を知らせます。

リザルトフェーズは、CPUがFDCデータレジスタ (リザルトレジスタ) の戻り値を読み、FDCコマンドの実行結果を得ることです。返されるリザルトデータは、0~7バイトの長さです。リザルトデータが返ってこないコマンドについては、FDCに検出割り込みステータスコマンドを送り、動作状況を把握する必要があります。

Linux 0.12のフロッピードライバでは、15個のコマンドのうち6個しか使用していないので、ここでは使用しているコマンドのみを紹介します。

1. 再校正コマンド (FD_RECALIBRATE)

このコマンドは、ヘッドをトラック0に戻すために使用します。通常、フロッピーディスクの操作にエラーが発生した場合に、ヘッドの再校正に使用します。コマンドコードは0x07、パラメータには指定したドライブレター (0~3) を指定します。

このコマンドは結果の段階を持たないため、プログラムは「割り込み状態の検出」コマンドを実

行して実行結果を得る必要があります。このコマンドのフォーマットを表9-16に示します。

表 9-16 再校正コマンド（FD_RECALIBRATE）のフォーマット

フェーズ	Seq	D7	D6	D5	D4	D3	D2	D1	D0	説明
コマンド	0	0	0	0	0	0	1	1	1	Recalibrationコマンドコード。0x07
パラメータ	1	0	0	0	0	0	0	US1	US2	ディスクドライブの番号。
実行										ヘッドが0トラックに移動
結果		なし。								コマンドを使用する必要があります。

			の実行結果です。
--	--	--	----------

2. ヘッドシークコマンド (FD_SEEK)

このコマンドは、選択されたドライブのヘッドを、指定されたトラックに移動させます。第1パラメータでは、ドライブ番号とヘッド番号を指定します。ビット0～1がドライブ番号、ビット2がヘッド番号で、その他のビットは無意味です。第2パラメータではトラック番号を指定します。

このコマンドは結果のフェーズを持たないため、プログラムは「割り込み状態の検出」コマンドを実行して実行結果を得る必要があります。このコマンドのフォーマットを表9-17に示します。

表 9-17 ヘッドシークコマンド (FD_SEEK) のフォーマット

フェーズ	Seq	D7	D6	D5	D4	D3	D2	D1	D0	説明
コマンド	0	0	0	0	0	1	1	1	1	ヘッドシークコマンドコード。0x0F
パラメータ	1	0	0	0	0	0	HD	US1	US2	ヘッドナンバー、ドライブナンバー
	2	C								トラックナンバー
実行										指定されたトラックにヘッドが移動します。
結果		なし。								コマンドを使用する必要があります。 の実行結果です。

3. セクターデータ読み出しコマンド (FD_READ)

このコマンドは、ディスク上の指定された位置から始まるセクターを読み取り、DMA コントローラを介してシステム・メモリ・バッファに転送するために使用されます。セクターが読み込まれるたびに、パラメータ4 (R) が自動的に1つインクリメントされ、DMAコントローラがフロッピー・ディスク・コントローラに送信カウント終了信号を送るまで、次のセクターの読み取りを続けます。このコマンドは、通常、ヘッド・シーク・コマンドが実行され、ヘッドがすでに指定されたトラック上にある後に開始されます。コマンドのフォーマットを表9-18に示します。

返された結果のうち、トラック番号Cとセクター番号Rは、現在のヘッドが位置する位置です。開始セクタ番号Rは、1つのセクタを読み取ると自動的に1つずつ増加するため、結果のRの値は次の未読セクタ番号となります。トラックの最後のセクタ（つまりEOT）が読み込まれると、トラック番号も1つインクリメントされ、R値は1にリセットされます。

表 9-18 セクターデータ読み出しコマンド (FD_READ) のフォーマット

フェーズ	Seq	D7	D6	D5	D4	D3	D2	D1	D0	説明
コマンド	0	MT	MF	SK	0	0	1	1	0	セクターコマンドコードの読み込み 0xE6 (MT=MF=SK=1)
パラメータ	1	0	0	0	0	0	0	US1	US2	ドライブ番号
	2	C								トラック番号（シリンダーアドレス、0～255）
	3	H								ヘッド番号（ヘッドアドレス、0または1）
	4	R								スタートセクター番号（セクターアドレス）
	5	N								セクターサイズコード（N=0...7: 128,256,512,16KB）
	6	EOT								トラックの最終セクター番号（エンドオブ

			トラック
	7	GPL	セクター間のギャップの長さ (3)
	8	DTL	セクター内のバイト数 (N=0の場合)
実行			ディスクからシステムへのデータ転送
結果	1	ST0	ステータスバイト 0

2	ST1	ステータスバイト1
3	ST2	ステータスバイト2
4	C	トラック番号
5	H	ヘッダー番号
6	R	セクター番号
7	N	セクターサイズコード (0~7: 128,256,512,....,16KB

その中でも、MT、MF、SKの意味は

- MTはマルチトラック動作を表します。MT=1は、2つのヘッドが同じトラックで連続して動作することを意味します。
- MFは録音方式を示す。MF=1はMFM録音モードを選択することを意味し、それ以外はFM録音モードとなります。
- SKは、削除フラグのあるセクタをスキップするかどうかを示します。SK=1はスキップを意味する。

返された3つのステータスバイトST0、ST1、ST2の意味をそれぞれ表9-19、表9-20、表9-21に示す。

表9-19 ステータスバイト0 (ST0)

ビット	名前	説明
7	ST0_INTR	割り込みの理由00 - 正常なコマンドの終了、01 - コマンドの異常終了、10 - 無効なコマンド、11 - 異常な終了 Pollingによって引き起こされる。
6		
5	ST0_SE	シークエンド。コントローラは、SEEKまたはRECALIBRATEコマンドを完了しました。 または、READ/WRITE with implied seek コマンドを使用します。
4	ST0_ECE	装備する。チェックエラー。Recalibration Track 0エラー。
3	ST0_NR	Not Readyです。フロッピーディスクドライブの準備ができていない。
2	ST0_HA	ヘッドアドレス。割り込みが発生したときの、現在のヘッド番号。
1	ST0_DS	ドライブセレクト。割り込み発生時のドライブ番号。 00~11はドライブ0~3に対応します。
0		

表9-20 ステータスバイト1 (ST1)

ビット	名前	説明
7	ST1_EOC	シリンダーの終了。トラックの最終セクターより先のセクターにアクセスしようとした。
6		未使用です。このビットは常に0です。
5	ST1_CRC	コントローラは、セクタのIDフィールドまたはDataフィールドでCRCエラーを検出しました。
4	ST1_OR	オーバーラン。データ転送のタイムアウト、DMAコントローラの故障
3		未使用 (0)。
2	ST1_ND	データがありません。指定されたセクターが見つかりませんでした。

1	ST1_WP	ライトプロテクト。
0	ST1_MAM	Missing Address Mask.セクターIDのアドレスマークが見つかりません。

表 9-21 ステータスバイト 2 (ST2)

ビット	名前	説明
-----	----	----

7		未使用 (0)。
6	ST2_CM	コントロールマークです。SK=0の場合、読み出したデータは削除フラグに遭遇します。
5	ST2_CRC	CRCエラー。セクターデータフィールドのCRCチェックエラーです。
4	ST2_WC	シリンダーが違います。セクターID情報のトラック番号Cが一致しません。
3	ST2_SEH	スキャン・イコール・ヒット。スキャン条件は条件を満たしていません。
2	ST2_SNS	Scan Not Satisfied: スキャン条件が要件を満たしていない
1	ST2_BC	バッドシリンダーです。セクターID情報でトラックC=0xFFとなっており、トラックが不良です。
0	ST2_MAM	Missing Address Mask. セクターIDデータのアドレスマークが見つかりませんでした。

4. セクターデータ書き込みコマンド (FD_WRITE)

このコマンドは、メモリ・バッファのデータをディスクに書き込むために使用します。DMA 転送モードでは、フロッピー・ドライブ・コントローラは、メモリ内のデータをディスクの指定されたセクタにシリアルに書き込みます。セクターが書き込まれるたびに、開始セクタ番号が自動的に1つずつ増加し、フロッピー・ドライブ・コントローラがDMAコントローラからカウント終了信号を受け取るまで、1セクターの書き込みを続けます。コマンドのフォーマットを表 9-22 に示します。省略名はリードコマンドと同じ意味です。

表 9-22 セクタデータ書き込みコマンド (FD_WRITE) のフォーマット

フェーズ	Seq	D7	D6	D5	D4	D3	D2	D1	D0	説明
コマンド	0	MT	MF	0	0	0	1	0	1	データ書き込みコマンドコード。0xC5 (MT=MF=1)
パラメータ	1	0	0	0	0	0	0	US1	US2	フロッピーディスクの番号。
	2	C								トラックナンバー
	3	H								ヘッドナンバー
	4	R								スタートセクター番号
	5	N								セクターサイズコード。
	6	EOT								トラックの最終セクター番号
	7	GPL								セクター間のギャップの長さ (3)
	8	DTL								セクター内のバイト数 (N=0の場合)
実行										データがシステムからディスクに転送される
結果	1	ST0								ステータスバイト 0
	2	ST1								ステータスバイト 1
	3	ST2								ステータスバイト 2
	4	C								トラック番号
	5	H								ヘッドナンバー
	6	R								セクター番号
	7	N								セクターサイズコード。

5. 割り込みステータスチェックコマンド (FD_SENSEI)

このコマンドを送信すると、フロッピーコントローラは直ちに通常の結果1と2（つまり、ステータスST0とヘッドがあるトラック番号PCN）を返します。これらは、コントローラが前のコマンドを実行した後の結果の状態です。割り込み信号は、通常、コマンドの実行後にCPUに送られます。リード/ライトセクタ、リード/ライトトラック、リード/ライトデリートフラグ、リードIDフィールド、フォーマットコマンド、スキャンコマンド、非DMA転送モードのコマンドによる割り込みについては、メインステータスレジスタのフラグをもとに、割り込みの原因を直接知ることができます。ドライブのレディ信号変化による割り込みの場合。

seek and recalibration (head return to zero)では、リターン結果がないため、コントローラがコマンドを実行した後に、このコマンドを使ってステータス情報を読み取る必要があります。本コマンドのフォーマットを表 9-23 に示します。

表 9-23 チェックインタラプトステータスコマンド (FD_SENSEI) のフォーマット

フェーズ	Seq	D7	D6	D5	D4	D3	D2	D1	D0	説明
コマンド	0	0	0	0	0	1	0	0	0	検出割込みの状態cmdコード。0x08
実行										
結果	1	ST0								ステータスバイト 0
	2	C								現在のヘッドのトラック番号。

6. ドライブパラメータ設定コマンド(FD_SPECIFY)

このコマンドは、3 つの初期タイマー値とフロッピーディスクコントローラ内部で選択された伝送モード、つまりドライブモータのステップレート (SRT)、ヘッドのロード/アンロード (HLT/HUT) 時間、伝送に DMA モードを使用するかどうかを設定し、フロッピードライブコントローラに送信します。このコマンドのフォーマットを表9-24に示します。時間単位は、データ転送速度が500KB/Sのときの値です。また、Linux 0.12 カーネルでは、コマンドフェーズのパラメータバイト 1 は floppy.c ファイルの 96-103 行目のオリジナルコメントに記載されている spec1、パラメータバイト 2 は spec2 となっています。オリジナルコメントと316行目のプログラムステートメントから、spec2は6に固定されており（つまり、HLT=3、ND=0）、ヘッドロード時間が6ミリ秒であること、DMAモードを使用することを示しています。

表 9-24 ドライブパラメータ設定コマンド (FD_SPECIFY) のフォーマット

フェーズ	Seq	D7	D6	D5	D4	D3	D2	D1	D0	説明
コマンド	0	0	0	0	0	0	0	1	1	セットパラメータのコマンドコード0x03
パラメータ	1	SRT (単位: 1ms)				HUT (単位: 16ms)				モーターステップレート、ヘッドアンロード時間
	2	HLT (単位: 2ms)							ND	ヘッドロード時間、非DMAモード
実行										コントローラを設定しても、割り込みは発生しません。
結果		なし。								なし。

9.6.3.4 フロッピーディスクコントローラのプログラミング

パソコンでは、フロッピーディスクコントローラには、一般的にNECのPD765やインテルの82078など、インテル8287Aの互換チップが使われています。フロッピーディスクのドライバは比較的複雑なので、このようなチップで構成されたフロッピーディスクコントローラのプログラミング方法を以下に詳しく説明します。一般的なディスク操作は、コマンドを送ってコントローラが結果を返すのを待つだけではない。フロッピーディスクドライブの制御は低レベルの操作であり、プログラムはその実行をさまざまな段階で干渉させる必要がある。

1. コマンドフェーズと結果フェーズの相互作用

上記のディスク操作コマンドやパラメータをフロッピーディスクコントローラに送信する前に、

まずコントローラのメインステータスレジスタ（MSR）を照会して、ドライブのレディ状態やデータ転送方向を知る必要があります。フロッピーディスク・ドライバーは、これを行うために `output_byte(byte)` 関数を使用します。この関数の等価ブロック図を図9-9に示します。

この関数は、メインステータスレジスタのデータポートレディフラグRQMが1、ディレクションフラグDIOが0（CPU→FDC）になるまでループし、その時点でコントローラはコマンドおよびパラメータのバイトを受け入れることができます。ループ文は、コントローラが次のような状態にならない場合に対応するため、タイムアウトカウント機能から開始します。

に対応しています。このドライバーでは、ループ回数を10000回に設定しています。プログラムが誤ったタイムアウトをしないように、ループ回数の選択には注意が必要です。Linuxカーネルバージョン0.1x～0.9xでは、ループ回数の調整が必要になることがよくあります。これは、当時使用されていたPCの速度が異なるため（16MHz～40MHz）、ループによる実際の遅延が大きく異なってしまうためです。このことは、初期のLinuxメーリングリストで多くの記事が議論されていることからわかります。この問題を完全に解決するには、システムのハードウェアクロックを使って固定周波数の遅延値を生成するのが一番です。

また、コントローラの結果バイトを読み出すリザルトフェーズでは、データ転送方向のフラグ要求がセット（FDC→CPU）されていることを除けば、送信コマンドと同じ操作方法を取る必要があります。本プログラムの対応する関数は `result()` で、結果ステータスバイトを `reply_buffer[]` バイト配列に格納します。

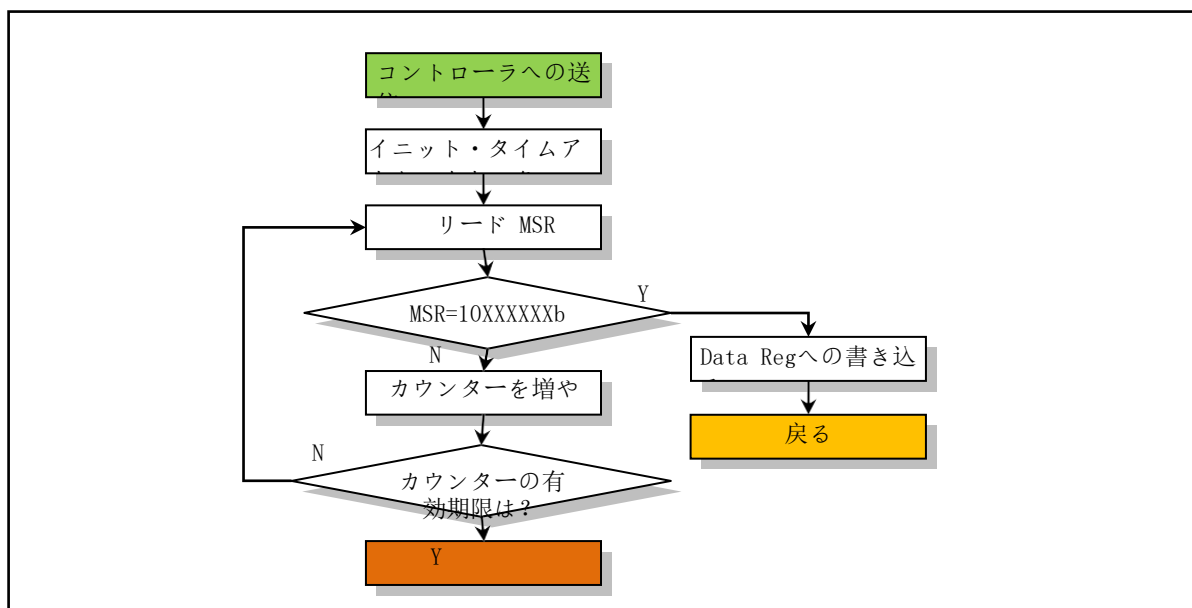


図9-9 フロッピーコントローラへのコマンドまたはパラメータバイトの送信

2. フロッピーディスクコントローラの初期化

フロッピーディスクコントローラの初期化では、コントローラがリセットされた後に、ドライブに適切なパラメータを設定します。コントローラのリセット操作は、FDCのリセットフラグ（DORのビット2）を0（リセット）にしてから

1.FDCリセット後は、「ドライブパラメータ指定」コマンドSPECIFYで設定した値が有効でなくなるため、再設定が必要です。floppy.cプログラムでは、リセット操作は、関数 `reset_floppy()` と割込みハンドラC関数 `reset_interrupt()` に含まれています。前者の関数は、DORレジスタのビット2を変更してコントローラをリセットするために使用されます。後者の関数は、コントローラがリセットされた後、SPECIFY コマンドを使用してコントローラ内のドライブパラメータを再設定するために使用されます。データ転送の準備段階で、FDC内の現在のドライブパラメータが実際のディスク仕様と異なることが検出された場合、転送関数 `transfer()` の最初に追加リセットされます。

コントローラがリセットされた後、データ転送速度を再初期化するために、指定された転送速度をデジタル制御レジスタDCRにも送信する必要があります。マシンがリセット操作（ウォームブート

など)を行った場合、データ転送レートはデフォルト値の250Kpbsになります。しかし、デジタル出力レジスタDORを介してコントローラに発行されたリセット操作は、データ転送速度に影響を与えません。

3. ドライブの再較正とヘッドシーク

ドライバの再校正（FD_RECALIBRATE）とヘッドシーク（FD_SEEK）は、2つのヘッド位置決めコマンドです。recalibrationコマンドはヘッドをゼロトラックに移動させ、head seekコマンドはヘッドを指定したトラックに移動させます。この2つのヘッドポジショニングコマンドは、一般的なリード/ライトコマンドとは異なり、結果のフェーズがありません。この2つのコマンドのいずれかが発行されると、コントローラは直ちにMSR（Main Status Register）のReady状態に戻り、バックグラウンドでヘッドの位置決め動作を行います。位置決め動作が完了すると、コントローラは割り込み要求サービスを生成します。このとき、「Detect Interrupt Status」コマンドを送信して割り込みを終了させ、位置決め動作後のステータスを読み出す必要があります。ドライブやモータのイネーブル信号は、デジタル出力レジスタ（DOR）によって直接制御されるため、ドライブやモータが起動していない場合は、位置決めコマンドを発行する前にDORを書き込む操作を行う必要があります。関連するフローチャートを図9-10に示します。

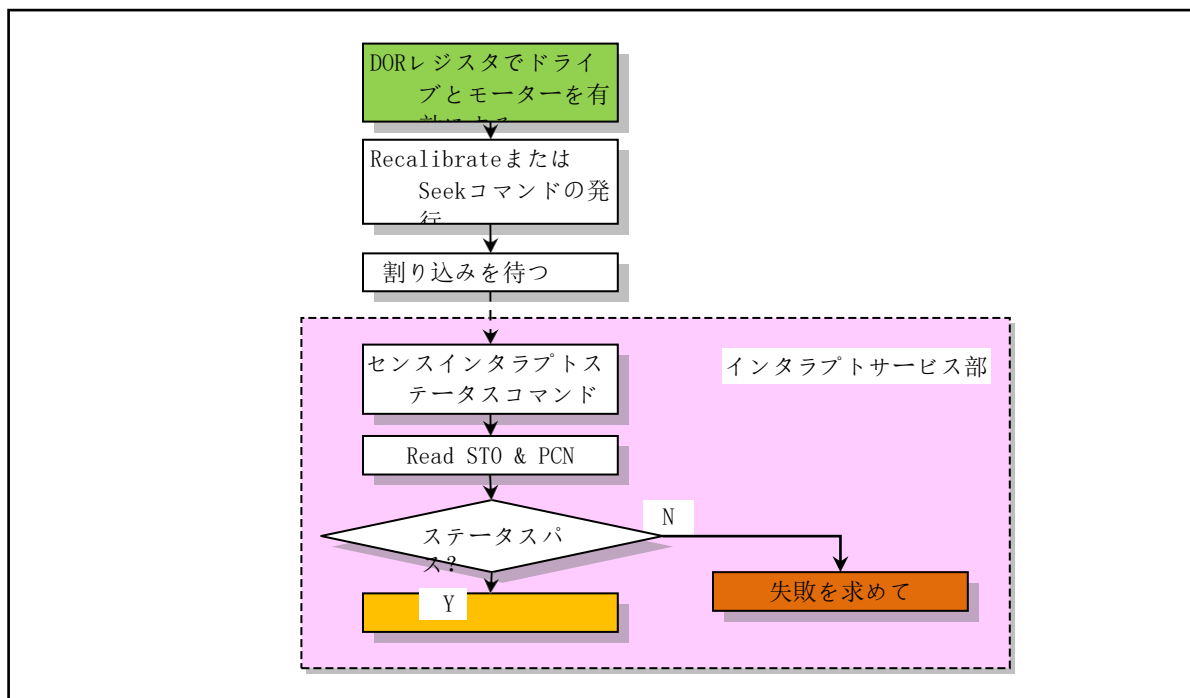


図9-10 再校正とシーク操作

4. データの読み取り/書き込み操作

データの読み取りまたは書き込み操作は、いくつかのステップを経て完了します。まず、ドライブモータの電源を入れ、ヘッドを正しいトラックに配置し、次にDMAコントローラを初期化し、最後にデータの読み取りまたは書き込みコマンドを送信する必要があります。さらに、エラーが発生したときの処理計画を決定する必要があります。典型的な動作フローチャートを図9-11に示します。

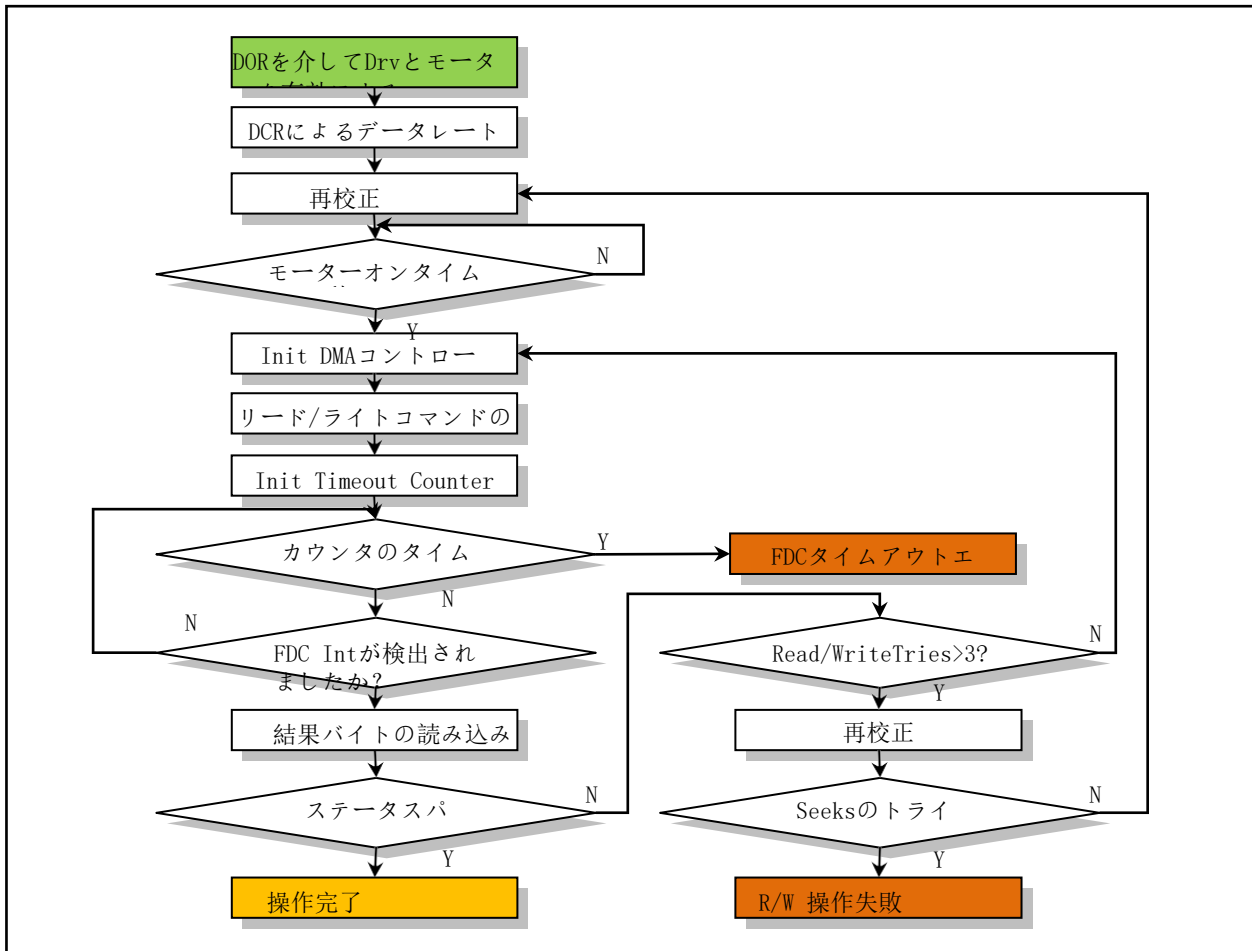


図9-11 データリード/ライトの動作フロー図

ディスクドライブのモーターが正常な動作速度に達してから、ディスクがデータの転送を開始します。ほとんどの31/2インチフロッピードライブでは、この起動時間は約300ms、51/4インチフロッピードライブでは約500msかかります。この起動遅延時間は、`floppy.c`プログラムで500msに設定しました。

モーター起動後、デジタルコントロールレジスタDCRを使って、現在のフロッピーディスク媒体に合ったデータ転送速度を設定する必要があります。

暗黙のシークモードが有効になっていない場合は、ヘッドを正しいトラックに配置するために、シークコマンドFD_SEEKを送信する必要があります。シーク動作が完了した後、ヘッドのロード時間もかかります。ほとんどのドライブでは、この遅延は少なくとも15msかかります。暗黙のシークモードを使用している場合、「ドライブパラメータの指定」コマンドで指定されたヘッドロードタイム（HLT）を使用して、最小のヘッド到着時間を決定することができます。例えば、データ転送レートが500Kbpsの場合、HLT=8であれば、有効なヘッドのインポジション時間は16msとなります。もちろん、すでにヘッドが正しいトラックに配置されていれば、この時間はかかりません。

その後、DMAコントローラが初期化され、リードコマンドとライトコマンドが実行されます。通常、データ転送が完了すると、DMAコントローラはTC（Termination Count）信号を発行します。この時点で、フロッピーディスクコントローラは現在のデータ転送を完了し、動作が結果段階に達したことを示す割り込み要求信号を発行します。動作中にエラーが発生した場合や、最終セクタ番号がトラックの最終セクタと等しい場合（EOT）、フロッピーディスクコントローラは直ちに結果段階に

入ります。

上のフロー図によると、結果ステータスバイトを読み取った後にエラーが見つかった場合、DMAコントローラを再初期化することで、データの読み取りまたは書き込み動作コマンドが再開されます。継続的なエラーは、通常、シーク操作によってヘッドが指定されたトラックに到達しなかったことを示し、再校正のための

ヘッドを複数回繰返し、シーク操作を再度行う必要があります。それでもエラーが発生した場合は、コントローラがドライバに読み取りまたは書き込み操作の失敗を報告します。

5. ディスクフォーマット操作

Linux 0.12カーネルでは、フロッピーディスクのフォーマット動作は実装されていませんが、参考までにディスクのフォーマット動作を簡単に説明しておきます。ディスクのフォーマット動作では、各トラックにヘッドを配置し、データフィールドを構成するための固定フォーマットフィールドを作成する。

モーターが起動し、正しいデータ転送レートが設定されると、ヘッドはゼロトラックに戻ります。この時点で、ディスクは500msの遅延時間内に通常の安定した動作速度に達する必要があります。

フォーマット動作でディスクに設けられた識別フィールド（IDフィールド）は、実行段階でDMAコントローラから提供されます。DMAコントローラは、各セクタ識別フィールドのトラック（C）、ヘッド（H）、セクタ番号（R）、セクタバイトの値を提供するように初期化されます。例えば、1トラックに9セクタあるディスクの場合、各セクタサイズは2（512バイト）となります。トラック7がヘッド1でフォーマットされている場合、DMAコントローラは36バイト（9セクタ×1セクタ4バイト）を送信するようにプログラムされている必要があります、データフィールドは次のようになります。7,1,1,2,7,1,2,2,7,1,3,2,...,7, 1,9,2. フロッピーディスクコントローラから提供されるデータは、フォーマットコマンドの実行時に識別フィールドとして直接ディスクに記録されるため、データの内容は任意である。そのため、保護されたディスクのコピーを防ぐためにこの機能を使う人もいます。

あるトラックの各ヘッドがフォーマット動作を行った後、次のトラックにヘッドを進めてフォーマット動作を繰り返すためには、シーク動作を行う必要があります。Formatted Trackコマンドには、暗黙のシーク操作が含まれていないため、シークコマンドSEEKを使用する必要があります。同様に、前述のヘッドインポジション時間もシークのたびに設定する必要があります。

9.6.3.5 DMAコントローラのプログラミング

DMA（Direct Memory Access）コントローラの主な目的は、外部デバイスがメモリに直接データを転送できるようにすることで、システムのデータ転送性能を向上させることにあります。通常は、マシンに搭載されているインテル8237Aチップまたはその互換チップによって実装されています。DMAコントローラをプログラムすることで、周辺機器とメモリー間のデータ転送をCPUとは独立して行うことができます。そのため、CPUはデータ転送中に他の作業を行うことができます。DMAコントローラがデータを転送する際の動作プロセスは以下の通りです。

1. DMAコントローラの初期化を行います。

プログラムは、DMAコントローラポートを介して初期化を行います。(1)DMAコントローラへの制御コマンドの送信、(2)転送用のメモリスタートアドレスの送信、(3)データ長の送信。送られたコマンドは、転送に使用するDMAチャネルの有無、メモリをペリフェラルに転送するのか（ライト）、ペリフェラルのデータをメモリに転送するのか、1バイト転送なのか、ブロック（ブロック）転送なのかを示しています。PCの場合、フロッピーディスクコントローラはDMAチャンネル2を使用するように指定されています。Linux 0.12カーネルでは、フロッピーディスク・ドライバーはシングルバイト転送モードを使用しています。インテル8237Aチップのアドレス端子は16本（うち8本はデータ線と兼用）しかないため、アドレスできるメモリ空間は64KBに限られます。PCは、1MBのアドレス空間にアクセスできるようにするために、表9-25のように、

LS670チップをDMAページレジスタとして使用し、1MBのメモリを16ページに分割して動作させています。そのため、転送されたメモリのスタートアドレスは、DMAページ値とページ内のオフセットアドレスに変換する必要があり、各転送のデータ長は64KBを超えることはできません。

このことから、メモリに設定する転送バッファは、1MBのアドレス空間内になければならないことがわかります。しかし、実際のデータバッファ（ユーザーバッファなど）が1MB空間の外にある場合、DMAが使用するための一時的な転送バッファをメモリの1MBアドレス領域に設定し、転送されたデータをコピーする必要があります。

データを一時的なバッファと実際のユーザーバッファの間に配置します。これはまさにLinux 0.12カーネルの使用方法であり、関数setup_DMA()を参照してください（プログラムfloppy.c、106行目および161行目）。

表9-25 DRAMページ対応メモリアドレス範囲

DMAページ	アドレス範囲（64KB
0x00	0x00000 - 0x0FFFF
0x01	0x10000 - 0x1FFFF
0x02	0x20000 - 0x2FFFF
0x03	0x30000 - 0x3FFFF
0x04	0x40000 - 0x4FFFF
0x05	0x50000 - 0x5FFFF
0x06	0x60000 - 0x6FFFF
0x07	0x70000 - 0x7FFFF
0x08	0x80000 - 0x8FFFF
0x09	0x90000 - 0x9FFFF
0x0A	0xA0000 - 0xAFFFF
0x0B	0xB0000 - 0xBFFFF
0x0C	0xC0000 - 0xCFFFF
0x0D	0xD0000 - 0xDFFFF
0x0E	0xE0000 - 0xEFFFF
0x0F	0xF0000 - 0xFFFFF

2. データ送信

初期化完了後、DMAコントローラのマスキレジスタを変更し、DMAチャンネル2をイネーブルにすることで、DMAコントローラはデータ転送を開始する。

3. 送信終了

転送すべきデータがすべて転送されると、DMAコントローラはフロッピーコントローラに「エンドオブプロセス」（EOP）信号を生成します。この時点で、フロッピーディスクコントローラは、ドライブモータをオフにし、CPUに割り込み要求信号を送るという終了操作を行うことができます。

PC/ATマシンでは、DMAコントローラは8つの独立したチャンネルを使用でき、そのうち最後の4つのチャンネルは16ビットです。フロッピーディスクコントローラは、DMAチャンネル2を使用するように指定されています。使う前にまず設定する必要があります。これには、ページレジスタポート、（オフセット）アドレスレジスタポート、データカウントレジスタポートの3つのポートに対する操作が必要です。DMAレジスタは8ビットで、アドレス値とカウント値は16ビットなので、2回送信する必要があります。まずローバイトを送信し、次にハイバイトを送信します。各チャンネルに対応するポートアドレスを表9-26に示します。

表9-26 各DMAチャンネルが使用するページ、アドレス、カウントレジスタポート

DMAチャネル	ページ登録	ベースアドレスレジス	ワードカウントレジ
---------	-------	------------	-----------

ル		タ	ス タ
0	0x87	0x00	0x01
1	0x83	0x02	0x03
2	0x81	0x04	0x05
3	0x82	0x06	0x07

4	0x8F	0xC0	0xC2
5	0x8B	0xC4	0xC6
6	0x89	0xC8	0xCA
7	0x8A	0xCC	0xCE

通常のDMAアプリケーションでは、DMAコントローラの動作と状態を制御する5つの共通レジスタがあります。これらは、表9-27に示すように、コマンドレジスタ、リクエストレジスタ、シングルマスクレジスタ、モードレジスタ、およびクリアプリ/ポストポインタトリガーです。Linux 0.12カーネルでは、表中の3つの網掛けのレジスタポート（0x0A、0x0B、0x0C）を主に使用しています。

表9-27 DMAプログラミングでよく使われるレジスタ

レジスタ名	リード/ライト	ポートアドレス	
		チャンネル0～3	チャンネル4～7
ステータスレジスタ/コマンドレジスタ	リード/ライト	0x08	0xD0
リクエスト登録	書く	0x09	0xD2
シングルチャンネルマスクレジスタ	書く	0x0A	0xD4
モードレジスタ	書く	0x0B	0xD6
クリアファースト/ラストフリップフロップ	書く	0x0C	0xD8
一時レジスタ/メータクリア	リード/ライト	0x0D	0xDA
マスクレジスタのクリア	書く	0x0E	0xDC
フルマスクレジスタ	書く	0x0F	0xDE

コマンドレジスタ

コマンドレジスタは、DMAコントローラチップの動作要件を指定し、DMAコントローラの全体的な状態を設定するために使用されます。通常、ブート初期化後に変更する必要はありません。Linux

0.12カーネルでは、フロッピードライバは起動後にROM

BIOSの設定を直接使用します。参考までに、コマンドレジスタのビットの意味を表9-

28のように示します。なお、同じポートを読み出す場合は、DMAコントローラのステータスレジスタの情報を取得します。

表9-28 DMAコマンドレジスタフォーマット

ビット	説明
7	DMA応答周辺信号DACK: 0-DACKアクティブロー、1-DACKアクティブハイ。
6	ペリフェラルリクエストDMA信号DREQ: 0-DREQアクティブロー、1-DREQアクティブハイ。
5	書き込みモードの選択。0はレイトライト、1はエクステンデッドライトを選択、X-ifビット3=1。

4	DMAチャンネルのプライオリティモード。0-固定優先、1-回転優先。
3	DMAサイクルの選択。0 - 通常のタイミングサイクル（5）、1 - 圧縮タイミングサイクル（3）、X - ビット0 = 1の場合。
2	DMAコントローラの起動：0 - コントローラを有効にし、1 - コントローラを無効にする。
1	チャンネル0のアドレスホールド。0 - チャンネル0のアドレスのホールドを無効にする；1 - チャンネル0のアドレスのホールドを許可する；X - 。 if bit 0 = 0.
0	メモリ転送モード。0 - メモリからメモリへの転送を無効にする、1 - メモリからメモリへの転送を有効にする。

リクエスト・レジスタ

--

8237Aは、DREQだけでなく、ソフトウェアによって開始されたDMAサービスのリクエストにも応答することができます。リクエスト・レジスタは、チャンネルに対するペリフェラルのリクエスト・サービス信号DREQを、チャンネルごとに1ビットずつ記録するためのものです。DREQがアクティブなときはポジション1に対応し、以下の場合には0に設定されます。

がDMAコントローラに応答することになります。また、DMAリクエスト信号DREQ端子を使用しない場合は、対応するチャンネルのリクエストビットをプログラミングで直接設定することで、DMAコントローラのサービスを要求することもできます。PCの場合、フロッピーディスクコントローラはDMAコントローラのチャンネル2に直接リクエスト信号DREQで接続されているので、Linuxカーネルでレジスタを操作する必要はありません。参考までに、リクエストチャンネルサービスのバイトフォーマットを表9-29に示す。

表9-29 DMAリクエストレジスタの各ビットの意味

ビット	説明
7-3	使われていません。
2	フラグを設定します。0 - リクエストビットがセットされます。1 - リクエストビットがリセットされます（0に設定されます）。
1	チャンネル選択。00-11はチャンネル0-3を選択します。
0	

マスクレジスタ

--

シングルマスクレジスタのポートは0x0A（16ビットチャンネルは0xD4）です。チャンネルがマスクされているということは、そのチャンネルを使用しているペリフェラルが発行したDMA要求信号DREQがDMAコントローラからの応答を得られないことを意味し、したがって、そのチャンネルではDMAコントローラを操作することができません。各チャンネルには、入力されたDREQを無効にするために設定可能なマスクビットが関連付けられています。本レジスタの各ビットの意味を表9-30に示します。

表9-30 DMAシングルマスクレジスタの各ビットの意味

ビット	説明
7-3	使われていません。
2	マスクフラグ。1 - マスクビットをセット、0 - マスクビットをクリア。
1	チャンネル選択。00-11はチャンネル0-3を選択します。
0	

モードレジスタ

--

モードレジスタは、DMAチャンネルの動作方法を指定するために使用されます。このレジスタの各ビットの意味を表9-31に示す。Linux

0.12カーネルでは、リードディスク(0x46)とライトディスク(0x4A)の2つの設定モードが使用されています。表9-

31によると、設定モード：DMAチャンネル2を使用、リードディスク（ライトメモリ）転送、自己初期化を無効にする、アドレスインクリメントを選択、シングルバイトモードを使用、0x4Aは設定モード：DMAチャンネル2を使用、ライトディスク（リードメモリ）転送、自己初期化を無効にする、アドレスインクリメントを選択、シングルバイトモードを使用を意味します。

表9-31 DMAモードレジスタの各ビットの意味

ビット	説明
7	転送モードを選択します。00-リクエストモード、01-シングルバイトモード、10-ブロックバイトモード。 11カスケードモード。
6	
5	アドレス方式。0 - アドレスのインクリメント、1 - アドレスのデクリメント。
4	Autoinitialization（自動初期化）。0-自動初期化無効、1-自動初期化有効。
3	転送タイプ。00-ペリファイ転送、01-ライトメモリ転送、10-リードメモリ転送。 11- 違法; XX - ビット6-7=11の場合。
2	
1	チャンネル選択。00～11でそれぞれチャンネル0～3を選択します。
0	

チャンネルアドレスおよびカウントレジスタは16ビットのデータを読み書きできるため、ローバイトとハイバイトの2つの書き込み動作を同時に行う必要があります。実際にどちらのバイトが書き込まれるかは、ソフトウェアコマンドのclear first/last flip-flopの状態によって決まります。このコマンドは、8237Aに新しいアドレスやワードカウント情報を書き込んだり読み出したりする前に実行する必要があります。これにより、フリップフロップが既知の状態に初期化され、その後、CPUがレジスタの内容にアクセスする際に、上位バイトと下位バイトが正しい順序でアドレス指定されるようになります。ポート0x0Cは、DMAコントローラのアドレス情報やカウント情報を読み書きする前に、バイトオーダーフリップフロップをデフォルトの状態に初期化するために使用されます。このフリップフロップが0のときは、下位バイトにアクセスし、1のときは上位バイトにアクセスします。フリップフロップは、1回の訪問で1回変化します。0x0Cポートを書き込むことで、クリアバイトファースト/ラストフリップフロップを0の状態にすることができます。

DMAコントローラを使用する場合、通常は一定の手順を踏む必要があります。以下では、DMAコントローラの方法を用いて、DMAの簡単なプログラミング手順を説明します。

1. 干渉を排除するために、割り込みをオフにします。
2. マスクレジスタ（ポート0x0A）を変更して、使用する必要のあるDMAチャンネルをマスクします。フロッピーディスクドライバの場合はチャンネル2です。
3. 0x0Cポートに書き込み、"clear first/last flip-flop"をデフォルトの状態にする。
4. モードレジスタ（ポート0x0B）を書き込んで、指定したチャンネルの動作モードワードを設定する。
5. アドレスレジスタ（ポート0x04）を書き込み、DMAで使用するメモリページのオフセットアドレスを設定します。最初に下位バイトを書き込み、次に上位バイトを書き込みます。
6. ページレジスタ（ポート0x81）を書き込み、DMAで使用するメモリページを設定します。
7. カウントレジスタ（ポート0x05）を書き込み、DMA転送のバイト数を設定します。これは転送長-1になるはずですが、また、ハイバイトとローバイトを1回ずつ書き込む必要があります。本書では、DMAコントローラが必要とするフロッピーディスクドライバの長さは1024バイトなので、書き込みDMAコントローラの長さは1023（つまり0x3FF）にします。
8. マスクレジスタ（ポート0x0A）を再度変更し、DMAチャンネルを有効にします。
9. 最後に、割り込みを有効にして、転送が完了した後にフロッピーコントローラがシステムに割り込み要求を発行できるようにします。

9.7 概要

本章では、まずブロックデバイスドライバが使用するリクエストアイテムやリクエストキューなどの主なデータ構造を紹介し、次にシステムプロセッサ、デバイスコントローラ、特定のブロックデバイスの関係から、ブロックデバイスの一般的な動作方法を紹介します。ブロックデバイスのデータへのアクセスは、割り込みハンドラ方式を採用しており（仮想記憶ディスクを除く）、デバイスが読み書きコマンドを発行した後は、直接呼び出し元のプログラムに戻り、割り込み禁止のスリープ待ちキューに入って、ブロックデバイスの操作が完了するのを待つことができることを、改めて確認して

おきましょう。また，上位層プログラムのブロックデバイスデータへのアクセスは，統一された低レベルのブロックデバイス読み書き関数 `ll_rw_block()` によって実現されている。

次の章では、もうひとつのデバイスである、ターミナルなどのキャラクターデバイスについてご紹介します。コンソール機器。システムと直接通信するインタラクティブデバイスの一つです。

