

## 6 Booting System

This chapter describes three assembly language files in the boot/ directory, as shown in Listing 6-1. As mentioned in the previous chapter, these three files are all assembly language programs, but use two different syntax formats. bootsect.S and setup.S are real-mode 16-bit code programs that use Intel's assembly language syntax and require the 8086 assembly compiler and linker as86 and ld86. However, head.s uses an AT&T assembly syntax format and runs in protected mode, which needs to be compiled with GNU's as (gas) assembler.

当時、リーナス・トーバルズが2つのアセンブラーを使っていた主な理由は、インテルx86プロセッサーの場合、1991年のGNUアセンブラーはi386以降の32ビットCPUコード命令しかサポートしていませんでした。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アセンブラーは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りました(GNUアセンブラーのマニュアルの「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				
Filename	Size	Last Modified Time (GMT)	Description	
<a href="#">bootsect.S</a>	7574 bytes	1992-01-14 15:45:22		
<a href="#">head.s</a>	5938 bytes	1992-01-11 04:50:17		

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、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

(boot/setup.S)をブートデバイス内のメモリ0x90200にロードします。残りのカーネル（システムモジュール）は、メモリアドレス0x10000（64KB）の先頭に読み込まれてロードされます。したがって、マシンのパワーオンからの逐次実行のプロセスは、図6-1のようになります。

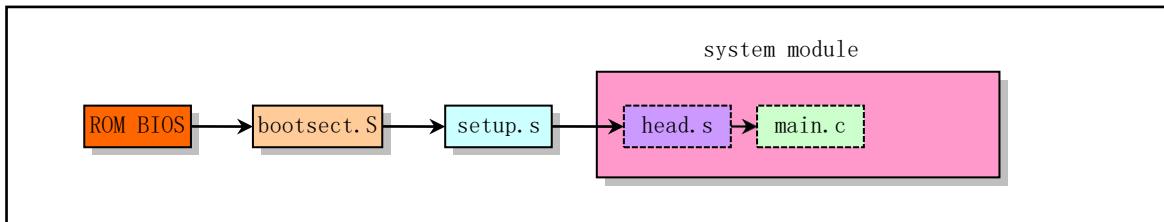


図6-1 システム電源投入時の実行順序

Because the length of the system module at that time will not exceed 0x80000 bytes (ie 512KB), therefore, when the bootsect program reads the system module into the physical address 0x10000 start position, it will not overwrite the bootsect and setup modules starting at 0x90000 (576KB). The subsequent setup program will also move the system module to the physical memory start location, so that the address of the code in the system module is equal to the actual physical address, which is convenient for operating the kernel code and data. Figure 6-2 clearly shows the dynamic location of these programs or modules in memory when the Linux system starts. In the figure, each vertical bar represents the image location map of each program in memory at a certain moment. The message "Loading..." will be displayed during system loading. Control is then passed to the code in boot/setup.S, which is another real-mode assembly language program.

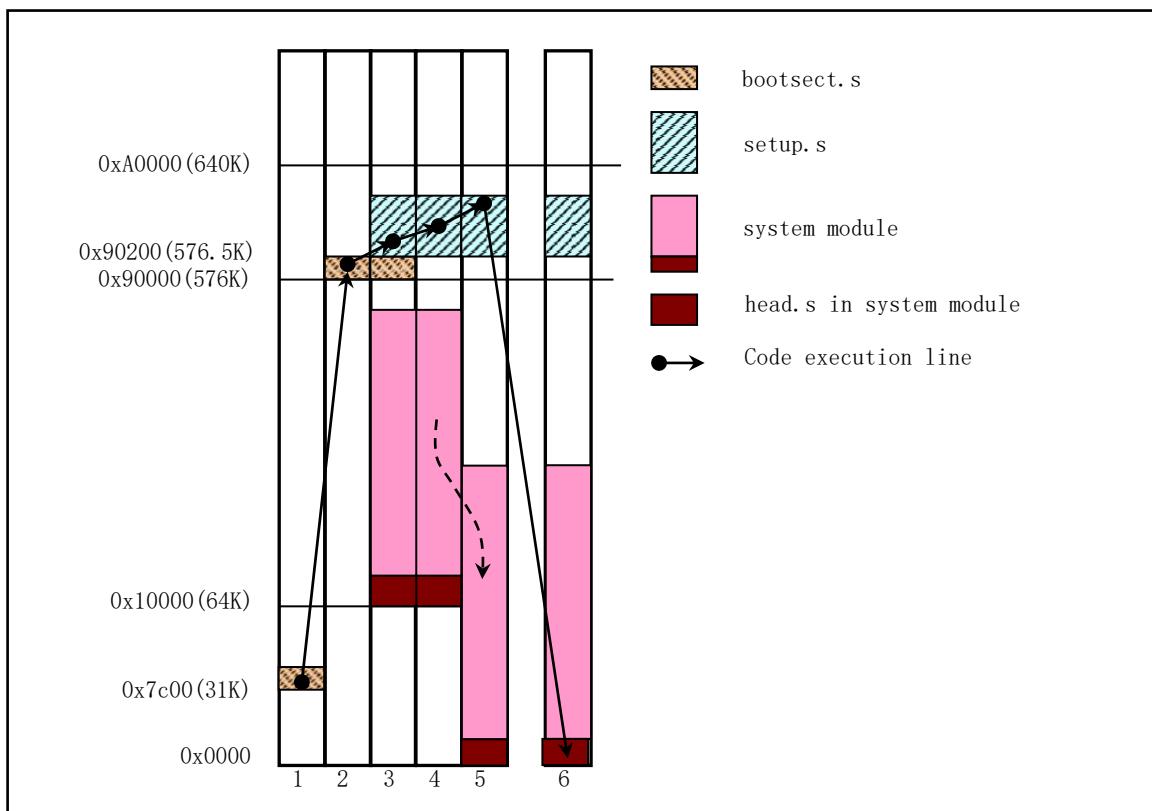


図6-2 メモリ内のカーネルの動的位置

The boot part identifies certain features of the host and the type of VGA card, and if required, it asks the user to select a display mode for the console. Then move the entire system from address 0x10000 to 0x0000, enter protected mode and jump to the rest of the system (at 0x0000). At this point, the boot settings for all 32-bit modes have been completed: IDT, GDT, and LDT are loaded, the processor and coprocessor are also confirmed, and paging is also set. Finally, the main() code in init/main.c is called. The source code for the above operation is in boot/head.s, which is probably the most tricky code in the entire kernel. Note that if something goes wrong in any of the previous steps, the computer will deadlock. It can't be handled before the operating system is fully operational.

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、と疑問に思う人がいるかもしれません。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コール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 Functional Description

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（シリンドー）、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。

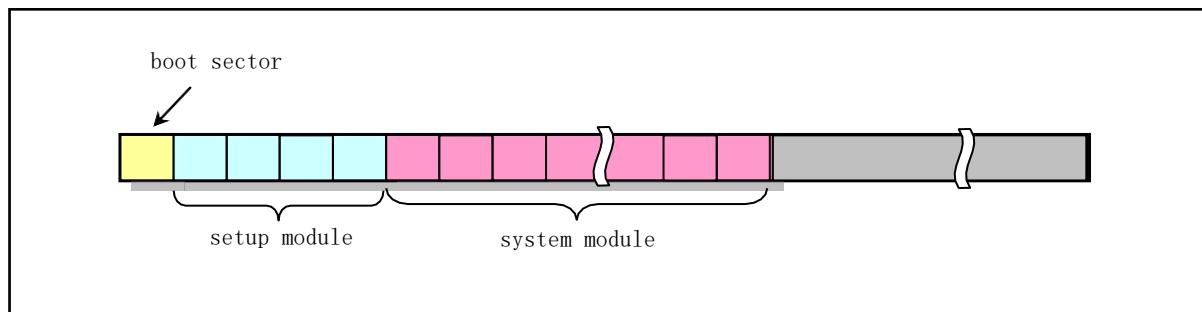


図 6-3 1.44MB のディスクに収められた Linux 0.12 カーネルの配布状況

The figure shows the distribution of sectors occupied by the Linux 0.12 kernel on a 1.44MB disk. There are 80 tracks (cylinders) on each side of the 1.44MB disk platter, each with 18 sectors and a total of 2880 sectors. The boot program code occupies the first sector, the setup module occupies the next four sectors, and the 0.12 kernel system module occupies approximately the next 260 sectors. There are still more than 2610 sectors left unused. These remaining unused space can be used to store a basic root file system, creating an integrated disk that can be used to run the system using a single disk. This will be covered in more detail in the chapter on block device drivers.

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ます。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。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、プログラム中の「#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 Code Comments

プログラム 6-1 linux/boot/bootsect.S

2 !ここでは、行頭の「！」や「;」がコメント文を表しています。1!

3 ! SYS\_SIZE is the number of clicks (16 bytes) to be loaded.

4 ! 0x3000 is 0x30000 bytes = 196kB, more than enough for current.

4 ! versions of linux

!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

ブートセクト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 - The default location for system module from disk.

```

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
13 ! 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
16 ! 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
19 ! 題ないはずです。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す。特に、minix のような21 ! バッファキャッ
21 ! シュが含まれていないので。
22 !
23 ! The loader has been made as simple as possible, and continuos
24 ! read errors will result in a unbreakable loop. Reboot by hand. It
25 ! loads pretty fast by getting whole sectors at a time whenever possible.
26
27 ! 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。
28 ! 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
29 !" 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。
30 ! 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。
31 ! 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。
32 ! 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
33 ! 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。
34 ! 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。
35 .globl begtext, begdata, begbss, endtext, enddata, endbss
36 .text                      ! text section.
37 begtext:
38 .data                      ! data section
39 begdata:
40 .bss                       ! uninitialized data section
41 begbss:
42 .テキスト
43
44 ! 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。
45 SETUPLEN = 4                ! nr of setup-sectors
46
47 BOOTSEG = 0x07c0            ! original address of boot-sector
48 INITSEG = DEF_INITSEG       ! we move boot here - out of the way
49
50 SETUPSEG = DEF_SETUPSEG    ! setup starts here
51
52 SYSSEG = DEF_SYSSEG        ! system loaded at 0x10000 (65536).
53 ENDSEG = SYSSEG + SYSSIZE   ! where to stop loading

```

42

43 ! ROOT\_DEV & SWAP\_DEV are now written by "build".

! 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す。

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 ! The root fs device uses the same boot device;

45 SWAP\_DEV = 0 ! The swap device uses the same boot device;

46

! ディレクティブのエントリは、リンカー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(ブートセクト)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

! jump to the label go of the moved code, that is, the next statement of the program.

47 entry start ! Tell the linker, program starts at the label start.

48 start:

49 mov ax, #BOOTSEG ! Set the ds segment register to 0x7C0;

50 mov ds, ax

51 mov ax, #INITSEG ! Set the es segment register to 0x9000;

52 mov es, ax

53 mov cx, #256 ! Set move count = 256 words (512 bytes);

54 sub si, si ! Source address ds:si = 0x07C0:0x0000

55 sub di, di ! Destination address es:di = 0x9000:0x0000

56 rep ! counting down, until cx = 0.

57 movw ! Move cx words from memory [si] to [di].

58 ! ジャンプインターフェースセグメント。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。

59 jmpi go, 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、スタックレジスターssとspを含むいくつかのセグメントレジスターを設定します。

! 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)、つまり  $sp = 0xfef4$  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がブートセクターを  $0x7c00$  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。

! One of the corrections is to remove line 65 and change line 109 to "mov ax, cs".  
go:        mov        ax, cs                      ! Set ds, es, and ss to segment after moved (0x9000).

```

60
61     mov    dx, #0fef4          ! arbitrary value >>512 - disk parm size
62
63     mov    ds, ax
64     mov    es, ax
65     push   ax                ! Temp save segment (0x9000) for 109 lines. (slipper!)
66
67     mov    ss, ax            ! put stack at 0x9ff00 - 12.
68     mov    sp, dx
69 /*
70 * Many BIOS's default disk parameter tables will not
71 * recognize multi-sector reads beyond the maximum sector number
72 * specified in the default diskette parameter tables - this may
73 * mean 7 sectors in some cases.
74 *
75 * Since single sector reads are slow and out of the question,
76 * we must take care of this by creating new parameter tables
77 * (for the first disk) in RAM. We will set the maximum sector
78 * count to 18 - the most we will encounter on an HD 1.44.
79 *
80 * High doesn't hurt. Low does.
81 *
82 * Segments are as follows: ds=es=ss=cs - INITSEG,
83 *                         fs = 0, gs = parameter table segment
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                ! set segment reg fs = 0.
88     pop    fs
89     mov    bx, #0x78          ! fs:bx is parameter table address
90
91     !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。
92     !"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
93     ! 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
94     ! es:di =  $0x9000:0xfef4$  が宛先アドレスとして使用されます。
95
96     seg fs
97     lgs    si, (bx)           ! gs:si is source.
98

```

```
93    mov    di, dx           ! es:di is destination ! dx = 0xfef4, set on line 61.  
94    mov    cx, #6           ! copy 12 bytes  
95    cld                  ! Clear direction flag. Incremente pointer when copying.  
96  
97    rep                  ! Copy the 12-byte table to 0x9000:0fef4.  
98    seg    gs  
99    movw  
100  
101   mov    di, dx           ! Es:di points to new table, then modifies the table.  
102   movb   4(di), *18        ! patch sector count
```

```

103
104     seg fs          ! Let interrupt vector 0x1E point to the new table.
105     mov    (bx), di
106     seg fs
107     mov    2(bx), es
108

```

109 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。

```

110     pop   ax
111     mov   fs, ax
112     mov   gs, 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。

113 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）。

```

114     xor   ah, ah      ! reset FDC
115     xor   dl, dl      ! dl = drive, here set to first disk drive.
116     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=シリンドーnrの下位8ビット、cl=セクターnr(bit0-5)シリンドーの上位2ビット(bit6-7)。

! dh = head number; dl = drive number (bit 7 set for hard disk);

!"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。

21 ! es:bx ->point to data buffer; 120

load\_setup:

```

122     xor   dx, dx      ! drive 0, head 0
123     mov   cx, #0x0002  ! sector 2, track 0
124     mov   bx, #0x0200  ! address = 512, in INITSEG
125     mov   ax, #0x0200+SETUPLEN ! service 2, nr of sectors
126     int   0x13         ! read it
127     jnc   ok_load_setup ! ok - continue
127
128     push  ax           ! dump error code
129     call   print_nl    ! print next line.
130     mov   bp, sp        ! ss:bp point to chars (word)
131     call   print_hex    ! display hex value.
132     pop   ax
133
134     xor   dl, dl        ! reset FDC !
135     xor   ah, ah

```

```
136     int    0x13
137     j      load_setup          ! j = jmp_
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 = 0x08      dl = drive number(bit 7 set for hard disk);
! CF is set if an error occurs, and ah = status code.
! ah = 0,        al = 0,          bl = drive type (AT/PS2);
! ch = low 8 bits of max cylinder nr;
! cl = max sector number (bit 5-0), high 2 bits of max cylinder number (bit 7-6);
! dh = max head number;         dl = number of drives;
! es:di -> drive parameter table (floppy only).

```

142

143 xor dl, dl

144 mov ah, #0x08 ! AH=8 is get drive parameters

145 int 0x13

146 xor ch, ch

! The following instruction indicates that the operand of the next statement is in cs  
! segment. It only affects its next statement. In fact, since the code and data are all  
! set in the same segment, the values of the segment registers cs and ds, es are the same.  
! Therefore, the instruction may not be used here.

147 seg cs

! The next sentence saves the number of sectors per track. For a floppy disk (dl=0), its  
! maximum track number will not exceed 256, and ch is enough to represent it, so bits 6-7  
! of cl must be zero. Also, since 146 lines have been set to ch=0, at this time, cx is the  
! number of sectors per track.

148 mov sectors, cx

149 mov ax, #INITSEG

150 mov es, ax ! Because the interrupt changed es, here restore it.

151

152 ! Print some inane message

! Using BIOS INT 0x10 function 0x03 and 0x13 to display the message: "Loading' + cr + lf",  
! which displays a total of 9 chars, including carriage return and line feed control chars.  
! The BIOS INT 0x10 (read cursor location) parameters are set as follows:  
! ah = 0x03, read cursor position and size;  
! bh = page number;  
! Return:  
! ch = start scan line; cl = end scan line;  
! dh = row (0x00 is top); dl = column (0x00 is left);  
!  
! The BIOS INT 0x10 (write string) parameters are set as follows:  
! ah = 0x13, write string;  
! al = write mode. 0x01-use attributes in bl, cursor stop at end of string.  
! bh = page number; bl = attributes; dh, dl = row, column at which to start writing.  
! cx = number of characters in string.  
! es:bp -> string to write.  
! Return: Nothing.

153

```
154    mov    ah, #0x03      ! read cursor pos
155    xor    bh, bh
156    int    0x10          ! position: dh - row(0--24), dl - column(0--79).
157
158    mov    cx, #9        ! Total 9 charachers.
```

```

159      mov     bx, #0x0007      ! page 0, attribute 7 (normal)
160      mov     bp, #msg1       ! es:bp point to message.
161      mov     ax, #0x1301      ! write string, move cursor
162      int     0x10
163
164 ! ok, we've written the message, now
165 ! we want to load the system (at 0x10000)
166
167      mov     ax, #SYSSEG
168      mov     es, ax          ! segment of 0x010000
169      call    read_it        ! load system module, es is parameter.
170      call    kill_motor      ! stop motor to know the drive status.
171      call    print_nl
172
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マジャーナンバーが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      seg cs
179      mov     ax, root_dev
180      or      ax, ax          ! root_dev is defined (not 0) ?
181      jne    root_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

182

!!!間違いなくAドライブです。

```

183      seg cs
184      mov     bx, sectors
184      mov     ax, #0x0208      ! /dev/ps0 - 1.2Mb
185      cmp     bx, #15         ! sectors = 15 ?
```

```
186      je      root_defined
187      mov     ax,#0x021c          ! /dev/PS0 - 1.44Mb
188      cmp     bx,#18
189      je      root_defined
190 undef_root:                      ! If not, then an infinite loop (dead).
```

```

191      jmp undef_root
192 root_defined:
193      seg cs
194      mov    root_dev, ax      ! Save the checked device number to root_dev.
195
196 ! その後（すべてがロードされた後）、198！ブ
197 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      jmpf    0, SETUPSEG      ! At this point, the bootsect code ends!!!

```

! ここではいくつかのサブルーチンをご紹介します。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
203 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
204 きるだけ速くロードするようにしています。

```

205 !

```

206 ! in:   es - starting address segment (normally 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      ! sectors read of current track
209 head:  .word 0              ! current head
210 track: .word 0             ! current track

```

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 & 0xffff) = 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の条件が満たされていない場合に使用します。

```

213
214      mov ax, es
215      test ax, #0x0fff
216 die:   jne die           ! es must be at 64kB boundary
217      xor bx, bx          ! bx is starting address within segment
218 rp_read:

```

! 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。

```
219      mov ax, es
220      cmp ax, #ENDSEG      ! have we loaded all yet?
221      jb ok1_read
222      ret
223 ok1_read:
```

! 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

224

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。

```

225      seg cs
226      mov ax, sectors          ! get nr of sectors per track.
227      sub ax, sread           ! Subtract the nr of sectors the track has been read.
228      mov cx, ax              ! cx = ax = the nr of unread sectors on the track.
229      shl cx, #9             ! cx = cx * 512 + current offset (bx).
230      add cx, bx             !     = total nr of bytes read after the operation.
231      jnc ok2_read          ! If it does not exceed 64KB, jump to ok2_read.
232      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

この時点での最大のバイト数は、(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

233

! 64KBの補数です。

```

234      xor ax, ax
235      sub ax, bx
236      shr ax, #9
234 ok2_read:

```

! 指定された開始セクタ (cl) 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は、次に  
をカウントし、トラックの最大セクタ数と比較します。

235

! '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。

```

236      call read_track        ! Reads data for given number of sectors on the track.
237      mov cx, ax            ! cx = the nr of sectors read by this time.
238      add ax, sread          ! plus the nr of sectors that have been read.
239      seg cs
240      cmp ax, sectors       ! If there are unread sectors on the track, jump to ok3_read
241      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

242

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。

```

243      mov ax, #1
244      sub ax, head          ! check current head no.
245      jne ok4_read          ! If it is head 0, then go get sectors on head 1.
246      inc track
245 ok4_read:
246      mov head, ax          ! store current head no to head.
247      xor ax, ax            ! Clear the number of sectors read.
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

249

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。

```

250      mov sread, ax          ! save the nr of sectors read on the track.
251      shl cx, #9           ! nr of sectors read * 512 bytes.

```

252        add bx, cx    ! Adjust the starting position of the data.  
253        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  
254        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。  
255        mov ax, es  
256        add ah, #0x10    ! Adjust segment base address to point to next 64KB.

```

257      mov es, ax
258      xor bx, bx          ! clear offse value.
259      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 - sectors to be read; es:bx - data buffer.

259 read_track:
! 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。
260 !"を1つ前の位置に表示します。ドット「...」を表示します。
261     pusha           ! push all registers.
262     pusha
263     mov ax, #0xe2e    ! loading... message 2e = .
264     mov bx, #7        ! character foreground color attribute.
265     int 0x10
266     popa
267

268 ! 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。
269     mov dx, track      ! current track.
270     mov cx, sread       ! sectors already read on the current track.
271     inc cx             ! cl = start reading sector nr.
272     mov ch, dl          ! ch = current head nr.
273     mov dx, head         ! get current head nr.
274     mov dh, dl          ! dh = head nr, dl = drive (0 for A drive)
275     and dx, #0x0100     ! head nr is no more than 1.
276     mov ah, #2           ! ah = 2, read sectors.

275
276     push dx           ! save for error dump
277     push cx
278     push bx
279     push ax
280

281     int 0x13
282     jc bad_rt          ! if error, jump to bad_rt
283     add sp, #8          ! if ok, discard status info.
284     popa
285     ret
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
bad_rt: push ax           ! save error code

```

```
287  
288     call print_all          ! ah = error, al = read  
289  
290  
291     xor ah, ah  
292     xor dl, dl  
293     int 0x13  
294  
295  
296     add sp, #10           ! Discard the info saved for the error condition.  
297     popa  
298     jmp read_track
```

```

299
300 /*
301 *      print_all is for debugging purposes.
302 *      It will print out all of the registers. The assumption is that this is
303 *      called from a routine, with a stack frame like
304 *          dx
305 *          cx
306 *          bx
307 *          ax
308 *          error
309 *          ret <- sp
310 *
311 */
312
313 print_all:
314     mov cx, #5           ! error code + 4 registers
315     mov bp, sp           ! Save current stack pointer sp.
316
317 print_loop:
318     push cx             ! save count left
319     call print_nl        ! nl for readability
320     jae no_reg          ! see if register name is needed
321                     ! if CF=0, registers are not displayed and jump.
322     Corresponding to the stack register order, display their names: "AX : " etc.
323     mov ax, #0xe05 + 0x41 - 1 ! ah = function 0x0e; al = char (0x05 + 0x41 -1)
324     sub al, cl
325     int 0x10
326     mov al, #0x58         ! X
327     int 0x10
328
329     mov al, #0x3a         ! :
330     int 0x10
331
332 ! bpが指すstack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。
333 no_reg:
334     add bp, #2            ! next register
335     call print_hex         ! print it
336     pop cx
337     loop print_loop
338     ret
339
340 ! 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。
341 print_nl:
342     mov ax, #0xe0d          ! CR
343     int 0x10
344     mov al, #0xa            ! LF
345     int 0x10
346     ret
347
346 /*
348 *      print_hex is for debugging purposes, and prints the word

```

349 \* pointed to by ss:bp in hexadecmial.

```

349 */
350
351     ! Call BIOS INT 0x10 to display the word pointed to by ss:bp in 4 hexadecimals.
352     print_hex:
353         mov    cx, #4          ! 4 hex digits
354         mov    dx, (bp)        ! load word into dx
355         print_digit:
356             ! The high byte is displayed first, so rotate dx by 4 to move high 4 bits to dx lower 4 bits.
357             rol    dx, #4          ! rotate so that lowest 4 bits are used
358             mov    ah, #0xe
359             mov    al, dl          ! mask off so we have only next nibble
360             and    al, #0xf        ! put in al, and get lower 4 bits only.
361             add    al, #0x30        ! Add '0' ASCII code 0x30 to convert the value to a char. If value in al exceeds 0x39, it
362             ! means that the value displayed exceeds number 9, so it needs to be represented by 'A'--'F'.
363             cmp    al, #0x39        ! check for overflow
364             jbe    good_digit
365             add    al, #0x41 - 0x30 - 0xa   ! 'A' - '0' - 0xa
366             good_digit:
367             int    0x10
368             loop   print_digit      ! cx--. If cx>0, the next value is displayed.
369             ret
370
371     /*
372     * This procedure turns off the floppy drive motor, so
373     * that we enter the kernel in a known state, and
374     * don't have to worry about it later.
375     */
376     kill_motor:
377         push   dx
378         mov    dx, #0x3f2        ! floppy controller port DOR.
379         xor    al, al          ! A drive, close FDC, disable DMA & int, close motor
380         outb   al              ! output al to port dx.
381         pop    dx
382         ret
383
384     sectors:
385
386     msg1:                   ! message to display, total 9 chars.
387         .byte 13,10          ! cr, lf.
388         .ascii "Loading"

```

---

```

389 ! Start at address 506 (0x1FA), so root_dev is in 2 bytes starting at 508 of boot sector.
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     .word 0xAA55
396
397
398 .text
399 endtext:
400 .data
401 enddata:
402 .bss
403 endbss:
404

```

---

## 6.2.3 Reference information

For the description of the bootsect.S program, a large amount of documents can be found online. Among them, Alessandro Rubini's article "Tour of the Linux kernel source" describes the kernel boot process more comprehensively. Since this program runs in CPU real mode, it will be easier to understand. If you still have difficulty reading at this time, then I suggest you review the 80x86 assembly and its hardware first, and then continue reading this book. For the newly developed Linux kernel, this program has not changed much, and basically maintains the appearance of the 0.12 version of the bootsect file.

### 6.2.3.1 Linux 0.12 Hard Disk Device Number

- 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。ブートセクト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 - memory;
- 2 - floppy disk;
- 3 - hard disk;
- 4 - ttyx;
- 5 - tty;
- 6 - parallel port;
- 7 - Unnamed pipe.

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。

Device nr	Device file	Description
0x0300	/dev/hd0	Represents the entire first hard driv
0x0301	/dev/hd1	The first partition of the first disk
0x0302	/dev/hd2	The second partition of the first disk
0x0303	/dev/hd3	The third partition of the first disk
0x0304	/dev/hd4	The fourth partition of the first disk
0x0305	/dev/hd5	Represents the entire second hard driv
0x0306	/dev/hd6	The first partition of the second disk
0x0307	/dev/hd7	The second partition of the second disk
0x0308	/dev/hd8	The third partition of the second disk
0x0309	/dev/hd9	The fourth partition of the second disk

### 6.2.3.2 Booting from hard disk

The bootsect program gives the default method and process for booting a Linux system from a floppy disk. If you want to boot your system from a hard drive, you usually need to use a different multi-OS bootloader to boot the system, such as multiple operating system bootloaders such as Shoelace, LILO, or Grub. At this point, the operations that bootsect.S needs to perform will be completed by these programs, and the bootsect program will not be executed. Because if you boot from the hard disk, usually the kernel image file will be stored in the root file system of an active partition of the hard disk. So you need to know where the kernel image file is in the file system and what file system it is, that is, your boot sector program needs to be able to recognize and access the file system and read the kernel image file from it.

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 Function Descriptions

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。

---

Size (bytes)	Name	Description
--------------	------	-------------

---

0x90000	2	Cursor Location	Colum (0x00-left), Row (0x00-top most)
0x90002	2	Extended Memory	Size of extended memory begin from address 1MB (in KB)
0x90004	2	Display page	Current display page
0x90006	1	Display mode	
0x90007	1	Char Columns	
0x90008	2	Char Rows ??	
0x9000A	1	Display memory	0x00-64k, 0x01-128k, 0x02-192k, 0x03=256k
0x9000B	1	Display status	0x00-Color, I/O=0x3dX; 0x01-Mono, I/O=0x3bX
0x9000C	2	Property Paras	Property parameters of display adapter.
0x9000E	1	Screen rows	Screen current display rows.
0x9000F	1	Screen columns	Screen current display columns.
...			
0x90080	16	Hd Paras Table	Hard disk parameter table for the first one.
0x90090	16	Hd Paras Table	Hd parameter table for the second one (zero if none).
0x901FC	2	Root devie no	Root file system device number (set in bootsec.s)

---

Then the setup program moves the system module from 0x10000-0x8ffff to the absolute address 0x00000 (At the time, it was considered that the length of the kernel system module system would not exceed this value: 512 KB). Next, load the interrupt descriptor table register (IDTR) and the global descriptor table register (GDTR), turn on the A20 address line, reconfigure the two interrupt control chips 8259A, and reconfigure the hardware interrupt number to 0x20 - 0x2f. Finally, the CPU's control register CR0 (also called the machine status word) is set, enters the 32-bit protected mode, and jumps to the head.s program at the forefront of the system module to continue running.

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。

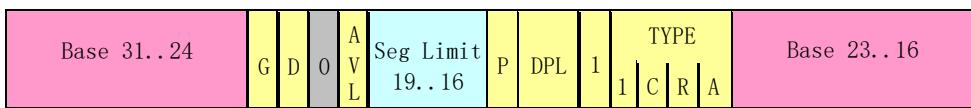
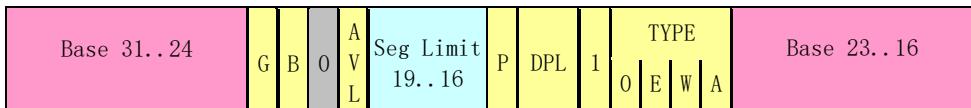


図6-4 コードおよびデータセグメントの記述子フォーマット

The segment descriptor is stored in the descriptor table. The descriptor table is actually an array of descriptor items in memory. There are two types of descriptor tables: Global descriptor table (GDT) and Local descriptor table (LDT). The processor locates the GDT table and the current LDT table by using the GDTR and LDTR registers. These two registers hold the base address of the descriptor table and the length of the table in a linear address. The instructions LGDT and SGDT are used to access the GDTR register; the instructions LLDT and SLDT are used to access the LDTR register. The LGDT uses a 6-byte operand in memory to load the GDTR register. The first two bytes represent the length of the descriptor table, and the last four bytes are the base address of the descriptor table. Note, however, that the operand used by the LLDT instruction to access the LDTR register is a 2-byte operand representing the selector of a descriptor entry in the global descriptor table GDT. The descriptor item in the GDT table corresponding to the selector should correspond to a local descriptor table.

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ジスが8MB (=0x7FF + 1) \* 4KB、リミットレンジスの値は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0xA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、セレクタについての説明をします。セレクタ部は、セグメントディスクリプタを指定するためのもので、ディスクリプターテーブルを指定し、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にセレクタのフォーマットを示す。

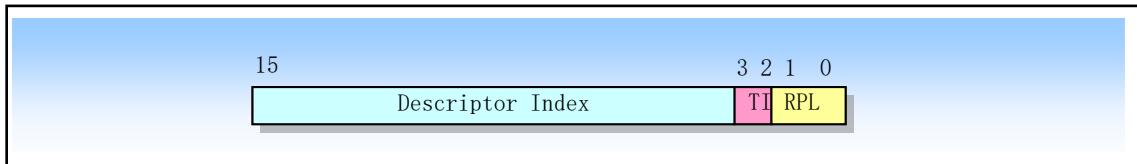


図6-5 セグメントセレクターのフォーマット

The index is used to select one of the 8192 ( $2^{13}$ ) descriptors in the specified descriptor table. The processor multiplies the index by 8, and adds the base address of the descriptor table to access the segment descriptor specified in the table. The Table Indicator (TI) is used to specify the descriptor table referenced by the selector. A value of 0 indicates that the GDT table is specified, and a value of 1 indicates that the current LDT table is specified. The Requestor's Privilege Level (RPL) is used to protect the mechanism.

GDTテーブルの最初の項目（インデックス値0）は使用されないので、インデックス値が0でテーブル・インジケータ値が0のセレクタ（つまりGDTの最初の項目を指すセレクタ）は、ヌル・セレクタ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・セレクタをロードしても、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る。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)がプログラムしたものですが、Linusは誰がd88-manなのか忘れてしまいました:-)。

### 6.3.2 Code Comments

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に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！

10 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

11 ! boot-block used to be. It is then up to the protected mode
12 ! system to read them from there before the area is overwritten
12 ! for buffer-blocks.

13 !
14
15 ! NOTE! These had better be the same as in bootsect.s!
16 ! config.hの定義です。DEF_INITSEG = 0x9000; DEF_SYSSEG = 0x1000; DEF_SETUPSEG = 0x9020
17 #include <linux/config.h>
17
18 INITSEG  = DEF_INITSEG           ! we move boot here - out of the way
19 SYSSEG   = DEF_SYSSEG           ! system loaded at 0x10000 (65536)._
20 SETUPSEG = DEF_SETUPSEG         ! this is the current segment
21
22 .globl begtext, begdata, begbss, endtext, enddata, endbss
23 .text
25 begtext:
26 .data
27 begdata:
28 .bss
29 begbss:
30 .text
30
31 entry start
32 start:
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     mov    ax, #INITSEG        ! Set ds to INITSEG (0x9000)
38     mov    ds, 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    mov     ah, #0x88
43    int     0x15
44    mov     [2], ax           ! store at 0x90002
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; b1 = 0x10 - return video configuration information.

! 戻る。

! bh = video state (0x00 - color, I/O port =0x3dX; 0x01 - mono, I/O port =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    mov     ah, #0x12
49    mov     bl, #0x10
50    int     0x10
51    mov     [8], ax           ! 0x90008 = ???
52    mov     [10], bx          ! 0x9000A = installed mem, 0x9000B = display state.
53    mov     [12], cx          ! 0x9000C = adapter features and settings.
```

! 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

mov ax, #0x5019 ! set default row and columns in ax (ah = 80, al = 25).

```
54
55    cmp     bl, #0x10        ! If bl is 0x10, it means not a VGA card, jump.
56    je      novga
57    call    chsvga         ! get card manufacturer & type, modify row col (line 215)
58 novga:  mov     [14], ax       ! Save screen row & column values (0x9000E, 0x9000F).
```

! 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);

---

```

60      mov     ah, #0x03      ! read cursor pos
61      xor     bh, bh
62      int     0x10          ! save it in known place, con_init fetches
63      mov     [0], dx      ! it from 0x90000.
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      mov     ah, #0x0f
67      int     0x10
68      mov     [4], bx      ! bh = display page
69      mov     [6], ax      ! al = video mode, ah = window width
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ー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0x900080に、2番目のテーブルは0x900090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

```
    mov    ax, #0x0000
74    mov    ds, ax
75    lds    si, [4*0x41]      ! Get INT 0x41 vector, addr of hd0 para table -> ds:si
76    mov    ax, #INITSEG
77    mov    es, ax
78    mov    di, #0x0080      ! Destination of replication: 0x9000:0x0080 -> es:di
79    mov    cx, #0x10          ! move total 16 bytes.
80    rep
81    movsb
82
```

83 ! hd1データの

取得 84

85

```
    mov    ax, #0x0000
86    mov    ds, ax
87    lds    si, [4*0x46]      ! INT 0x46 vector value -> ds:si
88    mov    ax, #INITSEG
89    mov    es, ax
90    mov    di, #0x0090      ! 0x9000:0x0090 -> es:di
91    mov    cx, #0x10
92    rep
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 - floppy (or other removable), change detection; 03 - hard disk).

! 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

```
    mov    ax, #0x01500
97    mov    dl, #0x81
98    int    0x13
99    jc     no_disk1
100
```

```
101      cmp     ah, #3          ! hard drive? (type == 3)?
102      je      is_disk1_
103 no_disk1:
104      mov     ax, #INITSEG    ! no 2nd hard drive, clean 2nd parameter table.
105      mov     es, ax
106      mov     di, #0x0090
107      mov     cx, #0x10
```

```

108      mov     ax, #0x00
109      rep
110      stosb
111 is_disk1:
112
113 ! now we want to move to protected mode ...
114
115      cli          ! no interrupts allowed !
116
117 ! first we move the system to it's rightful place
! The purpose of the following program is to move the entire system module to the 0x00000
! position, that is, move memory block (512KB) (0x10000 - 0x8ffff) to low end of memory.
118
119      mov     ax, #0x0000
120      cld          ! 'direction'=0, movs moves forward
121 do_move:
122      mov     es, ax      ! destination segment ! es:di (0x0:0x0 initially)
123      add     ax, #0x1000
124      cmp     ax, #0x9000    ! Has the last seg (64KB from 0x8000 seg) code moved?
125      jz      end_move    ! yes, jump.
126      mov     ds, ax      ! source segment ! ds:si (0x1000:0x0 initially)
127      sub     di, di
128      sub     si, si
129      mov     cx, #0x8000    ! move 0x8000 words (64K bytes)
130      rep
131      movsw
132      jmp     do_move
133
134 ! then we load the segment descriptors
! From here on, you will encounter 32-bit protected mode operation. See Chapter 4 for
! information on this. Before running into protected mode, we need to first set up the
! segment descriptor table to be used. Here you need to set the global descriptor table
! GDT and the interrupt descriptor table IDT.
!
! The instruction, LIDT, is used to load the interrupt descriptor table register. Its
! operand (idt_48) has 6 bytes. The first 2 bytes (bytes 0-1) are the size of descriptor
! table; the last 4 bytes (bytes 2-5) are the 32-bit linear base of the descriptor table.
! See the following 580--486 lines. Each 8-byte entry in the IDT table indicates the code
! information that needs to be called when an interrupt occurs. It is somewhat similar to
! the interrupt vector, but contains more information.
!
! The LGDT instruction is used to load the global descriptor table register with the same
! operand format as LIDT instruction. Each descriptor item (8 bytes) in the GDT describes
! the information of the data segment and code segment (block) in the protected mode. This
! includes the segment's maximum limit (16 bits), linear base address (32 bits), privilege
! level, in memory flag, read and write permissions, and other flags. See line 567--578.
135
136 end_move:
137      mov     ax, #SETUPSEG    ! right, forgot this at first. didn't work :-
138      mov     ds, ax          ! ds point to this code segment (setup)
139      lidt    idt_48         ! load idt with 0,0
140      lgdt    gdt_48         ! load gdt with whatever appropriate
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、A20に入ってからのテストも必要です。  
 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      call    empty_8042      ! Test 8042 status reg, wait for input buffer be empty.  

145          !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。  

146      mov     al,#0xD1      ! command write ! 0xD1 cmd code, write to P2 of 8042.  

147      out    #0x64,al      ! Bit 1 of P2 is used for strobing of A20 line.  

148      call    empty_8042      ! Waiting buffer to be empty, to see if cmd is accepted.  

149      mov     al,#0xDF      ! A20 on           ! parameters for the A20 line.  

150      out    #0x60,al      ! write to port 0x60.  

151      call    empty_8042      ! if input buffer is empty, then A20 line enabled.
```

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・インターパト・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  
 !(0x0eb)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、Linus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

!

! 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

```
161    mov    al,#0x11          ! initialization sequence
162    out    #0x20,al          ! send it to 8259A-1
163    .word   0x00eb,0x00eb    ! jmp $+2, jmp $+2      ! '$' is current address.
164    out    #0xA0,al          ! and to 8259A-2
165    .word   0x00eb,0x00eb
166    !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。
167    mov    al,#0x20          ! start of hardware int's (0x20)
168    out    #0x21,al          ! sned ICW2 cmd to master chip.
169    .word   0x00eb,0x00eb
170    mov    al,#0x28          ! start of hardware int's 2 (0x28)
```

```

169      out    #0xA1, al          ! sned ICW2 cmd to slave chip.
170      .word  0x00eb, 0x00eb
171      mov    al, #0x04          ! 8259-1 is master
172      out    #0x21, al          ! ICW3 cmd, chain pin IR2 to pin INT of slave chip.
173      .word  0x00eb, 0x00eb
174      mov    al, #0x02          ! 8259-2 is slave
175      out    #0xA1, al          ! ICW3 cmd, chain pin INT to pin IR2 on master chip.
176      .word  0x00eb, 0x00eb

```

! 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

177

! 初期化が終わり、チップの準備が整いました。

```

178      mov    al, #0x01          ! 8086 mode for both
179      out    #0x21, al          ! ICW4 cmd (8086 mode).
180      .word  0x00eb, 0x00eb
181      out    #0xA1, al          ! send ICW4 to slave chip.
182      .word  0x00eb, 0x00eb
183      mov    al, #0xFF          ! mask off all interrupts for now
184      out    #0x21, al
185      .word  0x00eb, 0x00eb
186      out    #0xA1, al

```

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ч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の「！」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 mov ax, #0x0001 ! protected mode (PE) bit

198        lmsw      ax                  ! This is it!  
199        jmpi      0,8                  ! jmp offset 0 of segment 8 (cs)

! 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レ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。セグメントセレクタ8 (0b0000, 0000, 0000, 1000) は、特権の  
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ので (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 ! This routine checks that the keyboard command queue is empty
202 ! No timeout is used - if this hangs there is something wrong with
203 ! the machine, and we probably couldn't proceed anyway.
```

! 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。

```
206     .word 0x00eb, 0x00eb
207     in    al, #0x64      ! 8042 status port
208     test   al, #2       ! is input buffer full?
209     jnz    empty_8042   ! yes - loop
210     ret
211
```

! 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。

```
212 ! Routine trying to recognize type of SVGA-board present (if any)
212 ! and if it recognize one gives the choices of resolution it offers.
```

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     push   ds          ! Save ds, will be popped out on line 231 (or 490 or 492).
217     push   cs          ! ds = cs
218     pop    ds
219     mov    ax, #0xc000
220     mov    es, ax        ! es points to 0xc000 seg. it's BIOS area on VGA card.
221     lea    si, msg1      ! ds:si points to null terminated message msg1.
222     call   prtstr       ! displays msg1.
```

! 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

! メークコード。押したボタンを離したときに発生するスキャンコードをブレークコードと呼ぶ。

! 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よりも大きい場合は、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。ブレークコードが0x9cの場合、ユーザーがボタンを押した/離したことを示します。

! 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

223 ! 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。

224 nokey: in al,#0x60 ! read in scan code from the controller buffer.  
225 cmp al,#0x82 ! compare with minimum break code 0x82  
226 jb nokey ! if less than it, no key is released yet.  
227 cmp al,#0xe0 ! if great than 0xe0, it's a prefix of code.  
228 ja nokey  
229 cmp al,#0x9c

```

230      je      svga          ! if break code is 0x9c, enter key is pressed/released.
231      mov     ax,#0x5019    ! otherwise set al = 25, ah = 80
232      pop     ds
233      ret

```

! 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。

234 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。

```

235 svga:   lea     si, idati      ! Check ATI 'clues'
236         mov     di,#0x31       ! the feature data is at 0xc000:0x0031
237         mov     cx,#0x09       ! 9 bytes
238         repe
239         cmpsb            ! If 9 bytes are the same, means we have an ATI card.
240         jne     noati

```

! OK、アダプターのバーノードがわかりました。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

241

! リスト（moati）からselmod (438行) にジャンプして処理を続けます。

```

242         lea     si, dscati
243         lea     di, moati
244         lea     cx, selmod
245         jmp     cx

```

! 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

246 ! al→ポートn、ah→ポートn+1。

```

247 noati:  mov     ax,#0x200f      ! Check Ahead 'clues'
248         mov     dx,#0x3ce       ! data port 0x0f -> 0x3ce port
249         out    dx,ax        ! set extend reg flag: 0x20 -> 0x3cf port
250         inc    dx          ! then check the flag to see if it has been set.
251         in     al,dx
252         cmp    al,#0x20      ! if it's 0x20, an Ahead A adapter found.
253         je     isahed       ! if it's 0x20, its an Ahead B adapter.
254         cmp    al,#0x21      ! if not a Ahead adapter, jump.

```

255 jne noahed

!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。

256 !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。

257 isahed: lea si, dscahead

258 lea di, moahead

259 lea cx, selmod  
260 jmp cx

! 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

261 IDが0xA5であることから、Chips & Tech社製のディスプレイカードであることがわかります。

262 noahed: mov dx, #0x3c3 ! Check Chips & Tech. 'clues'  
263 in al, dx ! read enable reg from port 0x3c3, add setup flag(bit 4).  
264 or al, #0x10  
265 out dx, al  
266 mov dx, #0x104 ! read chip id from global id port 0x104, stored in bl.  
267 in al, dx  
268 mov bl, al  
269 mov dx, #0x3c3 ! reset setup flag to port 0x3c3.  
270 in al, dx  
271 and al, #0xef  
272 out dx, al  
273 cmp bl, [idcandt] ! compare bl and id(0xA5) in idcandt( line 596).  
274 jne nocant

! 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。

275 ! 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。

276 lea si, dsccandt  
277 lea di, mocandt  
278 lea cx, selmod  
279 jmp cx

! 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。禁止されていないということは、Cirrusのディスプレイではないということです。

カードになります。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タート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、メモリスタート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。

280 !"の拡張子です。

281 nocant: mov dx, #0x3d4 ! Check Cirrus 'clues'  
282 mov al, #0x0c ! write reg index 0x0c to port 0x3d4 to get mem addr.  
283 out dx, al !  
284 inc dx ! read high byte of mem addr from port 0x3d5 to bl.

```
285    in     al, dx
286    mov    bl, al
287    xor    al, al
288    out    dx, al
289    dec    dx          ! write reg index 0x1f to port 0x3d4 to get Eagle ID.
290    mov    al, #0x1f
291    out    dx, al
292    inc    dx
```

```

293    in     al, dx      ! get Eagle ID from port 0x3d5, and store to bh.
294    mov    bh, al      ! swap nibbles and store to cl. left shift to ch.
295    xor    ah, ah      ! then put number 6 to cl.
296    shl    al, #4
297    mov    cx, ax
298    mov    al, bh
299    shr    al, #4
300    add    cx, ax
301    shl    cx, #8
302    add    cx, #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

303 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。

```

304    mov    ax, cx
305    mov    dx, #0x3c4
306    out   dx, ax
307    inc    dx

```

! 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ということです。

308

```

309    in     al, dx
310    and   al, al
311    jnz   nocirr

```

! 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。

312 ! 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。

```

313    mov    al, bh      !
314    out   dx, al      !
315    in     al, dx      !
316    cmp   al, #0x01
317    jne   nocirr

```

! 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

318 ! (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。

```

319    call   rst3d4
320    lea    si, dsccurrus
321    lea    di, mocirrus
322    lea    cx, selmod
323    jmp    cx

```

! 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

324 ! 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)。

325 rst3d4: mov dx, #0x3d4

```
326      mov     al, bl
327      xor     ah, ah
328      shl     ax, #8
329      add     ax, #0x0c
330      out    dx, ax          ! note, word output, al -> 0x3d4, ah -> 0x3d5.
331      ret
```

! 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。

! Everex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-Ultrographics II; 0x6200-Vision VGA; 0x6730-EVGA; 0x6780-Viewpoint。

332 ! di = BCDコードで表現されたビデオBIOSのバージョン番号。

```

333 nocirr: call    rst3d4          ! Check Everex 'clues'
320      mov     ax, #0x7000        ! int 0x10 with ax = 0x7000, bx=0x0000.
321      xor     bx, bx
322      int     0x10
323      cmp     al, #0x70          ! al should contain 0x70 for Everex card.
324      jne     noevrx
325      shr     dx, #4           ! ignore board fix number(bit3-0).
326      cmp     dx, #0x678         ! if board type is 0x678, its a Trident card.
327      je      istrnid
328      cmp     dx, #0x236         ! if board type is 0x236, also a Trident card.
329      je      istrnid

```

!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

lea si, dsceverex

```
330
331    lea    di, moeverex
332    lea    cx, selmod
333    jmp    cx
334 istrid: lea    cx, ev2tri      ! Everex card with a Trident type, jump to ev2tri
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  
noevrx: lea si, idgenoa ! Check Genoa 'clues'

```
336
337 xor ax, ax           ! ds:si points to feature data.
338 seg es
339 mov al, [0x37]        ! get feature data from VGA card at 0x37.
340 mov di, ax            ! es:di point to 0x37.
341 mov cx, #0x04
342 dec si
343 dec di
344 l1: inc si           ! compare the 4 feature bytes.
345 inc di
346 mov al, (si)
347 seg es
348 and al, (di)
349 cmp al, (si)
350 loope l1
351 cmp cx, #0x00
352 jne nogen
```

!さて、このカードがジエ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     lea    si, dscgenoa
354     lea    di, mogenoa
355     lea    cx, selmod
356     jmp    cx

```

! 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  
 357 ディスプレイカードのBIOSにある「！」の文字列（「VGA=」）。

```

358 nogen:   lea    si, idparadise    ! Check Paradise 'clues'
359         mov    di, #0x7d        ! es:di point to 0xc000:0x007d.
360         mov    cx, #0x04        ! there should be 4 bytes: "VGA="
361         repe
362         cmpsb
363         jne    nopara

```

!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

364 ! 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。

```

365     lea    si, dscparadise
366     lea    di, moparadise
367     lea    cx, selmod
368     jmp    cx

```

! 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

! this feature to check if it is a Trident display card.

```

367 nopara:  mov    dx, #0x3c4      ! Check Trident 'clues'
368         mov    al, #0x0e       ! output index 0x0e (mode ctrl reg 1) to port 0x3c4
369         out    dx, al       ! read original value from port 0x3c5 and store to al
370         inc    dx
371         in     al, dx
372         xchg   ah, 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、"0x02"となります。  
 ! 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

373

! モードコントロールレジスタ1、ah = 最後に読んだ値。

```

374     mov    al, #0x00
375     out    dx, al
376     in     al, dx
377     xchg   al, ah
378     mov    bl, al      ! Strange thing ... in the book this wasn't
379     and   bl, #0x02      ! necessary but it worked on my card which
380     jz    setb2      ! is a trident. Without it the screen goes
381     and   al, #0xfd      ! blurred ...
382     jmp    clrb2      !
383 setb2:  or    al, #0x02      !
383 clrb2:  out   dx, al
384     and   ah, #0x0f      ! get page number field (bit 3-0) (line 375)

```

385        cmp        ah, #0x02        ! if equal 0x02, it's a Trident card.  
386        jne        notrid

!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
! modes list (motrident), then jump to selmod (line 438) to continue setting.
387 ev2tri: lea    si,dsctrident
388      lea    di,motrident
389      lea    cx,selmod
390      jmp    cx

```

! ここで、Tsengカード (ET4000AXまたはET4000/W32) であるかどうかを確認します。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。  
レジスターの上位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（読み込みと

! 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

391 !!! レジスタはまだ0x55のはずです。

```

392 notrid: mov    dx,#0x3cd      ! Check Tseng 'clues'
393      in     al,dx       ! Could things be this simple ! :-
394      mov    bl,al       ! read original seg selector data from 0x3cd to bl.
395      mov    al,#0x55     ! write to it with value 0x55, and read again to ah.
396      out   dx,al
397      in     al,dx
398      mov    ah,al
399      mov    al,bl       ! restore original data.
400      out   dx,al
401      cmp    ah,#0x55     ! if read value equal to write, it's a Tseng card.
402      jne    notsen

```

! 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

403 ! 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。

```

404      lea    si,dsctseng
405      lea    di,motseng
406      lea    cx,selmod
407      jmp    cx

```

! 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。このレジスターのビット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。したがつ  
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。

408 !"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。

```

409 notsen: mov    dx,#0x3cc      ! Check Video7 'clues'
410      in     al,dx
411      mov    dx,#0x3b4      ! set dx to mono control index register port 0x3b4.
412      and    al,#0x01      ! If bit0 of mixed output reg is 0(mono), jump directly.
413      jz     even7        ! Otherwise set dx to color control index reg port 0x3d4.
414      mov    dx,#0x3d4
415 even7:  mov    al,#0x0c      ! Set index to 0x0c for the mem address high byte reg.

```

```
416    out    dx, al
417    inc    dx
418    in     al, dx          ! read high byte reg of vmem address , save to bl.
419    mov    bl, al
420    mov    al, #0x55        ! then write 0x55 to high byte reg and read it out.
```

```

418      out    dx, al
419      in     al, dx
! Then select the Video7 display card identification register whose index number is 0x1f
! through the CRT index register port 0x3b4 or 0x3d4. The contents of this register are
! actually the result of the XOR of the memory start address and the value 0xea.
420      dec    dx
421      mov    al, #0x1f
422      out    dx, al
423      inc    dx
424      in     al, dx          ! read Video7 card id register value and save it in bh.
425      mov    bh, al
426      dec    dx          ! select addr high byte reg to restore its original value.
427      mov    al, #0x0c
428      out    dx, al
429      inc    dx
430      mov    al, bl
431      out    dx, al
! Then we will verify that the "Video7 display card identification register value is the
! result value of the memory memory start address high byte and 0xea after XOR operation".
! Therefore, the result of the XOR operation of 0x55 and 0xea should be equal to the test
! value of the identification register. If it is not a Video7 card, set the default display
! row and column value (492 lines). Otherwise it is the Video7 card. So let si point to
! the display card row value table (dscvideo7), let di point to the number of extended
! mode and mode number list (movideo7).
432      mov    al, #0x55
433      xor    al, #0xea
434      cmp    al, bh
435      jne    novid7
436      lea    si, dscvideo7
437      lea    di, movideo7

```

! 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。  
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、それに合わせ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

! 対応する番号を押してモードを選択します。

!

[438](#) !以下のコードは、ヌル文字で終端された文字列 "Mode.COL\$ROWS: "を画面に表示します。COL\$ROWS:" を画面に表示します。

```
439 selmod: push    si
440           lea     si, msg2
441           call    prtstr
```

```

442      xor    cx, cx
443      mov    cl, (di)          ! cl is the extended modes of the checked card.
444      pop    si
445      push   si
446      push   cx

! 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

447 !"と表示され、ユーザーが選択できるようになっています。

448 tbl:   pop    bx           ! bx = total extend modes number.
449      push   bx
450      mov    al, bl
451      sub    al, cl
452      call   dprnt          ! display the value in decimal format.
453      call   spcinc          ! a dot, and then 4 spaces.
454      lodsw
455      xchg   al, ah          ! swap, al = columns.
456      call   dprnt          ! display column number.
457      xchg   ah, al          ! al = rows.
458      push   ax
459      mov    al, #0x78         ! show "x"
460      call   prnt1
461      pop    ax             ! al= row number
462      call   dprnt          ! display row number.
463      call   docr            ! cr, lf
464      loop   tbl             ! display next row columns, mode number decreased by 1.

465      ! 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。

466      pop    cx             ! cl = total extend modes number.
467      call   docr
468      lea    si, msg3        ! "Choose mode by pressing the corresponding number."
469      call   prtstr

! 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のブレー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、そのブレークコードは
!     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、リードブレークコードが0x82より小さい場合は、数字の
470      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意味します。

471      pop    si             ! pop up original row & column table pointer.
472      add    cl, #0x80        ! cl + 0x80 = the break code for the "number key -1".
469 nonum:  in    al, #0x60        ! Quick and dirty...
470      cmp    al, #0x82        ! less than 0x82 ? ignore it.
471      jb    nonum
472      cmp    al, #0x8b        ! scan code = 0x8b? it's number key 0.
473      je    zero
474      cmp    al, cl          ! great than the number of modes?
475      ja    nonum          ! non number key pressed.
476      jmp   nonzero

```

! 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。

477 !"の値を軸にしています。

```

478 zero:    sub     al, #0x0a      ! al = 0x8b - 0x0a = 0x81
479 nozero:   sub     al, #0x80      ! subtract 0x80 to obtain the mode selected by user.
480          dec     al           ! count from 0
481          xor     ah, ah       ! set display mode
482          add     di, ax
483          inc     di           ! di points to the mode number (skip the first).
484          push    ax
485          mov     al, (di)     ! mode number -> al, call int to set mode.
486          int     0x10
487          pop     ax
488          shl     ax, #1       ! mode nr x 2: pointer into the row & column table.
489          add     si, ax
490          lodsw
491          pop     ds           ! get row and colum to ax (ah = colum, al = row).
492          ret

```

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。

493 novid7: pop ds ! Here could be code to support standard 80x50, 80x30

```

492
493         mov     ax, #0x5019
494         ret
495

```

496 !次の列に「タブ」するルーチン 497

498 ドット(.)と4つのスペースを表示します。

```

499 spcing: mov     al, #0x2e      ! a dot '.'
500          call    prnt1
501          mov     al, #0x20
502          call    prnt1
503          mov     al, #0x20
504          call    prnt1
505          mov     al, #0x20
506          call    prnt1
507          mov     al, #0x20
508          call    prnt1
509          ret
510
511 ! Routine to print asciiz-string at DS:SI
512
513 prtstr: lods

```

```
514      jz      fin
515      call    prnt1          ! print a char in al
516      jmp     prtstr
517 fin:   ret
518
519 ! Routine to print a decimal value on screen, the value to be
520 ! printed is put in al (i.e 0-255).
521
522 dprnt: push    ax
```

```

523      push    cx
524      mov     ah, #0x00
525      mov     cl, #0x0a
526      idiv   cl
527      cmp     al, #0x09
528      jbe     lt100
529      call    dprnt
530      jmp     skip10
531 lt100: add    al, #0x30
532      call    prnt1
533 skip10: mov    al, ah
534      add    al, #0x30
535      call    prnt1
536      pop    cx
537      pop    ax
538      ret
539
540 ! Part of above routine, this one just prints ascii al
! This subroutine uses the interrupt 0x10 function 0x0E to write a character on the screen
! by telex. The cursor will automatically move to the next position. If a line is written,
! the cursor will move to the beginning of the next line. If the last line of a screen has
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: push   ax
543      push   cx
544      mov    bh, #0x00      ! page number.
545      mov    cx, #0x01
546      mov    ah, #0x0e
547      int    0x10
548      pop    cx
549      pop    ax
550      ret
551
552 ! Prints <CR> + <LF>
553
554 docr:  push   ax
555      push   cx
556      mov    bh, #0x00
557      mov    ah, #0x0e
558      mov    al, #0x0a
559      mov    cx, #0x01
560      int    0x10
561      mov    al, #0x0d
562      int    0x10
563      pop    cx
564      pop    ax
565      ret
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  
567 !(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。

568 gdt:  
569     .word 0,0,0,0                 ! dummy  
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レクタの値です。  
570     .word 0x07FF                 ! 8Mb - limit=2047 (0--2047, so 2048\*4096=8Mb)  
571     .word 0x0000                 ! base address=0  
572     .word 0x9A00                 ! code read/exec  
573     .word 0x00C0                 ! 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、カーネルデータセレクタの値です。  
575     .word 0x07FF                 ! 8Mb - limit=2047 (2048\*4096=8Mb)  
576     .word 0x0000                 ! base address=0  
577     .word 0x9200                 ! data read/write  
578     .word 0x00C0                 ! 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     .word 0                         ! idt limit=0  
582     .word 0,0                         ! 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     .word 0x800                     ! gdt limit=2048, 256 GDT entries  
586     .word 512+gdt, 0x9             ! gdt base = 0X9xxxx  
587  
588 msg1: .ascii "Press <RETURN> to see SVGA-modes available or any other key to continue."  
589   db 0x0d, 0x0a, 0x0a, 0x00  
590 msg2:                                 .ascii "Mode: COLSxROWS:"  
591   db 0x0d, 0x0a, 0x0a, 0x00  
592 msg3:                                 .ascii "Choose mode by pressing the corresponding number."  
593   db 0x0d, 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:                          .byte 0xa5                             ! idcandt means "ID of Chip AND Tech."

```
597 idgenoa:      .byte 0x77, 0x00, 0x66, 0x99  
598 idparadise:   .ascii "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 ! Manufacturer: Numofmodes: Mode:
601
602 moati:      .byte 0x02, 0x23, 0x33
603 moahead:     .byte 0x05, 0x22, 0x23, 0x24, 0x2f, 0x34
604 mocandt:     .byte 0x02, 0x60, 0x61
605 mocirrus:    .byte 0x04, 0x1f, 0x20, 0x22, 0x31
606 moeverex:   .byte 0x0a, 0x03, 0x04, 0x07, 0x08, 0x0a, 0x0b, 0x16, 0x18, 0x21, 0x40
607 mogenoa:    .byte 0x0a, 0x58, 0x5a, 0x60, 0x61, 0x62, 0x63, 0x64, 0x72, 0x74, 0x78
608 moparadise: .byte 0x02, 0x55, 0x54
609 motrident:  .byte 0x07, 0x50, 0x51, 0x52, 0x57, 0x58, 0x59, 0x5a
610 motseng:    .byte 0x05, 0x26, 0x2a, 0x23, 0x24, 0x22
611 movideo7:   .byte 0x06, 0x40, 0x43, 0x44, 0x41, 0x42, 0x45
612
! Below is a list of columns and rows for the modes that can be used with various brands
! of VGA cards. For example, line 615 indicates that the column and row values of the
! two extension modes for ATI card are 132 x 25 and 132 x 44, respectively.
613 !
       msb = Cols  lsb = Rows:
614
615 dscati:      .word 0x8419, 0x842c
616 dscahead:    .word 0x842c, 0x8419, 0x841c, 0xa032, 0x5042
617 dsccandt:   .word 0x8419, 0x8432
618 dsccirrus:  .word 0x8419, 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: .word 0x8419, 0x842b
622 dsctrident: .word 0x501e, 0x502b, 0x503c, 0x8419, 0x841e, 0x842b, 0x843c
623 dsctseng:   .word 0x503c, 0x6428, 0x8419, 0x841c, 0x842c
624 dscvideo7:  .word 0x502b, 0x503c, 0x643c, 0x8419, 0x842c, 0x841c
625
626 .text
627 endtext:
628 .data
629 enddata:
630 .bss
631 endbss:
```

### 6.3.3 Reference information

In order to get the basic parameters of the machine and display messages of the boot process to user, this program calls interrupt services in the BIOS multiple times and starts to involve some access operations to the hardware ports. The following briefly describes several BIOS interrupt services used and explains the cause of the A20 address line problem. Finally we also mentioned the issues of the 80X86 CPU 32-bit protection mode operation.

#### 6.3.3.1 Current memory image

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。

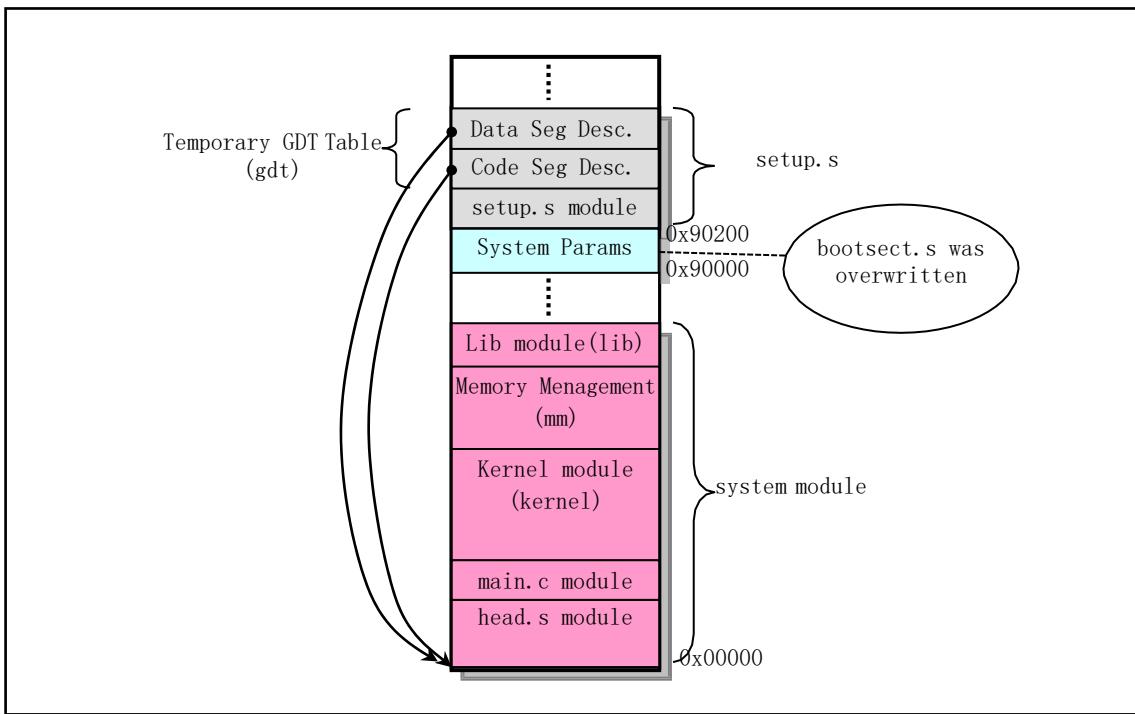


図6-6 setup.s終了後のメモリーマップの図

At this point, there are three descriptors in the temporary global table GDT. The first one is NULL not used, the other two are code and data segment descriptors. They all point to the beginning of the system module, which is the physical memory address 0x00000. Thus, when the last instruction 'jmp 0,8' (line 193) is executed, it will jump to the beginning of the head.s program to continue execution. The '8' in this instruction is the value of the segment selector, which is used to specify the descriptor item to be used. This is the code segment descriptor in the GDT. '0' is the offset in the code segment specified by the descriptor.

### 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 = )	Register	Description

0x10 ) Input/Return		
Input Information	ah	Function No. = 0x12, Obtain display card information.
	bl	Sub-Function No. = 0x10
Return Information	bh	Video Status: 0x00 - Color mode (the video hardware I/O port base address is 0x3DX); 0x01 - Mono mode (the video hardware I/O port base address is 0x3BX); (where the X value in the port address can be 0 -- F)
	bl	Installed video memory size: 00 = 64K, 01 = 128K, 02 = 192K, 03 = 256K
	ch	Feature connector bit information: Bits 0-1      Feature line 1-0, Status 2; Bits 2-3      Feature line 1-0, Status 1;

		Bits 4-7      Not used ( set to 0)																
	cl	<p>Video switch settings: Bits 0-3 correspond to switches 1-4. Bits 4-7 is not used.</p> <p>Original EGA/VGA switch settings:</p> <table> <tbody> <tr><td>0x00</td><td>MDA/HGC;</td><td>0x01-0x03</td><td>MDA/HGC;</td></tr> <tr><td>0x04</td><td>CGA 40x25;</td><td>0x05</td><td>CGA 80x25;</td></tr> <tr><td>0x06</td><td>EGA + 40x25;</td><td>0x07-0x09</td><td>EGA + 80x25;</td></tr> <tr><td>0x0A</td><td>EGA + 80x25Mono;</td><td>0x0B</td><td>EGA + 80x25Mono.</td></tr> </tbody> </table>	0x00	MDA/HGC;	0x01-0x03	MDA/HGC;	0x04	CGA 40x25;	0x05	CGA 80x25;	0x06	EGA + 40x25;	0x07-0x09	EGA + 80x25;	0x0A	EGA + 80x25Mono;	0x0B	EGA + 80x25Mono.
0x00	MDA/HGC;	0x01-0x03	MDA/HGC;															
0x04	CGA 40x25;	0x05	CGA 80x25;															
0x06	EGA + 40x25;	0x07-0x09	EGA + 80x25;															
0x0A	EGA + 80x25Mono;	0x0B	EGA + 80x25Mono.															

### 6.3.3.3 Hard Drive Basic Parameter Table ("INT 0x41")

In the ROM BIOS interrupt vector table, the interrupt vector location of 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。

Offset	Size	Name	Description
0x00	word	cyl	Number of cylinders
0x02	byte	head	Number of heads
0x03	word		Start cylinder to reducing the write current (only for PC/XT, others are 0)
0x05	word	wpcm	Pre-compensation cylinder number before start writing (multiplied by 4)
0x07	byte		Maximum ECC burst size (only for PC/XT, others are 0)

0x08	byte	ctl	Control byte (driver step selection): Bit 0 - Not used (0); Bit 1 - Reserved (0) (Close IRQ) Bit 2 - Allow reset; Bit 3 - Set if number of heads great than 8 Bit 4 - Not used (0); Bit 5 - Set if there is bad map at cylinder number+1 Bit 6 - Disable ECC retry; Bit 7 - Disable Access retry.
0x09	byte		Standard timeout value (only for PC/XT, others are 0)
0x0A	byte		Format timeout value (only for PC/XT, others are 0)
0x0B	byte		Detect drive timeout value (only for PC/XT, others are 0)
0x0C	word	lzone	Head landing (stop) cylinder number
0x0E	byte	sect	Number of sectors per track
0x0F	byte		Reserved.

#### 6.3.3.4 A20 address line problem

In August 1981, IBM's original personal computer IBM PC used a 16-bit Intel 8088 CPU. The CPU has a 16-bit internal (8-bit external) data bus and a 20-bit address bus width. Therefore, there are only 20 address lines (A0 – A19) in this PC, and the CPU can address only up to 1MB of memory range. At the time when the popular machine memory capacity was only a few tens of KB and several hundred KB, 20 address lines were enough to address the memory. The highest address that it can address is 0xffff:0xffff, which is 0x10ffef. For memory addresses that exceed 0x100000 (1MB), the CPU will default wrap around to the 0x0ffef position.

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ました。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。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ー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。しかし、このポートは他の互換性のあるマイクロコンピュータのデバイス（ディスプレイチップなど）にも使用され、システムエラー動作になる可能性があります。別の方針としては、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することができます。

### 6.3.3.5 Programming method of 8259A interrupt controller

- 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。

#### 2. 8059A chip working principle

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。

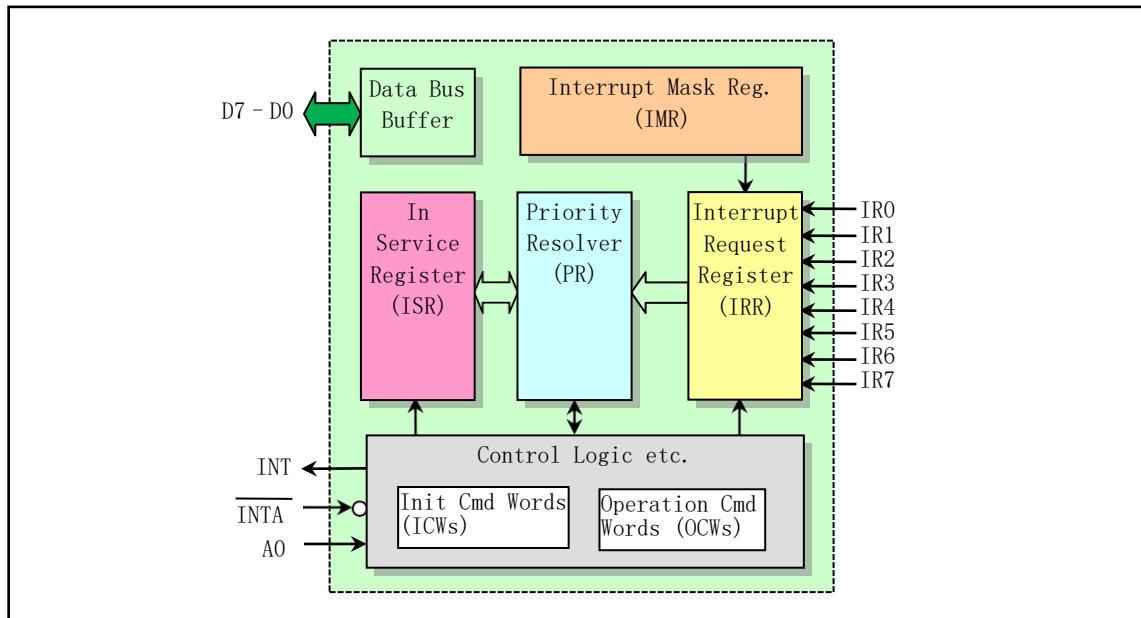


図6-7 プログラマブルインタラプトコントローラ 8259A チップダイアグラム

In the figure, the Interrupt Request Register (IRR) is used to store all the requested service interrupt levels on the interrupt request input pin. The 8 bits (D7-D0) of the register correspond to the pins IR7-IR0. The Interrupt Mask Register (IMR) is used to store the bits corresponding to the masked interrupt request line. The 8 bits of the register also correspond to 8 interrupt levels. Which bit is set to 1 masks which level of interrupt request. That is, the IMR processes the IRR, each bit of which corresponds to each request bit of the IRR. Masking high priority input lines does not affect the input of low priority interrupt request lines. The priority resolver (PR) is used to determine the priority of the bits set in the IRR, and the highest priority interrupt request is strobed into the in-service register (ISR). The ISR holds an interrupt request that is receiving service. The register set in the control logic block is used to accept two types of commands generated by the CPU. Before the 8259A can operate normally, the contents of the Initialization Command Word (ICW) registers must be set first. In the course of its work, you can use the Operation Command Words (OCW) registers to set and manage the 8259A's working mode at any time. The A0 line is used to select the register for the operation. In the PC/AT microcomputer system, when the A0 line is 0, the port address of the chip is 0x20 (master chip) and 0xA0 (slave chip), and when A0=1, the port is 0x21 and 0xA1.

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、最も優先度の高い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に割込み番号を送るよう<sup>う</sup>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。

### 3. Initialization command word programming

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も、ICW1に明記する必要があります。

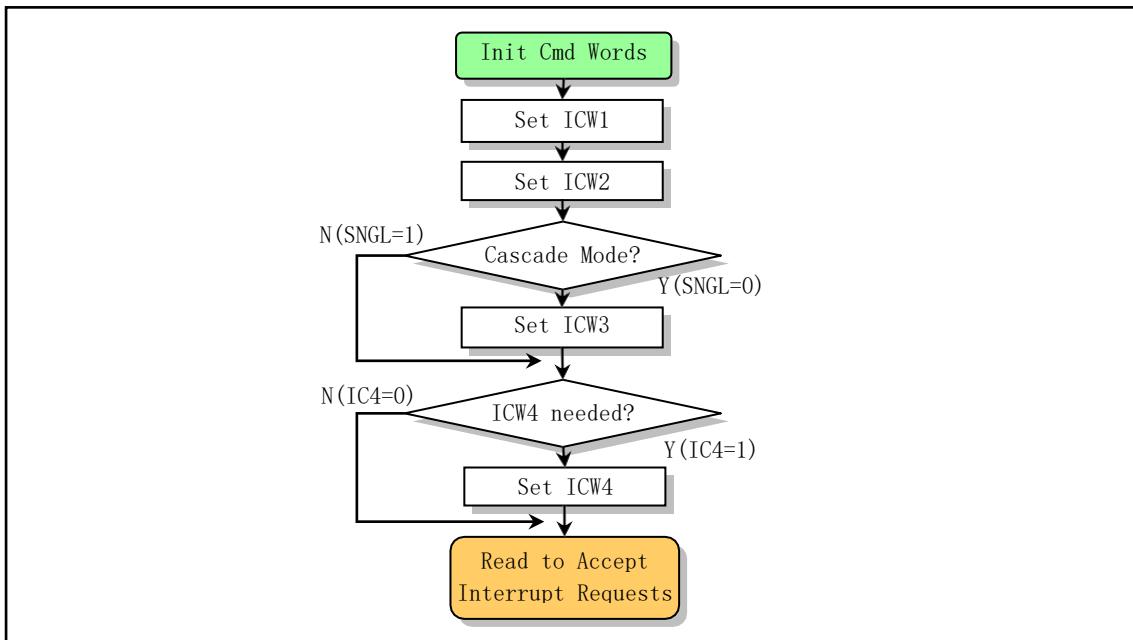


図 6-8 8259A 初期化シーケンス

(1) ICW1 When the transmitted byte 5th bit (D4) = 1 and the address line A0 = 0, it indicates that ICW1 is programmed. At this time, for the multi-chip cascading case of the PC/AT microcomputer system, the port address of the 8259A main chip is 0x20, and the port address of the slave chip is 0xA0. The format of ICW1 is shown in Table 6-5.

Bit	Name	Description
D7	A7	A7-A5 indicates the page start address used in the MCS80/85 for the interrupt service process. They are combined with A15-A8 in ICW2. These are not used in 8086/88.
D6	A6	
D5	A5	
D4	1	Always 1
D3	LTIM	1 - Level triggered interrupt mode; 0 – Edge triggered mode.
D2	ADI	The MCS80/85 used for the CALL instruction address interval. Not used in 8086/88.
D1	SNGL	1 - Single 8259A; 0 - Cascade mode.
D0	IC4	1 – requires ICW4; 0 – not required.

In the Linux 0.12 kernel, ICW1 is set to 0x11. It indicates that the interrupt request is edge triggered, multiple slices of 8259A are cascaded, and finally ICW4 needs to be sent.

(2) ICW2 This initialization command word is used to set the upper 5 bits of the interrupt number sent by the chip. After the ICW1 is set, the interrupt number indicates that ICW2 is set when A0=1. At this time, for the multi-chip cascading of the PC/AT microcomputer system, the port address of the 8259A main chip is 0x21, and the port address of the slave chip is 0xA1. The ICW2 format is shown in Table 6–6.

表6-6 インタラプト初期化コマンドワード ICW2

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	A15/T7	A14/T6	A13/T5	A12/T4	A11/T3	A10	A9	A8

In the MCS80/85 system, the A15-A8 indicated by bits D7-D0 and the A7-A5 set by ICW1 form the interrupt service program page address. In a system or compatible system using the 8086/88 processor, T7-T3 is the upper 5 bits of the interrupt number, and the lower 3 bits automatically set by the 8259A chip form an 8-bit interrupt number. When the 8259A receives the second interrupt response pulse INTA, it will be sent to the data bus for the CPU to read.

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 This command word is used to load an 8-bit slave register when multiple 8259A chips are cascaded. The port address is the same as above. The ICW3 format is shown in Table 6-7.

表6-7 インタラプト初期化コマンドワード ICW3

	A0	D7	D6	D5	D4	D3	D2	D1	D0
Master	1	S7	S6	S5	S4	S3	S2	S1	S0
Slave	1	0	0	0	0	0	ID2	ID1	ID0

The master chip bits S7\_S0 correspond to the cascaded slaves. Which bit is 1 means that the signal on the interrupt request pin IR of the master is from the slave, otherwise the corresponding IR pin is not connected to the slave. The slave chip bits of ID2\_ID0 correspond to the identification numbers of the slave chips, that is, the interrupt level connected to the master chip. When a slave receives a cascading line (CAS2 - CAS0) input value equal to its own ID2 - ID0, it means that the slave is selected. At this point, the slave should send the interrupt number of the interrupt request currently selected from the slave chip to the data bus.

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chipのIR2端子がスレーブchipに接続されていることを表します。スレーブchip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chip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 When IC0 bit 0 (IC4) is set, it indicates that ICW4 is required. Address line A0=1. The port address is the same as above. The ICW4 format is shown in Table 6-8.

表 6 - 8  イ ン タ ラ プ ト 初 期 化 コ マ ン ド ワ ー ド  I C W 4	Name	Description

Bit(s)		
D7-5		Always 0
D4	SFNM	1 - special fully nested mode; 0 - not a special fully nested mode.
D3	BUF	1 - buffer mode; 0 - unbuffered mode.
D2	M/S	1 - Buffered mode /Slave; 0 - Buffered mode /Master.
D1	AEOI	1 - Auto End of Interrupt mode; 0 - Normal End of Interrupt mode.
D0	$\mu$ PM	1 – 8086/88 processor system; 0 – MCS80/85 system.

The value of the ICW4 command word sent to the 8259A master chip and the slave chip by the Linux 0.12 core is 0x01. Indicates that the 8259A chip is set to a normal fully nested, unbuffered, non-automatic end interrupt mode and is used in the 8086 and its compatible systems.

#### 4. Operation command word programming

(1) 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可能です。

(2) OCW1 This operation command word is used to read/write the interrupt mask register IMR. Address line A0 needs to be 1. The port address description is the same as above. The OCW1 format is shown in Table 6-9.

表6-9 インタラプト操作コマンドワード OCW1

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

Bits D7-D0 correspond to 8 interrupt requests, 7 levels - 0 level mask bits M7 - M0. If M=1, the corresponding interrupt request level is masked; if M=0, the corresponding interrupt request level is allowed. In addition, masking high priority does not affect other low priority interrupt requests.

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、フロッピードライブ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。

(3) OCW2 is used to send EOI commands or set the automatic rotate mode for interrupt priority. When the bit D4D3 = 00, the address line A0 = 0 indicates that the OCW2 is programmed. The format of the operation command word OCW2 is shown in Table 6–10.

Bit(s)	Name	Description
D7	R	Priority rotation state.
D6	SL	Priority setting flag.
D5	EOI	Non-Automatic End of Interrupt flag.
D4-3		Always 0

D2	L2	L2 -- L0 - 3 bits form the level number, corresponding to the interrupt request level IRQ0--IRQ7 (or IRQ8-IRQ15).
D1	L1	
D0	L0	

The roles and meanings of the combination of bits D7-D5 are shown in Table 6–11. Those with an \* can specify the priority to reset the ISR by setting L2--L0, or select the special rotate priority to become the current lowest priority.

R(D7)	SL(D6)	EOI(D5)	Description	Type
0	0	1	Non-specific EOI command (fully nested mode).	End of Interrupt
0	1	1	Specific EOI command (not fully nested).	
1	0	1	Rotate on non-specific EOI command.	Automatic Rotation
1	0	0	Rotate in Automatic EOI mode (Set).	
0	0	0	Rotate in Automatic EOI mode (Clean).	
1	1	1	Rotate on Specific EOI command.	

---

1	1	0	Set priority command.	Specific rotation
0	1	0	No operation.	

The Linux 0.12 kernel uses only the operational command word to send an end interrupt EOI command to the 8259A before the end of the interrupt processing. The OCW2 value used is 0x20, indicating a non-special end interrupt EOI command in full nested mode.

(4) OCW3 is used to set the special mask mode and read register status (IRR and ISR). When D4D3=01 and address line A0=0, it means that OCW3 is programmed (read/write). However, this operation command word is not used in the Linux 0.12 kernel. The format of OCW3 is shown in Table 6–12.

	Name	Description
Bit		
D7		Always 0
D6	ESMM	Operate in a special mask mode: D6 -- D5: 11 - Set special mask; 10 - Reset special mask; 00,01 - No action.
D5	SMM	D6 -- D5: 11 - Set special mask; 10 - Reset special mask; 00,01 - No action.
D4		Always 0
D3		Always 1
D2	P	1 - Poll command; 0 - No poll command.
D1	RR	Read register status command on the next RD pulse:
D0	RIS	D1 -- D0: 11 - Read In Service Reg. ISR; 10 - Read Interrupt Reg. IRR

## 5. 8259A operation mode description

- (1) 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。以下は、8259Aチップがどのように動作するかをよりよく理解するために、いくつかの一般的な方法を詳しく説明しています。
- (2) Full nested mode
- (3) 初期化後、動作コマンド・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なります。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が割込みに応答すると、最も優先度の高い割込み要求が決定され、その割込み要求の割込み番号がデータバスに置かれます。また、割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込みサービス手順から戻る前に割込み終了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つの割込み要求をマスクしても、他の割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。

#### (4) End of Interrupt (EOI) method

上述したように、サービスレジスタISRで処理中の割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AE0I (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。

(5) 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。

#### (6) Special full nested mode

A. 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。

B. When an interrupt request from a slave chip is being serviced, the slave chip is not excluded by the priority of the master chip. Therefore, other higher priority interrupt requests issued from the chip will be recognized by the master chip, and the master chip will immediately issue an interrupt to the CPU. In the above conventional full nesting mode, when a slave interrupt request is being serviced, the slave chip is masked by the master chip. Therefore, a higher priority interrupt request issued from the slave chip cannot be processed.

C. When exiting the interrupt service routine, the program must check if the current interrupt service is the only interrupt request issued from the slave chip. The method of checking is to first issue a non-special interrupt EOI command to the slave chip and then read the value of its service register ISR. Check if the value is 0 at this time. If it is 0, it means that a non-special EOI command can be sent to the main chip. If it is not 0, there is no need to send an EOI command to the main chip.

#### (7) Cascade mode method

8259Aはマスターchip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chip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chip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chip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。

(8) 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、異なる方法で動作することができます。また、マスターchip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chip用とスレーブchip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。

#### (9) Automatic rotation priority mode

(10) 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。

#### (11) Interrupt mask mode

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、すべての低優先度の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。

## (12) Read register status

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の読み出しこmandoを出力する必要があります。

## 6.4 head.s

### 6.4.1 Function description

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、ディスク上のセットアップモジュールの後、ディスク上の第6セクターから始まるセクターに配置される。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。このダミー割込みベクターは、デフォルトの「割込み無視」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の手続きを指示します。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。

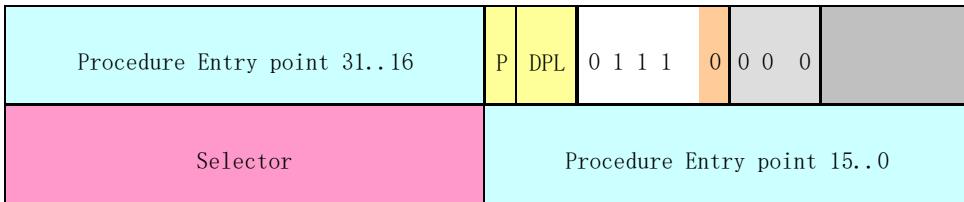


図6-9 IDTにおけるインタラプトゲート記述子のフォーマット

In the head.s program, the segment selector field in the interrupt gate descriptor is set to 0x0008, which indicates that the dummy interrupt service routine ignore\_int is in the kernel code. and the offset is set to the offset of ignore\_int interrupt service handler in the head.s program. Since the head.s program is moved to the beginning of memory address 0, the offset of the interrupt service handler is also the offset in the kernel code. Since the kernel code segment is always in memory and the privilege level is 0 (ie, P=1, DPL=00), it can be seen from the figure that the value of byte 5 and byte 4 of the interrupt gate descriptor should be 0x8E00.

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内に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、無限ループに入れます。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のテストを続け、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

Here, the attribute flag of each entry is set to 0x07 (P=1, U/S=1, R/W=1), indicating that the page exists and the user can read and write. The reason for setting the kernel page table attribute in this way is that both the segmentation and the paging management have protection methods. The protection flags (U/S, R/W) set in the page directory and page table entries need to be combined with the privilege level (PL) protection in the segment descriptor. But the PL in the segment descriptor plays a major role. The CPU will first check the segment protection and then check the page protection. If the current privilege level CPL < 3 (for example, 0), the CPU is running as a supervisor. At this point all pages can be accessed, and free to read and write. If CPL = 3, the CPU is running as user (user). At this point only the pages belonging to user (U/S=1) are accessible, and only pages marked as readable and writable (W/R = 1) are writable. At this time, the page belonging to the super user (U/S=0) can neither be written nor read. Because the kernel code is a bit special, it contains the code and data for task 0 and task 1. So setting the page property to 0x7 here will ensure that the two tasks can be executed in user mode, but they cannot access kernel resources arbitrarily.

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 Code Comments

プログラム 6-3 linux/boot/head.s

---

```

1  /*
2   * linux/boot/head.s_
3   *
4   * (C) 1991 Linus Torvalds_
5   */
6
7 /*
8  * head.s contains the 32-bit startup code. _
9  *
10 * NOTE!!! Startup happens at absolute address 0x00000000, which is also where
11 * the page directory will exist. The startup code will be overwritten by
12 * the page directory. _
13 */
14 .text
15 .globl _idt,_gdt,_pg_dir,_tmp_floppy_area
16 _pg_dir:           # The page directory will be stored here.

```

# 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レクタ # 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でセレクタ=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ーテーブルの項目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 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:           # set each data segment registers.
18     movl $0x10,%eax    # direct operand starts with '$', otherwise it's an address.
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, set system stack.
24     call setup_idt        # line 67--93
25     call setup_gdt        # line 95--107
26     movl $0x10,%eax        # reload all the segment registers
27     mov %ax,%ds            # after changing gdt. CS was already
28     mov %ax,%es            # reloaded in 'setup_gdt'
29     mov %ax,%fs            # GDT changed, all segs need to be reloaded.
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にです。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、0x0000000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) の値と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行がストップ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:     incl %eax          # check that A20 really IS enabled
34     movl %eax,0x000000        # loop forever if it isn't
35     cmpl %eax,0x100000
36     je 1b                  # '1b' means backward label 1.
                                # '5f' means forward 5.

38 37 /*
39 * NOTE! 486 should set bit 16, to check for write-protect in supervisor
40 * mode. Then it would be unnecessary with the "verify_area()" -calls.
41 * 486 users probably want to set the NE (#5) bit also, so as to use
42 * int 16 for math errors.
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・オンライン方式を # 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 $0x80000011,%eax    # Save PG,PE,ET_
45 /* "orl $0x10020,%eax" here for 486 might be good */
46     orl $2,%eax             # set MP
47     movl %eax,%cr0
48     call check_x87
49     jmp after_page_tables   # line 135
50
51 */

52 * ETが正しいかどうかに依存しています。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。# finit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:
55     fninit

```

```
56 fstsw %ax          # get status word -> ax
57 cmpb $0,%al        # status word should be 0 after fninit if has a math.
58 je 1f              /* no coprocessor: have to set bits */
59 movl %cr0,%eax
60 xorl $6,%eax      /* reset MP, set EM */
```

```

61      movl %eax,%cr0
62      ret

```

# .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 ( $2^2$ ) 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。

287 の fsetpm、387 では無視される \*/\*。

70 68 \* setup\_idt 69

```

*
-
71  * sets up a idt with 256 entries pointing to
72  * ignore_int, interrupt gates. It then loads
73  * idt. Everything that wants to install itself
74  * in the idt-table may do so themselves. Interrupts
75  * are enabled elsewhere, when we can be relatively
76  * sure everything is ok. This routine will be over-
77  * written by the page tables.
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

#続いて行われる初期化プロセスでは、カーネルは現在のデフォルトの設定を#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          # effective addr of ignore_int -> edx
80     movl $0x00080000,%eax       # store selector 0x0008 to high word of eax
81     movw %dx,%ax               /* selector = 0x0008 = cs */
82                                         # 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。
83     movw $0x8E00,%dx           /* interrupt gate - dpl=0, present */
84                                         # edx contains high 4 bytes of gate descriptor.
85     lea _idt,%edi             # _idt is IDT table address (offset).
86     mov $256,%ecx_
86 rp_sidt:
87     movl %eax,(%edi)           # store the dummy descriptor into IDT table

```

```
88    movl %edx,4(%edi)      # store eax to [edi+4]
89    addl $8,%edi          # edi point to the next item by plus 8.
90    dec %ecx
91    jne rp_sidt
```

```

92      lidt idt_descr          # Load the interrupt descriptor table register.
93      ret
94
95 /*
98 96 * setup_gdt 97
*
-
99 * This routines sets up a new gdt and loads it.
100 * Only two entries are currently built, the same
101 * ones that were built in init.s. The routine
102 * is VERY complicated at two whole lines, so this
103 * rather long comment is certainly needed :).
104 * This routine will be overwritten by the page tables.
104 */
105 setup_gdt:
106     lgdt gdt_descr          # contents on lines 234-238.
107     ret
108
110 109 /*
111 * I put the kernel page tables right after the page directory,
112 * using 4 of them to span 16 Mb of physical memory. People with
113 * more than 16MB will have to expand this.
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。

#ここで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。

# # 以下のコードやデータは、オフセット0x5000から始まります。

```

128 * tmp_floppy_area is used by the floppy-driver when DMA cannot
129 * reach to a buffer-block. It needs to be aligned, so that it isn't
130 * on a 64kB border.

```

```
131 */  
132 _tmp_floppy_area:  
133     .fill 1024,1,0          # 1024 bytes filled with 0.  
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ることになります。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  
after\_page\_tables:

```

135
136     pushl $0          # These are the parameters to main :-
137     pushl $0
138     pushl $0
139     pushl $L6          # return address for main, if it decides to.
140     pushl $_main        # '_main' is the internal representation of main().
141     jmp setup_paging   # jump to line 198.
142 L6:
143     jmp L6            # main should never return here, but
144                         # just in case, we know what happens.
145
146 /* This is the default interrupt "handler" :-)
147 int_msg:
148     .asciz "Unknown interrupt\n\r"      # 定义字符串“未知中断(回车换行)”。
149 .align 2                  # alignment with 4 bytes in memory.
150 ignore_int:
151     pushl %eax
152     pushl %ecx
153     pushl %edx
154     pushl %ds           # ds, es, fs, gs still occupy 2 words each when on stack.
155     pushl %es
156     pushl %fs
157     movl $0x10,%eax    # set selector (ds, es, fs points data descriptor in gdt)
158     movl %ax,%ds
159     movl %ax,%es
160     movl %ax,%fs
# Put printk() function's parameter pointer onto the stack. Note that if '$' is not added
# before int_msg, it means that the long word ('Unkn') at the int_msg symbol is pushed
# onto the stack. This function is in /kernel/printk.c. '_printk' is the internal
# representation in the printk compiled module.
161     pushl $int_msg

```

```
162     call _printk
163     popl %eax          # /kernel/printk.c
164     pop %fs
165     pop %es
166     pop %ds
167     popl %edx
168     popl %ecx
169     popl %eax
170     iret              # iret pop out CFLAGS too.
```

```

171
172
173 /*
174 * Setup_paging
175 *
176 175 *
177 * This routine sets up paging by setting the page bit
178 * in cr0. The page tables are set up, identity-mapping
179 * the first 16MB. The pager assumes that no illegal
180 * addresses are produced (ie >4Mb on a 4Mb machine).
181 180 *
182 * NOTE! Although all physical memory should be identity
183 * mapped by this routine, only the kernel page functions
184 * use the >1Mb addresses directly. All "normal" functions
185 * use just the lower 1Mb, or the local data space, which
186 * will be mapped to some other place - mm keeps track of
187 * that.
188 187 *
189 * For those with more memory than 16 Mb - tough luck. I've
190 * not got it, why should you :-) The source is here. Change
191 * it. (Seriously - it shouldn't be too difficult. Mostly
192 * change some constants etc. I left it at 16Mb, as my machine
193 * even cannot be extended past that (ok, but it was cheap :-)
194 * I've tried to show which constants to change by having
195 * some kind of marker at them (search for "16Mb"), but I
196 * won't guarantee that's all :-( )
197 */
# 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                      # align memory boundaries in 4-byte.
198 setup_paging:
199     movl $1024*5,%ecx        /* 5 pages - pg_dir+4 page tables */
200     xorl %eax,%eax
201     xorl %edi,%edi         /* pg_dir is at 0x000 */
202     cld;rep;stosl           # eax -> [es:edi], edi increased by 4
# 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。  
203 # 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す。

```
204     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(0~0xffff)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 -> points to the last item in the last page
208     movl $0xffff007, %eax      /* 16Mb - 4096 + 7 (r/w user, p) */
                                # the last item map to physical mem addr 0xffff000 + 7
209     std
210 1:    stosl              # set direction flag, edi is decreased (by 4 bytes).
211     subl $0x1000, %eax      /* fill pages backwards - more efficient :-)
212     jge 1b                # each time an item is filled, addr is reduced by 0x1000.
```

# Now set the page directory base register cr3, it contains the physical address of the  
# page directory table. Then set to start using paging (the PG flag of cr0, bit 31).

```
213     xorl %eax, %eax        /* pg_dir is at 0x0000 */
214     movl %eax, %cr3        /* cr3 - page directory start */
215     movl %cr0, %eax
216     orl $0x80000000, %eax  # add PG flag
217     movl %eax, %cr0        /* set paging (PG) bit */
218     ret                   /* this also flushes prefetch-queue */
```

# After changing the paging flag, it is required to use the branch instruction to refresh  
# the prefetch instruction queue. The ret instruction is used here. Another function of  
# this return instruction is to pop the address of the main() pushed by instructions on  
# line 140, and jump to the /init/main.c program to run. This program really ends here.

```
219
220 .align 2                # align memory boundaries in 4-byte.
221 .word 0                 # skip a word, so that line 224 are 4-byte aligned.
```

# 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。

idt\_descr:



```
222
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
227 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。
228 gdt_descr:
229     .word 256*8-1          # so does gdt (note that that's any
230     .long _gdt            # magic number, but it works for me :^)
230
231     .align 3              # align memory boundaries by 8 ( $2^3$ ) bytes.
232 _idt:   .fill 256,8,0    # idt is uninitialized
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 descriptor */
235     .quad 0x00c09a0000000fff        /* 16Mb */ # 0x08, kernel code seg.
236     .quad 0x00c0920000000fff        /* 16Mb */ # 0x10, kernel data seg.
237     .quad 0x0000000000000000      /* TEMPORARY - don't use */
238     .fill 252,8,0                /* space for LDT's and TSS's etc */

```

## 6.4.3 Reference Information

### 6.4.3.1 Memory map after the end of head.s program execution

After the execution of head.s program, the kernel code has officially completed the settings of the memory page directory and the page tables, and re-created the interrupt descriptor table IDT and the global descriptor table GDT. In addition, the program also opened a 1KB byte buffer for the floppy disk driver. At this time the detailed image of the system module in memory is shown in Figure 6-11.

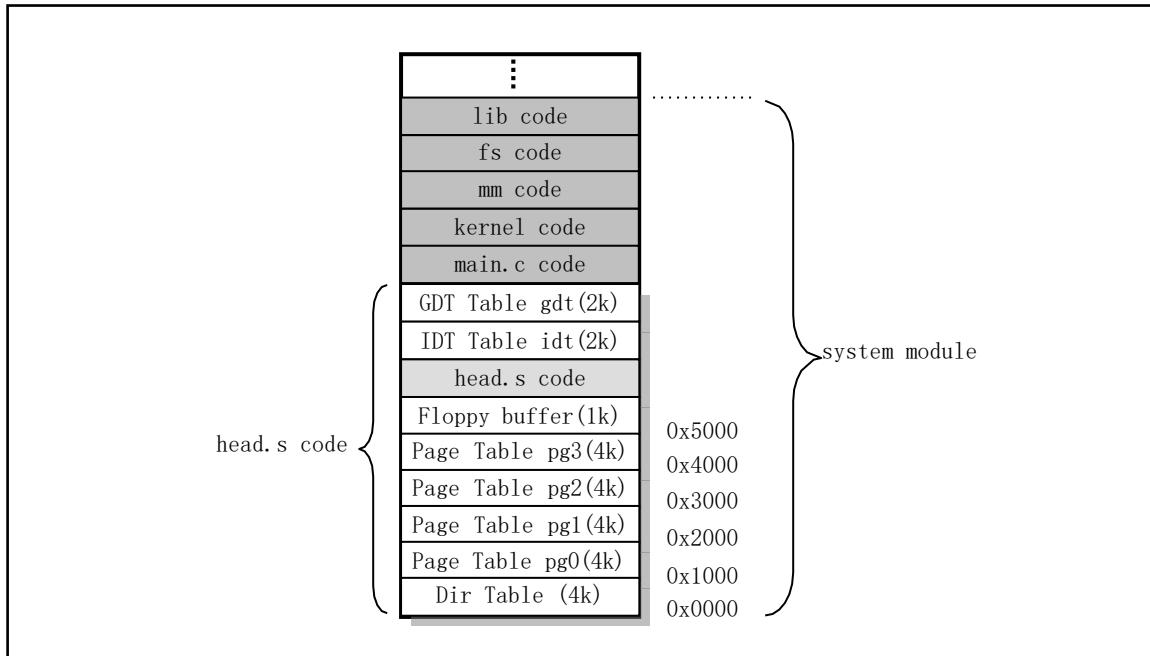


図6-11 メモリ内のシステムモジュールのマップ<sup>9</sup>

### 6.4.3.2 Intel 32-bit protection operation mechanism

The key to understanding this program is to know the operating mechanism of the Intel 80X386 32-bit protection mode. In order to be compatible with the 8086 CPU, the 80X86 protected mode was designed to be more complicated. See Chapter 4 for a detailed description of how the protected mode operates. Here we make a brief introduction to the protection mode by comparing the real mode and the protection mode.

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アイテム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ジスのほか、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ッシングモードは、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。

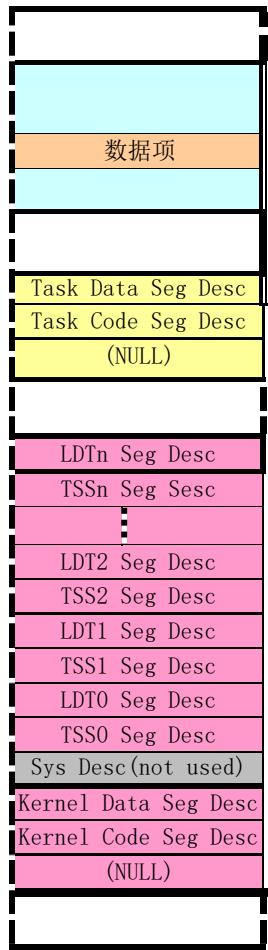


図6-12 Linuxカーネルが使用するディスクリプターテーブル

### 6.4.3.3 align directive

We have already explained the align directive when we introduced the assembler in Chapter 3. Here we will summarize it. The use of the directive `.align` is to instruct the compiler to populate the location counter (like the instruction counter) to a specified memory boundary at compile time. The goal is to increase the speed and efficiency with which the CPU can access code or data in memory. Its full format is:

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

1つ目のパラメータ値val1は必要なアライメント、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にアライメントが必

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 ( $2^{\text{Val1}}$ ) 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ー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.s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。

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



## 7 Initialization program (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ニ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っ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ーファイル(\*.h)については、要約の意味のみを示しています。ヘッダーファイルの具体的な詳細コメントは、コメントヘッダー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 Function description

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

Then, the kernel code performs various aspects of hardware initialization work. This includes trap gates, block devices, character devices, and ttys, as well as manually setting up the first task (task 0). After all initialization work is completed, the kernel sets the interrupt enable flag to turn on the interrupt and switches to task 0 to run. At this point, it can be said that the kernel has basically completed all the setup work. The kernel then creates several initial tasks through task 0, runs the shell program and displays the command line prompt, so that the Linux system is in normal operation.

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 Kernel initialization flow diagram

0. 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

1. Then, the system first calls the process creation function fork() to create a child process (init process) for running init(). The entire initialization process of the system is shown in Figure 7-2.

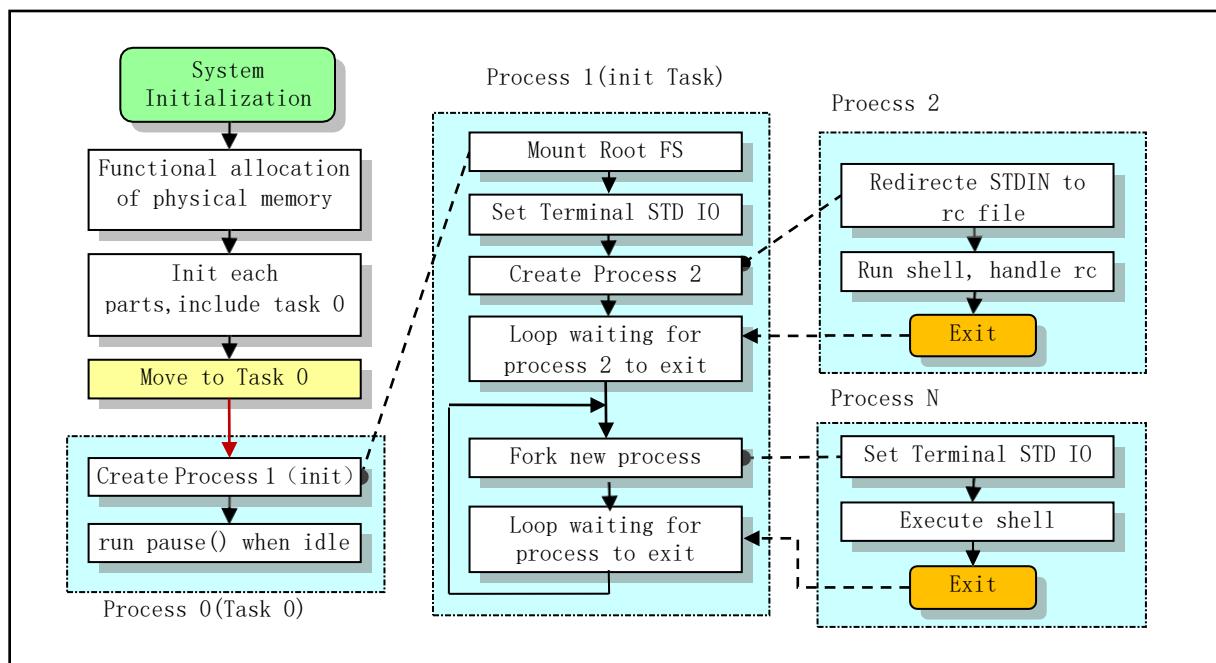


図7-2 カーネル初期化処理フロー図

As can be seen from the figure, the main.c program first determines how to allocate the system physical memory, and then calls the initialization functions of each part of the kernel to initialize the memory management, interrupt processing, block device and character device, process management, and hard disk and floppy disk hardware. After these operations are completed, the various parts of the system are already operational. The program then moves itself "manually" to task 0 (process 0) and uses the fork() call to create process 1 (init process) for the first time and calls the init() function there. In this function the program will continue to initialize the application environment and execute the shell login program. The original process 0 is scheduled to execute when the system is idle, so process 0 is also commonly referred to as the idle process. At this point, process 0 only executes the pause() system call and calls the scheduler function.

2. 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プログラムの実行。
  3. Install root file system
  4. 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トにロードしようとします。
  5. Display system information
  6. 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。
  7. Run resource configuration file
  8. 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、DOSOS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。
  9. Execute user login shell

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れます。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ます。その後、親プロセスは待機状態に入ります。その後、ユーザ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 Operations of the initial user stack

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でこのsyscallを実行します（プログラムの23行目を参照）。

By declaring an inline function, you can have gcc integrate the code of the function into the code that called it. This will speed up code execution because it saves the overhead of function calls. In addition, if any of the actual arguments is a constant, then these known values at compile time may make the code simpler without including all the code for the inline function. See the relevant instructions in Chapter 3.

### 23 static inline \_syscall0(int,fork)

Where \_syscall0() is the inline macro code in unistd.h, which calls the Linux system call interrupt int 0x80 in the form of embedded assembly. According to the macro definition on line 150 of the include/unistd.h file, we expand this macro and replace it with the above line. It can be seen that this statement is actually an int fork() creation process system call, as shown below.

---

```
// 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^)
153 151 type
name(void) ○152
{ ○152
154 long res; \
155 __asm__ volatile ("int $0x80" \
156 : "=a" ( res ) \
157 : "0" ( NR_##name )); \
158 if (_res >= 0) \
159     return (type) res; \
160 errno = - res; \
161 return -1; \
```

---

According to the above definition, we can expand \_syscall0(int, fork). After substituting the 23rd line, we can get the following statement:

---

```
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。
}
```

It can be seen that this is an inline function definition. gcc will insert the statement in the above "function" body

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状態になります。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る。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 Program Annotations

プログラム 7-1 linux/init/main.c

```
1 /*  
2  * linux/init/main.c  
3  *  
4  * (C) 1991 Linus 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。  
7 // 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。  
8 #define _LIBRARY  
9 #include <unistd.h>  
10#include <time.h>           // time type header file. tm structure, time function prototypes.  
11
```

```

12 11 /*
13 * we need this inline - forking from kernel space will result
14 * in NO COPY ON WRITE (!!!), until an execve is executed. This
15 * is no problem, but for the stack. This is handled by not letting
16 * main() use the stack at all after fork(). Thus, no function
17 * calls - which means inline code for fork too, as otherwise we
18 * would use the stack upon exit from 'fork()'.
```

18 \*

```

19 * Actually only pause and fork are needed inline, so that there
20 * won't be any messing with the stack from main(), but we define
21 * some others too.
```

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。この時点では、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で定義されている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  
// すべてのシステム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  
// of the initial task 0, and some embedded assembly function macro statements about  
// the descriptor parameter settings and acquisition.  
// <linux/head.h> 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が定義されています。

```
// along with several selector constants.  
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。  
// descriptors/interrupt gates, etc. is defined.  
// <asm/io.h> io のヘッダーファイルです。の io ポートを操作する関数を定義します。  
// form of a macro's embedded assembler.  
// <stddef.h> 標準定義のヘッダーファイルです。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。  
// of macros. It mainly describes one type (va_list) and three macros (va_start, va_arg  
// and va_end) for the vsprintf, vprintf, and vfprintf functions.
```

```

// <unistd.h> Linux標準のヘッダーファイルです。様々なシンボル定数や型が定義されている
// and various functions are declared. If _LIBRARY_is defined, it also includes the
// system call number and the inline assembly _syscall0().
// <fcntl.h> ファイル制御のヘッダーファイルです。演算制御定数の定義は
// symbol used for the file and its descriptors.
// <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のヘッダーファイル。ファイルテーブル構造の定義 (file, buffer_head, m_inode, etc.)
// <string.h> 文字列のヘッダーファイルです。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>      // teletype terminal.
29 #include <linux/sched.h>    // scheduler.
30 #include <linux/head.h>     // kernel head.
31 #include <asm/system.h>    // system machine.
32 #include <asm/io.h>        // port io.
33
34 #include <stddef.h>        // standard definition.
35 #include <stdarg.h>        // standard arguments.
36 #include <unistd.h>
37 #include <fcntl.h>          // file control.
38 #include <sys/types.h>      // date types.
39
40 #include <linux/fs.h>       // file system.
42 #include <string.h>         // string operations.
43
44 static char printbuf[1024];           // cache for kernel to display information.
45
46 extern char *strcpy();               // external functions defined elsewhere.
47 extern int vsprintf();              // Formatted output into a string (vsprintf.c, 92).
48 extern void init(void);             // Function prototype, init (168).
49 extern void blk_dev_init(void);     // Block dev init (blk_drv/ll_rw_blk.c, 210).
50 extern void chr_dev_init(void);    // Char dev init(chr_drv/tty_io.c, 402)
51 extern void hd_init(void);         // Hard disk init(blk_drv/hd.c, 378)
52 extern void floppy_init(void);     // Floppy init (blk_drv/floppy.c, 469)
53 extern void mem_init(long start, long end); // Memory init (mm/memory.c, 443)
54 extern long rd_init(long mem_start, int length); // Ramdisk init (blk_drv/ramdisk.c, 52)
55 extern long kernel_mkttime(struct tm * tm); // Calc boot time (kernel/mkttime.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
// format in which the output will be used, refer to the standard C language book. This
// 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。

```

59 57 static int sprintf(char \* str, const char \*fmt, ...) 58 {...

```

60     va_list args;
61     int i;
62
63     va_start(args, fmt);
64     i = vsprintf(str, fmt, args);
65     va_end(args);
66     return i;
67 }
```

68 /\*

```

69 * This is set up by the setup-routine at boot-time
70 */
// 以下の3行は、指定されたリニアアドレスを、強制的に
// 与えられたデータ型を指定して、その内容を取得します。カーネルコードのセグメントは
// 物理アドレス0から始まる位置にマッピングされた、これらのリニアアドレスは
// 対応する物理アドレスもの意味については、第6章の表6-3を参照してください。
// the memory values at these specified addresses. See line 125 below for the drive_info
// の構造になっています。
71 #define EXT_MEM_K (*(unsigned short *)0x90002) // Extented Mem size(KB) beyond 1MB
72 #define CON_ROWS ((*unsigned short *)0x9000e) & 0xff // Console rows and colums.
73 #define CON_COLS (((*unsigned short *)0x9000e) & 0xff00) >> 8)
74 #define DRIVE_INFO (*struct drive_info *)0x90080 // Harddisk parameter table(32 bytes)
75 #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC) // Root fs dev number.
76 #define ORIG_SWAP_DEV (*(unsigned short *)0x901FA) // Swap file dev number.
77
78 /*
79 * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
80 * and this seems to work. I anybody has more info on the real-time
81 * clock I'd be interested. Most of this was trial and error, and some
82 * bios-listing reading. Urghh.
83 */
84
// CMOSリアルタイムクロック情報の読み出しを行うマクロです。Outb_pとinb_pはポート入力
// 0x70は書き込みアドレスポートで、0x71は
85 //はリードデータポートです。0x80|addrは読み込まれるCMOSメモリのアドレスです。
86 #define CMOS_READ(addr) ({ \
87     outb_p(0x80|addr, 0x70); \
88     inb_p(0x71); \
89 })
// BCDコードをバイナリ値に変換するマクロを定義します。BCDコードはハーフバイトの
// (4ビット)で10進数を表現するため、1バイトで2つの10進数を表現することができます。
// '(val)&15'はBCDで表される10進数を、'((val)>>4)*10'は
// BCDの10進数の10桁目を10倍します。最後の2つは加算されます。
// 1バイトのBCDコードの実際のバイナリ値である。
90 #define BCD_TO_BIN(val) ((val)=((val)&15)+((val)>>4)*10)
91
// この関数は、CMOSリアルクロック情報をブートタイムとして、それを
// グローバル変数 startup_time (秒)。後述のCMOSメモリリストの説明を参照してください。
// 関数 kernel_mktime()を使用している。
// 1970年1月1日0:00から起動した日までを起動時間とします。
92 static void time_init(void) 93
{...
94     struct tm_time; // defined in include/time.h
95
// CMOSアクセスが非常に遅い。時間の誤差を小さくするために、すべてのデータを読み込んだ後に

```

```
// の値を次のループで表示します。このとき、CMOSで2番目の値が変化していれば
// すべての値が再び読み込まれます。これにより、コアはCMOSタイムエラーを制御して
/
/
1
秒
以
内
に
96
97     time.tm_sec = CMOS_READ(0);    // current time seconds (BCD code).
98     time.tm_min = CMOS_READ(2);    // current minutes.
99     time.tm_hour = CMOS_READ(4);   // current hours.
```

```

100         time.tm_mday = CMOS_READ(7);    // day in a month.
101         time.tm_mon = CMOS_READ(8);    // current month (1-12).
102         time.tm_year = CMOS_READ(9);    // current year.
103     } while (time.tm_sec != CMOS_READ(0));
104     BCD_TO_BIN(time.tm_sec);           // BCD to binary value.
105     BCD_TO_BIN(time.tm_min);
106     BCD_TO_BIN(time.tm_hour);
107     BCD_TO_BIN(time.tm_mday);
108     BCD_TO_BIN(time.tm_mon);
109     BCD_TO_BIN(time.tm_year);
110     time.tm_mon--;                  // range of months in tm_mon is 0-11.
111     startup_time = kernel_mktime(&time); // calc boot time (kernel/mktime.c, 41)
112 }
113

// main.cソースファイルからのみアクセス可能ない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;          // machine physical memory size (bytes).
115 static long buffer_memory_end = 0;   // buffer end address.
116 static long main_memory_start = 0;  // the main memory start position.
117 static char term[32];              // terminal settings (environment parameter).
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 }; // parameter string array
120 static char * envp_rc[] = { "HOME=/", NULL, NULL }; // environment string array
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、シェルプログラム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 }; // as above.
123 static char * envp[] = { "HOME=/usr/root", NULL, NULL };
124
125 struct drive_info { char dummy[32]; } drive_info; // hard disk parameter table.
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。初期化の後は
void main(void)      /* This really IS void, no error here. */

```

```

127
128 {                                     /* The startup routine assumes (well, ...) this */
129 /*
130 * Interrupts are still disabled. Do necessary setups, then
131 * enable them
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
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ーテーブルを
// 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
134     ROOT_DEV = ORIG_ROOT_DEV;           // ROOT_DEV defined in fs/super.c, 29
135     SWAP_DEV = ORIG_SWAP_DEV;          // SWAP_DEV defined in mm/swap.c, 36
136     sprintf(term, "TERM=%dx%d", CON_COLS, CON_ROWS);

```

```

137     envp[1] = term;
138     envp_rc[1] = term;
139     drive_info = DRIVE_INFO;           // Copy hd parameter table at 0x90080

// 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。
// キャッシュのエンドアドレス -> buffer_memory_end; マシンのメモリ容量 -> memory_end;
140     // メインメモリのスタートアドレス -> main_memory_start.

141     memory_end = (1<<20) + (EXT_MEM_K<<10); // mem size = 1MB + extended mem (bytes)
142     memory_end &= 0xfffff000;                 // Ignore less than 4KB (1 page)
143     if (memory_end > 16*1024*1024)          // If mem size > 16MB, calculated as 16MB.
144         memory_end = 16*1024*1024;
145     if (memory_end > 12*1024*1024)          // If mem size >12MB, set buffer end = 4MB
146         buffer_memory_end = 4*1024*1024;
147     else if (memory_end > 6*1024*1024)        // if size >6Mb, set buffer end = 2Mb
148         buffer_memory_end = 2*1024*1024;
149     else
150         buffer_memory_end = 1*1024*1024; // Otherwise set buffer end = 1Mb
151     main_memory_start = buffer_memory_end;

// If symbol RAMDISK is defined in the Makefile, the virtual disk is initialized. At this
152 //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。
153 #ifdef RAMDISK
154     main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
155 #endif
156 // 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。
157 // 読むときは、それぞれのinit関数に従う。

158     mem_init(main_memory_start, memory_end); // Main mem area (mm/memory.c, 443)
159     trap_init();                      // trap gate (hw vector) init (kernel/traps.c, 181)
160     blk_dev_init();                  // block dev init (blk_drv/ll_rw_blk.c, 210)
161     chr_dev_init();                  // char dev init (chr_drv/tty_io.c, 402)
162     tty_init();                     // tty init (chr_drv/tty_io.c, 105)
163     time_init();                   // setting boot time (line 92)
164     sched_init();                  // scheduler init (kernel/sched.c, 385)
165     buffer_init(buffer_memory_end); // buffer init(fs/buffer.c, 348)
166     hd_init();                     // harddisk init (blk_drv/hd.c, 378)
167     floppy_init();                // floppy init(blk_drv/floppy.c, 469)
168     sti();                        // All init has completed, so enable interrupt.

169 // スタック内のデータを操作してタスク0を開始し、割込みリターンを使用する
170 // 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
171 // 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
172 // 0を返し、元のプロセス（親プロセス）に対しては、プロセス
173 // 子プロセスのpid数。
174     move_to_user_mode(); // see head file include/asm/system.h.
175     if (!fork()) {
176         /* we count on this going ok */
177         init();           // run init() in task 1 (init process).
178     }

179 // 以下のコードはタスク0で実行されます。
180 // 168 /*

```

170 \* NOTE!! For any other task 'pause()' would mean we have to get a  
171 \* signal to awaken, but task0 is the sole exception (see 'schedule()')

```

172 * as task 0 gets activated at every idle moment (when no other tasks
173 * can run). For task0 'pause()' just means we go check if some other
174 * task can run, and if not we return here.
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ことを見つけます。
// 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。
81 179 static int printf(const char *fmt, ...) 180 ...
182     va_list args;
183     int i;
183
184     va_start(args, fmt);
185     write(1, printbuf, i=vsprintf(printbuf, fmt, args));
186     va_end(args);
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
// 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テムコールです。
// 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
// 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
194 // 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。
195     setup((void *) &drive_info);
// 端末のコンソールデバイス「/dev/tty0」を読み書き可能な状態で開く。これは
// 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（ファイルデスクリピタ）
// 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。
// UNIX系OSのデバイスstdin。ここにコピーされ、個別に開かれる
// 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//。
// 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

```

```
196 // 値を返さないようにする関数です。  
197     (void) open( "/dev/tty1", O_RDWR, 0 );  
198     (void) dup(0);           // duplicate file handle 0 to produce stdout (1).  
199     (void) dup(0);           // duplicate file handle 0 to produce stderr (2).
```

```

// Print the buffer blocks (1024 bytes per block) and total number of bytes, as well as
// free bytes in the main memory area.
198     printf ("%d buffers = %d bytes buffer space |n|r", NR_BUFFERS,
199             NR_BUFFERS*BLOCK_SIZE);
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、標準入力が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
201   // 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。
202   if (!(pid=fork())) {
203       close(0);
204       if (open ("/etc/rc", O_RDONLY, 0))
205           exit(1); // open failed, (lib/_exit.c, 10)
206       execve ("/bin/sh", argv_rc, envp_rc); // execute program /bin/sh
207       exit(2);
208
// 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。
209   if (pid>0)
210       while (pid != wait(&i))
211           /* nothing */;
212
// ここでコードが実行されると、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

```

```
212 // 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。  
213     while (1) {  
214         if ((pid=fork())<0) {  
215             printf("Fork failed in init|r|n");  
216             continue;  
217         }  
218         if (!pid) {           // new process  
219             close(0);close(1);close(2);  
220         }
```

```

218         setsid() ;                                // create a new session.
219         (void) open ("/dev/tty1", O_RDWR, 0);
220         (void) dup(0);
221         (void) dup(0);
222         exit (execve ("/bin/sh", argv, envp));
223     }
224     while (1)
225         if (pid == wait(&i))
226             break;
227         printf ("|n|rchild %d died with code %04x|n|r", pid, i);
228         sync();                                     // flush the buffer.
229     }
230     exit(0);        /* NOTE! _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IOなどを行い、その後、sys_exitを呼び出します。
231 }
232

```

## 7.1.3 Reference Information

### 7.1.3.1 CMOS

The CMOS memory of a PC is a 64- or 128-byte memory block powered by a battery, usually part of the real-time clock (RTC) chip. Some machines have a larger CMOS memory capacity. The 64-byte CMOS was originally used on IBM PC-XT machines to store clock and date information in a BCD format. Since this information only uses 14 bytes, the remaining bytes can be used to store some system configuration data.

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。

	Description	Address	Description

Address			
0x00	Current Seconds (real clock)	0x11	Reserved
0x01	Alarm Seconds	0x12	Hard drive type
0x02	Current Minutes (real clock)	0x13	Reserved
0x03	Alarm Seconds	0x14	Device byte
0x04	Current Hours (real clock)	0x15	Basic memory (low byte)
0x05	Alarm Hours	0x16	Basic memory (high byte)
0x06	Current day of the week (real clock)	0x17	Extended memory (low byte)

0x07	Date of the day in a month (real clock)	0x18	Extended memory (high byte)
0x08	Current month (real clock)	0x19-0x2d	Reserved
0x09	Current year (real time clock)	0x2e	Checksum (low byte)
0x0a	RTC Status Register A	0x2f	Checksum (high byte)
0x0b	RTC Status Register B	0x30	Extended memory above 1MB (low byte)
0x0c	RTC Status Register C	0x31	Extended memory above 1MB (high byte)
0x0d	RTC Status Register D	0x32	Current century
0x0e	POST diagnostic status byte	0x33	Information flag
0x0f	Shutdown status byte	0x34-0x3f	Reserved
0x10	Floppy disk drive type		

### 7.1.3.2 Forking a new process

Fork() is a system call function that copies the current process and creates a new entry in the process table that is almost identical to the original process (called the parent process) and executes the same code. But the new process (child process) has its own data space and environment parameters. The main purpose of creating a new process is to run the program concurrently, or use the exec() cluster function to execute other different programs in the new process.

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。

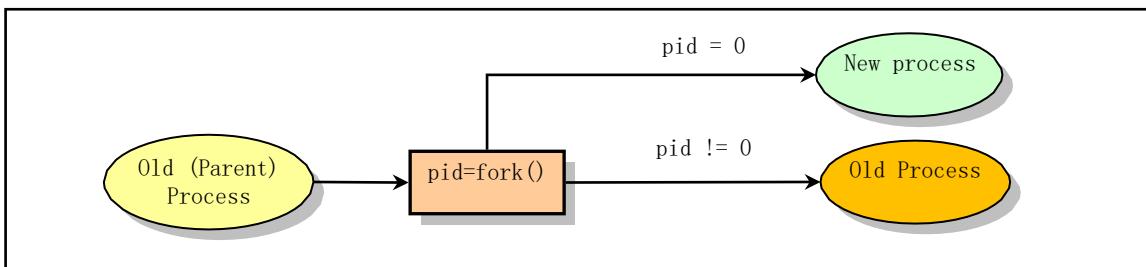


図7-3 fork()を呼び出して新しいプロセスを作る

The init() uses the return value of the fork() to distinguish and execute different code parts. Lines 201 and 216 in the main.c program are the code blocks that the child process starts executing (using the execve() system call to execute other programs, where sh is executed), and lines 208 and 224 are the code blocks executed by the parent process.

**7.1.3.3** 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。

### 7.1.3.4 Concept of session

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
```

---

Each of these commands: cat, grep, and more belong to a process group.

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。したがって、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。

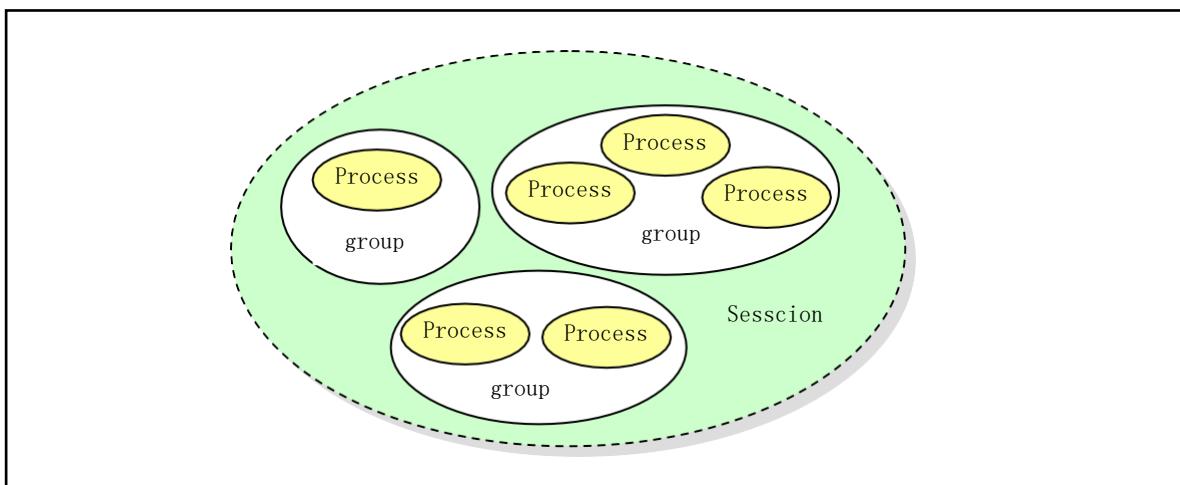


図7-4 プロセス、プロセスグループ、セッションの関係

Several process groups in a session are divided into a foreground process group and one or several background process groups. A terminal can only act as a control terminal for a session. A foreground process group is a process group that has a control terminal in the session, and other process groups in the session become a background process group. The control terminal corresponds to the /dev/tty device file, so if a process needs to access the control terminal, you can directly read and write the /dev/tty file.

## 7.2 Environment initialization

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ーフェースに入る。両者の実行関係を図7-5に示す。

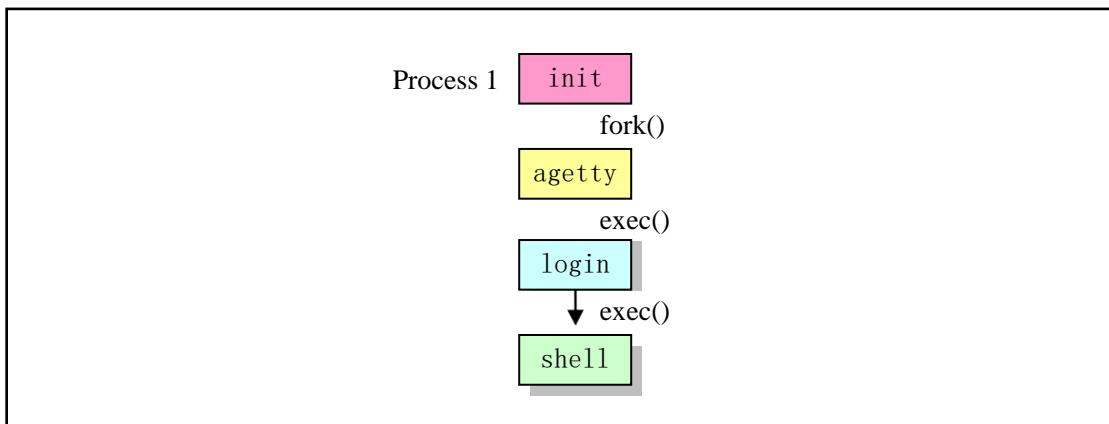


図7-5 環境初期化の手順

Although these programs (init, getty, login, shell) are not part of the kernel, a basic understanding of their role will facilitate understanding of the many functions that the kernel provides.

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行っています。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に、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がチェックされます。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ーフェースである。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'-'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
```

---

When executing the shell, in addition to some of the specified options, if the command line argument is

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 Summary

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来れば、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ルするのか。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()とタイマー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 Kernel Code (kernel)

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/			
Filename	Size	Last Modified Time (GMT)	Description
<a href="#">blk_drv/</a>		1992-01-16 14:39:00	
<a href="#">chr_drv/</a>		1992-01-16 14:37:00	
<a href="#">math/</a>		1992-01-16 14:37:00	
<a href="#">Makefile</a>	4034 bytes	1992-01-12 19:49:12	
<a href="#">asm.s</a>	2422 bytes	1991-12-18 16:40:03	
<a href="#">exit.c</a>	10554 bytes	1992-01-13 21:28:02	
<a href="#">fork.c</a>	3951 bytes	1992-01-13 21:52:19	
<a href="#">mktime.c</a>	1461 bytes	1991-10-02 14:16:29	
<a href="#">panic.c</a>	448 bytes	1991-10-17 14:22:02	
<a href="#">printk.c</a>	537 bytes	1992-01-10 23:13:59	
<a href="#">sched.c</a>	9296 bytes	1992-01-12 15:30:13	
<a href="#">signal.c</a>	5265 bytes	1992-01-10 00:30:25	
<a href="#">sys.c</a>	12003 bytes	1992-01-11 00:15:19	
<a href="#">sys_call.s</a>	5704 bytes	1992-01-06 21:10:59	
<a href="#">traps.c</a>	5090 bytes	1991-12-18 19:14:43	
 <a href="#">vsprintf.c</a>	4800 bytes	1991-10-02 14:16:29	

### 8.1 Main Functions

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。

General purpose		Hardware Service	System-call Service
schedule.c	mktime	asm.s	sys_call.s
panic.c		traps.c	fork.c, sys.c, exit.c, signal.c
pintk, vsprintf			

図8-1 各ファイルの呼び出しの階層関係

### 8.1.1 Interrupt Processing

There are two files related to interrupt handling: asm.s and traps.c. Asm.s is used to implement the assembly processing part of the interrupt service caused by most hardware exceptions, while the traps.c implements the C function part called in the asm.s interrupt processing. This C function part sometimes called Bottom-halves of the interrupt handling. In addition, several other hardware interrupt handlers are implemented in the files sys\_call.s and mm/page.s. See Figure 5-21 for the connection and function of the 8259A programmable interrupt control chip in the PC.

Linuxシステムでは、割込みサービス機能はカーネルが提供しているため、割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、ファーコール（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ー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

The asm.s code file mainly deals with the processing of Intel's reserved interrupt INT0--INT16, and the remaining reserved interrupts (INT17--INT31) are reserved for future expansion by Intel Corporation. The processing of 16 interrupts (INT32--INT47) corresponding to the IRQ pins of the PIC chip will be set in the initialization routines of various hardware such as clock, keyboard, floppy disk, math coprocessor, hard disk, etc. . The handler for the Linux system call interrupt INT128 (INT0x80) will be given in kernel/sys\_call.s. The specific definition of each interrupt is described in the Reference Information section given after the code file comment.

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。

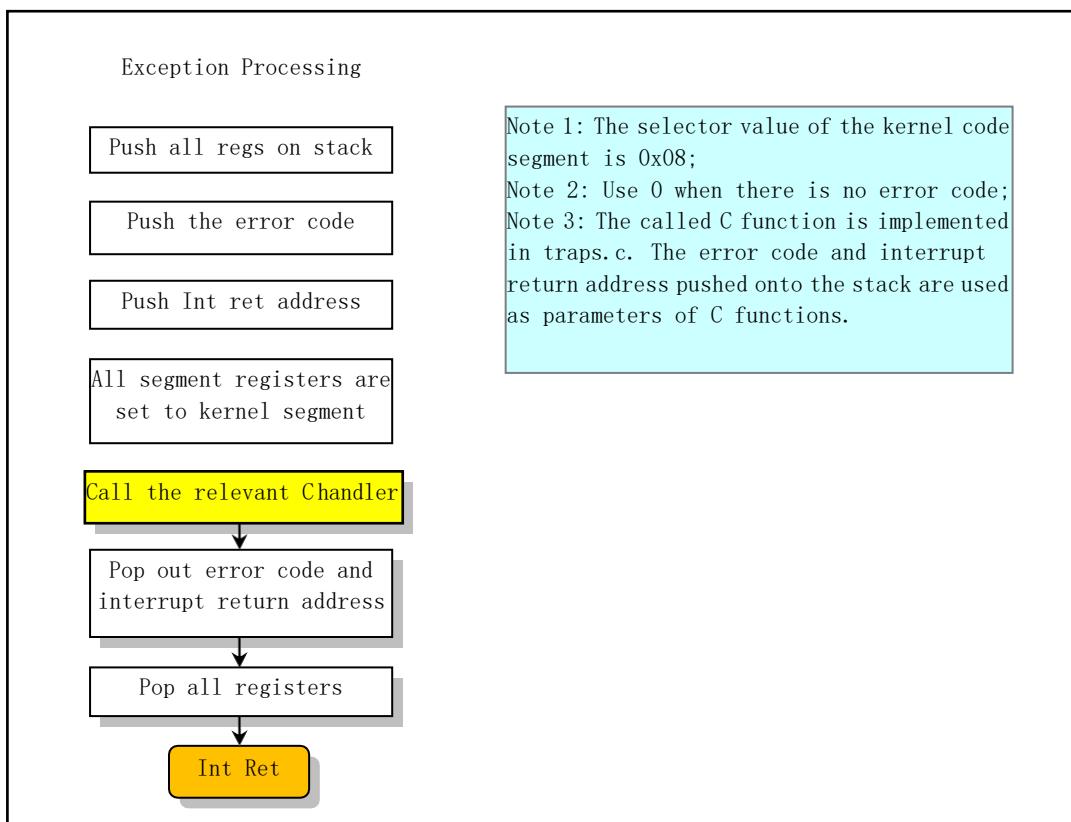


図 8-3 ハードウェア例外（フォールト、トラップ）処理フロー

### 8.1.2 System-call handling

In Linux, the application needs to use the interrupt INT 0x80 when using system resources, and the function call number needs to be placed in the register eax. If you need to pass parameters to the interrupt handler, you can use the registers ebx, ecx, and edx to store the parameters. Therefore the interrupt call is called a system call (syscall). The relevant files for implementing system calls include sys\_call.s, fork.c, signal.c, sys.c,

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プログラムのように動作します。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）。

**8.1.3** 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、C言語の'do\_'で始まる関数は、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()は基本的にすべ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固有のCプロセッサ関数です。

### 8.1.4 Other general-purpose programs

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となるスケジューラで、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したりするのに使われます。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 Function description

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push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、関数アドレスをebxレジスタにスワップします。その後の処理は、前述の図8-4(a)と同様です。

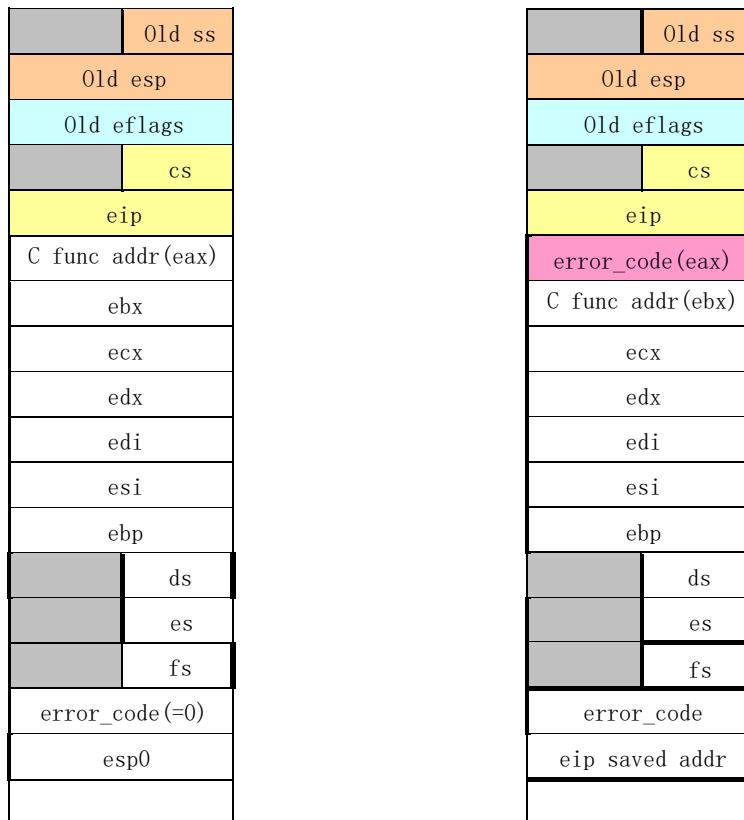


図8-4 エラー処理時のカーネルスタック変化の模式図

The reason for putting the error code and esp0 on the stack before the formal invocation to do\_divide\_error() is to use error code and esp0 as parameters do\_divide\_error(). In traps.c, the function signature is:

---

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

---

Therefore, the position and error code of the error can be printed in this C function. The processing of the remaining exceptions in the file is basically similar to the process described here.

## 8.2.2 Code Annotation

プログラム 8-1 linux/kernel/asm.s

```

1  /*
2  * linux/kernel/asm.s_
3  *
4  * (C) 1991 Linus Torvalds_
5  */
6
7 /*
8 * asm.s contains the low-level code for most hardware faults.
9 * page_exception is handled by the mm, so that isn't here. This
10 * file also handles (hopefully) fpu-exceptions due to TS-bit, as
11 * the fpu must be properly saved/resored. This hasn't been tested.
12 */
13 # TS - Task Switched、CR0のビット3。
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
15 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。
16 .globl _divide_error, _debug, _nmi, _int3, _overflow, _bounds, _invalid_op
17 .globl _double_fault, _coprocessor_segment_overrun
18 .globl _invalid_TSS, _segment_not_present, _stack_segment
19 .globl _general_protection, _coprocessor_error, _irq13, _reserved
20 .globl _alignment_check
21
22 # 以下の手順では、エラーコードのない例外を扱います。
23 # Int0 -- Handles errors of divided by zero.      Type: error;    Error code: None.
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
26 # 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
27 do_divide_error()から # コンパイルされたオブジェクトの対応する名前です。
28
29 # この関数はtraps.cの101行目にあります。
30 _divide_error:
31     pushl $_do_divide_error    # first push the C function address.
32 no_error_code:                  # no error code processing entry.
33     xchgl %eax, (%esp)       # _do_divide_error in stack exchanged with EAX
34     pushl %ebx
35     pushl %ecx
36     pushl %edx
37     pushl %edi
38     pushl %esi
39     pushl %ebp
40     pushl %ds                # occupied 4 bytes in stack.
41     pushl %es
42     pushl %fs
43     pushl $0                  # "error code"      # 0 as error code
44     lea 44(%esp), %edx        # get effective addr in esp to edx, points to the
45     pushl %edx                # ip of original interrupted code (at esp0).
46     movl $0x10, %edx          # Initialize ds, es, fs to kernel data seg selector.

```

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

```

39      mov %dx,%fs
# 以下の文の中の'*'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ルーチンを呼び出すことです。
40      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push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回実行したことと # 同じです。
41      call *%eax          # like: do_divide_error(long esp, long error_code)
42      addl $8,%esp
43      pop %fs
44      pop %es
45      pop %ds
46      popl %ebp
47      popl %esi
48      popl %edi
49      popl %edx
50      popl %ecx
51      popl %ebx
52      popl %eax          # recover original EAX
53      iret
53
# Int1 -- debug interrupt entry point. Type: Error/Trap (Fault/Trap); No error code.
# This exception is raised when the TF flag in EFLAGS is set. The CPU generates this
54 ハードウェアブレークポイントが検出されたとき、命令トレーストランプやタスクスワップ# ブ
# トラン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）。
55 _debug:
56     pushl $_do_int3      # _do_debug # push C routine address
57     jmp no_error_code    # line 22.
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ノリッジサイ
# 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
58 #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。
59 _nmi:
60     pushl $_do_nmi
61     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
# バッガによってコードに挿入され、処理が
_int3:

```

```
62
63     pushl $_do_int3
64     jmp no_error_code
65
# Int4 -- Overflow error interrupt entry point.  Type: Trap; no error code.
66 # 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。
67 _overflow:
68     pushl $_do_overflow
```

```

69      jmp no_error_code
69
70      # Int5 - バウンドチェックエラーの割り込みエントリーポイント。タイプです。エラー; エラーコードはありません
70      # 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
71      # 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
72      # い場合、# 5の例外が発生します。
72 _bounds:
73     pushl $_do_bounds
74     jmp no_error_code
73
74 _invalid_op:
75     pushl $_do_invalid_op
76     jmp no_error_code
77
77      # Int6 -- Invalid opcode interrupt entry point. Type: Error; no error code.
78      # The interrupt caused by the CPU actuator detecting an invalid opcode.
78 _invalid_op:
79     pushl $_do_invalid_op
80     jmp no_error_code
81
81      # Int9 -- The coprocessor segment overrun entry point. Type: Abandon; No error code.
82      # This exception is basically equivalent to coprocessor error protection. Because when
82      # 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
83      # ド # またはセーブする機会があります。
83 _coprocessor_segment_overrun:
84     pushl $_do_coprocessor_segment_overrun
85     jmp no_error_code
81
82      # Int15 -- インテルが予約している他の割り込みのエントリーポイントです。
83 _reserved:
84     pushl $_do_reserved
85     jmp no_error_code
85
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。
86      # 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
87      # 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
88      # す。以下の89行目では、0xF0がコプロセッサの
89      # 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、#
90      # 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
91      # 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
92      # るためのものです。
92 _irq13:
93     pushl %eax
94     xorb %al,%al
95     outb %al,$0xF0
96     movb $0x20,%al
97     outb %al,$0x20      # sent EOI (End of Interrupt) to 8259's master chip.
98     jmp 1f                # delay a while.
99 1:    jmp 1f
100 1:    outb %al,$0xA0      # sent EOI to 8259's slave chip.
101    popl %eax
102    jmp _coprocessor_error # code in system_call.s
103

```

# 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 -- ダブルフォールト タイプです。放棄；エラーコードがあります。
98 # 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。
99 _double_fault:
100     pushl $_do_double_fault    # addr of C routine pushed onto stack.
100 error_code:
101     xchgl %eax, 4(%esp)      # error code <-> %eax, original eax is in stored stack.
102     xchgl %ebx, (%esp)       # &function <-> %ebx, original ebx is stored in stack.
103     pushl %ecx
104     pushl %edx
105     pushl %edi
106     pushl %esi
107     pushl %ebp
108     push %ds
109     push %es
110     push %fs
111     pushl %eax              # error code
112     lea 44(%esp), %eax      # offset
113     pushl %eax
114     movl $0x10, %eax         # set kernel data segment selector.
115     mov %ax, %ds
116     mov %ax, %es
117     mov %ax, %fs
118     call *%ebx              # indirect invocation to C routine
119     addl $8, %esp            # discard used parameters.
120     pop %fs
121     pop %es
122     pop %ds
123     popl %ebp
124     popl %esi
125     popl %edi
126     popl %edx
127     popl %ecx
128     popl %ebx
129     popl %eax
130     iret
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。
_invalid_TSS:
```



```
132
133     pushl $_do_invalid_TSS
134     jmp error_code
135
# Int11 -- The segment does not present. Type: Error; there is an error code.
# The referenced segment is not in memory. The flag in the segment descriptor indicates
# that the segment is not in memory.
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るようとしたか、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。
```

140 # for the program.

```
141 _stack_segment:
142     pushl $_do_stack_segment
143     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。

144 # 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。

```
145 _general_protection:
146     pushl $_do_general_protection
147     jmp error_code
147
```

# Int17 -- バウンダリ・アライメント・チェック・エラー。

148 # 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。

```
149 _alignment_check:
150     pushl $_do_alignment_check
151     jmp error_code
151
```

# int7 -- \_device\_not\_available in file kernel/sys\_call.s, line 158.
# int14 -- \_page\_fault in file mm/page.s, line 14.
# 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 Information

### 8.2.3.1 Intel reserved interrupt vector definition

Here is a summary of the Intel reserved interrupt vector, as shown in Table 8-1.

	Name	Type	Error Code	Signal	Source

Vector No					
0	Devide error	Fault (Error)	No	SIGFPE	DIV and IDIV instructions.
1	Debug	Fault/ Trap	No	SIGTRAP	Any code or data reference or the INT instruction.
2	nmi	Interrupt	No		Non maskable external interrupt.
3	Breakpoint	Trap	No	SIGTRAP	INT 3 instruction.
4	Overflow	Trap	No	SIGSEGV	INTO instruction.
5	Bounds check	Fault	No	SIGSEGV	BOUND instruction.
6	Invalid Opcode	Fault	No	SIGILL	UD2 instruction or reserved opcode.
7	Device not available	Fault	No	SIGSEGV	Floating-point or WAIT/FWAIT instruction.

8	Double fault	Abort	Yes(0)	SIGSEGV	Any instruction that can generate an exception, NMI, or an INTR.
9	Coprocessor seg overrun	Abort	No	SIGFPE	Floating-point instruction.
10	Invalid TSS	Fault	Yes	SIGSEGV	Task switch or TSS access.
11	Segment not present	Fault	Yes	SIGBUS	Loading segment registers or accessing system segments.
12	Stack segment	Fault	Yes	SIGBUS	Stack operations and SS register loads.
13	General protection	Fault	Yes	SIGSEGV	Any memory reference and other protection checks.
14	Page fault	Fault	Yes	SIGSEGV	Any memory reference.
15	Intel reserved		No		
16	Coprocessor error	Fault	No	SIGFPE	Floating-point or WAIT/FWAIT
17	Alignment check	Fault	Yes(0)		Any data reference in memory.
20-31	Intel reserved.				
32-255	User Defined interrupts	Interrupt			External interrupt or INT n instruction.

## 8.3 traps.c

### 8.3.1 Function Description

The traps.c program mainly includes some C functions used in exception handling, which are used to be called by the exception handling low-level code asm.s. to display debugging message such as error location and error code. The die() generic function is used to display detailed error information in interrupt processing. The final initialization function trap\_init() of the program is called in the previous init/main.c to initialize the hardware exception handling interrupt vector (trap gate) and enable the interrupt request signal to arrive. Please refer to the previous asm.s file when reading this program.

8.3.2 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。

### 8.3.3 Code Annotation

プログラム 8-2 linux/kernel/traps.c

```

1 /*
2 * linux/kernel/traps.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * 'Traps. c' handles hardware traps and faults after we have saved some
9 * state in 'asm. s'. Currently mostly a debugging-aid, will be extended
10 * to mainly kill the offending process (probably by giving it a signal,
11 * but possibly by killing it outright if necessary).
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
//     defined, along with several selector constants.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_structや
//     data of the initial task 0, and some embedded assembly function macro statements
//     about the descriptor parameter settings and acquisition.
// <linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。
//     commonly used functions of the kernel.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
//     descriptors/interrupt gates, etc. is defined.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
//     for segment register operations.
// <asm/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。
//     of a macro's embedded assembler.

13 #include <string.h>
14
15 #include <linux/head.h>
16 #include <linux/sched.h>
17 #include <linux/kernel.h>
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 - セグメントセレクタ、addr - セグメント内に指定されたアドレス。
// Output: %0 - eax (_res); Input: %1 - eax (seg); %2 - memory address (*(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))); \
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 - セグメントセレクタ、addr - セグメント内に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^.. )
29     asm ("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
30         :"=a" ( res):"0" (seg), "m" (*(addr))); \
31     res;}) 33

// Function: fsセグメントレジスタの値(セレクタ)を取得する。
34 // 出力します。%0 - eax ( res ).
```

```
35 #define _fs() ({ \
36 register unsigned short res; \
36 asm ("mov %%fs,%%ax":=a" ( res)); ..^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 void divide_error(void);            // Int0 (kernel/asm.s, 20).
42 void debug(void);                  // int1 (kernel/asm.s, 54).
43 void nmi(void);                   // int2 (kernel/asm.s, 58).
44 void int3(void);                  // int3 (kernel/asm.s, 62).
45 void overflow(void);              // int4 (kernel/asm.s, 66).
46 void bounds(void);                // int5 (kernel/asm.s, 70).
47 void invalid_op(void);             // int6 (kernel/asm.s, 74).
48 void device_not_available(void);   // int7 (kernel/sys_call.s, 158).
49 void double_fault(void);          // int8 (kernel/asm.s, 98).
50 void coprocessor_segment_overrun(void); // int9 (kernel/asm.s, 78).
51 void invalid_TSS(void);           // int10 (kernel/asm.s, 132).
52 void segment_not_present(void);   // int11 (kernel/asm.s, 136).
53 void stack_segment(void);         // int12 (kernel/asm.s, 140).
54 void general_protection(void);   // int13 (kernel/asm.s, 144).
55 void page_fault(void);           // int14 (mm/page.s, 14).
56 void coprocessor_error(void);    // int16 (kernel/sys_call.s, 140).
57 void reserved(void);             // int15 (kernel/asm.s, 82).
58 void parallel_interrupt(void);   // int39 (kernel/sys_call.s, 295).
59 void irq13(void);                // int45 (kernel/asm.s, 86) Coprocessor handling.
60 void alignment_check(void);      // int46 (kernel/asm.s, 148).
61

```

// 本サブルーチンでは、エラーメッセージ、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 - エラーメッセージ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 {

```

65     long * esp = (long *) esp_ptr;
66     int i;
67

```

```

68     printk("%s: %04x|n|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:\t%04x:%p\n -- esp[1] segment selector(CS), esp[0] is EIP;

// (2) EFLAGS:\t%p -- esp[2] is EFLAGS;

// (3) ESP:\t%04x:%p\n -- esp[4] is SS, esp[3] is ESP

```

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->lvt[1]),get_limit(0x17));

```

```

72     if (esp[4] == 0x17) {           // if SS == 0x17, means in user segment,
73         printk("Stack: ");        // print 16 bytes data in user stack too.

```

```
74     for (i=0;i<4;i++)
75         printf ("%p ",get_seg_long(0x17,i+(long *)esp[3]));
```

```

76         printf( "|n" );
77     }
78     str(i); // get task no. (include/linux/sched.h, 210)
79     printf("Pid: %d, process nr: %d|n|r", current->pid, 0xffff & i);
80     for(i=0;i<10;i++)
81         printf("%02x ",0xff & get seg byte(esp[1], (i+(char *)esp[0])));
82     printf( "|n|r");
83     do_exit(11); /* play segment exception */
84 }
85
// 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。
86 void do_double_fault(long esp, long error_code) 87 {
87     die("double fault",esp,error_code);
88 }
89
90
91 void do_general_protection(long esp, long error_code) 92 {
92     die("general protection",esp,error_code);
93 }
94
95
96 void do_alignment_check(long esp, long error_code) 97 {
97     die("alignment check",esp,error_code);
98 }
99
100
101 void do_divide_error(long esp, long error_code) 102 {
102     die("divide error",esp,error_code);
103 }
104
105
// 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値です。
// 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。

```

```

114 // タスクレジスタの取得 TR -> tr
115     printf("eax|t|tebx|t|tecx|t|tedx|n|r%8x|t%8x|t%8x|t%8x|n|r",
116             eax, ebx, ecx, edx);
117     printf("esi|t|tedi|t|tebp|t|tesp|n|r%8x|t%8x|t%8x|t%8x|n|r",
118             esi, edi, ebp, (long) esp);
119     printf("|n|rds|tes|tfss|ttr|n|r%4x|t%4x|t%4x|t%4x|n|r",
120             ds, es, fs, tr);
121     printf("EIP: %8x CS: %4x EFLAGS: %8x|n|r", esp[0], esp[1], esp[2]);
122 }
123
124 void do_nmi(long esp, long error_code)
125 {
126     die("nmi",esp,error_code);

```

```

126 }
127
128 void do_debug(long esp, long error_code)
129 {
130     die("debug", esp, error_code);
131 }
132
133 void do_overflow(long esp, long error_code)
134 {
135     die("overflow", esp, error_code);
136 }
137
138 void do_bounds(long esp, long error_code)
139 {
140     die("bounds", esp, error_code);
141 }
142
143 void do_invalid_op(long esp, long error_code)
144 {
145     die("invalid operand", esp, error_code);_
146 }
147
148 void do_device_not_available(long esp, long error_code) 149 {_
149     die("device not available", esp, error_code);
150 }
151
152
153 void do_coprocessor_segment_overrun(long esp, long error_code) 154
154 {_
155     die("coprocessor segment overrun", esp, error_code);_
156 }
157
158 void do_invalid_TSS(long esp, long error_code) 159
159 {_
160     die("invalid TSS", esp, error_code);
161 }
162
163 void do_segment_not_present(long esp, long error_code)
164 {_
165     die("segment not present", esp, error_code);
166 }
167
168 void do_stack_segment(long esp, long error_code) 169
169 {_
170     die("stack segment", esp, error_code);_
171 }
172
173 void do_coprocessor_error(long esp, long error_code) 174 {_
174     if (last_task_used_math != current)
175         return;
176     die("coprocessor error", esp, error_code);
177 }
178

```

179180 void do\_reserved(long esp, long error\_code) 181{181182 die( "reserved (15, 17-47) error", esp, error\_code);183 }184

// 以下は、その割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

void trap\_init(void)

```

185 {
186     int i;
187
188     set_trap_gate(0, &divide_error);
189     set_trap_gate(1, &debug);
190     set_trap_gate(2, &nmi);
191     set_system_gate(3, &int3);           /* int3-5 can be called from all */
192     set_system_gate(4, &overflow);
193     set_system_gate(5, &bounds);
194     set_trap_gate(6, &invalid_op);
195     set_trap_gate(7, &device_not_available);
196     set_trap_gate(8, &double_fault);
197     set_trap_gate(9, &coprocessor_segment_overrun);
198     set_trap_gate(10, &invalid_TSS);
199     set_trap_gate(11, &segment_not_present);
200     set_trap_gate(12, &stack_segment);
201     set_trap_gate(13, &general_protection);
202     set_trap_gate(14, &page_fault);
203     set_trap_gate(15, &reserved);
204     set_trap_gate(16, &coprocessor_error);
205     set_trap_gate(17, &alignment_check);

// 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
207     for (i=18; i<48; i++)
208         set_trap_gate(i, &reserved);

// 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
// をクリアします。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

```
210     set_trap_gate(45, &irq13);
211     outb_p(inb_p(0x21)&0xfb, 0x21);           // enable IRQ2 of master chip.
212     outb_(inb_p(0xA1)&0xdf, 0xA1);           // enable IRQ13 of slave chip.
213     set_trap_gate(39, &parallel_interrupt); // set parallel 1 gate.
213 }
214
```

## 8.4 sys\_call.s

Linux uses the interrupt invocation method to implement the access interface between the user and the kernel resources. The sys\_call.s program mainly implements the system call INT 0x80 entry processing and signal detection processing, and gives the underlying interfaces of the two system functions, namely sys\_execve and sys\_fork. Interrupt handlers for coprocessor errors (INT 16), device not exist (INT7), clock interrupt (INT32), hard disk interrupt (INT46), floppy disk interrupt (INT38) are also listed.

### 8.4.1 Function descriptions

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する。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ー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。

**8.4.1.1** 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。

#### **8.4.1.2 Interrupt Service Entry Processing**

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「インターフェース」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に入力されたシステム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push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。. jmp \_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push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、システム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。

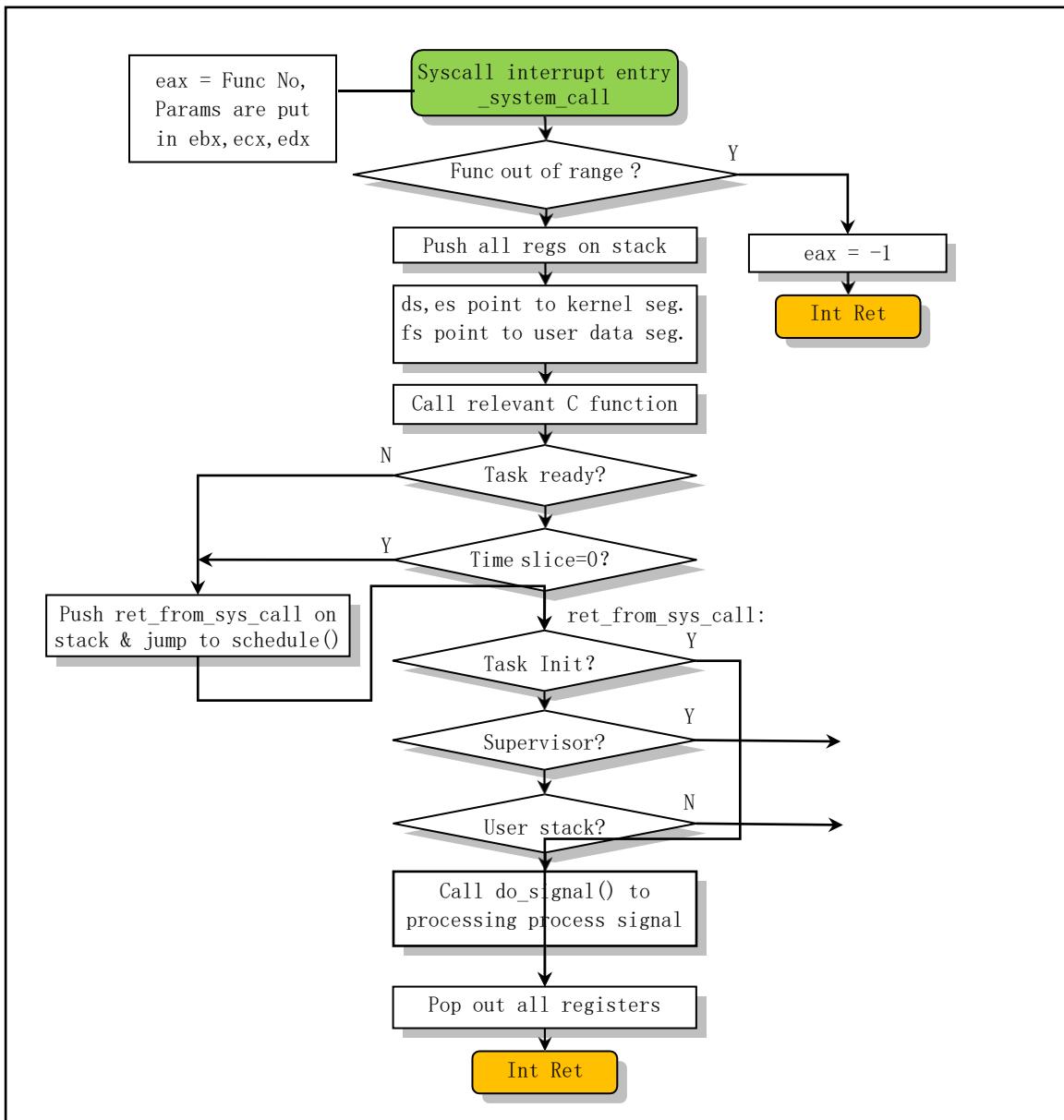


図8-5 システムコール処理フロー図

#### 8.4.1.3 Syscall parameter passing method

Regarding the parameter transfer issue in the system call INT 0x80, the Linux kernel uses several general-purpose registers as a channel for parameter passing. In the Linux 0.12 system, the program uses the registers EBX, ECX, and EDX to pass parameters, and can pass three parameters directly to the system call service procedure (not including syscall function number in the EAX register). If a pointer to a user-space data block is used, the user program can pass more data information to the system call procedure.

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。

8.4.2 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。

### 8.4.3 Code Annotation

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 */
6
7 /*
8 * system_call.s contains the system-call low-level handling routines.
9 * This also contains the timer-interrupt handler, as some of the code is
10 * the same. The hd- and floppy-interrupts are also here.
11 *
12 * NOTE: This code handles signal-recognition, which happens every time
13 * after a timer-interrupt and after each system call. Ordinary interrupts
14 * don't handle signal-recognition, as that would clutter them up totally
15 * unnecessarily.
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) - original %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) - %oldss
32 */
# 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 # 处理されると、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          # signal SIG_CHLD (child stop or end).

```

35

36 EAX

= 0x00

# offset of each register in the stack.

```

37 EBX      = 0x04
38 ECX      = 0x08
39 EDX      = 0x0C
40 ORIG_EAX = 0x10      # If not a syscall (other interrupts), the value is -1
41 FS       = 0x14
42 ES       = 0x18
43 DS       = 0x1C
44 EIP      = 0x20      # Line 44-48 is automatically pushed onto stack by CPU.
45 CS       = 0x24
46 EFLAGS   = 0x28
47 OLDESP   = 0x2C      # old SS:ESP is also pushed when privilege level changed
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          # these are offsets into the task-struct.
51 counter  = 4          # task runtime counts (ticks), time slice.
52 priority = 8          # counter=priority when task starts running, the longer it runs.
53 signal   = 12         # signal bitmap, signal = bit offset + 1
54 sigaction = 16         # MUST be 16 (=len of sigaction)
55 blocked  = (33*16)    # blocked signal offset
56
57 # offsets within sigaction           # see include/signal.h, line 55.
58 sa_handler = 0
59 sa_mask   = 4
60 sa_flags   = 8
61 sa_restorer = 12        # refer to the description of kernel/signal.c
62
63 nr_system_calls = 82     # total number of system calls in Linux 0.12.
64
65 ENOSYS = 38            # system-call number error code
66
67 /*
68 * Ok, I get parallel printer interrupts while using the floppy for some
69 * strange reason. Urgel. Now I just ignore them.
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
75 # 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。
76 .align 2                  # Memory is 4 bytes aligned.
77 bad_sys_call:
78     pushl $-ENOSYS        # set -ENOSYS in eax
79     jmp ret_from_sys_call

80 # 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。
81 # 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。
82 .align 2
83 reschedule:
84     pushl $ret_from_sys_call
85     jmp _schedule

```

```

85 ##### 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)。
86 .align 2
87 _system_call:
88     push %ds          # save original seg registers.
89     push %es
90     push %fs
91     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push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push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。
92 # 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。
93     pushl %edx
94     pushl %ecx        # push %ebx,%ecx,%edx as parameters
95     pushl %ebx        # to the system call

96 # 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をこのシステム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。
97     movl $0x10,%edx      # set up ds,es to kernel space
98     mov %dx,%ds
99     mov %dx,%es
100    movl $0x17,%edx      # fs points to local data space
101    mov %dx,%fs
102    cmpl _NR_syscalls,%eax    # syscall nr is valid ?
103    jae bad_sys_call

104 # 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。
105    call _sys_call_table(%eax,4)  # call C function indirectly.
106    pushl %eax            # the return value.

# 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、#スケジューラー
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:
103    movl _current,%eax      # structure pointer -> eax
104    cmpl $0,state(%eax)    # state
105    jne reschedule
106    cmpl $0,counter(%eax)  # counter

```

107 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 # 处理を行います。例えば、以下の131行目でプロセッサエラー  
# 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

108 これに#をつけることは、task[0]を参照することと同じです。

109 ret\_from\_sys\_call:

```
110     movl _current,%eax
111     cmpl _task,%eax          # task[0] cannot have signals
112     je 3f                  # forward jump to label 3 (line 129), exit
```

# 元々の呼び出しプログラムのコードセグメントセレ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。

113

# ここで、セレクタを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) のセレクタを表します。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レクタが # 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、こちらも終了します。

```
114     cmpw $0x0f,CS(%esp)      # was old code segment supervisor ?
115     jne 3f
116     cmpw $0x17,OLDSS(%esp)    # was stack segment = 0x17 ?
117     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用いて、 # 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

118

# 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。

```
119     movl signal(%eax),%ebx      # signal bitmap -> ebx.
120     movl blocked(%eax),%ecx     # signals blocked -> ecx.
121     notl %ecx
122     andl %ebx,%ecx           # get a bitmap of permissible signals
123     bsfl %ecx,%ecx           # scan the bitmap from bit0, located none zero bit.
124     je 3f                   # exit if none.
125     btrl %ecx,%ebx            # reset the signal.
126     movl %ebx,signal(%eax)      # store the new bitmap -> current->signal
127     incl %ecx                 # adjust signal starting from 1 (1--32).
128     pushl %ecx                # as parameter.
129     call _do_signal            # do_signal() (kernel/signal.c, 128)
130     popl %ecx                 # discard the parameter.
131     testl %eax, %eax           # check return value.
132     jne 2b                   # see if we need to switch tasks, or do more signals
```

```
133 3:    popl %eax          # contains ret code pushed at line 100.  
134      popl %ebx  
135      popl %ecx  
136      popl %edx  
137      addl $4, %esp        # skip orig_eax
```

```

134      pop %fs
135      pop %es
136      pop %ds
137      iret
138

##### 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。
# 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
# 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。
139 # math_error() を実行します。リターン後は、ラベル「ret_from_sys_call」にジャンプして
# 実行を継続します。
140 .align 2
141 _coprocessor_error:
142     push %ds
143     push %es
144     push %fs
145     pushl $-1           # fill in -1 for orig_eax    # not an syscall
146     pushl %edx
147     pushl %ecx
148     pushl %ebx
149     pushl %eax
150     movl $0x10, %eax      # ds, es point to kernel data seg.
151     mov %ax, %ds
152     mov %ax, %es
153     movl $0x17, %eax      # fs point to(user data seg)
154     mov %ax, %fs
155     pushl _ret_from_sys_call
156     jmp _math_error        # math_error() (kernel/math/error.c, 11).
156

##### Int7 -- The device or coprocessor does not exist. Type: Error; no error code.
# If the EM (analog) flag in control register CR0 is set, the interrupt is raised when
# 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
# ートさせるチャンスがある（181行目）。
# CR0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
# for execution (detection and processing of the signal).

```







```
157 .align 2
158 _device_not_available:
159     push %ds
160     push %es
161     push %fs
162     pushl $-1          # fill in -1 for orig_eax
163     pushl %edx
164     pushl %ecx
165     pushl %ebx
166     pushl %eax
167     movl $0x10,%eax    # ds,es to kernel data seg.
168     mov %ax,%ds
169     mov %ax,%es
```

170            movl \$0x17, %eax                # fs to user data seg.  
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すと、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。

172 # ret\_from\_sys\_call はリターン時に実行されます。

```
173         pushl $ret_from_sys_call
174         clts                      # clear TS so that we can use math
175         movl %cr0,%eax
176         testl $0x4,%eax          # EM (math emulation bit)
177         je _math_state_restore

#         pushl %ebp
```

# pushl %ebp

EM フラグが設定されている場合は、数学演算関数 mathemulate() を実行します

```
177          pushl %esi  
178          pushl %edi  
179          pushl $0          # temporary storage for ORIG EIP  
180
```

```
181      call _math_emulate          # (math/math_emulate.c, line 476)
182      addl $4,%esp              # discard temporary data.
183      popl %edi
184      popl %esi
185      popl %ebp
186      ret                      # ret to ret_from_sys_call
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。
.align 2
```

```
188
189 _timer_interrupt:
190     push %ds          # save ds,es and put kernel data space
191     push %es          # into them. %fs is used by _system_call
192     push %fs
193     pushl $-1          # fill in -1 for orig_eax
194     pushl %edx         # we save %eax,%ecx,%edx as gcc doesn't
195     pushl %ecx         # save those across function calls. %ebx
196     pushl %ebx         # is saved as we use that in ret_sys_call
197     pushl %eax
198     movl $0x10,%eax    # ds,es to kernel
199     mov %ax,%ds
200     mov %ax,%es
201     movl $0x17,%eax    # fs to user
202     mov %ax,%fs
203     incl _jiffies
204     movb $0x20,%al      # EOI to interrupt controller #1
205     outb %al,$0x20
```

# システムコールを実行するセレクタ(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。

呼び出し側の

```
# 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
.align 2
```

```
221
222 _sys_fork:
223     call _find_empty_process    # get last_pid (kernel/fork.c, 143)
224     testl %eax,%eax          # pid in eax, if negative then ret.
225     js 1f
226     push %gs
227     pushl %esi
228     pushl %edi
229     pushl %ebp
230     pushl %eax
231     call _copy_process        # copy_process() (kernel/fork.c, 68).
232     addl $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chip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      pushl %eax
hd in
terrupt
:
236
237      pushl %ecx
238      pushl %edx
239      push %ds
240      push %es
241      push %fs
242      movl $0x10,%eax          # ds,es poing to kernel data seg.
243      mov %ax,%ds
244      mov %ax,%es
```

```

245      movl $0x17,%eax          # fs point to user data seg.
246      mov %ax,%fs
247      movb $0x20,%al
248      outb %al,$0xA0          # EOI to interrupt controller #1
249      jmp 1f                  # give port chance 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

251 # 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。

252 1:   xorl %edx,%edx
253      movl %edx,_hd_timeout    # hd_timeout set to 0, controller produces INT in time.
254      xchgl _do_hd,%edx
255      testl %edx,%edx
256      jne 1f                  # if null, point to unexpected_hd_interrupt().
257      movl $_unexpected_hd_interrupt,%edx
258 1:   outb %al,$0x20          # send EOI to 8259A master chip.
259      call *%edx              # "interesting" way of handling intr.
260      pop %fs
261      pop %es
262      pop %ds
263      popl %edx
264      popl %ecx
265      popl %eax
266      iret

267

##### 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。
_floppy_interrupt:

```

```
267      pushl %eax
268      pushl %ecx
269      pushl %edx
270      push %ds
271      push %es
272      push %fs
273
274      movl $0x10, %eax          # ds, es point kernel data seg.
275      mov %ax, %ds
276      mov %ax, %es
277      movl $0x17, %eax          # fs point to user data seg.
278      mov %ax, %fs
279      movb $0x20, %al           # send EOI to 8259A master chip.
280      outb %al, $0x20           # EOI to interrupt controller #1
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:    call *%eax           # "interesting" way of handling intr.
287      pop %fs              # function pointed by do_floppy
288      pop %es
289      pop %ds
290      popl %edx
291      popl %ecx
292      popl %eax
293      iret
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      pushl %eax
296  parallel_interrupt:
297      movb $0x20,%al
298      outb %al,$0x20
299      popl %eax
300      iret

```

## 8.4.4 Reference Information

### 8.4.4.1 32-bit addressing for GNU assembly language

The GNU assembly language uses AT&T's assembly syntax. See Section 3.2.3 for a detailed introduction and examples of this. Here is just an introduction to the addressing method and some examples. The AT&T and Intel assembly language addressing operand formats are as follows:

~~AT&T: disp (base, index\*scale)~~ Intel: デ

Where disp is a optional offset, base is 32-bit base address, index is 32-bit index register, and scale is scale factor (1, 2, 4, 8, default is 1). Although the two assembly language addressing formats are slightly different, the specific addressing locations are actually identical. The addressing position calculation method in the above format is: disp + base + index \* 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、ディスプレーとベースには必ず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	AT&T format	Intel format
Addressing requirements		
Addressing a specified C variable 'booga'	_booga	[_booga]
Addressing the location pointed to by register	(%eax)	[eax]
Addressing a variable by using the contents of the register as the base address	_variable(%eax)	[eax + _variable]
Address a value in an int array (scale value is 4)	_array(,%eax,4)	[eax*4 + _array]
Use direct addressing offset *(p+1), where p is the char's pointer, placed in %eax.	1(%eax)	[eax+1]

Addresses the specified character in an 8-byte array of records. Where eax is the index, and ebx is the	<code>_array(%ebx, %eax, 8)</code>	<code>[ebx + eax * 8 + _array]</code>
--	------------------------------------	---------------------------------------

offset of the specified character in the record.		
--	--	--

#### 8.4.4.2 Adding a System Call

To add a new system-call to your kernel, we should first decide what its exact purpose is. Linux systems do not promote a system call for multiple purposes (except for ioctl() system calls). In addition, we need to determine the parameters, return values, and error code for the new system-call. The interface of the system call should be as simple as possible, so the parameters should be as few as possible. Also, the versatility and portability of system-calls should be considered in design. If we want to add a new system call to Linux 0.12, then we need to do the following things.

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を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)
{
    int i;

    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。
}
```

---

Then add the new system call number and prototype definition in the include/unistd.h file. For example, you can add a function number after line 149 and add a prototype definition after line 279:

---

```
// The new system call number.
#define __NR_sethostname 87

// int sethostname(char *name, int len);
```

Then add the external function declaration in the include/linux/sys.h file and insert the name of the new system call handler at the end of the function pointer table sys\_call\_table, as shown below. Note that the

function names must be arranged in strict order of syscall function numbers.

---

```
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。

---

Then modify line 63 of the sys\_call.s file to increase the total number of nr\_system\_calls by one. At this point you can recompile the kernel. Finally, refer to the implementation of the library function in the lib/ directory to add a new system call library function sethostname() to the libc library.

---

```
#define LIBRARY
#include <unistd.h>

_syscall2(int, sethostname, char *, name, int, len).
```

---

#### 8.4.4.3 Using System-Calls Directly in Assembly File

Below is a simple assembly example asm.s given by Mr. Linus in explaining the relationship and difference between as86 and GNU as. This example shows how to program a stand-alone program in assembly language on a Linux system. That is, it is not necessary to use a start code module (such as crt0.o) and a function in the library. The procedure is as follows:

---

```
.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"

---

There are two system-calls used: 4 - write file operation sys\_write() and 1 - exit program sys\_exit().The C function executed by the write system-call is declared as sys\_write(int fd, char \*buf, int len), see the program fs/read\_write.c, starting at line 83. It comes with 3 parameters. These three parameters are stored in registers EBX, ECX, and EDX before calling the system call. The steps to compile and execute the program are as follows:

---

```
[/usr/root]# as -o asm.o asm.s
[/usr/root]# ld -o asm asm.o
[/usr/root]# ./asm
```

---

## 8.5 mktimes.c

The mktimes.c program is used to calculate the boot time of the kernel-specific UNIX calendar time.

### 8.5.1 Functions

このプログラムには、カーネルでのみ使用される関数kernel\_mktimes()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mktimes()関数と全く同じものです。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 Code Annotation

プログラム 8-4 linux/kernel/mktimes.c

```

1  /*
2  *  linux/kernel/mktimes.c
3  *
4  *  (C) 1991 Linus Torvalds
5  */
6
// 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。
// 7 #include <time.h> 時刻に関する構造といくつかの関数のプロ
トタイプ。7 #include <time.h>
8
9 /*
10 * This isn't the library routine, it is only used in the kernel.
11 * as such, we don't care about years<1970 etc, but assume everything
12 * is ok. Similarly, TZ etc is happily ignored. We just do everything
13 * as easily as possible. Let's find something public for the library
14 * routines (although I think minix times is public).
15 */
16 /*
17 * PS. I hate whoever though up the year 1970 - couldn't they have gotten
18 * a leap-year instead? I also hate Gregorius, pope or no. I'm grumpy.
19 */
20 */
21 #define MINUTE 60           // 1 minute in seconds.
22 #define HOUR (60*MINUTE)   // 1 hour in seconds.
23 #define DAY (24*HOUR)      // 1 day in seconds.
24 #define YEAR (365*DAY)     // 1 year in seconds.
25 /*
26 * interestingly, we assume leap-years */
27 // 各月の初めの開始時間秒数は、年限で定義されています。
28 static int month[12] = {
29     0,
30     DAY*(31),
31     DAY*(31+29),
32     DAY*(31+29+31),
33     DAY*(31+29+31+30),

```

```

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
// 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。
// 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
// init/main.cで、CMOSから情報を取得しています。
41 long kernel_mktime(struct tm * tm) 42
{
44     long res;
45     int year;
45
// まず、1970年からの経過年数を計算します。があるからです。
// 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
// if (tm->tm_year<70) tm->tm_year += 100;
// 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
// 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。
// 1970年からの1年間は、 $1 + (y - 3) / 4$ 、つまり $(y + 1)/4$ となります。
// res = これらの年の秒数 + 各うるう年の秒数 + 秒数
// 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
// month[]配列には、うるう年の日数、つまり
// 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
// 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
// うるう年の検出方法は。 $(y + 2)$ を4で割ることができます、そうでない場合は
// うるう年ではありません。
46 // if (tm->tm_year<70) tm->tm_year += 100;
47     year = tm->tm_year - 70;
48 /* magic offsets (y+1) needed to get leapyears right. */
49     res = YEAR*year + DAY*((year+1)/4);
50     res += month[tm->tm_mon];
51 /* and (y+2) here. If it wasn't a leap-year, we have to adjust */
51     if (tm->tm_mon>1 && ((year+2)%4))
52         res -= DAY;
53     res += DAY*(tm->tm_mday-1);           // nr of days in the past month in secs.
54     res += HOUR*tm->tm_hour;             // the past hours of the day in secs.
55     res += MINUTE*tm->tm_min;            // the past minutes of the hour in secs.
56     res += tm->tm_sec;                  // nr of secs that have passed in 1 minute.
57     return res;                         // the nr of secs elapsed since 1970.
58 }
59

```

### 8.5.3 Information

### 8.5.3.1 Calculation method for leap year

The basic calculation method for leap years is:

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 Function description

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。

- 8.6.1.1** 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。

#### 8.6.1.2 Schedule function

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。

- 8.6.1.3** 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。

#### 8.6.1.4 Sleep and wake-up functions

この2つの関数は非常に短いのですが、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。

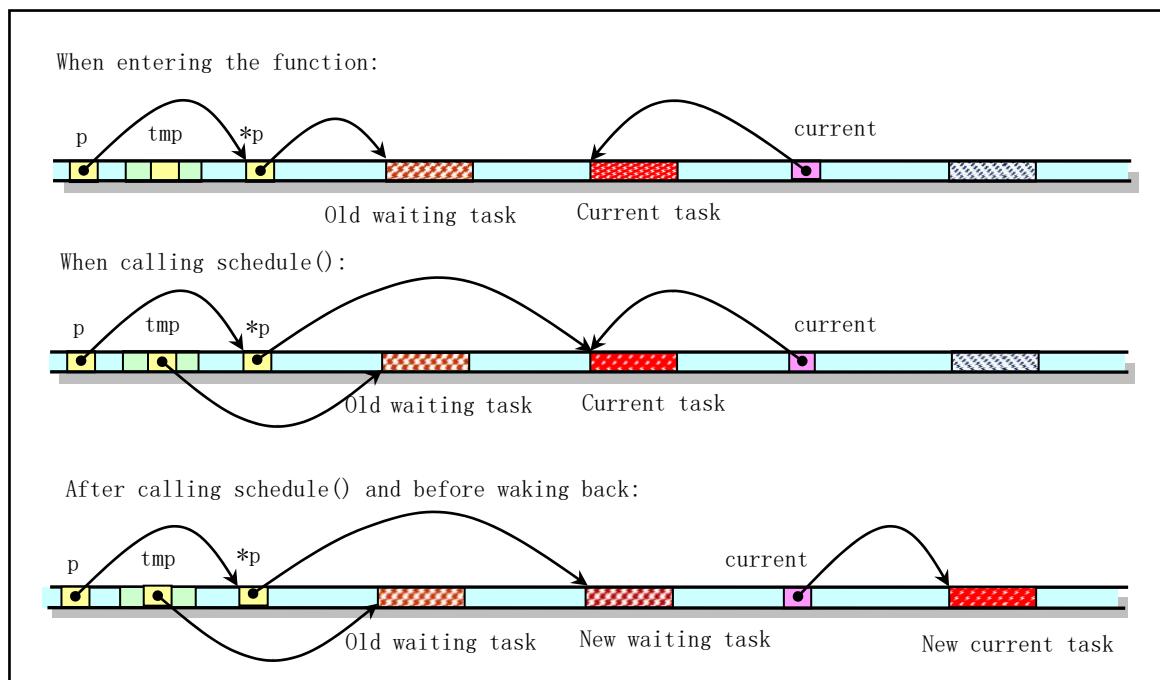


図8-6 sleep\_on()におけるポインタの変化の模式図。

When entering the function, the queue head pointer `*p` points to the task structure (process descriptor) that has been waiting in the wait queue. Of course, there is no waiting task on the wait queue when the system first starts executing. Therefore, the original waiting task in the above figure does not exist at the beginning, and `*p` points to NULL.

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、キューへ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が新しい現在のタスクに向かされ、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へッドポインタが指す元の待ちタスクを指し、キューへ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。

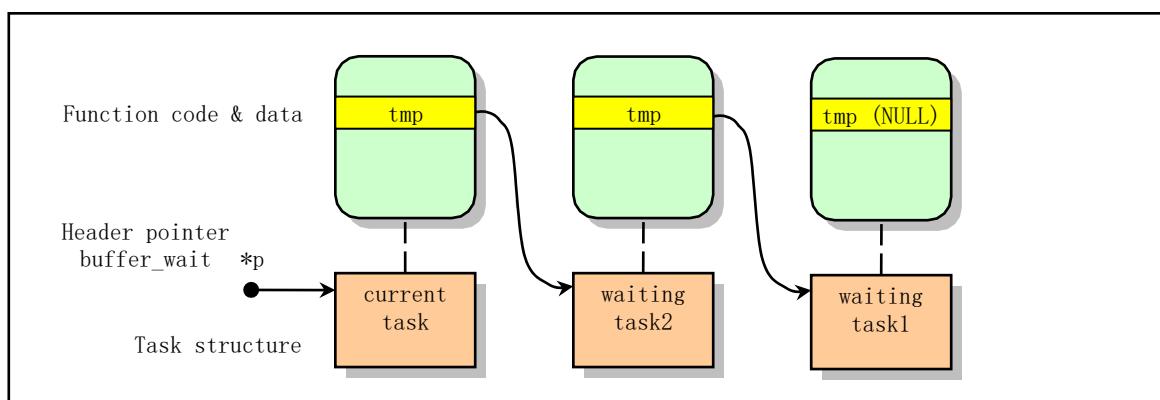


図8-7 sleep\_on()の暗黙のタスク待ち行列の様子

After inserting the process into the wait queue, the sleep\_on() function calls the schedule() function to execute another process. When the process is awakened and re-executed, the subsequent statements are executed, and a process that enters the waiting queue earlier than it wakes up. Note that the so-called wake-up here does not mean that the process is in the execution state, but in the ready state that can be scheduled to execute.

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つの関数が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 Code Annotation

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' is the main kernel file. It contains scheduling primitives
9 * (sleep_on, wakeup, schedule etc) as well as a number of simple system
10 * call functions (type getpid(), which just extracts a field from
11 * current-task
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
//      data of the initial task 0, and some embedded assembly function macro statements
//      about the descriptor parameter settings and acquisition.
// <linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。
//      commonly used functions of the kernel.
// <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

```

```
//      handlers, starting with 'sys_'.  
// <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。  
//      parameters.  
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。  
//      descriptors/interrupt gates, etc. is defined.  
// <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。  
//      of a macro's embedded assembler.
```

```

// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
//      for segment register operations.
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、および
//      signal manipulation function prototypes.
13 #include <linux/sched.h>
14 #include <linux/kernel.h>
15 #include <linux/sys.h>
16 #include <linux/fdreg.h>
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 - task no;      p - task structure pointer.
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_ptr->pid, p->p_cptr ? p->p_cptr->pid : -1); 32
32     i=0;
33     while (i<j && !((char *) (p+1))[i]) // Detects nr of zero bytes after task struct.
34         i++;
35     printk("%d/%d chars free in kstack|n|r", 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|r",
39                 p->p_ysptr ? p->p_ysptr->pid : -1,
40                 p->p_osptr ? p->p_osptr->pid : -1);
41     else
42         printk("|n|r");
43 }
44
// 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     printf( "|rTask->info: |n|r" );
50     for ( i=0; i<NR_TASKS; i++ )
51         if ( task[i] )
52             show_task( i, task[i] );
53 }
54
// 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
// 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
// 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
// ください。
// ライン438。
55 #define LATCH (1193180/HZ)
56
57 extern void mem_use(void);           // [??] not defined anywhere.
58
59 extern int timer_interrupt(void); // kernel/system_call.s, 189
60 extern int system_call(void);    // kernel/system_call.s, 84
61
// 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
す。
// 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
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。
// 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。
62 // 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）。
63 union task_union {
64     struct task_struct task;
65     char stack[PAGE_SIZE];
66 };
67 static union task_union init_task = {INIT_TASK}; 68
// 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
// チップの割り込みが発生します。
// 「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。
// 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。
// 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
// 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。
// 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
// 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
// この時点での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。
// 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。
// 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
// ジフティーを最適化するのではなく、場所を移動するのではなく、記憶からその値を取ること。
// 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

```

```
69 // その値。
70 unsigned long volatile jiffies=0;          // kernel pulse (ticks)
71 unsigned long startup_time=0;           // total seconds from 1970:0:0:0
72 int jiffies_offset = 0;           /* # clock ticks to add to get "true
73                                time". Should always be less than
74                                1 second's worth. For time fanatics
75                                who like to syncronize their machines
76                                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
// as the user mode stack for task 0. It is the kernel stack before running task 0 and is
// 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レクタ（0x10）に設定され、ESPは、データセグメントセレ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

84 82 long user_stack [ PAGE_SIZE>>2 ] ;
83
85 struct {
86     long * a;
87     short b;
88 } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
88 /*
89 * 'math_state_restore()' saves the current math information in the
90 * old math state array, and gets the new ones from the current task
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しなければなりません。
    // 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
95     if (last_task_used_math == current)
96         return;
97     __asm__ ("fwait");
98     if (last_task_used_math) {
99         __asm__ ("fnsave %0": "m" (last_task_used_math->tss.i387));
    }
    // 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。
100
101     // 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。

```

```
102     if (current->used_math) {  
103         __asm__("frstor %0": :"m" (current->tss.i387));  
104     } else {  
104         __asm__("fninit");           // send initial cmd to the math.  
105         current->used_math=1;      // set used math flag.  
106     }
```

```

107 }
108
110 /*  

111 * 'schedule()' is the scheduler function. This is GOOD CODE! There  

112 * probably won't be any reason to change this, as it should work well  

113 * in all circumstances (ie gives I/O-bound processes good response etc).  

114 * The one thing you might take a look at is the signal-handler code here.  

115 */  

116 * NOTE!! Task 0 is the 'idle' task, which gets called when no other  

117 * tasks can run. It can not be killed, and it cannot sleep. The 'state'  

118 * information in task[0] is never used.  

119 */  

120 void schedule(void)  

121 {  

122     int i, next, c;  

123     struct task_struct ** p;           // pointer's pointer of task struct.  

124 /* check alarm, wake up any interruptible tasks that have got a signal */  

125  

// 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  

// ループするとき。  

126     for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)  

127         if (*p) {  

// 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  

// 0で、タスクがTASK_INTERRUPTIBLEのスリープ状態であれば、readyにする  

// 状態 (TASK_RUNNING) です。  

128             if ((*p)->timeout && (*p)->timeout < jiffies) {  

129                 (*p)->timeout = 0;  

130                 if ((*p)->state == TASK_INTERRUPTIBLE)  

131                     (*p)->state = TASK_RUNNING;  

132             }  

// 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  

// (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SIGALRM  

// 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  

// タスクを終了させるためのシグナルです。  

133             if ((*p)->alarm && (*p)->alarm < jiffies) {  

134                 (*p)->signal |= (1<<(SIGALRM-1));  

135                 (*p)->alarm = 0;  

136             }  

// 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  

// タスクが割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。  

// '~(_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  

// とSIGSTOP信号をブロックすることはできません。  

137             if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&  

138                 (*p)->state==TASK_INTERRUPTIBLE)  

139                 (*p)->state=TASK_RUNNING;    // set ready.  

140             }  

141 /* this is the scheduler proper: */  

142
143

```

```
144      while (1) {
145          c = -1;
146          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。のカウンター（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。
150             while (--i) {
150                 if (!*--p)
151                     continue;
152                 if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
153                     c = (*p)->counter, next = i;
154             }
// 比較の結果、カウンターの値が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。
// loop (144 lines), execute the latter task switching macro(line 161). Otherwise, the
// 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ています。カウンタの値は次のように計算されます。
//     counter = counter / 2 + priority
// ここで計算過程では、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。
156             if (c) break;
157             for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
158                 if (*p)
159                     (*p)->counter = ((*p)->counter >> 1) +
160                         (*p)->priority;
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 {
167     current->state = TASK_INTERRUPTIBLE;
168     schedule();
169     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 172    sleep_on(struct task_struct **p, int state)
{
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         return;
180     if (current == &(init_task.task))
181         panic("task[0] trying to sleep");
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 = current;
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     // the task is inserted. Therefore, we should also awaken these subsequent tasks entered
198     // 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
199     // 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
200     // 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
201     // 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
202     // schedule()関数。
203
204     if (*p && *p != current) {
205         (**p).state = 0;
206         current->state = TASK_UNINTERRUPTIBLE;
207         goto repeat;
208     }
209
210     // 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
211     // キューへッドは、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
212     // 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
213     // 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
214     // 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。
215     // 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
216     // ウェイクアップ後に実行されるときは、ヘッダをNULLにする。
217
218     if (!*p)
219         printk("Warning: *P = NULL\n");

```

```
190         if (*p = tmp)
191             tmp->state=0;
192 }
193 // 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
// 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 shouldn't be in the kernel
217 * proper. They are here because the floppy needs a timer, and this
218 * was the easiest way of doing it.
219 */
220
// 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。
// フロッピードライバ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。
// で通常の速度で起動します。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。
222 //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};
223 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。
224      // 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。
225 unsigned char current_DOR = 0x0C;
226
227      // 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。
228      // Parameter nr is floppy drive number (0--3), the return value is the nr of ticks.
229      // The variable selected is the selected floppy drive flag (blk_drv/floppy.c, line 123).
230      // The mask is the start motor bits in the selected floppy drive DOR, and the upper 4 bits
231      // are the floppy drive start motor flags.
232
233 int ticks_to_floppy_on(unsigned int nr)
234 {
235     extern unsigned char selected;
236     unsigned char mask = 0x10 << nr;
237
238     // The system has up to 4 floppy drives. First, set the time (100 seconds) that the
239     // specified floppy drive nr needs to stop before it stops. Then take the current DOR
240     // value into variable mask, and set the motor start flag of the specified floppy drive
241     // in it.
242
243     if (nr>3)
244         panic("floppy_on: nr>3");
245     moff_timer[nr]=10000;           /* 100 s = very big :-) */
246     cli();                         /* use floppy_off to turn it off */
247     mask |= current_DOR;
248
249     // If the floppy drive is not currently selected, first reset the selection bits of other
250     // floppy drive, then set the floppy drive selection bit.
251
252     if (!selected) {
253         mask &= 0xFC;
254         mask |= nr;
255     }
256
257     // If the current value of DOR is different from the required value, a new value (mask) is
258     // output to the FDC digital output port, and if the motor that is required to start is
259     // not yet started, the motor start timer value of the corresponding floppy drive is set
260     // (HZ/2 = 0.5) Seconds or 50 ticks). If it has been started, set the startup timing to 2
261     // ticks, which can meet the requirements of the following depreciation in do_floppy_timer().
262
263     // The current digital output register current_DOR is updated thereafter.
264
265     if (mask != current_DOR) {
266         outb(mask, FD_DOR);
267         if ((mask ^ current_DOR) & 0xf0)
268             mon_timer[nr] = HZ/2;
269         else if (mon_timer[nr] < 2)
270             mon_timer[nr] = 2;
271         current_DOR = mask;
272     }
273
274     sti();                         // enable int.
275     return mon_timer[nr];           // return time value required to start motor.
276 }
277
278 // wait for a period of time required to start the floppy drive motor.
279 // Sets the delay time required for the motor of the specified floppy drive from start to
280 // 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
281 // 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、待機中のキューを入れる

253 // モーターを動作させます。そして、割り込みを開きます。

254 cli();

255 while (\_ticks\_to\_floppy\_on(nr))

256 sleep\_on(nr+wait\_motor);

257 sti();

258 }

259 // 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になるとOFFになります。

260 void floppy\_off(unsigned int nr) 261

{.

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、使用しているフロッピードライブを1つずつ確

認します。

// one. Skip if it is not the motor specified by DOR. If the motor start timer expires,

// プロセスを起こします。モーターOFF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 continue;

272 if (mon\_timer[i]) {

273 if (!--mon\_timer[i]) // if motor on timer expires

274 wake\_up(i+wait\_motor); // wake up the process.

275 } else if (!moff\_timer[i]) {

276 current\_DOR &= ~mask; // reset motor start bit

277 outb(current\_DOR, FD\_DOR); // update DOR.

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  
// カーネルのみでの使用を目的としています。
```

283 #define TIME\_REQUESTS 64

284

```

285     long jiffies;           // Timer ticks.
286
287     static struct timer_list {
288         void (*fn)();           // Timer handler.
289         struct timer_list * next; // points to the next timer.
290     } timer_list[TIME REQUESTS], * next_timer = NULL; // next_timer is timer queue head.
291
292     // 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
293     // 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
294     // モーターの起動や停止を行う // 操作です。
295     // jiffies - 時限数; *fn() - 時間切れの際に実行される関数。
296
297     void add_timer(long jiffies, void (*fn)(void)) {
298         struct timer_list * p;
299
300         // 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。
301         if (!fn)
302             return;
303         cli();
304
305         // 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
306         // リンクリストに追加されます。
307         if (jiffies <= 0)
308             (fn)();
309         else {
310
311             // 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。
312             for (p = timer_list ; p < timer_list + TIME REQUESTS ; p++)
313                 if (!p->fn)
314                     break;
315
316             // 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
317             // 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。
318             if (p >= timer_list + TIME REQUESTS)
319                 panic("No more time requests free");
320             p->fn = fn;
321             p->jiffies = jiffies;
322             p->next = next_timer;
323             next_timer = p;
324
325             // 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。
326             // 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
327             // 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。
328             while (p->next && p->next->jiffies < p->jiffies) {
329                 p->jiffies -= p->next->jiffies;
330                 fn = p->fn;
331                 p->fn = p->next->fn;
332                 p->next->fn = fn;
333                 jiffies = p->jiffies;
334                 p->jiffies = p->next->jiffies;
335                 p->next->jiffies = jiffies;
336                 p = p->next;
337             }
338         }
339     }

```

```
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)にあります。パラメータcplは現在の特権  
// 実行されているコードセレ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。

328

// 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。

329        if (blankcount || !blankinterval) {

330            if (blanked)

331                unblank\_screen();332                if (blankcount)333                    blankcount--;

334                blanked = 0;

// Otherwise, if the black screen flag is not set, the screen will be blank and the flag  
// will be set.

334        } else if (!blanked) {

335            blank\_screen();

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

338

// 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。

339        if (hd\_timeout)340            if (!--hd\_timeout)341                hd\_times\_out();                    // blk\_drv/hdc, line 318

341

// ビープ音のカウント数に達したら、ビープ音をオフにする。（ポート0x61にcmdを送信し、リセット

//        if (beepcount)                            // beep ticks (chr\_drv/console.c, 950)

ビ

ツ

ト

0

は

82

53

チ

ツ

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
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。
346 // 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。
347     if (cpl)
348         current->utime++;
349     else
350         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
351 //がnullに設定され、その後、タイマーが削除されます。
352     if (next_timer) {                  // timer list header.
353         next_timer->jiffies--;
354         while (next_timer && next_timer->jiffies <= 0) {

```

```

355         void (*fn)(void);           // a function pointer definition.
355
356         fn = next_timer->fn;
357         next_timer->fn = NULL;
358         next_timer = next_timer->next;
359         (fn)();                  // call the timer handler.
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。
362
// 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。
363     if (current_DOR & 0xf0)
364         do_floppy_timer();
365
// 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トは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
366
// 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。
367     if ((--current->counter)>0) return;
368     current->counter=0;
369     if (!cpl) return;           // kernel code
370     schedule();
371 }

// 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。
372 int sys_alarm(long seconds)
373 {
374     int old = current->alarm;
375
376     if (old)
377         old = (old - jiffies) / HZ;
378     current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
379     return (old);
380 }

// 現在のプロセスのpidを取得します。
381 int sys_getpid(void)
382 {
383     return current->pid;
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
// 有効なユーザーID - euid を取得します。
395 int sys_geteuid(void)
396 {
397     return current->euid;
398 }
399
// グループIDの取得 - gid
400 int sys_getgid(void)
401 {
402     return current->gid;
403 }
404
// 有効なグループID - egidを取得します。
405 int sys_getegid(void)
406 {
407     return current->egid;
408 }
409
// 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？）
// 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。
410 int sys_nice(long increment)
411 {
412     if (current->priority-increment>0)
413         current->priority -= increment;
414     return 0;
415 }
416
// カーネルスケジューラの初期化サブルーチンです。
417 void sched_init(void)
418 {
419     int i;
420     struct desc_struct * p;           // descriptor structure pointer
421
// 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
// 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
// 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
// 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、純粹に自分自身を戒めるためと
// カーネルのコードを変更する人。
422     if (sizeof(struct sigaction) != 16)      // signal struct
423         panic("Struct sigaction MUST be 16 bytes");
// 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
// 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。

```

```
// 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。  
// gdtは記述子の配列(linux/head.h)であり、これには  
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  
// 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。  
// asm/system.hの65行目を参照してください。
```

424        set\_tss\_desc(gdt+FIRST\_TSS\_ENTRY,&(init\_task.task.tss));

```

425     set_ldt_desc(gdt+FIRST_LDT_ENTRY, &(init_task.task.ldt));
    // Clear the task array and descriptor table entries (note that starting with i=1, so the
    // descriptor for the initial task is still there).
426     p = gdt+2+FIRST_TSS_ENTRY;
427     for(i=1;i<NR_TASKS;i++) {
428         task[i] = NULL;
429         p->a=p->b=0;
430         p++;
431         p->a=p->b=0;
432         p++;
433     }
434 /* Clear NT, so that we won't have troubles with that later on */
    // The NT flag in EFLAGS is used to control nested calls to tasks. When NT is set, the
    // current interrupt task will cause a task switch when the IRET instruction is executed.
    // NT indicates whether the back_link field in the TSS is valid. Invalid when NT=0.
435     __asm__ ("pushfl ; andl $0xfffffbfff, (%esp) ; popfl"); // reset NT

    // タスク0のTSSセグメントセレクタがタスクレジスタ(TR)にロードされる。また、LDT
    // セグメントセレクタは、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//セレクタが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 is task 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); /* binary, mode 3, LSB/MSB, ch 0*/
439     outb_p(LATCH & 0xff, 0x40); /* LSB */
440     outb(LATCH >> 8, 0x40); /* MSB */

    // 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); // change int mask, enable timer.
443     set_system_gate(0x80, &system_call);
444 }
445

```

## 8.6.3 Information

### 8.6.3.1 Floppy Drive Controller

For the application in the above program, only the I/O port used by the floppy disk controller (FDC) is briefly introduced here. For a detailed description of FDC programming, see the explanations in Chapter 9 after floppy.c. Four ports need to be accessed when programming the FDC. These ports correspond to one or more registers on the controller. For a normal floppy disk controller there are some ports shown in Table 8-3.

表8-3 フロッピーディスクコントローラのポート

I/O port	Port name	Read/Write	Register name

---

0x3f	FD_DOR	Write only	Digital output register (digiter controller register)
2	FD_STATUS	Read only	FDC main status register
	FD_DATA	Read/Write	FDCデータレジスタ
0x3f			
4			
0x3f			
5			
0x3f7	FD_DIR	Read only	Digital input register
0x3f7	FD_DCR	Write only	Drive control register (transfer rate control)

---

The digital output register DOR (or digital control) is an 8-bit register that controls driver motor turn-on, driver select, start/reset FDC, and enable/disable DMA and interrupt requests.

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) でなければならず、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ーは、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、その後に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にあり、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知る必要があります。

### 8.6.3.2 Programmable Timer/Counter Controller

#### 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ネルを備えています。各チャ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。

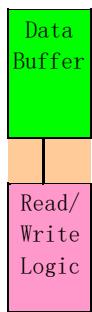


図 8-8 8253(8254) タイマ/カウンタチップの内部構造

The 3-state, bidirectional 8-bit Data Bus Buffer is used to interface with the system data bus. Read/Write Logic is used to receive input signals from the system bus and generate control signals that are input to other parts. Address lines A1, A0 are used to select one of the three counter channels or Control Word Registers that need to be read/written. Usually they are connected to the A0, A1 address line of the system. The read and write pins RD, WR and chip select pin CS are used by the CPU to control the read and write operations of the 8253 chip. The Control Word Register is used by the CPU to set how the specified counter works. It is a write-only register. But for the 8254 chip, you can use the Read-Back Command to read the status information. The three independent counter channels function exactly the same, each of which can work in different ways. The Control Word register will determine how each counter works. The CLK pin of each counter is connected to the clock frequency generator (crystal oscillator). The 8253 has a clock input frequency of up to 2.6MHz, while the 8254 can be up to 10MHz. Pin GATE is the gate control input of the counter, which is used to control the start and stop of the counter and the output state of the counter. Pin OUT is the output signal terminal of the counter.

図8-8(b)は、カウンターチャンネルの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。実際のカウンターは、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ーである。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の値は読みません。カ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つのレジスターは、カウンター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) chip programming

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。

	Name	Description

Bit		
7	SC1	SC1, SC0 are used to select counter channel 0-2, or to read back the command.
6	SC0	00 - Channel 0; 01 - Channel 1; 02 - Channel 2; 11 - Readback command (only in 8254).
5	RW1	RW1 and RW0 are used for counter read/write operation selection.
4	RW0	00 - indicates a register latch command; 01 - Reads/writes the low byte (LSB); 10 - Read/write high byte (MSB); 11 - Read/write low byte first, then high byte.
3	M2	M2-M0 is used to select the working mode of the specified channel.
2	M1	000 - mode 0; 001 - mode 1; 010 - mode 2;
1	M0	011 - mode 3; 100 - mode 4; 101 - mode 5.
0	BCD	Count value format selection. 0 - 16 bit binary count; 1 - 4 BCD code counts.

When the CPU performs a write operation, if the A1 and A0 lines are 11 (in this case, the corresponding port 0x43 on the PC microcomputer), the control word is written into the control word register. The contents of the control word specify the counter channel being programmed. The initial count value is written to the specified counter. When A1 and A0 are 00, 01, and 10 (corresponding to PC ports 0x40, 0x41, and 0x42, respectively), one of the three counters is selected. In a write operation, the control word must be written first and then the initial count value. The initial count value must be written in the format set in the control word (binary or BCD code format). When the counter starts working, we can still rewrite the new initial value to the specified counter at any time. This does not affect how the counters that have been set work.

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. Counter working mode

- (1) 8253/8254の3つのカウンター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。
  - (2) Mode 0 - Interrupt on terminal count
  - (3) 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のままであります。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。
  - (4) Mode 1 - Hardware Retriggerable One-shot
  - (5) 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。
  - (6) Mode 2 - Rate Generator
  - (7) 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。
  - (8) Mode 3 - Square Wave Mode
  - (9) 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。
  - (10) Mode 4 - Software Triggered Strobe
  - (11) 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（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。
  - (12) Mode 5 - Hardware Triggered Strobe
- 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ーチャネル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。

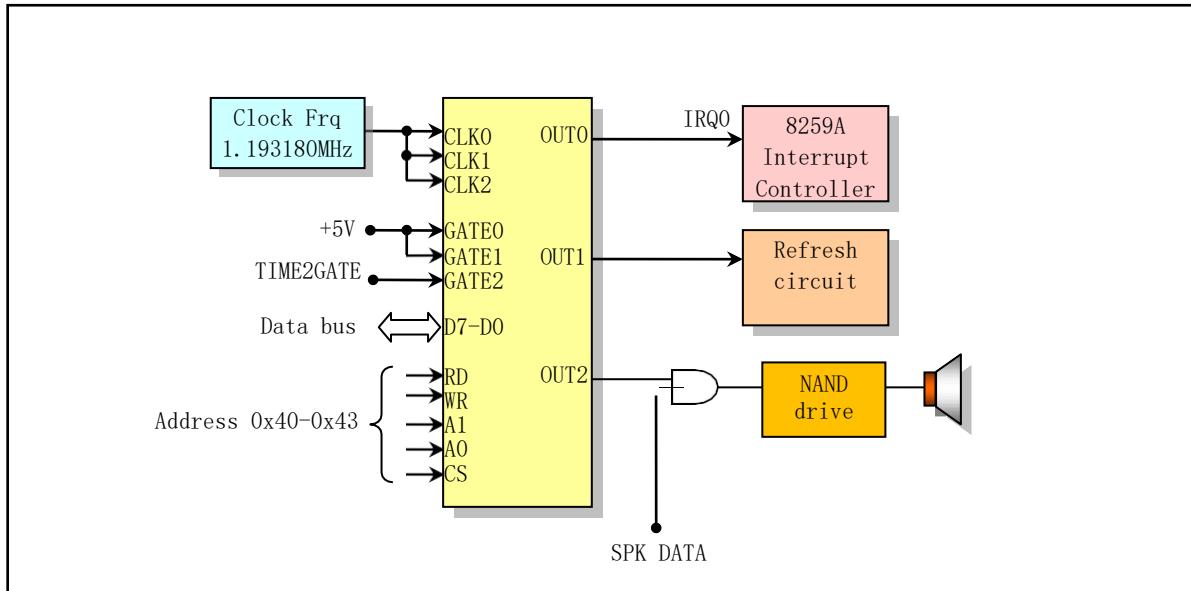


図8-9 PC内のタイマ/カウンタチップ接続図

For counter channel 0, its GATE pin is fixed high. When the system is powered on, it is set to work in mode 3 (square wave generator mode), and the initial count value is set to 0 by default, which means that the count value is 65536 (0--65535). Therefore, the OUT0 pin emits a square wave signal with a frequency of 18.2HZ ( $1.193180\text{MHz}/65536$ ) every second. OUT0 is connected to the level 0 interrupt requester of the programmable interrupt controller 8259 chip. Therefore, using the rising edge of the square wave can trigger an interrupt request, causing the system to issue an interrupt request every 54.9ms (1000ms/18.2).

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マイクロ秒ごとに送出され、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。このカウンターチャンネルは、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ン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、カーネルのタイミング操作を実現するために使用されます。

## 8.7 signal.c

### 8.7.1 Function Description

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つ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、正の整数であり、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、SIGCHLD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き、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。

**8.7.1.1** 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。

#### 8.7.1.2 Signals in Linux

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すために使用されるため、システムには最大32種類の信号が存在することになります。このバージョンの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。

1. 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。
2. Ignore this signal. Most signals can be ignored by the process. But there are two signals that can't be

ignored: SIGKILL and SIGSTOP. The reason is to give the superuser a definite way to terminate or stop any process specified. In addition, if the signal generated by some hardware exceptions is ignored (for example, divided by 0), the behavior or state of the process may become agnostic.

3. Capture the signal. In order to perform the capture operation, we must first tell the kernel to call our custom signal handler when the specified signal occurs. In this handler we can do anything. Of course, you can do nothing and play the same role of ignoring the signal. An example of a custom signal processing function is: If we create some temporary files during program execution, then we can define a function to capture the SIGTERM (terminate execution) signal and do some cleanup in the function. The SIGTERM signal is the default signal sent by the kill command.
4. Perform the default action. The process does not process the signal, and the signal is processed by the system's corresponding default signal handler. The kernel provides a default action for each type of signal. Usually these default actions are to terminate the execution of the process. See the description in the

### 8.7.1.3 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。

#### 8.7.1.4 Signal Processing Implementation

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。

1. 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（シグナルハンドル）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。ユーザー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、ユーザー定義のシグナル処理サービスプログラムを実行します。

#### 2. signal() function

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).
```

---

This signal() function takes two arguments. One is to specify the signal signr to be captured; the other is the new signal handler pointer void (\*handler)(int). This new signal handler is a function pointer with no return value and an integer parameter that is passed to the handler when the specified signal occurs.

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);
```

---

Then we can rewrite the prototype of the signal() function to the following simple form:

---

```
sigfunc *signal(int signr, sigfunc *handler);
```

---

The signal() function installs a new signal handler for the signal signr. The signal handler can be a signal processing function specified by the user, or it can be a specific function pointer SIG\_IGN or SIG\_DFL provided by the kernel. When the specified signal arrives, the signal is ignored if the associated signal processing handler is set to SIG\_IGN. If the signal handler is SIG\_DFL, then the default operation of the signal is performed. Otherwise, if the signal handler is set to a user's signal handler, the kernel first resets the signal handle to its default handle, performs an implementation-related signal blocking operation, and then invokes the specified signal handler.

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
```

---

They all represent function pointers with no return value, respectively, as required by the second parameter in the signal() function. The pointer values are 0 and 1, respectively. These two pointer values are logically the function address values that are not possible in the actual program. Therefore, in the signal() function, it is possible to judge whether to use the default signal processing handle or ignore the processing of the signal based on the two special pointer values. Of course, SIGKILL and SIGSTOP cannot be ignored. See the processing statements on lines 155-162 in the program listing below for related code.

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方法や処理方法は、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); // re-install the handler for the next capture.
    ...
}

メイン()
{
    signal(SIGINT, sig_handler); // set signal handler in main.
    ...
}
```

---

The reason that the signal() function is unreliable is that when the signal has already occurred and enters its own set of signal processing functions, it is possible that another signal will occur during this time before re-setting its own handler. But at this point the system has set the handle to the default value. Therefore, it is possible to cause signal loss.

### 3. sigaction() function

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);
```

---

The parameter sig is the signal we need to view or modify the signal processing handler, and the last two

parameters are pointers to the sigaction structure. When the parameter act pointer is not NULL, the behavior of

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);           // signal handler.
50     sigset_t sa_mask;                // signal mask.
51     int sa_flags;                    // flags for the handler.
52     void (*sa_restorer) (void);       // internal restorer pointer.
53 };

```

---

When modifying a signal processing method, if the processing handler sa\_handler is not the default SIG\_DFL or the ignore handler SIG\_IGN, then before the sa\_handler can be called, the sa\_mask field specifies one signal set that needs to be added to the process signal mask bitmap. If the signal handler returns, the system will restore the original signal mask bitmap of the process. This way we can block some of the specified signals when a signal handler is called. When the signal handler is called, the new signal mask bitmap automatically includes the currently transmitted signal, blocking the continued transmission of the signal. Thus, we can ensure that the same signal is blocked without being lost during the processing of a specified signal until the processing is completed. In addition, when a signal is blocked and occurs multiple times, usually only one sample is saved, that is, when the blocking is released, the signal processing handler is called again only once for the same multiple signals that are blocked. After we modify the processing handler of a signal, the handler is used until it is changed again. This is not the same as the traditional signal() function. The signal() function will restore it to the default handler of the signal after the handler processing has finished.

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           // ignore SIGCHLD if child stopped.
38 #define SA_INTERRUPT   0x20000000 // not restart syscall after interrupt by sig.
39 #define SA_NOMASK      0x40000000 // do not mask the current signal.
40 #define SA_ONESHOT     0x80000000 // restore to defulat handler after finished.

```

---

4. 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。

##### 5. do\_signal() function

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。

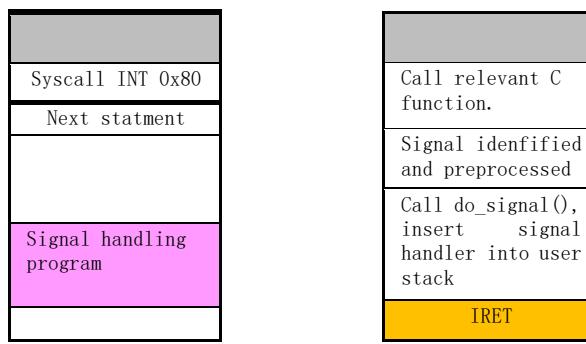


図8-10 シグナルハンドラーの呼び出され方

Before inserting the parameters of the signal handler into the user stack, the `do_signal()` function first expands the user program stack pointer down longs (see line 195 in the program below) and then adds the relevant parameters to it. See Figure 8-11. Since the code starting with line 193 of the `do_signal()` function is hard to understand, we will describe it in detail below.

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の左側のようになります。つまり、`do_signal()`の引数はカーネル・ストート・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

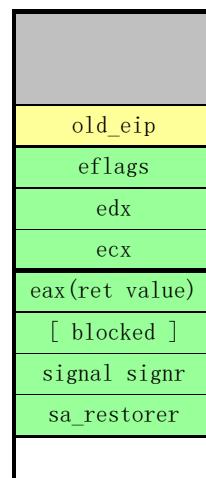


図8-12 ユーザー・ステート・スタックを修正する具体的なプロセス

After `do_signal()` determines and processes the two default signal handlers (`SIG_IGN` and `SIG_DFL`), if the user customizes the signal handler, then from the 193th line, `do_signal()` starts to prepare inserting the user-defined handler into the user state stack. It first saves the return execution point pointer `eip` of the original user program in the kernel state stack as `old_eip` in user stack, and then replaces the `eip` with the custom handler `sa_handler`, that is, the `eip` in the kernel state stack in the figure points to `sa_handler`. Next, the user state stack is extended downward by 7 or 8 long words by subtracting the longs value from the "original esp" saved in the kernel state. Finally, some of the register contents on the kernel stack are copied into this space, as shown in the right side of Figure 8-12.

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、これらのユーザー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。ユーザーが`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) // user defined signal handler.
```

---

```

{
    printf("The signal is %d\n", sig)と表示されます。
    (void) signal(SIGINT, SIG_DFL);           // restore default handler of SIGINT
}

int main()
{
    (void) signal(SIGINT, handler);           // set user defined handler for SIGINT.
    while (1) {
        printf("Signal test.n").
        sleep(1);                            // wait a second.
    }
}

```

---

Among them, the signal handler function () will be called when the signal SIGINT appears, and then return to the main program to continue execution. The function first outputs a message and then sets the processing of the SIGINT signal to the default signal handler. So when you press the Ctrl-C key combination a second time, SIG\_DFL will let the program finish running.

## 6. sa\_restorer function

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      # discard the signal signr
    popl %eax         # restore system-call ret value in eax
    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      # discard signal signr
    call __ssetmask   # set signal mask old blocking
    addl $4,%esp      # discard blocked.
    popl %ea
    x
    popl %ec
    x
    popl %ed
    x popfl ret

```

---

The main purpose of this function is to restore the return value and some register contents after the user program executes the syscall after the signal handler ends, and clear the signal value signr. When compiling a

user-defined signal handler, the compiler invokes the signal syscall in the libc library to insert the sa\_restorer() function into the user program. The function implementation of the signal syscall in the library file is shown below.

---

```

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) 08 {
09     void (*res)();
10    register int foobx asm ("bx") = sig;
11    _asm_("int $0x80": "=a" (res):
12        "0" (_NR_signal), "r" (foobx), "c" (func), "d" ((long) sig_restore));
13    return res;
14 }
15
16 // ユーザーが呼び出したsigaction()関数です。
17 int sigaction(int sig, struct sigaction * sa, struct sigaction * old) 17 {
18     register int foobx asm ("bx") = sig;
19     if (sa->sa_flags & SA_NOMASK)
20         sa->sa_restorer= sig_restore;
21     else
22         sa->sa_restorer= masksig_restore;
23     _asm_("int $0x80": "=a" (sig)
24         :"0" (_NR_sigaction), "r" (_foobx), "c" (sa), "d" (old));
25     if (sig>=0)
26         return 0;
27     errno = -sig;
28     return -1;
29 }
```

The sa\_restorer() is responsible for cleaning up the register value of the user program and the return value of the system-call after the signal handler is executed, as if it had not run the signal handler, but returned directly from the system-call.

**8.7.1.5** 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\_signel()が実行された後、sys\_call.sは、プロセスのカーネルステートスタック上の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はカーネルステートスタック上のcs:eip、eflags、ss:espをポップし、ユーザーステート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。

### 8.7.1.6 Process suspension

signal.cプログラムには、プロセスのシグナルマスクを一時的に引数で指定されたセットに置き換える、シグナルを受信するまでプロセスを一時停止するsys\_sigsuspend()というシステム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)
```

Where restart is a restart indicator. If it's 0, then we save the current mask in the oldmask, and then block the process until we receive a signal; if it's non-zero, then we restore the original mask from the saved oldmask, and return normally.

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)
```

This is the form in which the syscall is used in the C library. The first two parameters will be processed by the sigsuspend() library function. The general implementation of this library function is similar to the following code:

---

```
#define LIBRARY
#include <unistd.h>

int sigsuspend(sigset_t *sigmask)
{
    int res;

    register int fooebx asm ("bx") = 0;
    asm ("int $0x80"
         :"=a" (res)
         : "0" (NR_sigsuspend), "r" (fooebx), "c" (0), "d" (*sigmask)
         : "bx", "cx");
    if (res >= 0)
        return res; errno = -
    res; return -1;
}
```

---

Here, the register variable\_fooebx is the above 'restart'. When the syscall is called for the first time, it is 0, and the original blocking code is saved (old\_mask), and 'restart' is set to a non-zero value. So when the process calls the syscall for the second time, it will restore the blocking code that the process originally saved in old\_mask.

### 8.7.1.7 Restart of a system-call interrupted by signal

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, BUFSIZE)) <
    0 ) { if (errno == EINTR)
        goto again;           /* an interrupted syscall */
    }
```

---

In order to prevent the user program from having to deal with certain interrupted system-call situations, a restart (re-execute) function for some interrupted system-calls is introduced while processing the signal. System-calls that are automatically restarted include: ioctl, read, write, wait, and waitpid. The first three system-calls are interrupted by the signal only when operating on low-speed devices, while wait and waitpid are always interrupted when the signal is captured.

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なります。

## 8.7.2 Code Comments

---

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  
//      data of the initial task 0, and some embedded assembly function macro statements  
//      about the descriptor parameter settings and acquisition.  
//<linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。  
//      commonly used functions of the kernel.  
//<asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。  
//      descriptors/interrupt gates, etc. is defined.  
//<signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、および
```

```

//      signal manipulation function prototypes.
// <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。
//      (Linus was introduced from minix).
7 #include <linux/sched.h>
8 #include <linux/kernel.h>
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 {
28     /* fill in "set" with signals pending but blocked. */
29
// 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。
30
// セットポインタで示されます。
31     verify_area(set, 4);
32     put_fs_long(current->blocked & current->signal, (unsigned long *)set);
33     return 0;
34 }
35
/* atomically swap in the new signal mask, and wait for a signal.
36 *
37 * we need to play some games with syscall restarting. We get help
38 * from the syscall library interface. Note that we need to coordinate
39 * the calling convention with the libc routine.
40 *
41 * "set" is just the sigmask as described in 1003.1-1988, 3.3.7.
42 * It is assumed that sigset_t can be passed as a 32 bit quantity.
43 *
44 * "restart" holds a restart indication. If it's non-zero, then we

```

45 \*      *install the old mask, and return normally. If it's zero, we store*  
46 \*      *the current mask in old\_mask and block until a signal comes in.*  
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
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
52    // 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）。
53    if (restart) {
54        /* we're restarting */
55        current->blocked = old_mask;
56        return -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 -RESTARTNOINTR 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
57    // 走り続けるためには、システムコールに戻ることが必要です。
58    /* we're not restarting. do the work */
59    *(&restart) = 1;
60    *(&old_mask) = current->blocked;
61    current->blocked = set;
62    (void) sys_pause();           /* return after a signal arrives */
63    return -RESTARTNOINTR;       /* handle the signal, and come back */
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 {
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);
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()시스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
```

89

```
//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。
```

```
90     if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
91         return -EINVAL;
92     tmp.sa_handler = (void (*)(int)) handler;
93     tmp.sa_mask = 0;
94     tmp.sa_flags = SA_ONESHOT | SA_NOMASK;
95     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
```

96

```
// 元のシグナルハンドラ。
```

```
97     handler = (long) current->sigaction[signum-1].sa_handler;
98     current->sigaction[signum-1] = tmp;
99     return handler;
98 }
```

```
99
// Siganction() 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
100 // 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。
```

```
101 int sys_sigaction(int signum, const struct sigaction * action,  
102         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。

105

オールドアクションで // になります。

```
106     if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
107         return -EINVAL;
108     tmp = current->sigaction[signum-1];
109     get_new((char *) action,
110             (char *) (signum-1+current->sigaction));
111     if (oldaction)
112         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。

113

// 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。

```
114     if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
115         current->sigaction[signum-1].sa_mask = 0;
116     else
117         current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
118     return 0;
119 }
```

120 /\*

```
121 * Routine writes a core dump image in the current directory.
122 * Currently not implemented.
```

122 \*/

```
123 int core_dump(long signr)
124 {
125     return(0);      /* We didn't do a dump */
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、信号処理ハンドラーをユーザーの  
// 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ー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ー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  
// that are pushed onto kernel-state stack by the interrupt instruction;  
// (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。  
// ecx, and ebx pushed onto the stack just after entering system\_call on lines 85-91;  
// (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 is pushed.

128 // (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。

```
129 int do_signal(long signr, long eax, long ebx, long ecx, long edx, long orig_eax,
130               long fs, long es, long ds,
131               long eip, long cs, long eflags,
132               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 #ifdef notdef
141     printk("pid: %d, signr: %x, eax=%d, oeax = %d, int=%d\n",
142           current->pid, signr, eax, orig_eax,
143           sa->sa_flags & SA_INTERRUPT);
144 #endif
// The orig_eax value will be -1 if it's not a system-call intterrupt but is called during
// other interrupt service (see line 144 of sys_call.s). So when orig_eax is not equal to -1,
// it means that this function was called in the handling of a system-call. In the waitpid()
// function in kernel/exit.c, if SIGCHLD signal is received, or if reading data from a
// 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
// 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
// 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
// -ERESTARTNOINTR は、信号を処理した後、再び
// 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。
// and the return code eax of the system-call is equal to -ERESTARTSYS or -ERESTARTNOINTR,
// the following processing is performed (actually, it has not returned to user program).

```

```

145     if ((orig_eax != -1) &&
146         ((eax == -ERESTARTSYS) || (eax == -ERESTARTNOINTR))) {
147
148     // 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
149     // 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
150     // 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
151     // が
152     // 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。
153     if ((eax == -ERESTARTSYS) && ((sa->sa_flags & SA_INTERRUPT) ||
154         signr < SIGCONT || signr > SIGTTOU))
155         *(&eax) = -EINTR;
156     else {
157
158         // 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
159         // 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
160         // 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。
161         // 信号によって中断されます。
162
163         *(&eax) = orig_eax;
164         *(&eip) = old_eip -= 2;
165     }
166
167     // 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
168     // 処理されずにそのまま返されます。
169
170     sa_handler = (unsigned long) sa->sa_handler;
171     if (sa_handler==1)
172         return(1); /* Ignore, see if there are more signals... */
173
174     // 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
175     // 信号に応じた処理を行います。
176
177     if (!sa_handler) {
178         switch (signr) {
179             // シグ
180             // ナルが
181             // 次の2
182             // つであ
183             // った場
184             // 合も無
185             // 視して
186             // 戻しま
187             // す。
188
189             case SIGCONT:
190
191             break;
192
193             default:
194                 if (signr <= SIGRTMIN)
195                     do_rt_signal(signr);
196                 else
197                     do_sa_handler(signr);
198
199             break;
200
201         }
202     }
203
204     return(0);
205 }

```

```
161         case SIGCHLD:
162             return(1); /* Ignore, ... */
```

```

// If the signal is one of the following four signals, the current process state is set to
// the stop state TASK_STOPPED. If the parent process of the current process does not set
// the sigaction flag SA_NOCLDSTOP of the SIGCHLD signal, then the SIGCHLD signal is sent
// to the parent process. The SA_NOCLDSTOP flag indicates that the SIGCHLD signal is not
// generated when the child process stops executing or continues execution.
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); /* Reschedule another event */
174
// If the signal is one of the following 6 signals, if the signal generates a core dump,
// call do_exit() to exit with the exit code as signr|0x80. Otherwise the exit code is the
// signal value. The parameters of do_exit() are the return code and the exit status
// information provided by the program. It can be used as status information for the
// wait() or waitpid() functions. See lines 13-19 of the sys/wait.h file. wait() or
// waitpid() can use these macros to get the exit status code of the child process or the
// reason (signal) of the child process termination.
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             /* fall through */
184         default:
185             do_exit(signr);
186         }
187     }
188     /*
189      * OK, we're invoking a handler
190     */
191     // If the signal handler only needs to be called once, the handler is left blank. Note
192     // that the signal handler has been previously saved in the sa_handler pointer.
193     // When the system-call enters the kernel code, the user program return address (eip, cs)
194     // is saved in the kernel state stack. The following code modifies the code pointer eip on
195     // the kernel state stack to point to the signal handler, and also pushed sa_restorer,
196     // signr, process mask (if SA_NOMASK is not set), eax, ecx, edx as parameters onto the
197     // user stack. The original program return pointer and eflag register are also pushed onto
198     // the user stack. Therefore, when the syscall returns to the user program, the user's
199     // signal handler is executed first, and then the user program is continued.
200     if (sa->sa_flags & SA_ONESHOT)
201         sa->sa_handler = NULL;
202
// カーネルのステート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)" when assigning values to eip.
// The sa_mask field of the sigaction structure gives the set of signals that should be
// masked during the execution of the current signal handler. At the same time, the current
// signal will also be blocked. However, if the SA_NOMASK flag is used in sa_flags, the
// current signal will not be masked. If the signal handler is allowed to receive its own
// signal, the signal blocking code of the process also needs to be pushed onto the stack.
193     *(&eip) = sa_handler;
194     longs = (sa->sa_flags & SA_NOMASK)?7:8;
    // Extend the user stack pointer of the original user program by 7 (or 8) long words (used
    // to store the parameters of the signal handler, etc.) and check the memory usage (if the
    // memory is out of bounds, allocate a new page, etc.).
195     *(&esp) -= longs;
196     verify_area(esp, longs*4);
    // Store sa_restorer, signal signer, mask code blocked (if SA_NOMASK is set), eax, ecx,
    // edx, eflags, and user program code pointer from bottom to top in the user stack.
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))
        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; // Fill in with sa_mask bitmap
208     return(0);           /* Continue, execute handler */
209 }
210

```

## 8.7.3 Information

### 8.7.3.1 Signal Description

The signal in the process is a simple message used for communication between processes, usually a label value between 1 to 31, and does not carry any other information. The signals supported by the Linux 0.12 kernel are shown in Table 8–5.

表 8 - 5  プロセス信号 Sig	Name	Description	Default operation

1	SIGHUP	(Hangup) The kernel generates this signal when you no longer have a control terminal, or when you turn off Xterm or disconnect the modem. Since the background programs do not have control terminals, they often use SIGHUP to signal that they need to re-read their configuration files.	(Abort) Hang up control terminal or process.
2	SIGINT	(Interrupt) An interrupt from the keyboard. Usually the terminal driver will bind it to ^C.	(Abort) Terminate program.
3	SIGQUIT	(Quit) The exit interrupt from keyboard. Usually the terminal driver will bind it to ^\.	(Dump) Terminated and a dump core file is generated.
4	SIGILL	(Illegal Instruction) The program has an error or an illegal operation	(Dump)

		command has been executed.	
5	SIGTRAP	(Breakpoint/Trace Trap) Used for debugging, tracking breakpoints.	
6	SIGABRT	(Abort) Abandon execution and end abnormally.	(Dump)
6	SIGIOT	(IO Trap) Same as SIGABRT	(Dump)
7	SIGUNUSED	(Unused) Not used.	
8	SIGFPE	(Floating Point Exception) Floating exception.	(Dump)
9	SIGKILL	(Kill) The program was terminated. This signal cannot be captured or ignored. If you want to terminate a process immediately, you will send a signal 9. Note that the program will have no chance to do the cleanup.	(Abort)
10	SIGUSR1	(User defined Signal 1) User defined signal 1.	(Abort)
11	SIGSEGV	(Segmentation Violation) This signal is generated when the program references invalid memory. For example: addressing unmapped memory; addressing unlicensed memory.	(Dump)
12	SIGUSR2	(User defined Signal 2) Reserved for user programs for IPC or other purposes.	(Abort)
13	SIGPIPE	(Pipe) This signal is generated when a program writes to a socket or pipe, and there is no reader.	(Abort)
14	SIGALRM	(Alarm) This signal is generated after the delay time set by the user using alarm syscall. This signal is often used to determine syscall timeouts.	(Abort)
15	SIGTERM	(Terminate) Used to kindly require a program to terminate. It is the default signal for kill. Unlike SIGKILL, this signal can be captured so that it can be cleaned up before exiting.	(Abort)
16	SIGSTKFLT	(Stack fault on coprocessor) Coprocessor stack error.	(Abort)
17	SIGCHLD	(Child) The child process issued. The child process has been stopped or terminated. You can change its meaning and use it for other usage.	(Ignore) The child process stops or ends.
18	SIGCONT	(Continue) This signal causes the process stopped by SIGSTOP to resume operation. Can be captured.	(Continue) Resume process running.
19	SIGSTOP	(Stop) Stop process running. This signal cannot be captured or ignored.	(Stop) Stop process running.
20	SIGTSTP	(Terminal Stop) Send a stop key sequence to the terminal. This signal can be captured or ignored.	(Stop)
21	SIGTTIN	(TTY Input on Background) The background process attempts to read data from a terminal that is no longer under control, at which point the process will be stopped until the SIGCONT signal is received. This signal can be captured or ignored.	(Stop)
22	SIGTTOU	(TTY Output on Background) The background process attempts to output data to a terminal that is no longer under control, at which point the process will be stopped until the SIGCONT signal is received. This signal can be captured or ignored.	(Stop)

## 8.8 exit.c

### 8.8.1 Function Description

The exit.c program mainly implements the processing related to the termination and exit of process. These include process release, session (process group) termination, and program exit handlers, as well as system-calls such as killing processes, terminating processes, and suspending processes. It also includes signal sending function send\_sig(), and function tell\_father() that notifies parent the termination of child process.

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、現在のプロセスを中断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 Code Comments

```
1 /*
2 * linux/kernel/exit.c
```

```
3 *  
4 * (C) 1991 Linus Torvalds  
5 */  
6  
7 #define DEBUG_PROC_TREE  
8  
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。  
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナルの定義  
//      manipulation function prototypes.
```

```

// <sys/wait.h> Waitのヘッダーファイル。システムコール wait() core waitpid() および関連する
// constant symbols.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// of the initial task 0, and some embedded assembly function macro statements about the
// descriptor parameter settings and acquisition.
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// used functions of the kernel.
// <linux/tty.h> ttyヘッダーファイルは、tty_io、シリアルのパラメータと定数を定義しています。
// communication.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
// for segment register operations.
9 #include <errno.h>
10 #include <signal.h>
11 #include <sys/wait.h>
12
13 #include <linux/sched.h>
14 #include <linux/kernel.h>
15 #include <linux/tty.h>
16 #include <asm/segment.h>
17
18 int sys_pause(void);           // put in sleep until receive a signal (kernel/sched.c, 164).
19 int sys_close(int fd);         // close a file (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
// released. Finally execute the scheduler and exit immediately upon return. If the item
// 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
    if (!p)

```

```

25         return;
26     if (p == current) {
27         printk("task releasing itself\n|r");
28         return;
29     }
30
// The following loop statement scans the array of task structure pointers to find the specified
// task p. If found, the corresponding item in the task pointer array is set to NULL, and the
// associated pointer between the task is updated, and the memory page occupied by the task
// p data structure is released. Finally, exit after scheduler returns. If task p is not found,
// the kernel code is wrong, and an error message is displayed and the kernel crashes. In addition,
// the code that updates the links removes the task p from the doubly linked list.
31     for (i=1 ; i<NR_TASKS ; i++)
32         if (task[i]==p) {
33             task[i]=NULL;
34             /* Update links */
// 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。  
// pptr (parent pointer) points to the parent process of p.  
// cptr (child pointer) parent process points to the last created child.
```

```

35         if (p->p_osptr)
36             p->p_osptr->p_ysptr = p->p_ysptr;
37         if (p->p_ysptr)
38             p->p_ysptr->p_osptr = p->p_osptr;
39         else
40             p->p_pptr->p_cptr = p->p_osptr;
41         free_page((long)p);
42         schedule();
43         return;
44     }
45     panic("trying to release non-existent task");
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。
50 49 /*
51 * Check to see if a task_struct pointer is present in the task[] array
52 * Return 0 if found, and 1 if not found.
52 */
53 int bad_task_ptr(struct task_struct *p) 54 {...
55     int i;
56
57     if (!p)
58         return 0;
59     for (i=0 ; i<NR_TASKS ; i++)
60         if (task[i] == p)
61             return 0;
62     return 1;
63 }
64
66 65 /*
67 * This routine scans the pid tree and make sure the rep invariant still
68 * holds. Used for debugging only, since it's very slow....
68 *
69 * It looks a lot scarier than it really is.... we're doing nothing more
70 * than verifying the doubly-linked list found in p_ysptr and p_osptr,
71 * and checking it corresponds with the process tree defined by p_cptr and
72 * p_pptr;
73 */
74 void audit_ptree()
75 {
76     int i;
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

```

```

// four pointers (pptr, cptr, ysptr, and osptr). Skip if the task array slot is empty.
78     for (i=1 ; i<NR_TASKS ; i++) {
79         if (!task[i])
80             continue;
81         // 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
82         // 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
83         // cptr, ysptr, osptr にも同様の操作を行います。
84         if (bad_task_ptr(task[i]->p_pptr))
85             printk("Warning, pid %d's parent link is bad\n",
86                     task[i]->pid);
87         if (bad_task_ptr(task[i]->p_cptr))
88             printk("Warning, pid %d's child link is bad\n",
89                     task[i]->pid);
90         if (bad_task_ptr(task[i]->p_ysptr))
91             printk("Warning, pid %d's ys link is bad\n",
92                     task[i]->pid);
93         if (bad_task_ptr(task[i]->p_osptr))
94             printk("Warning, pid %d's os link is bad\n",
95                     task[i]->pid);
96         // 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。
97         // 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。
98         if (task[i]->p_pptr == task[i])
99             printk("Warning, pid %d parent link points to self\n");
100        if (task[i]->p_cptr == task[i])
101            printk("Warning, pid %d child link points to self\n");
102        if (task[i]->p_ysptr == task[i])
103            printk("Warning, pid %d ys link points to self\n");
104        if (task[i]->p_osptr == task[i])
105            printk("Warning, pid %d os link points to self\n");
106        // 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。
107        // 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
108        // 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。
109        if (task[i]->p_osptr) {
110            if (task[i]->p_pptr != task[i]->p_osptr->p_pptr)
111                printk(
112                    "Warning, pid %d older sibling %d parent is %d\n",
113                    task[i]->pid, task[i]->p_osptr->pid,
114                    task[i]->p_osptr->p_pptr->pid);
115            if (task[i]->p_osptr->p_ysptr != task[i])
116                printk(
117                    "Warning, pid %d older sibling %d has mismatched ys link\n",
118                    task[i]->pid, task[i]->p_osptr->pid);
119        }
120        // 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。
121        // 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。
122        // 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。
123        if (task[i]->p_ysptr) {
124            if (task[i]->p_pptr != task[i]->p_ysptr->p_pptr)
125                printk(
126                    "Warning, pid %d younger sibling %d parent is %d\n",
127                    task[i]->pid, task[i]->p_ysptr->pid,
128

```

```
118          task[i]->p_osptr->p_pptr->pid) ;  
119      if ( task[i]->p_ysptr->p_osptr != task[i])
```

```

119             printk(
120                 "Warning, pid %d younger sibling %d has mismatched os link\n",
121                         task[i]->pid, task[i]->p_ysptr->pid);
122         }
123         // タスクの最新の子ポインタ cptr が空でない場合、子の親が以下のようになっているかどうかをチェックします。
124         // 処理を行い、子の若いポインタ yspter 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
125         // のメッセージが表示されます。
126         if (task[i]->p_cptr) {
127             if (task[i]->p_cptr->p_ptr != task[i])
128                 printk(
129                     "Warning, pid %d youngest child %d has mismatched parent link\n",
130                         task[i]->pid, task[i]->p_cptr->pid);
131             if (task[i]->p_cptr->p_ysptr)
132                 printk(
133                     "Warning, pid %d youngest child %d has non-NULL ys link\n",
134                         task[i]->pid, task[i]->p_cptr->pid);
135         }
136     #endif /* DEBUG_PROC_TREE */
137     // Send a signal sig to task p with privilege priv.
138     // sig - the signal; p - a pointer to the task; priv - a flag that forces the signal to be sent,
139     // 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を持つ。
140     // この関数は、まずパラメータの正しさをチェックし、次に
141     // 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。
142     // 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。
143 static inline int send_sig(long sig, struct task_struct * p, int priv) {
144     // 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
145     // 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。
146     // 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。
147     if (!p)
148         return -EINVAL;
149     if (!priv && (current->euid!=p->euid) && !suser())
150         return -EPERM;
151     // If the signal to be send is SIGKILL or SIGCONT, then if the process p receiving signal
152     // is stopped at this time, it is set to the ready state (TASK_RUNNING). Then modify the
153     // signal bitmap process p to remove (reset) the signals SIGSTOP, SIGTSTP, SIGTTIN, and
154     // SIGTTOU that will cause the process to stop.
155     if ((sig == SIGKILL) || (sig == SIGCONT)) {
156         if (p->state == TASK_STOPPED)
157             p->state = TASK_RUNNING;
158         p->exit_code = 0;
159         p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1)) |
160                         (1<<(SIGTTIN-1)) | (1<<(SIGTTOU-1)) );
161     }
162     /* If the signal will be ignored, don't even post it */
163     if ((int) p->sigaction[sig-1].sa_handler == 1)
164         return 0;
165     /* Depends on order SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU */
166     // If the signal is one of SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU, then it is necessary to

```

// stop process p from running. Therefore (if SIGCONT is set in the signal bitmap of p), it

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     /* Actually deliver the signal */
157     p->signal |= (1<<(sig-1));
158     return 0;
159 }
160
// 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 pggrp.
// コードは、タスク配列をスキャンし、グループID pggrpを持つプロセスを探し、そのセッションを返す
// idです。指定されたグループ pggrp 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。

```

### 161 int session\_of\_pggrp(int pggrp)

```

162 {
163     struct task_struct **p;
164
165     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
166         if ((*p)->pggrp == pggrp)
167             return ((*p)->session);
168     return -1;
169 }
170
// 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）。
// パラメータ：grp - プロセスグループID、sig - シグナル、priv - 特権。
// つまり、シグナルsigはプロセスグループpggrpの各プロセスに送られます。限り、それは
// が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
// グループ pggrp の場合、エラーコード -ESRCH が返されます。グループが pggrp であるプロセスが
// 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。

```

### 171 int kill\_pg(int pggrp, int sig, int priv) 172 {

```

174     struct task_struct **p;
175     int err, retval = -ESRCH;           // ESRCH - error search.
176     int found = 0;
177
// 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
// キャンします。
// システムになります。グループID pggrp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ら
// れます。
// 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。
178     if (sig<1 || sig>32 || pggrp<=0)
179         return -EINVAL;
180     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
181         if ((*p)->pggrp == pggrp) {
182             if (sig && (err = send_sig(sig,*p,priv)))
183                 retval = err;
184             else
185                 found++;
186         }
187     return (found ? 0 : retval);
188 }

```

```
// プロセスをキルする（プロセスにシグナルを送る）。
// パラメータ：pid - プロセスID、sig - シグナル、priv - 特権。
// 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
// 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 ; --p)
196         if ((*p)->pid == pid)
197             return(sig ? send_sig(sig,*p,priv) : 0);
198     return(-ESRCH);
199 }
200
201 /*
202 * POSIX specifies that kill(-1,sig) is unspecified, but what we have
203 * is probably wrong. Should make it like BSD or SYSV.
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 - 送信する必要のあるシグナル。
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
218 int sys_kill(int pid,int sig) 206 {
219     struct task_struct **p = NR_TASKS + task;      // points to the last item.
220     int err, retval = 0;
221
222     if (!pid)
223         return(kill_pg(current->pid, sig, 0)); 212
224     if (pid == -1) {
225         while (--p > &FIRST_TASK)
226             if (err = send_sig(sig,*p,0))
227                 retval = err;
228         return(retval);
229     }
230     if (pid < 0)
231         return(kill_pg(-pid, sig, 0));
232     /* Normal kill */
233     return(kill_proc(pid, sig, 0));
234 }
235
236 /*
237 * Determine if a process group is "orphaned", according to the POSIX
238 * definition in 2.2.2.52. Orphaned process groups are not to be affected
239 * by terminal-generated stop signals. Newly orphaned process groups are

```

229 \* to receive a SIGHUP and a SIGCONT.  
229\*

230 \* "I ask you, have you ever known what it is to be an orphan?"

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
// "orphan." The connection between a process group and a parent outside its group depends
// 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
// の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プロセスではありません。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。
int is_orphaned_pgrp(int pgrp)
```

```

232
233 {
234     struct task_struct **p;
235
236     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
237         if (!(*p) ||
238             ((*p)->pgrp != pggrp) ||
239             ((*p)->state == TASK_ZOMBIE) ||
240             ((*p)->p_pptr->pid == 1))
241             continue;
242         if (((*p)->p_pptr->pgrp != pggrp) &&
243             ((*p)->p_pptr->session == (*p)->session))
244             return 0;
245     }
246     return(1);      /* (sighing) "Often!" */
247 }
248
// Check if the process group contains a job (process group) that is in a stopped state.
// Returns 1 if there is none; returns 0 if none. The search method is to scan the entire task
// 配列を作成し、グループ pggrp 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 pggrp) 250
{
251     struct task_struct ** p;
252

```

```

253     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
254         if ((*p)->pgrp != pggrp)
255             continue;
256         if ((*p)->state == TASK_STOPPED)
257             return(1);
258     }
259     return(0);
260 }
261

// プログラム終了処理関数です。以下の365行目のsyscall sys_exit()によって呼び出されます。
// 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。
// 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。
// 他のプロセスを実行するために、戻りません。
262 volatile void do_exit(long code) 263
{
265     struct task_struct *p;
266     int i;
267

// 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。
// 関数 free_page_tables() の第一引数 (get_base() の戻り値) は、以下を示します。
// 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
// 値) は、解放するバイト長を示します。get_base()マクロの current->lvt[1] は
// はプロセスコードセグメント記述子の位置を示し、current->lvt[2]は
// データセグメント記述子の位置。get_limit()の0x0fはコードのセレクタです。
// セグメント、0x17はデータセグメントのセレクタとなります。つまり、セグメントのベースアドレスが
// 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
// の制限（長さ）がかかっている場合、セグメントセレクターはパラメータとして使用されます（指定された
// セレ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）。
// free_page_tables()は、mm/memory.cファイルの69行目の冒頭にあります。
268     free_page_tables(get_base(current->lvt[1]), get_limit(0x0f));
269     free_page_tables(get_base(current->lvt[2]), get_limit(0x17));

// 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
// 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。
// process to zombie state (TASK_ZOMBIE) and set process exit code.

```

```
269     for (i=0 ; i<NR_OPEN ; i++)
270         if (current->filp[i])
271             sys_close(i);
272     iput(current->pwd);
273     current->pwd = NULL;
274     iput(current->root);
275     current->root = NULL;
276     iput(current->executable);
277     current->executable = NULL;
278     iput(current->library);
279     current->library = NULL;
280     current->state = TASK_ZOMBIE;
281     current->exit_code = code;
282     /*
```

283  
284

*\* Check to see if any process groups have become orphaned  
\* as a result of our exiting, and if they have any stopped*

```

285      * jobs, send them a SIGUP and then a SIGCONT. (POSIX 3.2.2.2)
286      *
287      * Case i: Our father is in a different pgrp than we are
288      * and we were the only connection outside, so our pgrp
289      * is about to become orphaned.
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      /* Let father know we died */
299      current->p_pptr->signal |= (1<<(SIGCHLD-1));
300
301      /*
302      * This loop does two things:
303      *
304      * A. Make init inherit all the childprocesses
305      * B. Check to see if any process groups have become orphaned
306      *      as a result of our exiting, and if they have any stopped
307      *      jons, send them a SIGUP and then a SIGCONT. (POSIX 3.2.2.2)
308      */

```

// 現在のプロセスに子プロセスがある場合（その p\_cptr 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  
// the init process (parent).

```

309     if (p = current->p_cptr) {
310         while (1) {
311             p->p_pptr = task[1];
312             if (p->state == TASK_ZOMBIE)
313                 task[1]->signal |= (1<<(SIGCHLD-1));
314             /*
315             * process group orphan check
316             * Case ii: Our child is in a different pggrp
317             * than we are, and it was the only connection
318             * outside, so the child pggrp is now orphaned.
319             */
// 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。
// loop through these sibling processes.

```



```
320     if ((p->pgrp != current->pgrp) &&
321         (p->session == current->session) &&
322         is_orphaned_pgrp(p->pgrp) &&
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             continue;
330         }
331     /*
332      * This is it; link everything into init's children
333      * and leave
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。この時点では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。
336         p->p_osptr = task[1]->p_cptr;
337         task[1]->p_cptr->p_ysptr = p;
338         task[1]->p_cptr = current->p_cptr;
339         current->p_cptr = 0;
340         break;
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。
342 // 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）。
343     if (current->leader) {
344         struct task_struct **p;
345         struct tty_struct *tty;
346
347         if (current->tty >= 0) {
348             tty = TTY_TABLE(current->tty);
349             if (tty->pgrp>0)
350                 kill_pg(tty->pgrp, SIGHUP, 1);
351             tty->pgrp = 0;
352             tty->session = 0;
353         }
354         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
355             if ((*p)->session == current->session)
356                 (*p)->tty = -1;
357     }
// 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に設定して
// 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ルして、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。
358     if (last_task_used_math == current)
359         last_task_used_math = NULL;

```

```
359 #ifdef DEBUG_PROC_TREE  
360     audit_ptree();  
361 #endif  
362     schedule();
```

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信した場合、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、このシステム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 - the process id; *stat_addr - pointer to the state info; options - waitpid option.
```

370 int sys\_waitpid(pid\_t pid,unsigned long \* stat\_addr,int options) 371 { ...

```
373         int flag;           // selected child in ready or sleep state.
374         struct task_struct *p;
375         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ヤンされます。
    verify_area(stat_addr, 4);
```

376

```
377 repeat:  
378     flag=0;  
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。  
381         if (pid>0) {  
382             if (p->pid != pid)  
383                 continue;  
 // それ以外の場合、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意味する  
 // 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ると、390行目にジャンプして続行します。
// 実行します。
385         } else if (!pid) {
386             if (p->pgrp != current->pgrp)
387                 continue;
// 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。スキヤンされたプロセス 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
389         } else if (pid != -1) {
390             if (p->pgrp != -pid)
391                 continue;
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、スキヤ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
// ステータスマッセージ 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 なので、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。
391     switch (p->state) {
392         case TASK_STOPPED:
393             if (!(options & WUNTRACED) ||
394                 !p->exit_code)
395                 continue;
396             put_fs_long((p->exit_code << 8) | 0x7f,
397                         stat_addr);
398             p->exit_code = 0;
399             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。
401         case TASK_ZOMBIE:
402             current->cutime += p->utime;
403             current->cstime += p->stime;
404             flag = p->pid;

```

```
405          put_fs_long(p->exit_code, stat_addr);  
406          release(p);  
407 #ifdef DEBUG PROC TREE  
408          audit_ptree();  
409 #endif  
410          return flag;
```

```

// 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
412         default:
413             flag=1;
414             continue;
412         }
413     }

// タスク配列のスキャンが終了した後、フラグが設定されていれば、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
// if (flag) {
が
保
存
さ
れ
、
SI
G
C
H
L
D
信
号
を
受
信
可
能
な
よ
う
に
修
正
さ
れ
た
後
、
ス
ケ
ジ
ュ
ー
ラ
ー
を
実
行
し
ま
す
。
414
415     if (options & WNOHANG)
416         return 0;
417     current->state=TASK_INTERRUPTIBLE;

```

```

418     oldblocked = current->blocked;
419     current->blocked &= ~(1<<(SIGCHLD-1));
420     schedule();
421
422     // 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
423     // 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
424     // 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。
425     current->blocked = oldblocked;
426     if (current->signal & ~(current->blocked | (1<<(SIGCHLD-1))))
427         return -ERESTARTSYS;
428     else
429         goto repeat;
430
431     // 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。
432     // のコードが返されます（子プロセスが存在しない）。
433     return -ECHILD;
434 }
435

```

## 8.9 fork.c

### 8.9.1 Function Description

The fork() system-call is used to create child process. All processes in Linux are child processes of process 0 (task 0). The fork.c program includes a set of auxiliary processing functions for sys\_fork() (starting at line 222 in kernel/sys\_call.s). It gives two C functions used in the sys\_fork() system-call: find\_empty\_process() and copy\_process(). It also includes the process memory area validation and memory allocation functions verify\_area() and copy\_mem().

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レ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）単位で動作する必要があるため、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。

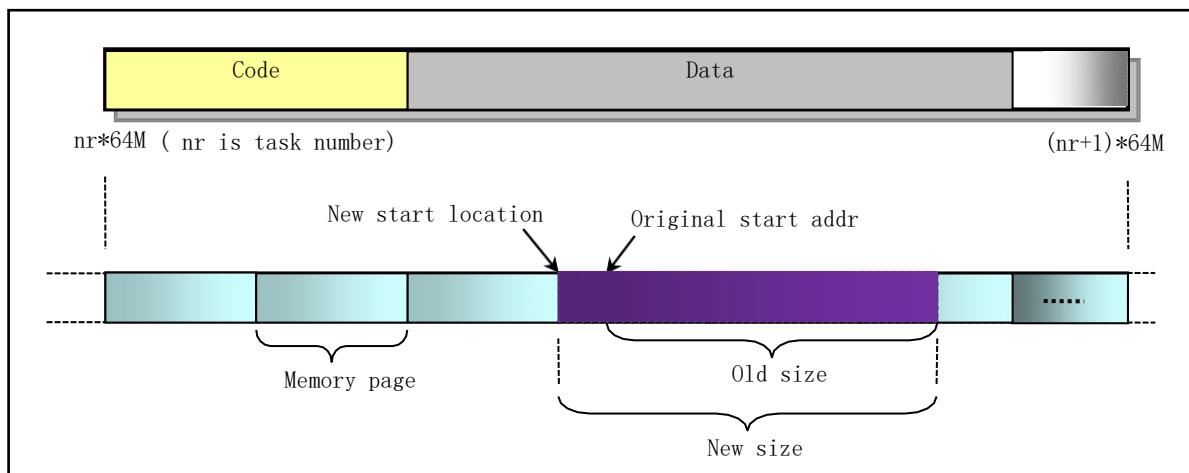


図8-13 メモリ検証範囲と開始位置の調整

The role of fork() is briefly described above based on the purpose of each function in the fork.c program. Here we will give a little more explanation on it. In general, fork() will first apply for a page of memory for the new process to copy the task data structure (also known as process control block, PCB) information of the parent process, and then modify the copied task data structure for the new process. These fields include resetting the registers of the TSS structure in the task structure by using the registers that are gradually pushed into stack when the system-call interrupt occurs (ie, the parameter of copy\_process()), so that the state of the new process keeps the parent process state before entering the interrupt. The program then determines the starting position (nr \* 64MB) in the linear address space for the new process. For the segmentation mechanism of the CPU, the

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 Code Comments

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' contains the help-routines for the 'fork' system call
9 * (see also system_call.s), and some misc functions ('verify_area').
10 * Fork is rather simple, once you get the hang of it, but the memory
11 * management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'
12 */

// <errno.h> エラー番号のヘッダーファイルです。システムの様々なエラー番号を含みます。
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義
//      が含まれています。
//      used functions of the kernel.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
//      segment register operations.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
//      descriptors/interrupt gates, etc. is defined.

13 #include <errno.h>
14
15 #include <linux/sched.h>
16 #include <linux/kernel.h>
17 #include <asm/segment.h>
18 #include <asm/system.h>
19

// 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

```

照してください。

20 extern void write\_verify(unsigned long address); 21

22 long last\_pid=0; // The latest process id, generated by get\_empty\_process().  
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。  
ユーザースペースの//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は
// 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
// 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온라이트）が行われます。
```

24 void verify\_area(void \* addr,int size) 25 {.26 unsigned long start;.27

```
// 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
```

```
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
```

```
// 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もの
// が
```

```
// 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
```

```
// 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
```

28

```
// 上の図8-13。
```

29 start = (unsigned long) addr;30 size += start & 0xffff;31 start &= 0xfffffff000; // now start is logical address.

```
// 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
```

```
// 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
```

32

```
// 書き込み可能なので、ページをコピーします(mm/memory.c, line 274)。
```

33 start += get\_base(current->lvt[2]); // include/linux/sched.h, line 27734 while (size>0) {33 size -= 4096;34 write\_verify(start);35 start += 4096;36 }37 }38 }

```
// メモリのページテーブルをコピーします。
```

```
// 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
```

```
// 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。
```

```
// 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온라이트技術を使用しているので
```

```
// 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ティリとページテーブルエンティリのみが設定されます。
```

```
// 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  
// 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。  
// それ以外の場合は、エラーコードを返します。
```

```
41 39 int copy_mem(int nr,struct task_struct * p) 40 {  
42     unsigned long old_data_base, new_data_base, data_limit;  
43     unsigned long old_code_base, new_code_base, code_limit;  
43  
    // 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を取る。
```

```

// 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です。
// 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
45     code_limit=get_limit(0x0f);
46     data_limit=get_limit(0x17);
47     old_code_base = get_base(current)>lvt[1]);
48     old_data_base = get_base(current)>lvt[2]);
49     if (old_data_base != old_code_base)
50         panic("We don't support separate I&D");
51     if (data_limit < code_limit)
52         panic("Bad data_limit");
53
54     // 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。
55     // 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
56     // 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ティリとページテーブル
57     // エントリを設定します。
58     // 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ティリとページテーブルエンティリをコピーしま
59     // す。
60     // 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
61     // copy_page_tables()は0を返します。それ以外の場合は、エラーを示し、そのページのエンティリが
62     // 適用されたばかりのものがリリースされます。
63
64     new_data_base = new_code_base = nr * TASK_SIZE;
65     p->start_code = new_code_base;
66     set_base(p->lvt[1], new_code_base);
67     set_base(p->lvt[2], new_data_base);
68     if (copy_page_tables(old_data_base, new_data_base, data_limit)) {
69         free_page_tables(new_data_base, data_limit);
70     return -ENOMEM; 59
71     }
72
73 /*
74 * Ok, this is the main fork-routine. It copies the system process
75 * information (task[nr]) and sets up the necessary registers. It
76 * also copies the data segment in its entirety.
77 */
78
79     // プロセス情報をコピーします。
80     // 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
81     // 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
82     // 徐々にプロセスのカーネルステートスタックに押し込まれていきます。の値(パラメータ)は
83     // sys_call.sファイルのincludeでスタックにpushされます。
84     // 1) 実行時にpush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
85     //    the INT instruction;
86     // 2) ds, es, fs, and edx, ecx, ebxが入力直後の85-91行目でスタックにpushされる。

```

```
// 3) 97行目でsys_call_tableのsys_fork()が呼ばれたときにpushされたリターンアドレス  
//      (represented by none);  
// 4) 226-230行目で、gs, esi, edi, ebp, eax(nr)がcopy_process()を呼び出す前にpushされる。  
68 // 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。  
69 int copy_process(int nr, long ebp, long edi, long esi, long gs, long none,  
70                  long ebx, long ecx, long edx, long orig_eax,  
71                  long fs, long es, long ds,
```

```
72         long eip, long cs, long eflags, long esp, long ss)
72 {
73     struct task_struct *p;
74     int i;
75     struct file *f;
76
// 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、次のように返します）。
// 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
// 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。
// memory page p just applied.
    p = (struct task_struct *) get_free_page();
77
78     if (!p)
```

```

79             return -EAGAIN;
80     task[nr] = p;
81     *p = *current; /* NOTE! this doesn't copy the supervisor stack */

// その後、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、タスク
// 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
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
// 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。
// 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
ダーやリセットします。
// フラグ、カーネルおよびユーザー mode での実行時間の統計、システム時間の start_time
82   プロセスの実行を開始したときの//。
83   p->state = TASK_UNINTERRUPTIBLE;
84   p->pid = last_pid;           // new pid obtained from find_empty_process()
85   p->counter = p->priority;  // run time slice value (number of ticks).
86   p->signal = 0;              // signal bitmap.
87   p->alarm = 0;               // alarm timer.
88   p->leader = 0;              /* process leadership doesn't inherit */
89   p->utime = p->stime = 0;   // user state and core state running time.
90   p->cutime = p->cstime = 0; // child's user state and core state running time.
91   p->start_time = jiffies;   // the start time of the process (current time ticks).

// 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）。
// 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 + )
// ss0:esp0はスタックとして使用されます。
プログラムがカーネルモードで実行されるためには、//。
// 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
// 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
// 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
// です。
// 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
92   // 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。
93   p->tss.back_link = 0;
94   p->tss.esp0 = PAGE_SIZE + (long) p; // Task kernel state stack pointer.
95   p->tss.ss0 = 0x10;                  // selector for the kernel state stack.
96   p->tss.eip = eip;
97   p->tss.eflags = eflags;
98   p->tss.eax = 0;                   // This is why the new process will return 0.
99   p->tss.ecx = ecx;
100  p->tss.edx = edx;
101  p->tss.ebx = ebx;
102  p->tss.esp = esp;

```

```

103     p->tss.ebp = ebp;
104     p->tss.esi = esi;
105     p->tss.edi = edi;
106     p->tss.es = es & 0xffff;           // The segment register has only 16 bits.
107     p->tss.cs = cs & 0xffff;
108     p->tss.ss = ss & 0xffff;
109     p->tss.ds = ds & 0xffff;
110     p->tss.fs = fs & 0xffff;
111     p->tss.gs = gs & 0xffff;
112     p->tss.ldt = LDT(nr);        // The selector for the task LDT descriptor (in GDT).
113     p->tss.trace_bitmap = 0x80000000;    // (High 16 bits are valid).

// 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。
114     // 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)
115     if (last_task_used_math == current)
116         __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
117     // 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。
118     if (copy_mem(nr, p)) {                // The return is not 0, indicating an error.
119         task[nr] = NULL;
120         free_page((long) p);
121         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。
119     // 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1ずつ。
120     for (i=0; i<NR_OPEN; i++)
121         if (f=p->filp[i])
122             f->f_count++;
123     if (current->pwd)
124         current->pwd->i_count++;
125     if (current->root)
126         current->root->i_count++;

```

```
127     if (_current->executable)
128         _current->executable->i_count++;
129     if (_current->library)
130         _current->library->i_count++;
// 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ト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。
```

```
131
132     set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY, &(p->tss));
133     set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY, &(p->ldt));
134     p->p_pptr = current;           // parent pointer.
135     p->p_cptr = 0;
136     p->p_ysptr = 0;
137     p->p_osptr = current->p_cptr; // old sibling.
138     if (p->p_osptr)             // if old sibling exist, its young
139         p->p_osptr->p_ysptr = p; // sibling points to this new process.
140     current->p_cptr = p;        // I am the current new child.
141     p->state = TASK_RUNNING;    /* do this last, just in case */
142     return last_pid;
143 }
144
145 // 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
146 // タスク配列の番号（配列項目）です。
147 // 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
148 // 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。
149 // 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。もしあれば、タスク配列の
150 // 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
151 // 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
152 // 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
153 // 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
154 int find_empty_process(void)
```

```
143 {
144     int i;
145
146     repeat:
147         if ((++last_pid)<0) last_pid=1;
148         for(i=0 ; i<NR_TASKS ; i++)
149             if (task[i] && ((task[i]->pid == last_pid) ||
150                             (task[i]->pgrp == last_pid)))
151                 goto repeat;
```

```
153     for(i=1 ; i<NR_TASKS ; i++)           // Task 0 is excluded.  
154         if (!task[i])  
155             return i;  
156     return -EAGAIN;  
157 }  
158
```

---

## 8.9.3 Information

### 8.9.3.1 Task Status Segment (TSS)

Figure 8-14 below shows the contents of the task state segment (TSS). The TSS for each task is saved in the task data structure task\_struct. Please refer to Chapter 4 for a detailed description of it.

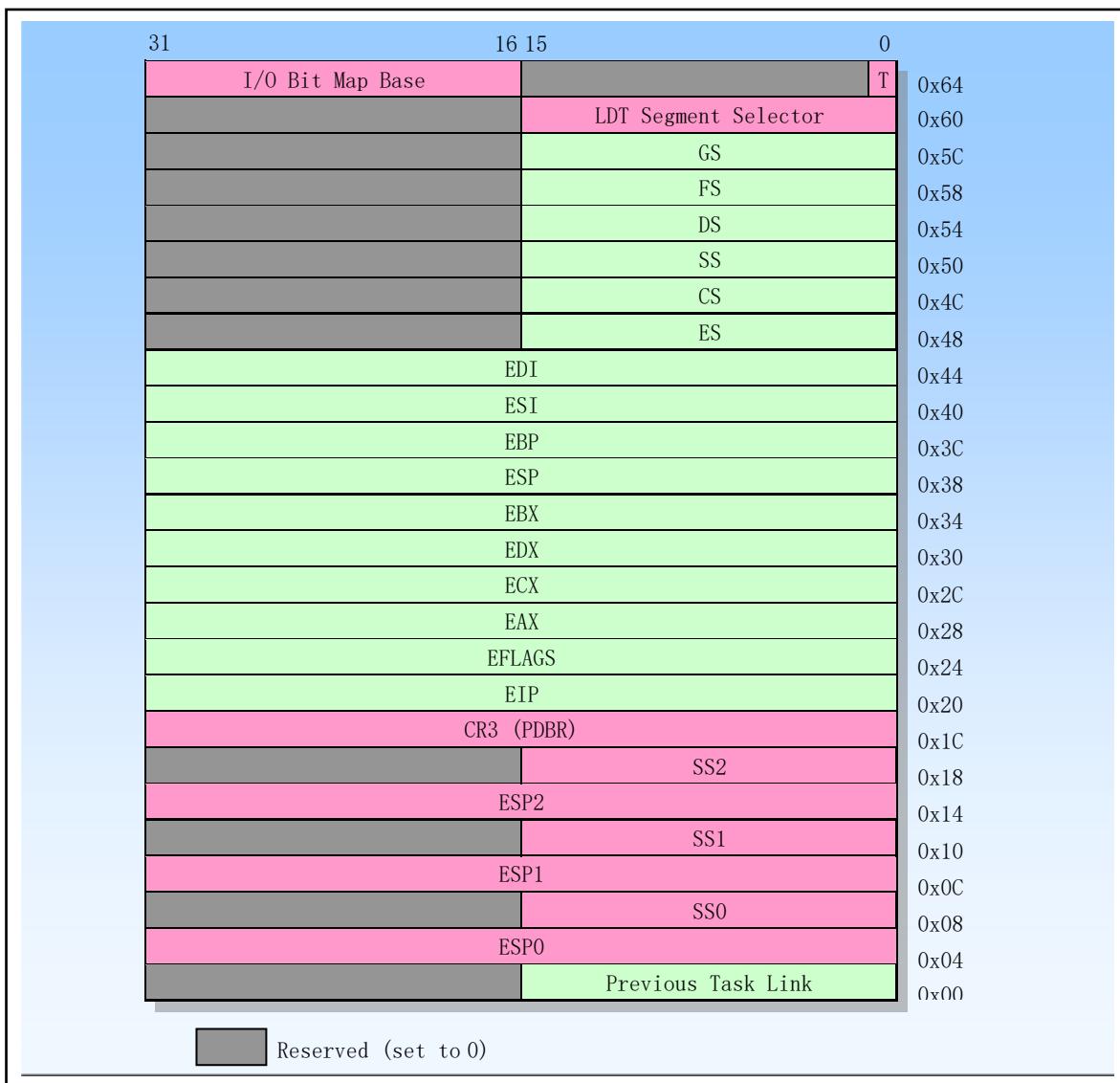


図8-14 タスクステータスセグメントTSSの情報

All information required by the CPU management task is stored in a special type of segment, task state segment (TSS). The figure shows the TSS format for performing the 80386 task.

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セレクタ、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ため、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 Function Description

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。

Type	User ID	Group ID
Process	gid - User ID, indicating the user who owns the process.	gid - the group ID that indicates the user group that owns the process.
Efficient	euid - A efficient user ID indicating the access rights to the file.	egid - the efficient group ID. Indicate the permission to access the file.
Saved	suid - the saved user ID. When the set-user-ID flag of the executable file is set, the suid of the execution file is saved in the suid. Otherwise suid is equal to the euid of the process.	sgid - the saved group ID. When the set-group-ID flag of the execution file is set, the gid of the execution file is stored in the sgid. Otherwise sgid is equal to the process's egid.

The uid and gid of the process are the user ID and group ID of the process owner, that is, the real user ID (ruid) and the real group ID (rgid) of the process. Superusers can modify them using the functions `set_uid()` and `set_gid()`. The effective user ID and effective group ID are used for permission judgment when the process accesses the file.

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ー、実ユーザー・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ー 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なるプログラムです。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る。

**8.10.2** 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。

### 8.10.3 Code comments

プログラム 8-9 linux/kernel/sys.c

```

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

// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.

```

```

// <linux/tty.h> ttyヘッダーファイルは、tty_io、シリアルのパラメータと定数を定義しています。
// communication.
// <linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。
// commonly used functions of the kernel.
// <linux/config.h> カーネル設定用のヘッダーファイルです。キーボード言語やハードディスクを定義する
// type (HD_TYPE) options.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
// segment register operations.
// <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。
// the process.
// <sys/utsname.h> システム名構造体のヘッダーファイルです。
// <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。
// resources used by processes.
// <string.h> 文字列のヘッダーファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。7 #include <errno.h>
8
9 #include <linux/sched.h>
10 #include <linux/tty.h>
11 #include <linux/kernel.h>
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 * The timezone where the local system is located. Used as a default by some
22 * programs who obtain this value by using gettimeofday.
23 */
24 struct timezone sys_tz = { 0, 0 };
25
26 extern int session_of_pgrp(int pgrp); 27
27 // 日付と時刻を取得します ( ftime - fetch time ) 。
28 // 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。
29 int sys_ftime()
30 {
31     return -ENOSYS;
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     return -ENOSYS;
46 }
47
// 端末の回線設定情報を取得します。
48 int sys_getty()
49 {
50     return -ENOSYS;
51 }
52
int sys_rename()

53 {
54     return -ENOSYS;
55 }
56
57
58 int sys_prof()
59 {
60     return -ENOSYS;
61 }
62
63 /*
64 * This is done BSD-style, with no consideration of the saved gid, except
65 * that if you set the effective gid, it sets the saved gid too. This
66 * makes it possible for a setgid program to completely drop its privileges,
67 * which is often a useful assertion to make when you are doing a security
68 * audit over a program.
69 *
70 * The general idea is that a program which uses just setregid() will be
71 * 100% compatible with BSD. A program which uses just setgid() will be
72 * 100% compatible with POSIX w/ Saved ID's.
73 */
// Set the real and/or effective group ID (gid) of the current task. If the task does not have
// superuser privileges, then only its real group ID and effective group ID can be swapped.
// If the task has superuser privileges, you can set the effective and real group IDs arbitrarily,

```

```
// and the saved gid (sgid) is set to a effective gid (egid). The real group ID (rgid) refers
// to the current gid of the process.
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     else
81         return(-EPERM);
```

```

82      }
83      if (egid>0) {
84          if ((current->gid == egid) ||
85              (current->egid == egid) ||
86              suser()) {
87                  current->egid = egid;
88                  current->sgid = egid;
89          } else
90              return (-EPERM);
91      }
92      return 0;
93 }
94
95 /*
96 * setgid() is implemented like SysV w/SAVED_IDS
97 */
98 int sys_setgid(int gid) 99 {
99
100     if (suser()) {
101         current->gid = current->egid = current->sgid = gid;
102     else if ((gid == current->gid) || (gid == current->sgid))
103         current->egid = gid;
104     else
105         return -EPERM;
106     return 0;
107 }
108
109 // プロセスの課金をオンまたはオフにする
110 int sys_acct()
111 {
112     return -ENOSYS;
113 }
114
115 // 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。
116 int sys_phys()
117 {
118     return -ENOSYS;
119 }
120
121 int sys_lock()
122 {
123     return -ENOSYS;
124 }
125
126 int sys_mpx()
127 {
128     return -ENOSYS;

```

127 }  
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、ユーザー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);      // Verify mem capacity is sufficient (4 bytes).
141         put_fs_long(i, (unsigned long *)tloc);
142     }
143     return i;
144 }
145
146 /*
147 * Unprivileged users may change the real user id to the effective uid
148 * or vice versa. (BSD-style)
149 *
150 */
151 * When you set the effective uid, it sets the saved uid too. This
152 * makes it possible for a setuid program to completely drop its privileges,
153 * which is often a useful assertion to make when you are doing a security
154 * audit over a program.
155 */
156 * The general idea is that a program which uses just setreuid() will be
157 * 100% compatible with BSD. A program which uses just setuid() will be
158 * 100% compatible with POSIX w/ Saved ID's.
159 */
160
// 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
int sys_setreuid(int ruid, int euid)

```

```
159
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     else
169         return(-EPERM);
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);
180             }
181         }
182     return 0;
183 }
184 /*
185 * setuid() is implemented like SysV w/ SAVED_IDS
186 *
187 * Note that SAVED_ID's is deficient in that a setuid root program
188 * like sendmail, for example, cannot set its uid to be a normal
189 * user and then switch back, because if you're root, setuid() sets
190 * the saveduid too. If you don't like this, blame the bright people
191 * in the POSIX committee and/or USG. Note that the BSD-style setreuid()
192 * will allow a root program to temporarily drop privileges and be able to
193 * regain them by swapping the real and effective uid.
194 */
195 // 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権限を持っていない場合は、setuid()
196 // 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
197 // がスーパーユーザー権限を持っている場合、ruid、euid、suidには
198 int sys_setuid(int uid)

```

```

199 {
200     if (suser()) {
201         current->uid = current->euid = current->suid = uid;
202     } else if ((uid == current->uid) || (uid == current->suid))
203         current->euid = uid;
204     else
205         return -EPERM;
206     return(0);
207 }
208
209 // 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。
210 // 1970年1月1日のGMT 00:00:00から。
211 // 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
212 // 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で
213 // す。
214 // 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

```

```
// 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でユーザ空間の値にアクセスする。関数パラメタで提供される現在時刻の値  
// 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  
// /秒単位です。
```

```
209 207 int sys_stime(long * tptr) 208  
210 {  
211     if (!suser())  
212         return -EPERM;  
213     startup_time = get_fs_long((unsigned long *)tptr) - jiffies/HZ;
```

```

212     jiffies_offset = 0;
213     return 0;
214 }
215
// // Get the current task runtime statistics.
// Returns the task runtime statistics of the tms structure at the user data space pointed to
// by tbuf. The tms structure includes the process user runtime, kernel runtime, childuser
// runtime, and child kernel runtime. The return value of the function is the ticks that the
// system runs to the current time.
216 int sys_times(struct tms * tbuf)
217 {
218     if (tbuf) {
219         verify_area(tbuf, sizeof *tbuf);
220         put_fs_long(current->utime, (unsigned long *)&tbuf->tms_utime);
221         put_fs_long(current->stime, (unsigned long *)&tbuf->tms_stime);
222         put_fs_long(current->cutime, (unsigned long *)&tbuf->tms_cutime);
223         put_fs_long(current->cstime, (unsigned long *)&tbuf->tms_cstime);
224     }
225     return jiffies;
226 }
227
// // Set the program's end position in memory.
// When the value of the parameter end_data_seg is reasonable and the system does have enough
// memory and the process does not exceed its maximum data segment size, the function sets the
// value specified by end_data_seg at the end of the data segment. This value must be greater
// than the end of the code and less than 16KB of the end of the stack. The return value is
// the new end value of the data segment (if the return value is different from the required
// value, an error has occurred). This function is not directly called by the user, but is wrapped
// by the libc library function, and the return value is not the same.
228 int sys_brk(unsigned long end_data_seg)
229 {
// If the parameter is greater than the end of the code and is less than (stack - 16KB), set
// the new data segment end value.
230     if (end_data_seg >= current->end_code &&
231         end_data_seg < current->start_stack - 16384)
232         current->brk = end_data_seg;
233     return current->brk;           // Returns the current data segment end value.
234 }
235
236 /*
237 * This needs some heavy checking...
238 * I just haven't get the stomach for it. I also don't fully
239 * understand sessions/pgrep etc. Let somebody who does explain it.
240 */
241
242 * OK, I think I have the protection semantics right.... this is really
243 * only important on a multi-user system anyway, to make sure one user
244 * can't send a signal to a process owned by another. -TYT, 12/12/91
245 */
246
// // 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

```

```

// same session. In this case, the parameter pgid specifies the existing process group ID to
// join, and the session ID of the group must be the same as the process to be joined (L263).
245 int sys_setpgid(int pid, int pgid)
246 {
247     int i;
248
// If the parameter pid is 0, the pid is set to the pid of the current process. If the parameter
// pgid is 0, then pgid is also the pid of the current process. If pgid is less than 0, an invalid
// error code is returned.
249     if (!pid)
250         pid = current->pid;
251     if (!pgid)
252         pgid = current->pid;
253     if (pgid < 0)
254         return -EINVAL;
// Scan the task array for the task with the specified process pid. If the process with the
// 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
// 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。

```

275

```

279         return -EPERM;
280     current->leader = 1;
281     current->session = current->pgrp = current->pid;
282                         // 現在のプロセスには制御端子がありません。
283
284 * Supplementary group ID's
285 */
286
287     //// 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。
288     // 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。
289     // 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
290     //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
291     // グループIDは配列アイテムに保存されます。
292     // 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。
293     // 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
294     // 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。
295
296 int sys_getgroups(int gidsetsize, gid_t *grouplist) 297 {
297     int i;
298
299     // 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
300     // 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
301     // 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
302     // 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
303     // 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
304     // 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
305     // コピーされたユーザーグループIDの数を返します。
306
307     if (gidsetsize)
308         verify_area(grouplist, sizeof(gid_t) * gidsetsize);
309
310     for (i = 0; (i < NGROUPS) && (current->groups[i] != NOGROUP);
311          i++, grouplist++) {
312         if (gidsetsize) {
313             if (i >= gidsetsize)
314                 return -EINVAL;
315             put_fs_word(current->groups[i], (short *) grouplist);
316         }
317     }
318     return(i);           // Returns the number of user group ids.
319
320     // 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。
321     // 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
322     // ユーザグループのIDを含むユーザ空間のキャッシュ。307
323
324 int sys_setgroups(int gidsetsize, gid_t *grouplist) 325 {
325     int i;

```

310

```
// 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  
// 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  
// グループ[NGROUPS]配列の容量を超えてます。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  
311 // 関数は0を返します。

```

312     if (!suser())  

313         return -EPERM;  

314     if (gidsetsize > NGROUPS)  

315         return -EINVAL;  

316     for (i = 0; i < gidsetsize; i++, grouplist++) {  

317         current->groups[i] = get_fs_word((unsigned short *) grouplist);  

318     }  

319     if (i < NGROUPS)  

320         current->groups[i] = NOGROUP;  

321     return 0;  

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     int      i;
```

// 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。

```

328     if (grp == current->egid)  

329         return 1;  

330  

331     for (i = 0; i < NGROUPS; i++) {  

332         if (current->groups[i] == NOGROUP)  

333             break;  

334         if (current->groups[i] == grp)  

335             return 1;  

336     }  

337     return 0;  

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

339 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".

340 static struct utsname thisname = {

```
341     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     if (!name) return -ERROR;  
347 }
```

```
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     return 0;
352 }
353 */
354 /*  
355 * Only sethostname; gethostname can be implemented by calling uname()  
356 */
357 int sys_sethostname(char *name, int len) 358
358 {
359     int     i;
360 // cannot exceed the maximum length MAXHOSTNAMELEN.
```

```

361     if (!suser())           // ユーザー権限がない場合
362         return -EPERM;
363     if (len > MAXHOSTNAMELEN) // ホスト名長さがMAXHOSTNAMELENを超える場合
364         return -EINVAL;
365     for (i=0; i < len; i++) {
366         if ((thisname.nodename[i] = get_fs_byte(name+i)) == 0)
367             break;
368     }
// After the copy is completed, if the string provided by the user does not contain NULL
// characters, if the length of the copied hostname does not exceed MAXHOSTNAMELEN, a NULL
// is added after the host name string. If MAXHOSTNAMELEN characters have been filled, change
// the last character to NULL.
369     if (thisname.nodename[i]) {
370         thisname.nodename[i]>MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
371     }
372     return 0;
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,           // Current (soft) limits.
381                 (unsigned long *) rlim);
382     put_fs_long(current->rlim[resource].rlim_max,           // System (hard) limits.
383                 ((unsigned long *) rlim)+1);
384     return 0;
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
391 // 元の処理を「新しい」構造体の情報に変更し、0を返します。
392     if (resource >= RLIM_NLIMITS)
393         return -EINVAL;
394     old = current->rlim + resource;           // old = current->rlim[resource]
395     new.rlim_cur = get_fs_long((unsigned long *) rlim);
396     new.rlim_max = get_fs_long(((unsigned long *) rlim)+1);
397     if (((new.rlim_cur > old->rlim_max) ||
398          (new.rlim_max > old->rlim_max)) &&
399          !suser())
400         return -EPERM;
401     *old = new;
402     return 0;
403 }

404 /*
405 * It would make sense to put struct rususage in the task_struct,
406 * except that would make the task_struct be *really big*. After
407 * task_struct gets moved into malloc'ed memory, it would
408 * make sense to do this. It will make moving the rest of the information
409 * a lot simpler! (Which we're not doing right now because we're not
410 * measuring them yet).
411 */
412

// 指定されたプロセスのリソース使用情報を取得します。
// このシステムコールは、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_getusage(int who, struct rusage *ru) 413 {。
414     struct rusage r;
415     unsigned long *lp, *lpend, *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
417 // 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。
418     if (who != RUSAGE_SELF && who != RUSAGE_CHILDREN)
419         return -EINVAL;
420     verify_area(ru, sizeof *ru);
421     memset((char *) &r, 0, sizeof(r));           // at the end of include/strings.h

// 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ラ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。
422 // jiffies_offsetは、システムパラメータの誤差調整です。
423     if (who == RUSAGE_SELF) {
424         r.ru_utime.tv_sec = CT_TO_SECS(current->utime);
425         r.ru_utime.tv_usec = CT_TO_USECS(current->utime);
426         r.ru_stime.tv_sec = CT_TO_SECS(current->stime);
427         r.ru_stime.tv_usec = CT_TO_USECS(current->stime);
428     } else {
429         r.ru_utime.tv_sec = CT_TO_SECS(current->cutime);
430         r.ru_utime.tv_usec = CT_TO_USECS(current->cutime);
431         r.ru_stime.tv_sec = CT_TO_SECS(current->cstime);
432         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
432 // ユーザースペースのルーズさを解消し、0を返す。
433     lp = (unsigned long *) &r;
434     lpend = (unsigned long *) (&r+1);
435     dest = (unsigned long *) ru;
436     for (; lp < lpend; lp++, dest++)
437         put_fs_long(*lp, dest);
438     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。
```

```

// 現在のシステムのティックを秒とマイクロ秒で表現します。これらは以下で定義されています。
// include/linux/sched.hファイルを参照してください。Jiffies_offsetは、システムティックの誤差調整です。
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));
446         put_fs_long(CT_TO_USECS(jiffies+jiffies_offset),
447                     ((unsigned long *) tv)+1);
448     }
449     if (tz) {
450         verify_area(tz, sizeof *tz);
451         put_fs_long(sys_tz.tz_minuteswest, ((unsigned long *) tz));
452         put_fs_long(sys_tz.tz_dsttime, ((unsigned long *) tz)+1);
453     }
454     return 0;
455 }
456 */
457 /*
458 * The first time we set the timezone, we will warp the clock so that
459 * it is ticking GMT time instead of local time. Presumably,
460 * if someone is setting the timezone then we are running in an
461 * environment where the programs understand about timezones.
462 * This should be done at boot time in the /etc/rc script, as
463 * soon as possible, so that the clock can be set right. Otherwise,
464 * various programs will get confused when the clock gets warped.
465 */
// システムの現在時刻を設定します。
// 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
// をユーザーデータ領域の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。
// 両方とも空の場合は何もせず、0を返します。
466 int sys_settimeofday(struct timeval *tv, struct timezone *tz) 467 {
468     static int      firsttime = 1;
469     void           adjust_clock();
470
// 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
// 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、つまりユーザーのタイムゾーン構造をコピーする
// 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
//     if (!suser())
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。

```

471
472         return -EPERM;
473     if (tz) {
474         sys_tz.tz_minuteswest = get_fs_long((unsigned long *) tz);
475         sys_tz.tz_dsttime = get_fs_long(((unsigned long *) tz)+1);
476         if (firsttime) {
477             firsttime = 0;
478             if (!tv)
479                 adjust_clock();
480         }
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);
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     return 0;
492 }
493
494 /*
495 * Adjust the time obtained from the CMOS to be GMT time instead of
496 * ローカルタイム
497 */
498
499 * This is ugly, but preferable to the alternatives. Otherwise we
500 * would either need to write a program to do it in /etc/rc (and risk
501 * confusion if the program gets run more than once; it would also be
502 * hard to make the program warp the clock precisely n hours) or
503 * compile in the timezone information into the kernel. Bad, bad....
504 *
505 * XXX Currently does not adjust for daylight savings time. May not
506 * は、BIOSの賢さ(馬鹿さ?)に応じて、何かをする必要があります。
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 * network.... - TYT, 1/1/92
509 */
510
// 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。
511 void adjust_clock()
512 {
513     startup_time += sys_tz.tz_minuteswest*60;
514 }
515
// 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。
516 int sys_umask(int mask)
517 {
518     int old = current->umask;
519     current->umask = mask & 0777;
520     return (old);
521 }
522

```

## 8.11 vsprintf.c

### 8.11.1 Function Description

The program mainly includes the vsprintf() function, which formats the parameters and outputs them to the buffer. Since this function is a standard function in the C library, there is basically no content related to the working principle of the kernel, so it can be skipped. In order to understand its application in the kernel, you can directly read the instructions of the function after the code. Please also refer to the C library function manual for how to use the vsprintf() function.

### 8.11.2 Code Comments

プログラム 8-10 linux/kernel/vsprintf.c

```

1  /*
2  *  linux/kernel/vsprintf.c
3  *
4  *  (C) 1991 Linus Torvalds
5  */
6
7 /* vsprintf.c -- Lars Wirzenius & Linus Torvalds. */
8 /*
9 * Wirzenius wrote this portably, Torvalds fucked it up :-)
10 */

// 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
// 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。
// 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
// 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）、次のように
// 認めています。
// 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
// 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
// います。
// 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ページ
// は http://liw.iki.fi/liw/
11
// #include <stdarg.h> 標準の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。
//      form of macros. It mainly describes a type (va_list) and three macros (va_start, va_arg
//      and va_end) for the vsprintf, vprintf, vfprintf functions.
// #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
// います。
//      string operations.
12 #include <stdarg.h>
13 #include <string.h>
14
15 /* we use this so that we can do without the ctype library */
16 #define is_digit(c) ((c) >= '0' && (c) <= '9') // check if it's a digital char.
17
// 文字列を整数に変換します。入力は、数値の文字列へのポインタです。

```

// 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。

```
18 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     return i;
25 }
26
27 // 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。
28 #define ZEROPAD 1           /* pad with zero */
29 #define SIGN    2           /* unsigned/signed long */
30 #define PLUS    4           /* show plus */
31 #define SPACE   8           /* space if plus */
32 #define LEFT    16          /* left justified */
33 #define SPECIAL 32          /* 0x */
34 #define SMALL   64          /* use 'abcdef' instead of 'ABCDEF' */
35
36 // 割り算の演算。入力 : nは配当、baseは除数、結果 : nは商。
37 // 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。
38
39 #define do_div(n,base) ({ 36
40     int res; ..^w^.. )
41
42     37 asm ("divl %4":=a" (n),"=d" ( res): "0" (n), "1" (0), "r" (base)); 38 res; })
43
44     // 整数を指定された基底の文字列に変換します。
45     // 入力: num - 整数; base - 基底; size - 文字列の長さ。
46     // precision - 数値の長さ (精度) 、 type - 型のオプション。
47     // 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
48     // 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。
49     , int type)
50 }
```

```

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42 {
43     char c, sign, tmp[36];
44     const char *digits="0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ";
45     int i;
46
47     // If 'type' indicates a lowercase letter, a lowercase set of letters is defined. If 'type'
48     // 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
49     // 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
50     // 2-32の間のベースを持つ//数字。
51     if (type&SMALL) digits="0123456789abcdefghijklmnopqrstuvwxyz";
52     if (type&LEFT) type &= ~ZEROPAD;
53     if (base<2 || base>36)
54         return 0;
55
56     // '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。
57     // '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
58     // 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

```

52

// 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。

```

53     c = (type & ZEROPAD) ? '0' : ' ';
54     if (type&SIGN && num<0)
{ 53       sign='-' ;
54       num = -num;
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。

```

57     // 結果の前にゼロを置く)。
58     if (sign) size--;
59     if (type&SPECIAL)
60         if (base==16) size -= 2;
61     else if (base==8) size--;
62
63     // 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。
64     // 与えられた基底に応じてもし、桁数が精度よりも大きい場合は
65     // 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
66     // 文字が格納されます。
67
68     i=0;
69     if (num==0)
70         tmp[i++]='0';
71     else while (num!=0)
72         tmp[i++]=digits[do_div(num, base)];
73     if (i>precision) precision=i;
74     size -= precision;

75     // From here on, the resulting conversion result is gradually formed and temporarily placed
76     // in the string str. If there is no zero-fill and left-aligned flags in the 'type', Spaces
77     // indicated by the remaining width is first filled in str. If the sign is needed, store the
78     // sign symbol in it.
79
80     if (!(type&(ZEROPAD+LEFT)))
81         while(size-->0)
82             *str++ = ' ';
83     if (sign)
84         *str++ = sign;
85
86     // If the 'type' indicates a special conversion, a '0' is placed for the first location the
87     // octal result; '0x' is stored for the hexadecimal value.
88
89     if (type&SPECIAL)
90         if (base==8)
91             *str++ = '0';
92         else if (base==16) {
93             *str++ = '0';
94             *str++ = digits[33]; // 'X' 或' x'
95         }
96
97     // If there is no left adjust flag in the 'type', the c ('0' or space) is stored in the remaining
98     // width, see line 51.
99
100    if (!(type&LEFT))
101        while(size-->0)
102            *str++ = c;
103
104    // At this time, i holds the number of digits of num. If the number of digits is less than the
105    // precision, put (precision - i) '0' in str. Then the converted numeric characters are also
106    // 記入したスト i の合計です。
107
108    while(i<precision--)
109        *str++ = '0';
110    while(i-->0)
111        *str++ = tmp[i];
112
113    // 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。
114
115    // '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。
116
117    while(size-->0)
118        *str++ = ' ';
119
120    return str;           // Returns the pointer to the end of the converted string.
121 }
```

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 len;

93     int

94     vsprintf(cha

95         \*bu

96         con

97         char

98         \*Fm

99         va\_i

100         Tis

101         args)

102     {

```

94
95     int i;
96     char * str;           // Used to hold strings during the conversion.
97     char *s;
98     int *ip;
99
100    int flags;           /* flags to number() */
101
102    int field_width;      /* width of output field */
103    int precision;        /* min. # of digits for integers; max
104                                number of chars for from string */
105    int qualifier;         /* 'h', 'l', or 'L' for integer fields */
106
// 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。
// 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。
// 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。
// 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
// ディレクティブはstrに順番に格納されます。
107    for (str=buf ; *fmt ; ++fmt)
108    {
109        if (*fmt != '%') {
110            *str++ = *fmt;
111            continue;
112        }
113
// 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。
114        /* process flags */
115        flags = 0;
116        repeat:
117            ++fmt;           /* this also skips first '%' */
118            switch (*fmt) {
119                case '-': flags |= LEFT; goto repeat;
120                case '+': flags |= PLUS; goto repeat;
121                case ' ': flags |= SPACE; goto repeat;
122                case '#': flags |= SPECIAL; goto repeat;
123                case '': flags |= ZEROPAD; goto repeat;
124            }
125
// 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
// 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
// '*'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。
// 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
// '-' (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。
// 絶対値としての幅の値。
126        /* get field width */
127        field_width = -1;
128        if (is_digit(*fmt))
129            field_width = skip_atoi(&fmt);
130        else if (*fmt == '*') {

```

```
131             /* it's the next argument */      // bug here, should add "++fmt;"  
132             field_width = va_arg(args, int);  
133             if (field_width < 0) {  
134                 field_width = -field_width;  
134             flags |= LEFT;  
135         }  
136     }  
137  
// 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。  
// 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  
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。  
// 精度欄が文字'*'であれば、次のパラメータで  
// the field precision value is taken as 0.  
/* get the precision */
```

```

138     precision = -1;
139     if (*fmt == '.' ) {
140         ++fmt;
141         if (is digit(*fmt))
142             precision = skip atoi(&fmt);
143         else if (*fmt == '*' ) {
144             /* it's the next argument */ // should add ++fmt;
145             precision = va arg(args, int);
146         }
147         if (precision < 0)
148             precision = 0;
149     }
150 }
151

// 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
// h、l、Lの//は、リストの後の説明を参照してください。
153     /* get the conversion qualifier */
154     qualifier = -1;
155     if (*fmt == 'h' || *fmt == 'I' || *fmt == 'L') {
156         qualifier = *fmt;
157         ++fmt;
158     }

// 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。
// 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。
// このとき、フラグフィールドが左寄せでないことを示している場合は、そのフィールドの前に
// 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
// 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す
// パラメータの文字です。
160     switch (*fmt) {
161     case 'c':
162         if (!(flags & LEFT))
163             while (--field_width > 0)
164                 *str++ = ' ';
165         *str++ = (unsigned char) va arg(args, int);
166         while (--field_width > 0)
167             *str++ = ' ';

```

167  
168      break;

```
// 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  
// パラメーター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  
// 'width - string length' spaces after the string.  
case 's':
```

```

169
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++;
182     while (len < field_width--)
183         *str++ = ' ';
184     break;
185
// If the format character is 'o', it means that the corresponding parameter needs to be
// 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         break;
190
191
// 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。
// 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
// 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。
192     case 'p':
193         if (field_width == -1) {
194             field_width = 8;
195             flags |= ZEROPAD;
196         }
197         str = number(str,
198                     (unsigned long) va_arg(args, void *), 16,
199                     field_width, precision, flags);
200         break;
201
// 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。
// を16進数で出力します。'x' は小文字を意味します。

```

```
202         case 'x':
203             flags |= SMALL;
204         case 'X':
205             str = number(str, va_arg(args, unsigned long), 16,
206                             field_width, precision, flags);
207             break;
208
// 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         case 'd':
209
210         case 'i':
211             flags |= SIGN;
212         case 'u':
213             str = number(str, va_arg(args, unsigned long), 10,
214                             field_width, precision, flags);
215             break;
216
217         // 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
218         // 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。
219         // 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
220         // ポインタが指す位置に。
221         case 'n':
222             ip = va_arg(args, int *);
223             *ip = (str - buf);
224             break;
225
226         // 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
227         // 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
228         // means that it has been processed to the end of the format string, then exit the loop.
229         default:

```

```
221     if (*fmt != '%')  
222         *str++ = '%';  
223     if (*fmt)  
224         *str++ = *fmt;  
225     else  
226         --fmt;  
227         break;  
228     }  
229 }  
230 *str = '\0';           // add null to the end of the output string.  
231 return str - buf;    // return string length.  
232 }  
233 }  
234 }
```

---

### 8.11.3 Information

#### 8.11.3.1 Format string of vsprintf()

The vsprintf() function is one of the printf() series. These functions all produce formatted output: A format string fmt that determines the output format is received, and the parameters are formatted with the format string to produce a formatted output. The function declaration form is as follows:

```
int vsprintf(char *buf, const char *fmt, va_list args)
```

---

The other functions in the printf() series declare a form similar to this. Printf will send the output directly to the standard output device stdout (the display / console), so there is no first parameter (buffer pointer) in the above declaration. cprintf also sends the output to the console. fprintf sends the output to a file, so the first argument will be a file handle. The printf with a 'v' character (for example, vfprintf) indicates that the arguments

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. The format string

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. The format indicator string

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]

---

Each conversion indication string needs to start with a percent sign (%). The meaning of each part is as follows:

- [flags] is an optional sequence of flag characters;
- [width] is an optional width indicator;
- [. prec] is an optional precision indicator;
- [|h|l|L] Is an optional input length modifier;
- [type] is a conversion type character (or a conversion indicator).

- flags control output alignment, numeric symbols, decimal points, trailing zeros, binary, octal, or hexadecimal, see the notes in lines 27-33 above. The flag characters and their meanings are as follows:

'#' 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。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 specifies the output string width, which specifies the minimum width value of the field. If the

result of the conversion is smaller than the specified width, then the left side (or the right side, if the right adjustment flag is given) needs to be filled with spaces or zeros (determined by the flags) and so on. In addition to using numbers to specify the width field, you can also use '\*' to indicate that the width of the field is given by the next integer parameter. When the width of the conversion value is greater than the width specified by width,

は、いかなる状況でも結果を切り捨てるはありません。フィールドの幅が拡大され、結果が完全に表示されます。

- precision is the number that describes the minimum number of output. For d, I, o, u, x, and X conversions, the precision value indicates the number of digits at least. For e, E, f, and F, this value indicates the number of digits that appear after the decimal point. For g or G, indicate the maximum number of significant digits. For s or S conversions, the precision value specifies the maximum number of characters in the output string.

- The length modifier indicator describes the output type form after the integer conversion. In the following description, the 'integer number conversion' represents d, i, o, u, x or X conversion.

'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' Indicates that the subsequent integer conversion corresponds to a signed integer or unsigned short integer parameter.

'l' Indicates that the subsequent integer conversion corresponds to a long integer or unsigned long integer argument.

'll' Indicates that the subsequent integer conversion corresponds to a long long integer or unsigned long long integer argument.

'L' Indicates that the e, E, f, F, g or G conversion result corresponds to a long double precision parameter.

- type is the format of the input parameter type and output that are accepted. The meaning of each conversion indicator is as follows:

'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、出力する必要の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、出力する必要の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、精密度以上の場合は、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、その結果を示す。

が出力されます。

s' 入力が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、変換は行われないことを示す。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 Function Description

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された場合、dsレジスタが指すカーネルデータセグメント内にあることになります。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。

8.12.2 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。

### 8.12.3 Code Comments

プログラム 8-11 linux/kernel/printk.c

```

1  /*
2  * linux/kernel/printk.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8 * When in kernel-mode, we cannot use printf, as fs is liable to
9 * point to 'interesting' things. Make a printf with fs-saving, and
10 * all is well.
11 */

```

//<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。

// of macros. It mainly describes one type (va\_list) and three macros (va\_start, va\_arg and  
// va\_end) for the vsprintf, vprintf, and vfprintf functions.

//<stddef.h> 標準定義のヘッダファイルです。NULL, offsetof(TYPE, MEMBER)が定義されています。

```
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義  
が含まれています。  
//     used functions of the kernel.  
12 #include <stdarg.h>  
13 #include <stddef.h>  
14
```

```
15 #include <linux/kernel.h>
16
17 static char buf[1024];           // Temporary buffer for display.
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。
// は、実際には文字ポインタ型です。
26
// 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。
27     va_start(args, fmt);
28     i=vsprintf(buf, fmt, args);
29     va_end(args);
30     console_print(buf);          // chr_drv/console.c, line 995.
31     return i;
32 }
```

## 8.13 panic.c

### 8.13.1 Function Description

The `panic()` function is used to display kernel error messages and put the system into an infinite dead loop. In many places in the kernel, this function is called if the kernel code has a serious error during execution. Calling the `panic()` function in many cases is a straightforward approach. This approach follows the UNIX "as concise as possible" principle.

### 8.13.2 Code Comments

プログラム 8-12 linux/kernel/panic.c

```
1 /*
2 *  linux/kernel/panic.c
3 *
4 *  (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * This function is used through-out the kernel (include in mm and fs)
9 * to indicate a major problem.
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。
//      used functions of the kernel.
```

```

// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
11 #include <linux/kernel.h>
12 #include <linux/sched.h>
13
14 void sys_sync(void); /* it's really int */ // fs/buffer.c, line 44.
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: %s\n|r", s);
19     if (current == task[0])
20         printk("In swapper task - not syncing\n|r");
21     else
22         sys_sync();
23     for(;;);
24 }
25

```

## 8.14 Summary

This chapter mainly studies the 12 source files in the linux/kernel directory, and gives the implementation of some of the most important mechanisms in the kernel, including system calls, process scheduling, process replication, and process termination processing.

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。

## 9 Block Device Driver

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・デバイスについては、次の章で説明します。

表 9 - 1  L i n u x  0 .1 2 カ 一 ネ ル の 主 要	Name	Device type	Description

デバイス番号について Major No.			
0	None	None	None
1	ram	Block/Char	Ram devices (virtual disk)
2	fd	Block	floppy device
3	hd	Block	harddisk device
4	ttyx	Char	device (virtual or serial terminal)
5	tty	Char	tty device
6	lp	Char	lp printer

The Linux 0.12 kernel mainly supports three types of block devices: hard disk, floppy disk and memory virtual disk. Since block devices are primarily related to file systems and caches, you can quickly take a look at the contents of the file system chapter before proceeding with this chapter. The source code files covered in this chapter are shown in List 9-1.

リスト 9-1 linux/kernel/blk\_drv

Filename	Size	Last modified time (GMT)	Desc.
<a href="#">Makefile</a>	2759 bytes	1992-01-12 19:49:21	
<a href="#">blk.h</a>	3963 bytes	1991-12-26 20:02:50	
<a href="#">floppy.c</a>	11660 bytes	1992-01-10 03:45:33	
<a href="#">hd.c</a>	8331 bytes	1992-01-16 06:39:10	
<a href="#">11_rw_blk.c</a>	4734 bytes	1991-12-19 21:26:20	
<a href="#">ramdisk.c</a>	2740 bytes	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ーで、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ーhd.c、フロッピーディスクドライバelfloppy.c、メモリー仮想ディスクドライバー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 Main Functions

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利用して、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。バッファーマネージャはまず、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、バッファー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。

**9.1.1** 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。

### 9.1.2 Block Device Requests and Request Queues

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

とがわかります。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[]を使って管理します。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);           // A function pointer of requests.
    struct request * current_request;   // current request structure pointer.
}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)。
```

The first field is a function pointer that is used to manipulate the request item of the corresponding block device. For example, for a hard disk drive, it is `do_hd_request()`, and for a floppy device it is `do_floppy_request()`. The second field is the current request item structure pointer, which is used to indicate the request item currently being processed by the block device. All the request items in the table are set to NULL at the time of initialization.

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。

Main Dev No	Type	Description	Request function
0	None	None	NULL
1	Block/Char	ram, memory dev (ramdisk etc)	<code>do_rd_request()</code>
2	Block	fd, floppy disk device	<code>do_fd_request()</code>
3	Block	hd, hardware disk device	<code>do_hd_request()</code>
4	Char	ttyx device (virtual or serial terminal etc)	NULL
5	Char	tty device	NULL

6	Char	lp printing device	NULL
---	------	--------------------	------

When the kernel issues a block device read or write or other operation request, the ll\_rw\_block() function will create a device request item using the operation function do\_XX\_request() according to the command specified in its parameter and the device no in the data buffer block header, and inserts it into the request queue using elevator algorithm. The 'XX' in the function name can be one of 'rd', 'fd' or 'hd', representing memory, floppy disk and hard disk block devices. The request item queue consists of items in the request item table (array). There are a total of 32 items. The data structure of each request item is as follows:

---

```
struct request {
    int dev;                                // The device no used (-1 means empty).
    int cmd;                                 // Command (READ or WRITE).
    int errors;                             // The nr of errors during operation.
    unsigned long sector;                   // Starting sector. (1 block = 2 sectors)
    unsigned long nr_sectors;                // Read/write sector number.
    char * buffer;                           // Data buffer.
    struct task_struct * waiting;           // The place where the task waits for operation.
    struct buffer_head * bh;                // Buffer header (include/linux/fs.h, 68).
    struct request * next;                  // next request item.
};

extern struct request request[NR_REQUEST]; // Request item array (NR_REQUEST = 32).
```

---

The current request item pointer for each block device, together with the request link list for the device in the request array, constitutes the request queue for the device. Between the items, the next pointer field is used to form a linked list. Therefore, the block device item and the associated request queue form the structure shown in Figure 9-1. The main reason why the request item adopts the array and linked list structure is to satisfy two purposes: First, the array structure of the request item can be used to perform loop operations when searching

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2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。

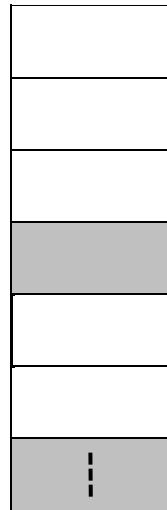


図9-1 デバイステーブルエントリーとリクエスト項目

For a currently idle block device, when the `ll_rw_block()` function establishes the first request item for it, the current request item pointer `current_request` directly points to the newly created request item, and immediately invokes the request function to start the block device read and write operations. When a block device already has a linked list of several request items, `ll_rw_block()` will use the elevator algorithm to insert the newly created request into the appropriate position of the linked list according to the principle of minimum head movement distance.

**9.1.3** 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、書き込み操作用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のために特別に使用します。

#### 9.1.4 Block device access scheduling

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/ライトヘッドの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。

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、反対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す。

9.1.5 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。

### 9.1.6 Block device operation method

- 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見なすことができ、動作ステップは以下のようになります。

- First, the system indicates the C function that the controller should invoke during the interrupt caused by the execution of the command, and then sends a read, write, reset or other operation command to the block device controller;
- When the controller completes the specified command, it will issue an interrupt request signal, which will cause the system to execute the interrupt processing of the block device, and call the specified C function to post-process the read/write or other commands after these commands are finished.

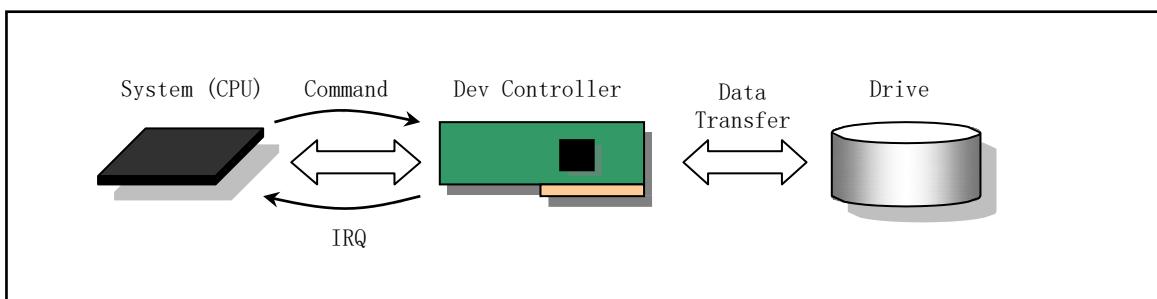


図9-2 システム、ブロックデバイスコントローラ、およびドライブ

For disk write operations, the system needs to wait for the controller to give a response to allow data to be written to the controller after issuing a write command (using `hd_out()`), ie, waiting for the data request service flag DRQ of the controller status register to be set. Once DRQ is set, the system can send a sector of data to the controller buffer.

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ます。まだ読むべきデータがある（デクリメント結果の値が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、ロックデバイスのIOの完了を待ちます。例えば、キャッシングマネージャのリードロック関数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 Function Description

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 Code annotation

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          // nr of device types
5 /*
6 * NR_REQUEST is the number of entries in the request-queue.

```

- 7 \* NOTE that writes may use only the low 2/3 of these: reads
- 8 \* take precedence.
- 9 \*
- 10 \* 32 seems to be a reasonable number: enough to get some benefit

```

11 * from the elevator-mechanism, but not so much as to lock a lot of
12 * buffers when they are in the queue. 64 seems to be too many (easily
13 * long pauses in reading when heavy writing/syncing is going on)
14 */
15 #define NR_REQUEST      32
16
18 17 /*
19 * Ok, this is an expanded form so that we can use the same
20 * request for paging requests when that is implemented. In
21 * paging, 'bh' is NULL, and 'waiting' is used to wait for
22 * read/write completion.
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

23 // 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。

```

24 struct request {
25     int dev;           /* -1 if no request */ // device requested.
26     int cmd;           /* READ or WRITE */
27     int errors;        // count of error.
28     unsigned long sector; // start sector no.
29     unsigned long nr_sectors; // nr sectors needed.
30     char * buffer;    // data buffer.
31     struct task_struct * waiting; // waiting queue of tasks.
32     struct buffer_head * bh; // Buffer header (include/linux/fs.h, 73).
33     struct request * next; // points to the next request.
33 };
34
36 35 /*
37 * This is used in the elevator algorithm: Note that
38 * reads always go before writes. This is natural: reads
39 * are much more time-critical than writes.
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  
// 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ージ）。

```

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);           // request handling function.
48     struct request * current_request;  // current request item.
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ーの配列。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                                // current major number used.
57
59 58 /*
60  * Add entries as needed. Currently the only block devices
61  * supported are hard-disks and floppies.
61 */
62
63 // 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。
64 #if (MAJOR_NR == 1)
65 /* ram disk */
66 #define DEVICE_NAME "ramdisk"
67 #define DEVICE_REQUEST do_rd_request           // request handler.
68 #define DEVICE_NR(device) ((device) & 7)         // Sub-device nr (0 - 7).
69 #define DEVICE_ON(device)                      // Turn on (ram disk doesn't need this).
69 #define DEVICE_OFF(device)
70
71 // 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。
72 #elif (MAJOR_NR == 2)
73 /* floppy */
74 #define DEVICE_NAME "floppy"
75 #define DEVICE_INTR do_floppy                 // device interrupt handler.
75 #define DEVICE_REQUEST do_fd_request          // request handler.
76 #define DEVICE_NR(device) ((device) & 3)         // Sub-device nr (0 - 3).
77 #define DEVICE_ON(device) floppy_on(DEVICE_NR(device))
78 #define DEVICE_OFF(device)
floppy_off(DEVICE_NR(device)) 79
80 // 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。
81 #elif (MAJOR_NR == 3)
82 /* harddisk */
83 #define DEVICE_NAME "harddisk"
84 #define DEVICE_INTR do_hd                     // device interrupt handler.
84 #define DEVICE_TIMEOUT hd_timeout             // device timeout.
85 #define DEVICE_REQUEST do_hd_request          // request handler.
86 #define DEVICE_NR(device) (MINOR(device)/5)   // harddisk no (0, 1).

```

```
87 #define DEVICE_ON(device) // the harddisk always running.  
88 #define DEVICE_OFF(device)  
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
97 (blk_dev[MAJOR_NR].current_request) 97 #define
98 CURRENT_DEV DEVICE_NR(CURRENT->dev)
99
// 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) =
101 NULL; 101 #endif
102 // 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
103 // 0で、SET_INTR()マクロが定義されています。
104 #ifdef DEVICE_TIMEOUT
105 int DEVICE_TIMEOUT = 0;
106 #define SET_INTR(x) (DEVICE_INTR) = (x), DEVICE_TIMEOUT = 200
107 #else
108 #define SET_INTR(x) (DEVICE_INTR) = (x)
109 #endif
110 // シンボルのDEVICE_REQUESTは、引数なし、戻り値なしのスタティックな関数ポインタであること
111 // を宣言しています。
112 static void (DEVICE_REQUEST)(void);
113 // 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
114 // 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
115 // 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
116 // は、バッファブロックのヘッダポインタです。
117 extern inline void unlock_buffer(struct buffer_head * bh) 111 {
118     if (!bh->b_lock)
119         printk(DEVICE_NAME ": free buffer being unlocked\n");
120     bh->b_lock=0;
121     wake_up(&bh->b_wait);
122 }
123
124 // 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。
125 // 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
126
127

```

// 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。

```
20 118 extern inline void end_request(int uptodate) 119 {  

121     DEVICE OFF(CURRENT->dev); // turn off the device.  

122     if (CURRENT->bh) { // the current request item pointer.  

123         CURRENT->bh->b_uptodate = uptodate; // set flag.  

124         unlock buffer(CURRENT->bh);
```

```

124      }
125      if (!uptodate) {                                // upodate flag not set...
126          printk(DEVICE_NAME " I/O error\n|r");
127          printk("dev %04x, block %d\n|r", CURRENT->dev,
128                                         CURRENT->bh->b_blocknr);
129      }
130      wake_up(&CURRENT->waiting);                // Wake up process waiting for the request.
131      wake_up(&wait_for_request);                 // Wake up process waiting for idle request.
132      CURRENT->dev = -1;                          // Release the request item.
133      CURRENT = CURRENT->next;                   // Point to the next request item.
134  }
135

// 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

136 // "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。
137 #ifdef DEVICE_TIMEOUT
138 #define CLEAR_DEVICE_TIMEOUT DEVICE_TIMEOUT = 0;
139 #else
140 #define CLEAR_DEVICE_TIMEOUT
141 #endif
142

// 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

143 // "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。
144 #ifdef DEVICE_INTR
145 #define CLEAR_DEVICE_INTR DEVICE_INTR = 0;
146 #else
147 #define CLEAR_DEVICE_INTR
148 #endif
149

// 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
// 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。
// マクロが定義されています。このマクロは、現在の
// リクエストアイテムです。作業内容は以下の通りです。
// デバイスの現在の要求項目が空(NULL)の場合、そのデバイスには
// 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
// 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
// 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ーが文字化けてしまい、カーネル
// 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用されたバッファブロック
// 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
150 // のメッセージが表示され、機械も停止します。
151 #define INIT_REQUEST \
149 repeat: \
150     if (!CURRENT) {\\                                // no more requests need to processed.
151         CLEAR_DEVICE_INTR \

```

```
152     CLEAR_DEVICE_TIMEOUT \
153     return; \
154 } \
155 if (MAJOR(CURRENT->dev) != MAJOR_NR) \
156     panic(DEVICE_NAME ": request list destroyed"); \
157 if (CURRENT->bh) { \
158     if (!CURRENT->bh->b_lock) \
159         panic(DEVICE_NAME ": block not locked"); \
160 }
161 }
```

---

[162 #endif](#)  
[163](#)  
[164 #endif](#)  
[165](#)

---

## 9.3 hd.c

### 9.3.1 Function description

The hd.c program is the hard disk controller driver. It provides read and write operations to the hard disk controller and block devices, as well as hard disk initialization processing. All functions in the program can be divided into five categories according to their functions:

- Functions that initialize the hard disk and set the data structure information used by the hard disk, such as sys\_setup() and hd\_init();
- The function hd\_out() that sends the command to the hard disk controller;
- The function do\_hd\_request() that handles the current request item of the hard disk;
- C functions called during hard disk interrupt handling, such as read\_intr(), write\_intr(), bad\_rw\_intr(), and recal\_intr(). The do\_hd\_request() function will also be called in read\_intr() and write\_intr();
- The hard disk controller operates auxiliary functions such as controller\_ready(), drive\_busy(), win\_result(), hd\_out(), and reset\_controller().

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）。

(1) 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。

(2) First determine if the current request item exists. If the current request pointer is empty, it indicates that the current hard disk block device has no pending request items, so the program is immediately exited. This is the statement executed in the macro INIT\_REQUEST. Otherwise continue processing the current request item.

(3) verifying the reasonableness of the device number specified in the current request item and the

(4) 要求されたディスクの開始セクタ番号

- (5) calculating the disk track number, head number and cylinder number of the request data according to the information provided by the current request item;
- (6) If the reset flag has been set, the hard disk recalibration flag is also set, and the hard disk is reset, and the "create drive parameter" command (WIN\_SPECIFY) is resent to the controller. This command does not cause a hard disk interrupt;
- (7) If the recalibration flag is set, send the hard disk recalibration command (WIN\_RESTORE) to the controller, and pre-set the C function (recal\_intr()) that needs to be executed in the interrupt caused by the command, and drop out. The main function of the recal\_intr() is to re-execute this function when the controller terminates the command and raises an interrupt.
- (8) If the current request item specifies a write operation, first set the C function to write\_intr() to send a command parameter block of the write operation to the controller. The controller's status register is queried cyclically to determine if the request service flag (DRQ) is set. If the flag is set, it indicates that the controller has "agreed" to receive the data, and then the data in the buffer pointed to by the request item is written into the data buffer of the controller. If the flag is still not set after the loop query times out, the operation failed. The bad\_rw\_intr() function is then called, and it is determined whether to abandon the current request item or to set a reset flag according to the number of occurrences of the error to continue reprocessing the current request item.
- (9) If the current request item is a read operation, set the C function to read\_intr(), and send a read operation command to the controller.

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の中止するか、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が判断される。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（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（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

Time

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

As can be seen from the above analysis, the four most important functions in this program are `hd_out()`, `do_hd_request()`, `read_intr()`, and `write_intr()`. Understand the role of these four functions and understand the operation process of the hard disk drive.

**9.3.2** `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。

### 9.3.3 Code annotation

プログラム 9-2 linux/kernel/blk\_drv/hd.c

```

1 /*
2  * linux/kernel/hd.c
3  *
4  * (C) 1991 Linus Torvalds
5 */
6
7 /*
8  * This is the low-level hd interrupt support. It traverses the
9  * request-list, using interrupts to jump between functions. As
10 * all the functions are called within interrupts, we may not

```

```

11 * sleep. Special care is recommended.
12 *
13 * modified by Drew Eckhardt to check nr of hd's from the CMOS.
14 */
15
// <linux/config.h> カーネル設定用のヘッダーファイルです。キーボード言語やハードディスクを定義する
// type (HD_TYPE) options.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// of the initial task 0, and some embedded assembly function macro statements about the
// descriptor parameter settings and acquisition.
// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する (file, buffer_head,
// m_inode, etc.).
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義
// が含まれています。
// used functions of the kernel.
// <linux/hdreg.h> ハードディスクパラメータのヘッダーファイルです。ハードディスクのレジスタポートへのアカ
// セスを定義します。
// status code, partition table and other information.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
// descriptors/interrupt gates, etc. is defined.
// <asm/io.h> Ioのヘッダーファイルです。の形で、ioポートを操作する関数を定義します。
// a macro's embedded assembler.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
// segment register operations.
16 #include <linux/config.h>
17 #include <linux/sched.h>
18 #include <linux/fs.h>
19 #include <linux/kernel.h>
20 #include <linux/blk.h>
21 #include <asm/system.h>
22 #include <asm/io.h>
23 #include <asm/segment.h>
24
// ハードディスクのメジャー番号記号定数を定義します。ドライバでは、メジャーデバイス番号
// は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。このシンボルはblk.hファイルで使用されま
// す
// に関連する他のシンボル定数やマクロを決定します。
25 #define MAJOR_NR 3           // harddisk major no is 3.
26 #include "blk.h"
27
// Read the CMOS parameter マクロ機能。このマクロは、CMOSのハードドライブ情報を読み取る。
// outb_p, inb_p は include/asm/io.h で定義されているポート入出力マクロで、正確には
28 // init/main.cでCMOSクロック情報を読み取るのと同じマクロです。
29 #define CMOS_READ(addr) ({ \
30     outb_p(0x80|addr, 0x70); \
31     inb_p(0x71); \
32 })
33 /* Max read/write errors/sector */
34 #define MAX_ERRORS    7

```

```
35 #define MAX HD          2           // nr of hard disks supported by the system.  
36  
36 // Recalibration. ハードディスクの割り込みで呼び出される再キャリブレーション関数（311行目）は  
37 // リセット操作時のハンドラーです。  
38 static void recal_intr(void);  
38 // 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
39      // 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)。
40 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
40 // ヘッドを0番のシリンダーに
41 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
42 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。
43 static int reset = 0;
42
43 /*
44 * This struct defines the HD's and their types.
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
46 // 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。
47 struct hd_i_struct {
48     int head, sect, cyl, wpcom, lzone, 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。
49 // 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。
50 #ifdef HD_TYPE
51 struct hd_i_struct hd_info[] = { HD_TYPE };
52 #define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))           // count hd nr.
52 #else
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。
57 // 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。
58 static struct hd_struct {
59     long start_sect;                      // The partition start sector in the hard disk.
60     long nr_sects;                        // The total nr of sectors in the partition.
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
67 // 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。
68 #define port_write(port, buf, nr) \

```

69 \_\_asm ("cld;rep;outsw": : "d" (port), "S"(buf), "c" (nr) : "cx", "si")  
69

```

70 extern void hd_interrupt(void);      // Harddisk interrupt handler (sys_call.s, line 235).
71 extern void rd_load(void);          // The ram disk load function (ramdisk.c, line 71).
72
73 /* This may be used only once, enforced by 'static int callable' */
// 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するために使用され、RAMの仮想ディスクをロードしようとします
// ルートファイルシステムです。

```

//呼び出しを一度だけに制限するために使用されます。

```

// 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のメモリに格納します。
82
83     if (!callable)
84         return -1;
85     callable = 0;
86 #ifndef HD_TYPE                                // Read if HD_TYPE is not defined.
87     for (drive=0 ; drive<2 ; drive++) {
88         hd_info[drive].cyl = *(unsigned short *) BIOS;      // cylinders.
89         hd_info[drive].head = *(unsigned char *) (2+BIOS);    // headers.
90         hd_info[drive].wpcom = *(unsigned short *) (5+BIOS); // write pre-com.
91         hd_info[drive].ctl = *(unsigned char *) (8+BIOS);    // control byte.
92         hd_info[drive].lzone = *(unsigned short *) (12+BIOS); // landing zone.
93         hd_info[drive].sect = *(unsigned char *) (14+BIOS);   // sectors/track.
94     BIOS += 16;                                  // Each hd parameter table is 16 bytes long.
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ドーの番号があるかどうかを確認すると
95 // 0であれば、2台目のハードディスクがあるかどうかがわかります。
96     if (hd_info[1].cyl)
97         NR HD=2;                                // The nr of hard disks is set to 2.
98     else

```

```
99      NR_HD=1;  
100#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;                                // starting sector number.
102        hd[i*5].nr_sects = hd_info[i].head*
103            hd_info[i].sect*hd_info[i].cyl; // total nr of sectors.
104    }
105
106 /*
107     We query CMOS about hard disks : it could be that
108     we have a SCSI/ESDI/etc controller that is BIOS
109     compatible with ST-506, and thus showing up in our
110     BIOS table, but not register compatible, and therefore
111     not present in CMOS.
112
113     Furthermore, we will assume that our ST-506 drives
114     <if any> are the primary drives in the system, and
115     the ones reflected as drive 1 or 2.
116
117     The first drive is stored in the high nibble of CMOS
118     byte 0x12, the second in the low nibble. This will be
119     either a 4 bit drive type or 0xf indicating use byte 0x19
120     for an 8 bit type, drive 1, 0x1a for drive 2 in CMOS.
121
122     Needless to say, a non-zero value means we have
123     an AT controller hard disk for that drive.
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。
128 // システムのハードディスク。
129 if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
130     if (cmos_disks & 0x0f)
131         NR_HD = 2;
132     else
133         NR_HD = 1;
134 else
135     NR_HD = 0;
// 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。
```

```
// array hd[]. First, read the first data block of the hard disk (fs(buffer.c, line 267) by
// 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
// partition structure array hd[]. Finally release the bh buffer.
```

```

139     for (drive=0 ; drive<NR_HD ; drive++) {
140         if (! (bh = bread(0x300 + drive*5, 0))) {           // 0x300、0x305 is dev no.
141             printk("Unable to read partition table of drive %d|r",
142                   drive);
143             panic(")");
144         }
145         if (bh->b_data[510] != 0x55 || (unsigned char)
146             bh->b_data[511] != 0xAA) {                         // check flag 0xAA55
147             printk("Bad partition table on drive %d|r", drive);
148             panic(")");
149         }
150         p = 0x1BE + (void *)bh->b_data;    // partition table is located at 0x1BE.
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);                                // Release the buffer.
156     }
// Now count the total number of data blocks in each partition and save it in the harddisk
// 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
// size array point to this array.
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
// file system.

```



```
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.
163     init_swapping();                          // mm/swap.c, line 199.
164     mount_root();                            // fs/super.c, line 241.
165     return (0);
166 }
167 // Check and cycle to wait for the hard disk controller to be ready.
```

```
// 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。
// コントローラーの状態です。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);           // get status.
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);        // if ERR_STAT is set, read HD_ERROR.
184     return (1);
185 }
```

//// 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。
187 // "register char res asm("ax");" のようになります。
188 static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
189                     unsigned int head,unsigned int cyl,unsigned int cmd,
190                     void (*intr_addr)(void))
190 {
191     register int port asm("dx");           // define a register variable.
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  
 193 // と表示されて停止してしまいます。

```
194     if (drive>1 || head>15)
195         panic("Trying to write bad sector");
196     if (!controller_ready())
197         panic("HD controller not ready");
```

// 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)で指定されたドライブの制御モードを確立します。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。

```
199     SET_INTR(intr_addr);           // do_hd = intr_addr
200     outb_p(hd_info[drive].ctl, HD_CMD); // out control byte.
201     port=HD_DATA;                // (0x1f0)
202     outb_p(hd_info[drive].wpcom>>2, ++port); // Param: write pre-comp. (divided by 4)
203     outb_p(nsect, ++port);        // Param: total nr of r/w sectors.
204     outb_p(sect, ++port);         // Param: starting sector.
205     outb_p(cyl, ++port);          // Param: The cylinder nr low 8 bits.
206     outb_p(cyl>>8, ++port);      // Param: The cylinder nr high 8 bits.
207     outb_p(0xA0 | (drive<<4) | head, ++port); // Param: drive no + head no.
208     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

{。

```
212     unsigned int i;
213     unsigned char c;
```

// 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);           // get main status byte.
216         c &= (BUSY_STAT | READY_STAT | SEEK_STAT);
217         if (c == (READY_STAT | SEEK_STAT))
218             return 0;
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、再読み込み許可) 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     int      i;
227
228     outb(4, HD_CMD);           // 4 is reset byte.
229     for(i = 0; i < 1000; i++) nop();    // wait a while.
230     outb(hd_info[0].ctl & 0x0f, HD_CMD); // normal control byte(allow retry, reread).
231     if (drive_busy())
232         printk("HD-controller still busy\n|r");
233     if ((i = inb(HD_ERROR) != 1)
234         printk("HD-controller reset failed: %02x\n|r", i);
235 }
236

```

//// ハードディスクのリセット操作。

//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 //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れます。

//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。

//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

//コマンド。

237 static void reset\_hd(void) 238

{。

```

239     static int i;
240

```

//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  
//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。  
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  
//割り込み信号を表示します。この時点で、この関数は割込みハンドラによって再び実行されます。  
//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  
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  
//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。  
//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  
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  
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。  
// the do\_hd\_request() function is called again to start processing the request item.



```
241 repeat:  
242     if (reset) {  
243         reset = 0;  
244         i = -1;           // initialize the current hd number.  
245         reset_controller();  
246     } else if (win_result()) {  
247         bad_rw_intr();  
248         if (reset)  
249             goto repeat;  
250     }  
251     i++;                // handling next hd.  
252     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();           // request item processing..
257 }
258
259     //// The default function called by an unexpected hard disk interrupt.
260     // This function is the default C function called in the hard disk interrupt handler when an
261     // unexpected hard disk interrupt occurs. This function is invoked when the called function
262     // pointer is NULL. See kernel/sys_call.s, line 256. The function sets the reset flag after
263     // displaying the warning message, and then continues to invoke the request item function
264     // go_hd_request() and performs a reset processing operation therein.
265 void unexpected_hd_interrupt(void)
266 {
267     printk("Unexpected HD interrupt\n|r");
268     reset = 1;
269     do_hd_request();
270 }
271
272     //// The processing function for handling hard disk read/write failure.
273     // If the number of errors in the read sector operation is greater than or equal to 7, the current
274     // request item is terminated and the process waiting for the request is awake, and the
275     // corresponding buffer update flag is reset, indicating that the data is not updated. If the
276     // number of errors in reading/writing a sector operation has been greater than 3 times, it
277     // is required to perform a controller reset operation (set reset flag).
278 static void bad_rw_intr(void)
279 {
280     if (++CURRENT->errors >= MAX_ERRORS)
281         end_request(0);
282     if (CURRENT->errors > MAX_ERRORS/2)
283         reset = 1;
284 }
285
286     //// 割り込みで呼び出されたリードセクタ機能。
287     // 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。
288     // ハードディスクのリードコマンドの//。の後、コントローラは割込み要求信号を生成します。
289     // 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
290     // 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
291     // 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。
292 static void read_intr(void) 275
293 {
294     // 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
295     // 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
296     // 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
297     // 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。
298     //
299     // 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
300     // 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下の場合は
301     // 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
302     // 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
303     // 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
304     // 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。  
276     if (win_result()) {                                // if there is error...
```

```

277         bad_rw_intr();           // r/w failure handling.
278         do_hd_request();       // continue handling request items.
279         return;
280     }

// 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が呼び出されるたびに、関数ポインタをNULLに設定する。
281 kernel/sys_call.sファイルの251～253行目を参照してください。

282     port_read(HD_DATA, CURRENT->buffer, 256);    // put in buffer.
283     CURRENT->errors = 0;                      // Clear the number of errors.
284     CURRENT->buffer += 512;                     // adjust the buffer pointer.
285     CURRENT->sector++;                        // increase sectors read.
286     if (--CURRENT->nr_sectors) {                // if there is still data to read
287         SET_INTR(&read_intr);                  // points to read_intr() again.
288         return;
289     }

// 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
289 // 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。
290     end_request(1);                            // set data updated flag.
291     do_hd_request();                         // Perform other request operations.
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
296     if (win_result()) {                      // If controller returns error messages,
297         bad_rw_intr();                      // r/w error handling.
298         do_hd_request();                    // continue handling request items.
299         return;
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なります。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     if (--CURRENT->nr_sectors) {           // If there are still sectors to write...
301         CURRENT->sector++;
302         CURRENT->buffer += 512;
303         SET_INTR(&write_intr);           // do_hd points write_intr() again.
304         port_write(HD_DATA, CURRENT->buffer, 256);
305         return;
306     }
307     // 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
308     // 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
309     // 他のリクエストを処理します。
310     end_request(1);                      // set data updated flag.
311     do_hd_request();                   // Perform other request operations.
312 }

313     ///////////////////////////////////////////////////////////////////
314     // 割り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。
315     // 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
316     // 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。
317     // 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。
318     // 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
319     // が継続または処理されます。311
320     static void recal_intr(void) 312 {
321         if (win_result())                  // if error invoke bad_rw_intr()
322             bad_rw_intr();
323         do_hd_request();
324     }

325     ///////////////////////////////////////////////////////////////////
326     // ハードディスクのタイムアウト処理機能。
327     // 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
328     // の後、コントローラが割り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
329     // //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
330     // //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。
331     // 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
332     // 所定の時間（200ティック）以内に、ハードディスクの割り込みハンドラの実行を開始します。
333     // 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。
334     // 機能です。
335
336 void hd_times_out(void)
337 {
338     // 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
339     // 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
340     // 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。
341     // 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
342     // 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
343     // // 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
344     // // は、リクエストアイテムハンドラdo_hd_request()で実行されます。
345
346     if (!CURRENT)
347         return;
348     printk("HD timeout");

```

```
324     if (++CURRENT->errors >= MAX_ERRORS)
325         end_request(0);
326     SET_INTR(NULL); // let do_hd = NULL, time_out = 200
327     reset = 1;
328     do_hd_request();
```

328 }329

```
//// ハードディスクの読み取り/書き込み要求操作を行う。
// 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
// 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。
現在の要求項目にある // セクター番号情報を、コマンドに応じて
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
そのような
// 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
// の操作が最初に行われます。
// リクエストアイテムが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) 。
// 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) 。
// 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
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。
// 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。
// handling. See kernel/sys_call.s, line 235.
```

330 void do\_hd\_request(void)331 {

```
333     int i, r;
334     unsigned int block, dev;
335     unsigned int sec, head, cyl;
336     unsigned int nsect;
```

336

```
// 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
// 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
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
// 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
// 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
// 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
// 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
// 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。次に
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る
パーティションの開始セクタ番号を加算します。
// 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。
// description of the hard disk device number, see Table 6-1 in Section 6.2.3.
```

```
337     INIT_REQUEST;
338     dev = MINOR(CURRENT->dev);
339     block = CURRENT->sector;                      // starting sector.
340     if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
341         end_request(0);
342         goto repeat;                                // in blk.h, line 149.
343     }
344     block += hd[dev].start_sect;
345     dev /= 5;                                     // dev is now the hd no (0, or 1).
```

```
// 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
// セクターナンバー (sec) 、シリンドーナンバー (cyl) 、ヘッドナンバー (head) を計算します。
// 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使
用されています。
// 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。
// 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。
// セクター番号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、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));
348     _asm_( "divl %4": "=a" (cyl), "=d" (head): "0" (block), "1"(0),
349             "r" (hd_info[dev].head));
350     sec++;                                // current track sector number is adjusted.
351     nsect = CURRENT->nr_sectors;          // The number of sectors to read/write.

// 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
// コントローラー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。
352     //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。
353     if (reset) {
354         recalibrate = 1;                  // need to recalibrate.
355         reset_hd();
356         return;
357     }
// この時点での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
358     if (recalibrate) {
359         recalibrate = 0;
360         hd_out(dev, hd_info[CURRENT_DEV].sect, 0, 0, 0,
361                 WIN_RESTORE, &recal_intr);
362         return;
363     }
// 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が判断されます。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  
// data register port HD_DATA.
```

---

```
363     if (CURRENT->cmd == WRITE) {
364         hd_out(dev, nsect, sec, head, cyl, WIN_WRITE, &write_intr);
365         for(i=0 ; i<10000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
```

```

366             /* nothing */;
367         if (!r) {
368             bad_rw_intr();
369             goto repeat;           // label is in blk.h, line 149.
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
372     // コントローラ。コマンドが無効な場合は停止します。
373     } else if (CURRENT->cmd == READ) {
374         hd_out(dev, nsect, sec, head, cyl, WIN_READ, &read_intr);
375     } else
376         panic("unknown hd-command");
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。
// interrupt handler (kernel/sys_call.s, line 235). The hard disk interrupt is INT 0x2E (46),
// 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);                  // set interrupt handler
382     outb_p(inb_p(0x21)&0xfb, 0x21);                   // reset mask bit 2 on master chip.
383     outb_(inb_p(0xA1)&0xbf, 0xA1);                   // reset mask bit 6 on slave chip.
384 }
385

```

## 9.3.4 Information

### 9.3.3.1 AT Hard Disk Interface Register

The programming register port description for the AT hard disk controller is shown in Table 9–3. Also see the description in the include/linux/hdreg.h header file.

	Name	Read operation	Write operation

Port			
0x1f0	HD_DATA	Data Register - Sector data (read, write and format)	
0x1f1	HD_ERROR,HD_PRECOMP	Error Register (HD_ERROR)	Write Precomp Register (HD_PRECOMP)
0x1f2	HD_NSECTOR	Total Sectors Register - Total number of sectors (read, write, check and format)	
0x1f3	HD_SECTOR	Sector Number Register - Start sector number (read, write and check)	
0x1f4	HD_LCYL	Cylinder Number Register - Cylinder number low byte (read,write,check and format)	
0x1f5	HD_HCYL	Cylinder Number Register - Cylinder number high byte (read,write,check and format)	
0x1f6	HD_CURRENT	Drive/Head Register - Drive/Head No.(101dhhh, d=Drvie No, h=Head No)	
0x1f7	HD_STATUS,HD_COMMAND	Main Status Register (HD_STATUS)	Command Register (HD_COMMAND)

0x3f6	HD_CMD	---	Harddisk Control Register (HD_CMD)
0x3f7		Digital Input Register (1.2MB Floopy)	---

The port registers are described in detail below.

- ◆ Data register (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。

- ◆ Error register (read) / write precompensation register (write) (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ド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。

What is write pre-compensation?

Early hard disks have a fixed number of sectors per track, and since each sector has a fixed 512 bytes, the physical track length occupied by each sector is shorter as it gets closer to the center of the disk. The ability to cause magnetic media to store data is reduced. Therefore, for the hard disk head, it is necessary to take certain measures to put the data of one sector into a relatively small sector at a relatively high density. The common method used is the Write Precompensation technique. That is, starting from the edge of the disc to a position near a track (cylinder) in the center of the disc, the write current in the head is adjusted in some way.

The specific adjustment method is as follows: the representation of the binary data 0, 1 on the disk is recorded by a magnetic recording and encoding method (for example, FM, MFM, etc.). If the adjacent recording bits are magnetized and flipped twice, magnetic field overlap may occur. Therefore, the peak value of the corresponding electrical waveform will drift when the data is read at this time. If the recording density is increased, the degree of peak drift is increased, and sometimes the data bits cannot be separated and recognized, resulting in read data errors. The way to overcome this problem is to use pre-write compensation or post-read compensation techniques. Pre-write compensation means that the pulse compensation is written in advance in the opposite direction to the peak drift of the readout before the write data is sent to the driver. If the peak value of the signal drifts forward when read, the signal is delayed to be written; if the signal drifts backward when read, the signal is written in advance. Thus, when reading, the position of the peak can be close to the normal position.

	Dianostic command	Other commands

Value		
0x01	No error	Data flag missing
0x02	Controller error	Track 0 error
0x03	Sector buffer error	
0x04	ECC part error	Command abandon
0x05	Control processor error	
0x10		ID not found
0x40		ECC error
0x80		Bad sector

- ◆ Sector Number Register (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。
- ◆ Cylinder number register (HD\_LCYL, HD\_HCYL, 0x1f4, 0x1f5)
  - ◆ 2つのシリンダーレジスターには、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
- ◆ Driver/head register (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101dhhh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 ブ / ヘ ツ ド レ ジ ス タ  Bit	Name	Description
0	HS0	Head number lowest bit
1	HS1	
2	HS2	
3	HS3	Head number highest bit
4	DRV	Select drive, 0 - Select drive 0; 1 - Select drive 1
5	Reserved	Always 1
6	Reserved	Always 0
7	Reserved	Always 1

◆ Main Status Register (Read) / Command Register (Write) (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。

Bit	Name	Mask	Description
9			
8			
7			
6			
5			
4			
3			
2			
1			
0	ERR_STAT	0x01	Command execution error. When this bit is set, the previous command ends with an error. At this point, the bits in the error register and status register contain some information that caused the error.
1	INDEX_STAT	0x02	Received the index. This bit is set when an index flag is encountered while the disk is spinning.
2	ECC_STAT	0x04	ECC checksum error. This bit is set when a recoverable data error is encountered and has been corrected. This situation does not interrupt a multi-sector read operation.
3	DRQ_STAT	0x08	Data request service. When this bit is set, it indicates that the drive is ready to transfer one word or one byte of data between the host and the data port.
4	SEEK_STAT	0x10	The drive seek ends. When this bit is set, it indicates that the seek operation has been

			completed and the head has stopped on the specified track. This bit does not change when an error occurs. This bit will again indicate the completion status of the current seek after the host has read the status register.
5	WRERR_STAT	0x20	Drive failure (write error). This bit does not change when an error occurs. This bit will again indicate the error status of the current write operation only after the host has read the status register.
6	READY_STAT	0x40	The drive is ready. Indicates that the drive is ready to receive commands. This bit does not change when an error occurs. This bit will again indicate the current drive ready state after the host has read the status register. At power-on, this bit should be reset until the drive speed is normal and the command can be received.
7	BUSY_STAT	0x80	<p>The controller is busy. This bit is set when the drive is operating by the controller of the drive. At this point, the host cannot send a command block. A read of any of the command registers will return the value of the status register. This bit will be set under the following conditions:</p> <ul style="list-style-type: none"> <li>(*) It is within 400 nanoseconds after the machine reset signal RESET becomes negative or the SRST of the device control register is set. The set state of this bit is required to be no longer than 30 seconds after a machine reset.</li> <li>(*) The host is within 400 nanoseconds of writing commands such as recalibration, read, read buffer, initialization of drive parameters, and execution of diagnostics.</li> <li>(*) Within 5 microseconds of 512 bytes of data transferred during a write operation, write buffer, or format track command.</li> </ul>

When a write operation is performed, the port corresponds to a command register. It accepts hard disk control commands from the CPU. There are 8 commands, as shown in Table 9–7. The last column is used to describe the actions taken by the controller after the end of the corresponding command (causing an interrupt or doing nothing).

Command Name		表9-7 ATハ ードディス クコントロ ーラコマン ド一覧		Default value	Command End Form
		Command Code Byte	High 4 bits		
WIN_RESTORE	Drive Recalibration (Reset)	0x1	R R R R	0x10	Interrupt
WIN_READ	Read Sector	0x2	0 0 L T	0x20	Interrupt
WIN_WRITE	Write Sector	0x3	0 0 L T	0x30	Interrupt
WIN_VERIFY	Sector Check	0x4	0 0 0 T	0x40	Interrupt
WIN_FORMAT	Format track	0x5	0 0 0 0	0x50	Interrupt
WIN_INIT	Controller Initialization	0x6	0 0 0 0	0x60	Interrupt
WIN_SEEK	Seek Track	0x7	R R R R	0x70	Interrupt
WIN_DIAGNOSE	Controller Diagnostic	0x9	0 0 0 0	0x90	Interrupt or Idle
WIN_SPECIFY	Build Drive Parameters	0x9	0 0 0 1	0x91	Interrupt

The lower 4 bits of the command code byte in the table are additional parameters, which means:

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。

(1) これらのコマンドの詳細は以下の通りです。

(2) 0x1X -- (WIN\_RESTORE), drive recalibrate command

(3) 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シリンドーに移動させます。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。

(4) 0x20 -- (WIN\_READ) retryable read sector; 0x21 -- no retry read sector.

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、ドライバ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入ると、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つ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・ビットを設定し、DRQ\_STATを設定し、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ド番号、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ド番号、ヘッド番号、セクタ番号が格納されます。

- (5) 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ド番号、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。
- (6) 0x30 -- (WIN\_WRITE) retryable write sector; 0x31 -- no retry write sector.  
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フィールドに配置されます。

IDフィールドが正しく読み取られると、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、コマンドロックレジスタに最後に書き込まれたセクタのシリンド番号、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。セクタの転送が完了すると、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・ロック・レジスタに、最後に書き込まれたセクタのシリンド番号、ヘッド番号、セクタ番号が格納されます。

- (7) 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、コマンドロックレジスタには、エラーが発生したセクタのシリンド番号、ヘッド番号、セクタ番号が格納されます。

(8) 0x40 -- (WIN\_VERIFY) retryable sector read verification; 0x41 -- no retry sector verification.

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・ロック・レジスタには、最後に検証されたセクタのシリンド番号、ヘッド番号、セクタ番号が格納されます。

- (9) 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、コマンドロックレジスタには、エラーが発生したセクタのシリンド番号、ヘッド番号、セクタ番号が格納されます。

(10) 0x50 -- (WIN\_FORMAT) Format the track command.

- (11) 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

ツ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。

(12) 0x60 -- (WIN\_INIT) controller initialization.

(13) 0x7X -- (WIN\_SEEK) seek operation.

(14) 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 フラグをセットし、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。

(15) 0x90 -- (WIN\_DIAGNOSE) drive diagnostic command.

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

(16) 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。

(17) 0x91 -- (WIN\_SPECIFY) Create a drive parameter command.

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バ/ヘッド・レジスタで、具体的に選択されたドライバに応じてドライブ・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。

◆ Hard disk control register (write) (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ーテーブルのシフト0x08のバイト記述と同じです。

Bit	Control Byte Description (drive step selection)
9	
-	
8	
ハ ー ド デ ィ ス ク 制 御 バ イ ト	

Offset		の意味
0x08	0	Not used
	1	Reserved (0) ( Close IRQ)
	2	Allow reset
	3	Set if the number of heads is greater than 8
	4	Not used (0)
	5	If there is a manufacturer's bad area map at the cylinders +1, set 1
	6	Prohibit ECC retry
	7	Prohibit access retry

### 9.3.3.2 AT hard disk controller programming

When operating the hard disk controller, you need to send parameters and commands at the same time. The command format is shown in Table 9–9. First, you need to send a 6-byte parameter, and finally issue a 1-byte command code. No matter what command, you need to completely output the 7-byte command block, and write to port 0x1f1 -- 0x1f7 in turn. Once the command block register is loaded, the command begins execution.

表 9 - 9  ハ ード デ ィ ス ク ・ コ ン ト ロ ー ラ の コ マ ン	Description

---

ド フ オ 一 マ ツ ト Port	
0x1f1	Write pre-compensation start cylinder number
0x1f2	Number of sectors

0x1f3	Starting sector number
0x1f4	Cylinder number low byte
0x1f5	Cylinder number high byte
0x1f6	Drive number / head number
0x1f7	Command code

First, the CPU outputs a control byte to the control register port (HD\_CMD, 0x3f6) to establish a corresponding hard disk control mode. After the mode is established, parameters and commands can be sent in the above order. The steps are:

1. Detect controller idle state: The CPU reads the main status register. If bit 7 (BUSY\_STAT) is 0, the controller is idle. If the controller is always busy within the specified time, it is judged as a timeout error. See the controller\_ready() function on line 168 in hd.c.
2. Check if the drive is ready: The CPU determines if the main status register bit 6 (READY\_STAT) is 1 to see if the drive is ready. If 1, the CPU can output parameters and commands. See the drive\_busy() function on line 209 in hd.c.
3. Output command block: Outputs parameters and commands to the corresponding ports in sequence. See the hd\_out() function starting with line 187 in hd.c.
4. CPU waits for interrupt generation: After the command is executed, the hard disk controller will generate an interrupt request signal (IRQ14 - corresponding int46) or set the controller state to idle, indicating the end of the operation or the request for sector transfer (multi-sector read/write). The function called in the program hd.c during the interrupt processing is shown in the code 237--293. There are 5 functions corresponding to 5 cases: hard disk reset, unexpected interrupt, bad read/write interrupt, read interrupt and write interrupt.
5. Detection operation result: The CPU reads the main status register again. If bit 0 is equal to 0, the command execution is successful, otherwise it fails. If it fails, you can further query the error register (HD\_ERROR) to get the error code. See the win\_result() function on line 176 of hd.c.

### 9.3.3.3 硬盘分区表

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。

	Name	Size (Byte)	Description

Offset			
0x000	MBR Code	446	Boot program code and data.
0x1BE	Partition table entry 1	16	The first partition table entry, a total of 16 bytes.
0x1CE	Partition table entry 2	16	The second partition table entry, 16 bytes.
0x1DE	Partition table entry 3	16	The third partition table entry, 16 bytes.
0x1EE	Partition table entry 4	16	The fourth partition table entry, 16 bytes.
0x1FE	Boot flag	2	Valid boot sector flags: 0x55, 0xAA

In addition to the initial 446-byte boot executable code, the MBR also contains a hard disk partition table with a total of four entries. The partition table is stored at the 0x1BE--0x1FD offset position of the 1st sector of

は、ハードディスクの0番のシリンドーです。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、シリンド番号、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。

	Name	Size	Description
Offset			
0x00	boot_ind	1 byte	Boot index. Only one partition of the 4 partitions can be booted at a time. 0x00 - Do not boot from this partition; 0x80 - Boot from this partition.

0x01	head	1 byte	Partition start head number. The head number ranges from 0 to 255.
0x02	sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the beginning of the partition.
0x03	cyl	1 byte	The lower 8 bits of the cylinder number at the starting of the partition.
0x04	sys_ind	1 byte	Partition type. 0x0b - DOS; 0x80 - Old Minix; 0x83 - Linux . . .
0x05	end_head	1 byte	The head number at the end of the partition. It ranges from 0 to 255.
0x06	end_sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the end of the partition .
0x07	end_cyl	1 byte	The lower 8 bits of the cylinder number at the end of the partition.
0x08-0x0b	start_sect	4 byte	The physical sector number at the beginning of the partition. It counts from 0 in the order of the sector number of the entire hard disk.
0x0c-0x0f	nr_sects	4 byte	The number of sectors occupied by the partition.

The fields 'head', 'sector', and 'cyl' in the table represent the head number, sector number, and cylinder number at the beginning of a partition, respectively. The range of the 'head' number ranges from 0 to 255. The lower 6 bits in the 'sector' byte field represent the sector number counted in the current cylinder. The sector number count range is 1-63. The upper 2 bits of the 'sector' field form a 10-bit cylinder number with the 'cyl' field, which ranges from 0 to -1023. Similarly, the 'end\_head', 'end\_sector', and 'end\_cyl' fields in the table indicate the head number, sector number, and cylinder number at the end of the partition, respectively. Therefore, if we use C to indicate the cylinder number, H for the head number, and S for the sector number, the starting CHS value of the partition can be expressed as:

$$H = \text{head}.$$

$$S = \text{sector} \& 0x3f;$$

$$C = (\text{sector} \& 0xc0) << 2) + \text{cyl};$$

The 'start\_sect' field in the table is the 4-byte partition start physical sector number. It represents the sector number of the entire hard disk compiled from 0. The encoding method is: starting from CHS for 0 cylinder, 0 head and 1 sector (0, 0, 1), first encode the sectors in the current cylinder in order, and then encode the head from 0 to the maximum head number. Finally, the cylinder is counted. If the total number of heads of a hard disk is MAX HD and the total number of sectors per track is MAX\_SECT, then the physical sector number phy\_sector of the hard disk corresponding to a CHS value is:

---

```
phy_sector = (C * MAX_HEAD + H) * MAX_SECT + S - 1
```

The first sector of the hard disk (0 cylinder 0 header 1 sector) has the same purpose as the first sector (boot sector) on the floppy disk except for one partition table. Only its code will move itself from 0x7c00 to 0x6000 during execution to free up space at 0x7c00, and then find out which active partition is based on the information in the partition table. Then load the first sector of the active partition to 0x7c00 to execute. A partition from which cylinder, head and sector of the hard disk are recorded in the partition table. Therefore, it is possible to know from the partition table where the first sector of an active partition (ie, the boot sector of the partition) is on the hard disk.

#### 9.3.3.4 Relationship between absolute sector and current cylinder, sector, head.

■ 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ド番号を 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ドー番号、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 = The quotient is tracks, and the remainder is sec;
- tracks / dev\_heads = the quotient is cyl, and the remainder is head;
- On the current track, the sector number starts from 1, so you need to increase sec by 1.

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ー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 Function Description

This program is mainly used to perform low-level block device read/write operations. It is the interface program between all block devices (hard drive, floppy drive and virtual ram disk) in this chapter and other parts of the system. By calling the program's read/write function ll\_rw\_block(), other programs in the system can asynchronously read and write data from the block device. The main purpose of this function is to create block device read and write request items for other programs and insert them into the specified block device request queue. The actual read and write operations are done by the request handling function request\_fn() of the device. For hard disk operations, the function is do\_hd\_request(); for floppy operations, the function is do\_fd\_request(); for virtual disks it is do\_rd\_request().

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信することなくなり、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。

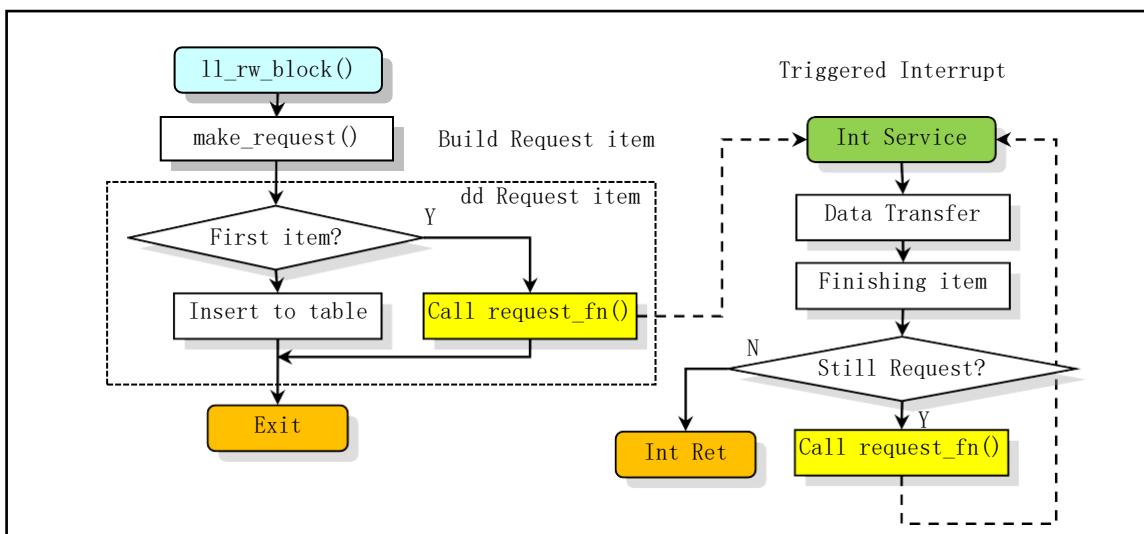


図9-5 ll\_rw\_blockのコールシーケンス

For the virtual disk device, since its read and write operations do not involve the above-mentioned synchronous operation with the external hardware device, there is no interrupt processing described above. The read and write operations of the current request item to the virtual device are completely implemented in `do_rd_request()`.

#### 9.4.2 Code Annotation

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 * This handles all read/write requests to block devices
9 */
// <errno.h> エラー番号のヘッダファイルです。システムの様々なエラー番号を含みます。
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ

```

```
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
//<linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定
義が含まれています。
```

```

//      used functions of the kernel.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
//      descriptors/interrupt gates, etc. is defined.
10 #include <errno.h>
11 #include <linux/sched.h>
12 #include <linux/kernel.h>
13 #include <asm/system.h>
14
15 #include "blk.h"           // request structure, linked list queue.
16
17 /*
18 * The request-struct contains all necessary data
19 * to load a nr of sectors into memory
20 */
21 struct request request[NR_REQUEST];
22
23 /*
24 * used to wait on when there are no free requests
25 */
26 struct task_struct *wait_for_request = NULL; 27
27 /* blk_dev_struct is:
28 *      do_request-address
29 *      next-request
30 */
31 // 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
32 // の内容は、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
33 // 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
34 // 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。
35 struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
36     { NULL, NULL },          /* no_dev */
37     { NULL, NULL },          /* dev mem */
38     { NULL, NULL },          /* dev fd */
39     { NULL, NULL },          /* dev hd */
40     { NULL, NULL },          /* dev tty */
41     { NULL, NULL }           /* dev lp */
42 };
43 /*
44 * blk_size contains the size of all block-devices:
45 * blk_size[MAJOR][MINOR]
46 *
47 * if (!blk_size[MAJOR]) then no minor size checking is done.
48 */
49 // デバイスデータブロックの総数を示すポインターの配列。各ポインタの項目は
50 // 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
51 // の所有するデータブロックの総数に相当します。
52 // 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) 。

```

```
49 int * blk_size[NR_BLK_DEV] = { NULL, NULL, };      // nr of block devices, NR_BLK_DEV = 7.  
50
```

// 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）。

```
53 51 static inline void lock_buffer(struct buffer_head *bh) 52 {  

54      cli();                      // disable int.  

55      while (bh->b_lock)          // sleeps if locked, until buffer is unlocked.  

56          sleep_on(&bh->b_wait);  

57      bh->b_lock=1;              // locked the buffer immediately.  

58      sti();                      // enable int.  

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 60 static inline void unlock_buffer(struct buffer_head * bh) 61 {  

61      if (!bh->b_lock)           // if not locked...  

62          printk("ll_rw_block.c: buffer not locked|r");  

63      bh->b_lock = 0;            // reset lock flag.  

64      wake_up(&bh->b_wait);    // wake up the task waiting for this buffer.  

65 }  

66  

67
```

```
68 /*  

69 * add-request adds a request to the linked list.  

70 * It disables interrupts so that it can muck with the  

71 * request-lists in peace.  

72 *  

73 * Note that swapping requests always go before other requests,  

74 * and are done in the order they appear.  

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項目のポインタを持つもので、reqはリクエスト  
// 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 76 static void add_request(struct blk_dev_struct * dev, struct request * req) 77 {
```

```
77     struct request * tmp;  

78  

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  

81     req->next = NULL;  

82     cli();  

83     if (req->bh)  

84         req->bh->b_dirt = 0;      // clear the buffer dirty flag.  

// 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。

85

```
86     if (!(tmp = dev->current_request)) {
87         dev->current_request = req;
88         sti(); // Enable int.
89         (dev->request_fn)(); // runs request function, ie. do_hd_request().
90         return;
91     }
```

//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  
ア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

90

//キューの正しい位置。

```
91     for ( ; tmp->next ; tmp=tmp->next) {
92         if (!req->bh)
93             if (tmp->next->bh)
94                 break;
95             else
96                 continue;
97         if ((IN_ORDER(tmp, req) ||
98             !IN_ORDER(tmp, tmp->next)) &&
99             IN_ORDER(req, tmp->next))
100             break;
101     }
102     req->next=tmp->next;
103     tmp->next=req;
104     sti();
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。

```
1     struct request * req;
0
6
s
```

t\_a\_t\_i\_c  
v\_o\_i\_d  
m\_a\_k\_e\_=r\_e\_q\_u\_e\_s\_t( i\_n\_t  
m\_a\_l\_o\_r\_+i\_n\_t  
r\_w\_2  
s\_t\_r\_u\_c\_t  
b\_u\_f\_f\_e\_r\_i\_h\_e\_a\_d  
\* - b\_h )  
1

```
0  
7  
{  
.108  
109     int rw_ahead;  
110  
111 /* WRITEA/READA is special case - it is not really needed, so if the */  
112 /* buffer is locked, we just forget about it, else it's a normal read */  
// 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。  
// a prefetch/write command before modifying the command.
```

```

113     if (rw_ahead = (rw == READA || rw == WRITEA)) {
114         if (bh->b_lock)
115             return;
116         if (rw == READA)
117             rw = READ;
118         else
119             rw = WRITE;
120     }
121     if (rw!=READ && rw!=WRITE)
122         panic("Bad block dev command, must be R/W/RA/WA");
123     lock_buffer(bh);
124     if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
125         unlock_buffer(bh);
126         return;
127     }
128 repeat:
129 /* we don't allow the write-requests to fill up the queue completely:
130 * we want some room for reads: they take precedence. The last third
131 * of the requests are only for reads.
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に開始されます。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から空のエントリー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、リクエスト項目の配列がヘッダーを越えて検索されている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
// item), and then search the request queue after a while.

```



```

133     if (rw == READ)
134         req = request+NR REQUEST;           // for read, search from the end.
135     else
136         req = request+((NR REQUEST*2)/3); // for write, search from 2/3 backward.
137 /* find an empty request */
138     while (--req >= request)
139         if (req->dev<0)
140             break;
141 /* if none found, sleep on new requests: check for rw_ahead */
142     if (req < request) {                  // no free item ...
143         if (rw_ahead) {                   // exit if it's read/write ahead.
144             unlock buffer(bh);
145             return;
146         }
147         sleep on(&wait for request);
148         goto repeat;                    // line 128.
149     }
150 /* fill up the request-info, and add it to the queue */

```

// 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。

151

// 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。

```
152     req->dev = bh->b_dev;           // device no.
153     req->cmd = rw;                 // command (READ/WRITE).
154     req->errors=0;                // error count.
155     req->sector = bh->b_blocknr<<1; // start sector, (1block = 2 sectors).
156     req->nr_sectors = 2;          // number of sectors read/written.
157     req->buffer = bh->b_data;    // data buffer location.
158     req->waiting = NULL;        // waiting list for the operation.
159     req->bh = bh;                // buffer header.
160     req->next = NULL;          // points to next request.
161     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。

65

163 void ll\_rw\_page(int rw, int dev, int page, char \* buffer) 164 {。

```
166     struct request * req;
167     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

168

// 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。

```
169     if (major >= NR_BLK_DEV || !(blk_dev[major].request_fn)) {
170         printk("Trying to read nonexistent block-device\n|r");
171         return;
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（

174 // 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。

175 repeat:

```
176     req = request+NR_REQUEST;           // points to the end.
177     while (--req >= request)
178         if (req->dev<0)
179             break;
```

```
180     if (req < request) {
181         sleep_on(&wait_for_request);      // sleep at wait_for_request.
182         goto repeat;
182     }
183 /* fill up the request-info, and add it to the queue */
// 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

184

// 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。

```

185     req->dev = dev;                      // device no.
186     req->cmd = rw;                      // command.
187     req->errors = 0;                    // error count.
188     req->sector = page<<3;           // starting sector.
189     req->nr_sectors = 8;                // nr of sectors read/written.
190     req->buffer = buffer;              // data buffer.
191     req->waiting = current;          // waiting queue.
192     req->bh = NULL;                  // buffer header.
193     req->next = NULL;                // points to next request.
194     current->state = TASK_UNINTERRUPTIBLE;
195     add_request(major+blk_dev, req); // add to request queue.
196     schedule();
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; // major device no ( 3 for hard disk).

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

// 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

202

// をリクエストキューに入れることができます。

203 if ((major=MAJOR(bh->b\_dev)) >= NR\_BLK\_DEV ||

204 !(blk\_dev[major].request\_fn)) {

205 printk("Trying to read nonexistent block-device\n|r");

206 return;

207 }

208 make\_request(major, rw, bh);

209 }

//// The block device initialization function, called by the initialization program main.c.

// Initializes the request array and sets all request items as free items (dev = -1). There

// are 32 items (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 Function

This file is a virtual Ram Disk driver created by Theodore Ts'o. A ram disk device is a way to use physical memory to simulate the actual disk storage data. Its purpose is mainly to improve the speed of reading and writing of "disk" data. In addition to taking up some valuable memory resources, the main disadvantage is that once the system is shut down or crashes, all the data in the virtual disk will disappear. Therefore, the virtual disk usually only stores some common tool programs or temporary data such as system commands, rather than important input documents.

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配置されます。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のメモリの先頭に設定します。このとき、メモリの割り当ては図9-6のようになります。

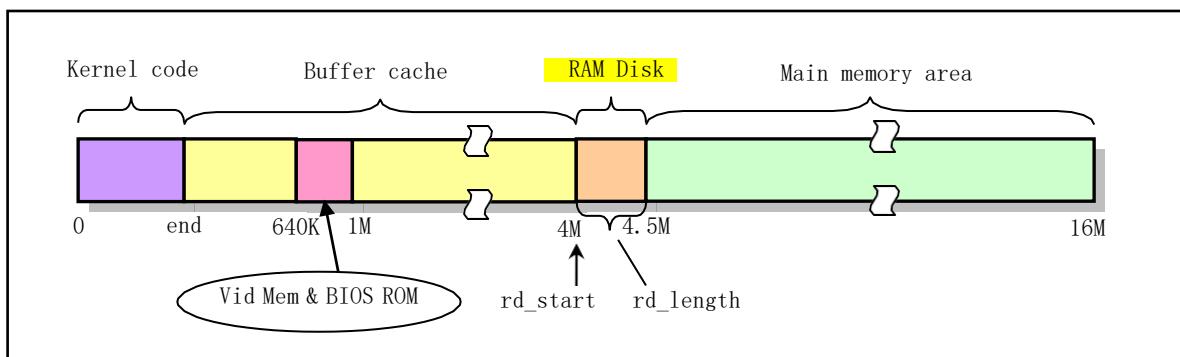


図9-6 16MBメモリシステムにおけるラムディスクの具体的な位置について

The read and write access operations to the ram disk are in principle the same as those for ordinary disks, and they need to be operated according to the access mode of the block device. Since the implementation does not involve synchronization with an external controller or device, its implementation is relatively simple. For data transfer between the system and the device, it is only necessary to perform an in-memory block copy operation.

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。

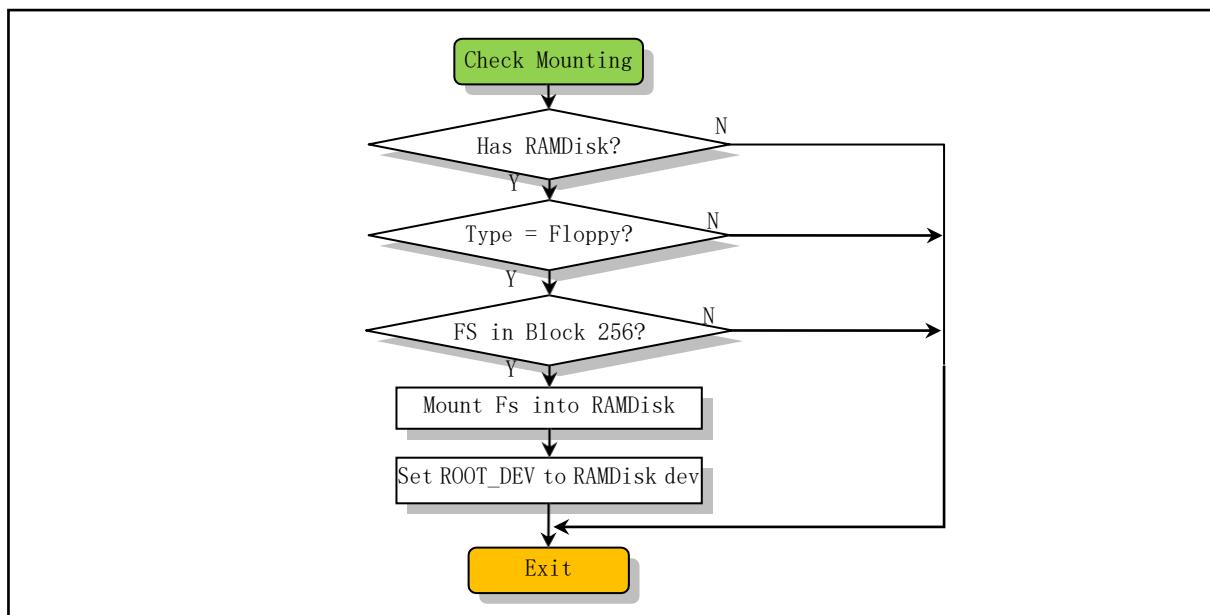


図9-7 ラムディスクにルートファイルシステムをロードするためのフローチャート

If the symbol RAMDISK and its size are defined in the linux/Makefile configuration file when compiling the Linux 0.12 kernel source code, then after booting and initializing the RAMDISK area, it will first try to check the location at the 256th disk block on the disk, Is there a root file system? The detection method is to determine whether there is a valid file system super block in the 257th disk block. If so, the file system is loaded into the RAMDISK area in memory and used as the root file system. So we can use a boot disk that integrates the root file system to boot the system to the shell command prompt. If a valid root file system is not stored at the specified disk block location (256th disk block) on the boot disk, the kernel will prompt to insert the root file system disk. After the user presses the Enter key to confirm, the kernel will read the root file system on a separate disk into the virtual disk area for execution.

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。

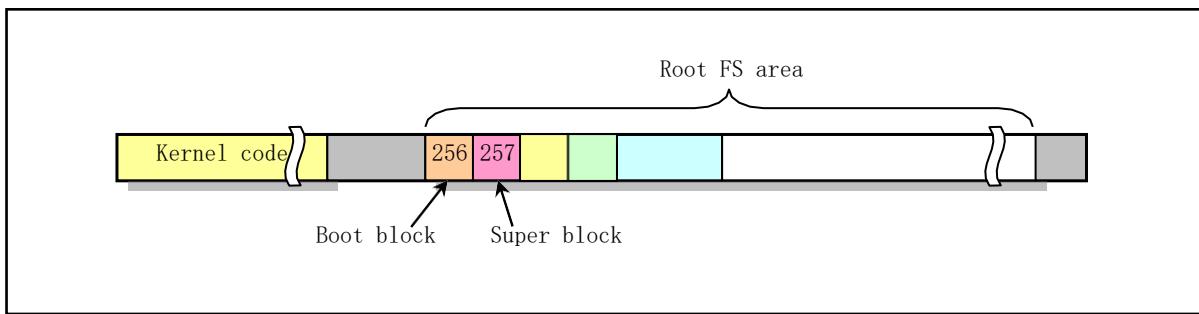


図9-8 集積ディスクのデータブロックレイアウト

### 9.5.2 Code Annotation

プログラム 9-4 linux/kernel/blk\_drv/ramdisk.c

```

1  /*
2   *  linux/kernel/blk_drv/ramdisk.c
3   *
4   *  Written by Theodore Ts'o, 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することで、安定性、復元性、信頼性が大幅に向上了しました。
// 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/ttys/
6
// <string.h> 文字列のヘッダーファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <linux/config.h> カーネル設定用のヘッダーファイルです。キーボード言語やハードディスクを定義する
// type (HD_TYPE) options.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// of the initial task 0, and some embedded assembly function macro statements about the
// descriptor parameter settings and acquisition.
// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する (file, buffer_head,
// m_inode, etc.).
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義
// が含まれています。
// used functions of the kernel.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
// descriptors/interrupt gates, etc. is defined.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数が定義されています。
// segment register operations.
// <asm/memory.h> メモリコピーのヘッダーファイルです。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>
```

```

13 #include <asm/system.h>
14 #include <asm/segment.h>
15 #include <asm/memory.h>
16
17 // 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
18 // 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、//。
19 // 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。
20 #define MAJOR_NR 1
21 #include "blk.h"
22
23 // 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
24 // 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。
25 char *rd_start; // the starting address of the ram disk in memory.
26 int rd_length = 0; // memory size (in bytes) occupied by the ram disk.
27
28 // ラムディスクのカレントリクエスト操作機能です。
29 // この関数の構造は、ハードディスクドライバーのdo_hd_request()に似ています。
30 // (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
31 // 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
32 // 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
33 // 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ス
34 // 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
35 // 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
36 // が WRITE の場合、リクエストで指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
37 // 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
38 // 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
39 // 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。
40 void do_rd_request(void)
41 {
42     int len;
43     char *addr;
44
45 // 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)
46 // 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。
47 // 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。
48 // 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
49 // 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
50 // はblk.hでは「(blk_dev[MAJOR_NR].current_request)」と定義されています。
51     INIT_REQUEST;
52     addr = rd_start + (CURRENT->sector << 9);
53     len = CURRENT->nr_sectors << 9;
54
55 // 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
56 // 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
57 // 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

```

```
32 // 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)。
33     if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd_start+rd_length)) {
34         end_request(0);
35         goto repeat;
36     }
// 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。
```

```

// the command does not exist and crashes.
35     if (CURRENT-> cmd == WRITE) {
36         (void ) memcpy(addr,
37                         CURRENT->buffer,
38                         len);
39     } else if (CURRENT->cmd == READ) {
40         (void) memcpy(CURRENT->buffer,
41                         addr,
42                         len);
43     } else
44         panic( "unknown ramdisk-command" );
45 // 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
46 // デバイスのリクエスト項目を処理します。
47     end_request(1);
48     goto repeat;
49 }
50 /*
51 * Returns amount of memory which needs to be reserved.
52 */
53
// 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、仮想エクステント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。
54
55 long rd_init(long mem_start, int length) 53 {
56     int i;
57     char *cp;
58
59     blk_dev[MAJOR_NR].request_fn = DEVICE REQUEST;      // do_rd_request()
60     rd_start = (char *) mem_start;                      // 4MB for the 16MB machine.
61     rd_length = length;                                // size of the ram disk.
62     cp = rd_start;
63     for (i=0; i < length; i++)                          // cleared.
64         *cp++ = '|0';
65     return(length);
66 }
67 /*
68 * If the root device is the ram disk, try to load it.
69 * In order to do this, the root device is originally set to the
70 * floppy, and we later change it to be ramdisk.
71 */
72
73 /**
74 * 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。
75 // この関数は、カーネルのセットアップ関数setup() (hd.c, line 162)で呼び出されます。この関数は
76 // 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     struct buffer_head *bh;           // cache buffer head pointer.  
74     struct super_block      s;  
75     int          block = 256;    /* Start at block 256 */  
76     int          i = 1;  
77     int          nblocks;       // The amount of file system disk blocks.  
78     char         *cp;           /* Move pointer */  
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。  
// displayed. If the root file device is not a floppy device at this time, it also exits.  
if (!rd_length)
```

```

80
81         return;
82     printf("Ram disk: %d bytes, starting at 0x%x\n", rd_length,
83             () rd_start);
84     if (MAJOR(ROOT_DEV) != 2)
85         return;
// 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
87     bh = breada(ROOT_DEV, block+1, block, block+2, -1);
88     if (!bh) {
89         printf("Disk error while looking for ramdisk!\n");
90         return;
91     }
92     *((struct d_super_block *) &s) = *((struct d_super_block *) bh->b_data);
93     brelse(bh);
94     if (s.s_magic != SUPER_MAGIC)
95         /* No ram disk image present, assume normal floppy boot */
96         return;
// Then we try to read the entire root file system into the memory virtual disk extent. For
// 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 * 2^s_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。
97     nblocks = s.s_nzones << s.s_log_zone_size;
98     if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
99         printf("Ram disk image too big! (%d blocks, %d avail)\n",
100                nblocks, rd_length >> BLOCK_SIZE_BITS);
101    return;

```

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、シングルブ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す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。  
// in the display string indicates that a tab is displayed.  
printf("Loading %d bytes into ram disk... 0000k",
```

```

102
103     nblocks << BLOCK_SIZE_BITS);
104     cp = rd_start;
105     while (nblocks) {
106         if (nblocks > 2)                                // need read ahead ?
107             bh = breada(ROOT_DEV, block, block+1, block+2, -1);
108         else                                         // read one block each time.
109             bh = bread(ROOT_DEV, block);
110         if (!bh) {
111             printk("I/O error on block %d, aborting load\n",
112                     block);
113             return;
114         }
115         () memcpy(cp, bh->b_data, BLOCK_SIZE); // copy to location cp.
116         brelse(bh);
117         printk("|010|010|010|010|010%4dk", i);      // nr of blocks loaded.
118         cp += BLOCK_SIZE;                            // next disk block.
119         block++;
120         nblocks--;
121         i++;
122     }
// 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。
// "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。
// device number 0x0101, and finally returned.
123     printk("|010|010|010|010|010done |n");
124     ROOT_DEV=0x0101;
125 }
126

```

## 9.6 floppy.c

### 9.6.1 Function description

This program is a floppy disk controller driver. Like other block device drivers, the program also uses the request item operation function (do\_fd\_request() for the floppy disk drive) to perform read and write operations on the floppy disk. The main difference from the hard disk driver is that the floppy disk driver uses more timing functions and operations.

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が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。

**9.6.2** 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いる説明書や、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。

### 9.6.3 Code Annotation

プログラム 9-5 linux/kernel/blk\_drv/floppy.c

```

1 /*
2  * linux/kernel/floppy.c
3  *
4  * (C) 1991 Linus Torvalds
5 */
6 /*
7 * 02.12.91 - Changed to static variables to indicate need for reset
8 * and recalibrate. This makes some things easier (output_byte reset
9 * checking etc), and means less interrupt jumping in case of errors,
10 * so the code is hopefully easier to understand.
11 */
12 */
13
14 */
15 * This file is certainly a mess. I've tried my best to get it working,
16 * but I don't like programming floppies, and I have only one anyway.
17 * Urgel. I should check for more errors, and do more graceful error
18 * recovery. Seems there are problems with several drives. I've tried to
19 * correct them. No promises.
20 */

```

23   21  
      22 /\*  
24    *\* As with hd.c, all routines within this file can (and will) be called*  
25    *\* by interrupts, so extreme caution is needed. A hardware interrupt*  
26    *\* handler may not sleep, or a kernel panic will happen. Thus I cannot*  
27    *\* call "floppy-on" directly, but have to set a special timer interrupt*

```

28 * etc.
28 *
29 * Also, I'm not certain this works on more than 1 floppy. Bugs may
30 * abund.
31 */
32
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_structや
//   data of the initial task 0, and some embedded assembly function macro statements
//   about the descriptor parameter settings and acquisition.
// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する (file,
//   buffer_head, m_inode, etc.).
// <linux/kernel.h> カーネルのヘッダーファイルです。のプロトタイプ定義が含まれています。
//   commonly used functions of the kernel.
// <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
//   parameters.
// <asm/system.h> システムのヘッダーファイルです。を定義する埋め込みアセンブリマクロです。
//   modifies descriptors/interrupt gates, etc. is defined.
// <asm/io.h> Io のヘッダーファイルです。の io ポートを操作する関数を定義します。
//   form of a macro's embedded assembler.
// <asm/segment.h> セグメント操作用のヘッダーファイルです。埋め込みアセンブリ関数の定義
//   for segment register operations.
33 #include <linux/sched.h>
34 #include <linux/fs.h>
35 #include <linux/kernel.h>
36 #include <linux/fdreg.h>
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。
41 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。
42 #define MAJOR_NR 2           // floppy drive major number.
43 #include "blk.h"           // block dev header file. requests, queues are defined.
43
44 static int recalibrate = 0; // flag: recalibrate head position (return to zero).
45 static int reset = 0;       // flag: reset operation needed.
46 static int seek = 0;        // flag: perform a seek operation.
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\\Yn":": "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)           // Type of floppy drive (2--1.2Mb, 7--1.44Mb).
54 #define DRIVE(x) ((x)&0x03)          // The floppy drive number (0--3 corresponds to A--D).
55 /*
56 * Note that MAX_ERRORS=8 doesn't imply that we retry every bad read
57 * max 8 times - some types of errors increase the errorcount by 2,
58 * so we might actually retry only 5-6 times before giving up.
59 */
60 #define MAX_ERRORS 8
61
62 /*
63 * globals used by 'result()'
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 returns up to 7 bytes of results.
66 static unsigned char reply_buffer[MAX_REPLIES]; // used to store the response results.
67 #define ST0 (reply_buffer[0])          // result status byte 0.
68 #define ST1 (reply_buffer[1])          // result status byte 1.
69 #define ST2 (reply_buffer[2])          // result status byte 2.
70 #define ST3 (reply_buffer[3])          // result status byte 3.
71
72 /*
73 * This struct defines the different floppy types. Unlike minix
74 * linux doesn't have a "search for right type"-type, as the code
75 * for that is convoluted and weird. I've got enough problems with
76 * this driver as it is.
77 *
78 * The 'stretch' tells if the tracks need to be boubled for some
79 * types (ie 360kB diskette in 1.2MB drive etc). Others should
80 * be self-explanatory.
81 */

```

// フロッピードライブ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

```

82 static struct floppy_struct {
83     unsigned int size, sect, head, track, stretch;
84     unsigned char gap, rate, spec1;
85 } floppy_type[] = {
86     { 0, 0, 0, 0, 0x00, 0x00, 0x00 }, /* no testing */
87     { 720, 9, 2, 40, 0, 0x2A, 0x02, 0xDF }, /* 360kB PC diskettes */
88     { 2400, 15, 2, 80, 0, 0x1B, 0x00, 0xDF }, /* 1.2 MB AT-diskettes */
89     { 720, 9, 2, 40, 1, 0x2A, 0x02, 0xDF }, /* 360kB in 720kB drive */
90     { 1440, 9, 2, 80, 0, 0x2A, 0x02, 0xDF }, /* 3.5" 720kB diskette */
91     { 720, 9, 2, 40, 1, 0x23, 0x01, 0xDF }, /* 360kB in 1.2MB drive */
92     { 1440, 9, 2, 80, 0, 0x23, 0x01, 0xDF } /* 720kB in 1.2MB drive */

```

93      { 2880, 18, 2, 80, 0, 0x1B, 0x00, 0xCF },      */\* 1.44MB diskette \*/*  
94 };

```

95
96 /*
97 * Rate is 0 for 500kb/s, 2 for 300kbps, 1 for 250kbps
98 * Spec1 is 0xSH, where S is stepping rate (F=1ms, E=2ms, D=3ms etc),
99 * H is head unload time (1=16ms, 2=32ms, etc)
100 */
101 * Spec2 is (HLD<<1 / ND), where HLD is head load time (1=2ms, 2=4 ms etc)
102 * and ND is set means no DMA. Hardcoded to 6 (HLD=6ms, use DMA).
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。
105 // 割り込みトラップゲート記述子を初期化する。
106 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
107 // の範囲です。
108 extern char tmp_floppy_area[1024];
109
110 /*
111 * These are global variables, as that's the easiest way to give
112 * information to interrupts. They are the data used for the current
113 * request.
114 */
115
// 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。
116 static int cur_spec1 = -1; // current spec1.
117 static int cur_rate = -1;
118 static struct floppy_struct * floppy = floppy_type; // floppy point to the floppy_type[].
119 static unsigned char current_drive = 0;
120 static unsigned char sector = 0; 118
static unsigned char head = 0; 119 static
unsigned char track = 0
121 static unsigned char seek_track = 0;
122 static unsigned char current_track = 255;
123 static unsigned char command = 0; // read/write command.
124 unsigned char selected = 0; // drive selected flag.
125 struct task_struct * wait_on_floppy_select = NULL; // wait floppy queue.
126
//// 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

```

```
{  
129     if (nr != (current_DOR & 3))  
130         printk("floppy_deselect: drive not selected\n|r");  
131     selected = 0;  
132     wake_up(&wait_on_floppy_select);  
132 }  
133
```

```

135 134 */
136 * floppy-change is never called from an interrupt, so we can relax a bit
137 * here, sleep etc. Note that floppy-on tries to set current_DOR to point
138 * to the desired drive, but it will probably not survive the sleep if
139 * several floppies are used at the same time: thus the loop.
139 */

    //// 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。
    // 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
    // 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
    // の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。
    // 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。
    // fs/buffer.c (119行目).

140 int floppy_change(unsigned int nr)
141 {
    // 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。
    // 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
    // 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。
    // 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。
    // (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
142 // 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。
143 repeat:
144     floppy_on(nr);           // Start and wait for the specified floppy drive nr.
    // 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
    // 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、現在選択さ
    // れている
    // 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。
    // 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
    // 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が現
    // 在
    // 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
    // ィスクドライブがまだ
    // 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
    // 機能の//と再循環します。
145
146     while ((current_DOR & 3) != nr && selected)
147         sleep_on(&wait_on_floppy_select);
148     if ((current_DOR & 3) != nr)
149         goto repeat;
    // ここで、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
    // た。
    // 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
    // れは
    // 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
    // 合は
        if (inb(FD_DIR) & 0x80) {

```

```
148
149         floppy_off(nr) ;
150     return 1;
151 }
152     floppy_off(nr) ;
153     return 0;
154 }
155
156 ///////////////////////////////////////////////////////////////////
157 ///////////////////////////////////////////////////////////////////
158 ///////////////////////////////////////////////////////////////////
159 ///////////////////////////////////////////////////////////////////
160 ///////////////////////////////////////////////////////////////////
```

148  
149        floppy\_off(nr) ;  
150      return 1;  
151 }  
152        floppy\_off(nr) ;  
153      return 0;  
154 }  
155  
156 ///  
157 ///  
158 ///  
159 ///  
160 ///

      ///Copy 1024 bytes of data from the memory address 'from' to the address 'to'.  
#define copy\_buffer(from,to) \  
  \_\_asm\_\_( "cld ; rep ; movsl" \  
          : : "c" (BLOCK\_SIZE/4), "S" ((long)(from)), "D" ((long)(to)) \  
          : "cx", "di", "si")  
      /// Setup (initialize) the floppy disk DMA channel.

```

// 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;           // current request buffer address.
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際には、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();
166     if (addr >= 0x100000) {
167         addr = (long) tmp_floppy_area;
168         if (command == FD_WRITE)
169             copy_buffer(CURRENT->buffer, tmp_floppy_area);
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
// の設定です。シングルチャ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」を転送するバイト数0x3ff (0--1023) を書き込む。
// 最後に、DMAチャネル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 /* mask DMA 2 */
172     immoutb_p(4|2, 10);                      // port 10
173 /* output command byte. I don't know why, but everyone (minix, */
174 /* sanches & canton) output this twice, first to 12 then to 11 */
// 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
    __asm__ ("outb %%al, $12|n|tjmp 1f|n1:|tjmp 1f|n1:|t"

```

```

175      "outb %%al,$11\n\tjmp 1f|n1:\tjmp 1f|n1:";;
176      "a" ((char) ((command == FD READ)?DMA READ:DMA WRITE));
177 /* 8 low bits of addr */
178 // Write the base/current address register (port 4) to DMA channel 2.
179     immoutb_p(addr, 4);
180     addr >>= 8;
181 /* bits 8-15 of addr */
182     immoutb_p(addr, 4);
183     addr >>= 8;
184 /* bits 16-19 of addr */
185 // 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
186 // ページレジスタ（ポート0x81）。
187     immoutb_p(addr, 0x81);
188 /* low 8 bits of count-1 (1024-1=0x3ff) */
189 // 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 /* high 8 bits of count-1 */
     immoutb_p(3, 5);

189 /* activate DMA 2 */
190     immoutb_p(0|2, 10);
191     sti();
192 }
193 }

194
    ///////////////////////////////////////////////////////////////////
    // Output a byte command or parameter to the floppy drive controller.
    // Before sending a byte to the controller, the controller needs to be in a ready state, and
    // the data transfer direction must be set from CPU to FDC, so the function needs to read the
    // controller state information first. The loop query method is used here for proper delay.
    // If an error occurs, the reset flag is set.

195 static void output_byte(char byte)
196 {
197     int counter;
198     unsigned char status;
199

    // First, the state of the main state controller FD_STATUS (0x3f4) is cyclically read. If the
    // read status is STATUS_READY and the direction bit STATUS_DIR = 0 (CPU ⇨ FDC), the specified
    // byte is output to the data port.

200     if (reset)
201         return;
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);
206             return;
207         }
208     }
    // If it cannot be sent after the end of the cycle of 10,000 times, the reset flag is set and
    // エラーメッセージが表示されます。

209     reset = 1;

```

```

211     printk( "Unable to send byte to FDC|n|r");
211 }
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、配列のreply_buffer[].Returns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、メインストートコントローラ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テータスが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、コントローラのステータスが 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_REPLYES(7)です。
217     if (reset)

```

```

218         return -1;
219     for (counter = 0 ; counter < 10000 ; counter++) {
220         status = inb_p(FD_STATUS)&(STATUS_DIR|STATUS_READY|STATUS_BUSY);
221         if (status == STATUS READY)
222             return i;
223         if (status == (STATUS_DIR|STATUS_READY|STATUS_BUSY)) {
224             if (i >= MAX_REPLIES)
225                 break;
226             reply_buffer[i++] = inb_p(FD_DATA);
227         }
228     }
229 // 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。
230     reset = 1;
231     printk("Getstatus times out\n|r");
232     return -1;
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
// 以後の番組では、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
236    // 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）。
237    CURRENT->errors++;
238    if (CURRENT->errors > MAX_ERRORS) {
239        floppy_deselect(current_drive);
240        end_request(0);
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。

241    // 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。
242    if (CURRENT->errors > MAX_ERRORS/2)
243        reset = 1;
244    else
245        recalibrate = 1;
245 }

246
247 /*
249 * Ok, this interrupt is called after a DMA read/write has succeeded,

```

250 \* so we check the results, and copy any buffers.

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。
// フロッピーディスク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
// フロッピーディスク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 & 0xbff) || (ST2 & 0x73)) {
254         if (ST1 & 0x02) { // 0x02 = ST1_WP - Write Protected.
255             printk("Drive %d is write protected\n|r", current_drive);
256             floppy_deselect(current_drive);
257             end_request(0);
258         } else
259             bad_flp_intr();
260         do_fd_request();
261     }
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ている場合、フロッピーディスクの内容が
// 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、他のフロッピーディスク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);
266     end_request(1);
267     do_fd_request();
268 }
269 }
```

```
//// DMAチャネル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 {。
273     setup DMA() ;           // Initialize the floppy disk DMA channel.
274     do_floppy = rw_interrupt; // set function called in the int.
275     output byte(command) ;    // send command.
276     output byte(head<<2 | current drive) ; // param: head no + drive no.
277     output byte(track) ;      // param: track no.
278     output byte(head) ;       // param: head no.
279     output byte(sector) ;     // param: start sector no.
```

```

280     output_byte(2);           /* sector size = 512 */
281     output_byte(floppy->sect); // param: sectors per track.
282     output_byte(floppy->gap); // param: gap between sectors.
283     output_byte(0xFF);        /* sector size (0xff when n!=0 ?) */
284
285 // If any of the above output_byte() operations fail, the reset flag is set. The reset processing
286 // code in do_fd_request() is executed immediately.
287     if (reset)
288         do_fd_request();
289 }
290
291 /*
292 * This is the routine called after every seek (or recalibrate) interrupt
293 * from the floppy controller. Note that the "unexpected interrupt" routine
294 * also does a recalibrate, but doesn't come here.
295 */
296
297 ///////////////////////////////////////////////////////////////////
298 // The C function called during the interrupt process after the seek operations.
299 // First, the detection interrupt status command is sent, and the status information ST0 and
300 // the track information of the head are obtained. If an error occurs, the error count detection
301 // process is executed or the floppy operation request item is canceled. Otherwise, set the
302 // current track variable according to the status information, then call the function
303 // setup_rw_floppy() to set the DMA and output the read/write commands and parameters.
304 static void seek_interrupt(void)
305 {
306
307 // The check interrupt status command is sent first to obtain the result of the seek operation
308 // execution. This command takes no arguments. The returned result is two bytes: ST0 and the
309 // current track number of the head. Then read the result information of the FDCexecution.
310 // If the number of returned result bytes is not equal to 2, or ST0 is not the end of the seek,
311 // or the track on which the head is located (ST1) is not equal to the set track, an error has
312 // occurred. Then, the error counting is processed, and then the execution of the floppy disk
313 // request item or the execution of the reset processing is continued. Note that the sense
314 // interrupt status command (FD_SENSEI) should return 2 result bytes, that is the result() return
315 // value should equal 2.
316
317 /* sense drive status */
318     output_byte(FD_SENSEI);
319     if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {
320         bad_flp_intr();
321         do_fd_request();
322         return;
323     }
324
325 // If the seek operation is successful, the floppy disk operation of the current request is
326 // continued, that is, the command and parameters are sent to the floppy disk controller.
327     current_track = ST1;      // set current track.
328     setup_rw_floppy();       // set DMA, output floppy commands and parameters.
329 }
330
331 /*
332 * This routine is called when everything should be correctly set up
333 * for the transfer (ie floppy motor is on and the correct floppy is
334 * selected).
335 */
336
337 ///////////////////////////////////////////////////////////////////
338 // Read/write data transfer function.
339 static void transfer(void)
340 {

```

// 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

```
313     if (cur_spec1 != floppy->spec1) {           // check the current parameters.  

314         cur_spec1 = floppy->spec1;  

315         output_byte(FD_SPECIFY);                // send set disk parameters command.  

316         output_byte(cur_spec1);                /* hut etc */  

317         output_byte(6);                      /* Head load time =6ms, DMA */  

318     }  

319     if (cur_rate != floppy->rate)            // check current rate.  

320         outb_p(cur_rate = floppy->rate, FD_DCR);  

    // 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
```

320

// do\_fd\_request()はすぐに実行されます。

```
321     if (reset) {  

322         do_fd_request();  

323         return;  

324     }  

    // 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信されて  

    // 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
```

324

// recalibrationコマンドを実行し、ヘッドをゼロトラックに戻します。

```
325     if (!seek) {  

326         setup_rw_floppy();                  // Send command & parameter block.  

327         return;  

328     }  

329     do_floppy = seek_interrupt;          // set invoked function.  

330     if (seek_track) {                    // start track.  

331         output_byte(FD_SEEK);             // send seek command.  

332         output_byte(head<<2 | current_drive); // param: head + current drive.  

333         output_byte(seek_track);           // param: track no.  

334     } else {  

335         output_byte(FD_RECALIBRATE);      // send recalibrate command.  

336         output_byte(head<<2 | current_drive); // param: head + current drive.  

337     }  

    // 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。
```

337

// 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。

```
338     if (reset)  

339         do_fd_request();  

340     }  

341 /*
```

342 \* Special case - used after a unexpected interrupt (or reset)

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。もし、戻り値が

//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
// bytes, that is the result() return value should equal 2.
static void recal_interrupt(void)
```

```

344
345 {
346     output_byte(FD_SENSE);           // send sense interrupt status cmd.
347     if (_result() != 2 || (ST0 & 0xE0) == 0x60) // reset if there are errors
348         _reset = 1;
349     else
350         _recalibrate = 0;
351     do_fd_request();
352 }
353
354 void unexpected_floppy_interrupt(void)
355 {
356     output_byte(FD_SENSE);           // send sense interrupt status cmd.
357     if (_result() != 2 || (ST0 & 0xE0) == 0x60) // reset if there are errors.
358         _reset = 1;
359     else
360         _recalibrate = 1;
361 }
362
363 static void recalibrate_floppy(void) 364 {
364     _recalibrate = 0;
365     _current_track = 0;
366     do_floppy = _recal_interrupt;        // point to recal function.
367     output_byte(FD_RECALIBRATE);        // cmd: recalibrate.
368     output_byte(head << 2 | _current_drive); // param: head no + drive no.
369
370 // and the reset processing code in do_fd_request() is executed immediately.

```



```
370     if (reset)
371         do_fd_request();
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 {
377     output_byte(FD_SENSE);                                // send sense interrupt status cmd.
378     (void) result();                                     ;
379     output_byte(FD_SPECIFY);                            // send drive param setting cmd.
380     output_byte(cur_spec1);                            /* hut etc */
```



```

391     reset = 0;
392     cur_spec1 = -1;           // invalidated.
393     cur_rate = -1;
394     recalibrate = 1;         // set recalibration flag.
395     printk("Reset-floppy called\n|r");
396     cli();
397     do_floppy = reset_interrupt;    // point reset function.
398     outb_p(current_DOR & ~0x04, FD_DOR); // do reset command.
399     for (i=0 ; i<100 ; i++)          // delay for a while.
400         __asm__ ("nop");
401     outb(current_DOR, FD_DOR);      // enable controller again.
402     sti();
403 }
404
405 static void floppy_on_interrupt(void)           // floppy_on() interrupt.
406 {
407 /* We cannot do a floppy-select, as that might sleep. We just force it */
408
409     selected = 1;                // set drive selected flag.
410     if (current_drive != (current_DOR & 3)) {
411         current_DOR &= 0xFC;      // clear selected drive.
412         current_DOR |= current_drive; // set current drive.
413         outb(current_DOR, FD_DOR); // send current DOR.
414         add_timer(2, &transfer);   // add timer and related function.
415     } else

```

```

415         transfer();
416     }
417
    //// Floppy disk read/write request item processing function.
    // This is the main function in the floppy driver. Its main uses are: (1) Processing the case
    // where the reset flag or the re-correction flag is set; (2) Obtaining the parameter block
    // of the floppy drive by the request item using the device number in the request item; (3)
    // Starting the floppy disk read/write operation by using the kernel timer.
418 void do_fd_request(void)
419 {
420     unsigned int block;
421
    // First check if there is a reset flag or a recalibration flag. If one of them exists, the
    // function returns immediately after processing the relevant flag.
422     seek = 0;                                // reset seek flag.
423     if (reset) {
424         reset_floppy();
425         return;
426     }
427     if (recalibrate) {
428         recalibrate_floppy();
429         return;
430     }
    // The important aspects of this function start here. First use the INIT_REQUEST macro in the
    // 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
    // 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
    // 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
    // るために使用されます。
    // 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
    // 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。
431     // 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。
432     INIT_REQUEST;
433     floppy = (MINOR(CURRENT->dev)>>2) + floppy_type;
    // 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
    // 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
    // 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。
434     // 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。
435     if (current_drive != CURRENT_DEV) // the drive specified in the request.
436         seek = 1;
437     current_drive = CURRENT_DEV;
    // 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内なので
    // 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。
    // 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。
    // and the floppy disk request item is terminated to execute the next request item. Then
    // セクタ番号、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。
    // 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）。
438
439     block = CURRENT->sector;
440     if (block+2 > floppy->size) {

```

```
441         end_request(0);
442         goto repeat;
440     }
441     sector = block % floppy->sect; // the sector number on the track.
```

```

442     block /= floppy->sect;           // track number.
443     head = block % floppy->head;    // head no.
444     track = block / floppy->head;    // track no.
445     seek track = track << floppy->stretch; // seek track number related to drive type.

// 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
446 //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。
447     if (seek track != current track)
448         seek = 1;
449     sector++;                      // sectors count from 1.
450     if (CURRENT->cmd == READ)
451         command = FD READ;
452     else if (CURRENT->cmd == WRITE)
453         command = FD WRITE;
454     else
455         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。
// written, which takes a certain amount of time. So here ticks_to_floppy_on() is used to
// calculate the start delay time, and then use this delay to set a timer. The function
// floppy_on_interrupt() is called when the time expires.
add timer(ticks_to_floppy_on(current drive),&floppy on interrupt);

```

```
455  
456 }  
457 // The total number of data blocks contained in various types of floppy disk.  
458 static int floppy_sizes[] ={ // initial data for array blk_size[].  
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онт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);           // reset floppy int mask bit..
475 }
476

```

---

## 9.6.4 Information

### 9.6.3.1 Device number of the floppy disk drive

In Linux, the floppy drive's major number is 2, and the minor device number is determined by the floppy drive type and the floppy drive sequence number, which is:

---

FDマイナーNo.=TYPE \* 4 + DRIVE

---

Among them, DRIVE is 0-3, corresponding to floppy drive A, B, C or D; TYPE is the type of floppy drive, for example, 2 means 1.2M floppy drive, 7 means 1.44M floppy drive, as shown in Table 9-12. That is, it is the index value of the floppy type array (floppy\_type[]) defined in floppy.c, line 85.

表 9 - 1 2  フ ロ ッ ピ ー デ イ ス ク の 種 類  Type	Description
0	Not used.
1	360KB PC Floppy drive.
2	1.2MB AT Floppy drive.
3	360kB Floppy disk used in the 720kB drive.
4	3.5" 720kB Floppy drive.
5	360kB Floppy disk used in the 1.2MB drive.
6	720kB Floppy disk used in the 1.2MB drive.
7	1.44MB Floppy drive.

For example, type 7 indicates a 1.44MB drive, drive number 0 indicates an A drive, because  $7*4 + 0 = 28$ , so (2, 28) refers to the 1.44M drive A, the device number is 0x021C, and the corresponding device file name is /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 Floppy Drive Controller

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、データブロックを送信する際にはDMAコントローラ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（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には、表9-13に示すようなポートがあります。

表 9 - 1 3  フ ロ ッ ピ ー デ イ ス ク コ ン ト ロ ー ラ の ポ ー ト  I/O port	Name	Reed/Write	Register Name
0x3f2	FD_DOR	Write only	Digital Output Register (DOR)
0x3f4	FD_STATUS	Read only	Main Status Register (STATUS)

0x3f5	FD_DATA	Read only	Result Register(RESULT)
		Write only	Data Register(DATA)
0x3f7	FD_DIR	Read only	Digital Input Register (DIR)
	FD_DCR	Write only	Drive Control Register (DCR)(Transfer Rate)

The digital output register (DOR) port is an 8-bit register that controls the driver motor turn-on, drive select, start/reset FDC, and enable/disable DMA and interrupt requests. The meaning of each bit of this register is shown in Table 9-14.

表 9 - 1 4  デ ジ タル 出 力 レ ジ ス タ 定 義  Bit	Name	Description
7	MOT_EN3	Motor control for drive D: 1- Start motor; 0-Stop motor.
6	MOT_EN2	Motor control for drive C: 1- Start motor; 0-Stop motor.
5	MOT_EN1	Motor control for drive B: 1- Start motor; 0-Stop motor.
4	MOT_EN0	Motor control for drive A: 1- Start motor; 0-Stop motor.
3	DMA_INT	DMA and IRQ channel: 1-Enabled; 0-Disabled.
2	RESET	Controller reset: 1-Controller enabled; 0-Reset controller.
1	DRV_SEL1	
0	DRV_SEL0	00-11 is used to select floppy drive A-D respectively.

The FDC's Main Status Register (MSR) is also an 8-bit register that reflects the basic state of the floppy disk controller FDC and floppy disk drive FDD operation. Typically, the status bits of the main status register are read before the CPU sends a command to the FDC or before the FDC obtains the result of the operation to determine if the current FDC data register is ready and to determine the direction of data transfer. The definition of each bit of the MSR is shown in Table 9-17.

表 9 - 1 5  M S R の 各 ビ ツ ト の 定 義  Bit	Name	Description
7	RQM	Data port ready: FDC data register is ready.
6	DIO	Transfer direction: 1 - FDC -> CPU; 0 - CPU -> FDC
5	NDM	Non DMA mode: 1 - Controller not in DMA; 0 - DMA mode
4	CB	Controller busy: FDC is busy in executing a command.
3	DDB	Drive D is busy.
2	DCB	Drive C is busy.
1	DBB	Drive B is busy.
0	DAB	Drive A is busy.

The data port of the FDC corresponds to multiple registers: a write-only command and parameter register and a read-only result register. Only one register can appear on data port 0x3f5 at a time. When accessing a write-only register, the direction bit DIO of the main status register must be 0 (CPU -> FDC), and vice versa when accessing a read-only register. When accessing the write-only register to send commands and parameters, the command is 1 byte and the related parameter is 1-8 bytes. When accessing the read-only register to read the result, the result is only read after the FDC is not busy, and usually the result data has a maximum of 7 bytes.

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 Floppy disk controller command

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ーでは、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. Recalibration command (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。

	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description

Phase									
Command	0	0	0	0	0	0	1	1	Recalibration command code: 0x07
Parameter	1	0	0	0	0	0	0	US1	Disk drive number.
Execution									The head moves to 0 track
Result	None.								You need to use the command to get the

										execution result.
--	--	--	--	--	--	--	--	--	--	-------------------

## 2. Head seek command (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9-17に示します。

表 9 - 1 7 ヘ ッ ド シ ー ク コ マ ン ド ( F D - S E E K ) の フ オ 一 マ ツ	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description

ト Phase											
Command	0	0	0	0	0	1	1	1	1	Head seek command code: 0x0F	
Parameter	1	0	0	0	0	0	HD	US1	US2	Head number, Drive number.	
	2	C									
Execution											The head moves to the specified track.
Result		None.									You need to use the command to get the execution result.

### 3. Read sector data command (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込み、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ドィスク・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 - 1 8 セ ク タ 一 デ 一 タ 読 み 出 し コ	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description

マ ン ド ( F D -R E A ) の フ オ ー マ ツ ト Phase														
Command	0	MT	MF	SK	0	0	1	1	0	Read sector command code: 0xE6 (MT=MF=SK=1)				
Parameters	1	0	0	0	0	0	0	US1	US2	Drive number				
	2	C								Track number (Cylinder address, 0 to 255)				
	3	H								Head number (Head address, 0 or 1)				
	4	R								Start sector number (Sector address)				
	5	N								Sector size code (N=0..7: 128,256,512,,16KB)				
	6	EOT								Final sector number. of the track (End of Track)				
	7	GPL								Length of gap between sectors (3)				
	8	DTL								The number of bytes in sector, when N=0				
Execution										Data is transferred from disk to system				
Result	1	ST0								Status byte 0				

2	ST1	Status byte 1
3	ST2	Status byte 2
4	C	Track number
5	H	Header number
6	R	Sector number
7	N	Sector size code (0..7: 128,256,512,...,16KB)

Among them, the meanings of MT, MF and SK are:

- MT represents multi-track operation. MT = 1 indicates that two heads are allowed to operate continuously on the same track.
- MF indicates the recording method. MF=1 means to select the MFM recording mode, otherwise it is the FM recording mode.
- SK indicates whether to skip the sector with the delete flag. SK=1 means skip.

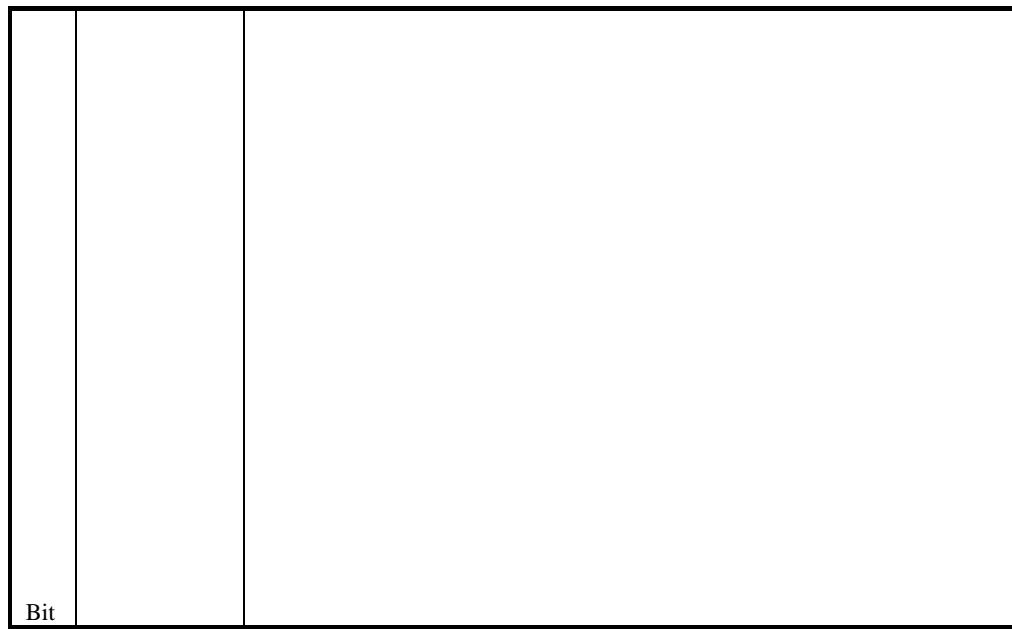
返された3つのステータスバイト ST0, ST1, ST2の意味をそれぞれ表9-19, 表9-20, 表9-21に示す。

Bit	Name	Description
7		Reason for interrupt. 00 - Normal termination of command; 01 - Abnormal termination of command; 10 - Invalid command; 11 - Abnormal termination caused by Polling.
6	ST0_INTR	
5	ST0_SE	Seek End. The controller completed a SEEK or RECALIBRATE command, or a READ/WRITE with implied seek command.
4	ST0_ECE	Equip. Check Error. Recalibration track 0 error.

3	ST0_NR	Not Ready. Floppy disk drive not ready.
2	ST0_HA	Head address. The current head number when interrupt occurs.
1	ST0_DS	Drive Select. Drive number when interrupt occurs.
0		00 - 11 corresponds to drive 0 - 3 respectively

Bit	Name	Description
7	ST1_EOC	End of Cylinder. Tried to access a sector beyond the final sector of the track.
6		Unused. This bit is always 0.
5	ST1_CRC	The controller detected a CRC error in ID field or the Data field of a sector.
4	ST1_OR	Over Run. Data transfer timeout, DMA controller failure
3		Unused (0).
2	ST1_ND	No Data. The specified sector was not found.
1	ST1_WP	Write Protect.
0	ST1_MAM	Missing Address Mask. Sector ID address mark not found.

	Name	Description



7		Unused (0).
6	ST2_CM	Control Mark. When SK=0, the read data encounters the delete flag.
5	ST2_CRC	CRC error. The sector data field CRC check error.
4	ST2_WC	Wrong Cylinder. The track number C of the sector ID info does not match.
3	ST2_SEH	Scan Equal Hit. The scanning conditions meet the requirements.
2	ST2_SNS	Scan Not Satisfied: Scanning conditions do not meet the requirements
1	ST2_BC	Bad Cylinder. The track C = 0xFF in the sector ID info, the track is bad.
0	ST2_MAM	Missing Address Mask. The sector ID data address mark not found.

#### 4. Write sector data command (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。略称はリードコマンドと同じ意味を持ちます。

	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description

Phase												
Command	0	MT	MF	0	0	0	1	0	1	Write data command code: 0xC5 (MT=MF=1)		
Parameters	1	0	0	0	0	0	0	US1	US2	Floppy drive number.		
	2	C								Track number.		
	3	H								Head number.		
	4	R								Start sector number		
	5	N								Sector size code.		
	6	EOT								Final sector number. of the track		
	7	GPL								Length of gap between sectors (3)		
	8	DTL								The number of bytes in sector, when N=0		
Execution										Data is transferred from the system to the disk		
Result	1	ST0								Status byte 0		
	2	ST1								Status byte 1		
	3	ST2								Status byte 2		
	4	C								Track number		
	5	H								Head number		
	6	R								Sector number		
	7	N								Sector size code.		

##### 5. Check interrupt status command (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、ステート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 - 2 3  チ エ ツ ク イ ン タ ラ プ ト ス テ ー タ ス コ マ ン ド ( F D - S E N S E H ) の	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
--	-----	----	----	----	----	----	----	----	----	-------------

フォーマット Phase										
Command	0	0	0	0	0	1	0	0	0	Detection interrupt status cmd code: 0x08
Execution										
Result	1	ST0								Status byte 0
	2	C								Track number of the current head.

#### 6. Set drive parameter command (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传送モード、つまりドライブモータのステッププレート (SRT)、ヘッドのロード/アンロード (HLT/HUT) 時間、传送に 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	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description

マ ン ド ( F D - S P E C I F Y ) の フ オ 一 マ ツ ト Phase												
Command	0	0	0	0	0	0	1	1	Set parameter command code: 0x03			
Parameters	1	SRT (Unit: 1ms)	HUT (Unit: 16ms)				Motor step rate, head unloading time					
	2	HLT (Unit: 2ms)				ND	Head load time, non-DMA mode					
Execution												
Result		None.										

#### 9.6.3.4 Floppy disk controller programming

In a PC, the floppy disk controller generally uses a compatible chip of NEC PD765 or Intel 8287A, such as Intel's 82078. Since the driver of the floppy disk is relatively complicated, the programming method of the floppy disk controller composed of such a chip is described in detail below. Typical disk operations include not only sending commands and waiting for controllers to return results. The control of a floppy disk drive is a low-level operation that requires the program to interfere with its execution at different stages.

##### 1. Interaction between command and result phases

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行わないように、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マ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もわかります。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。

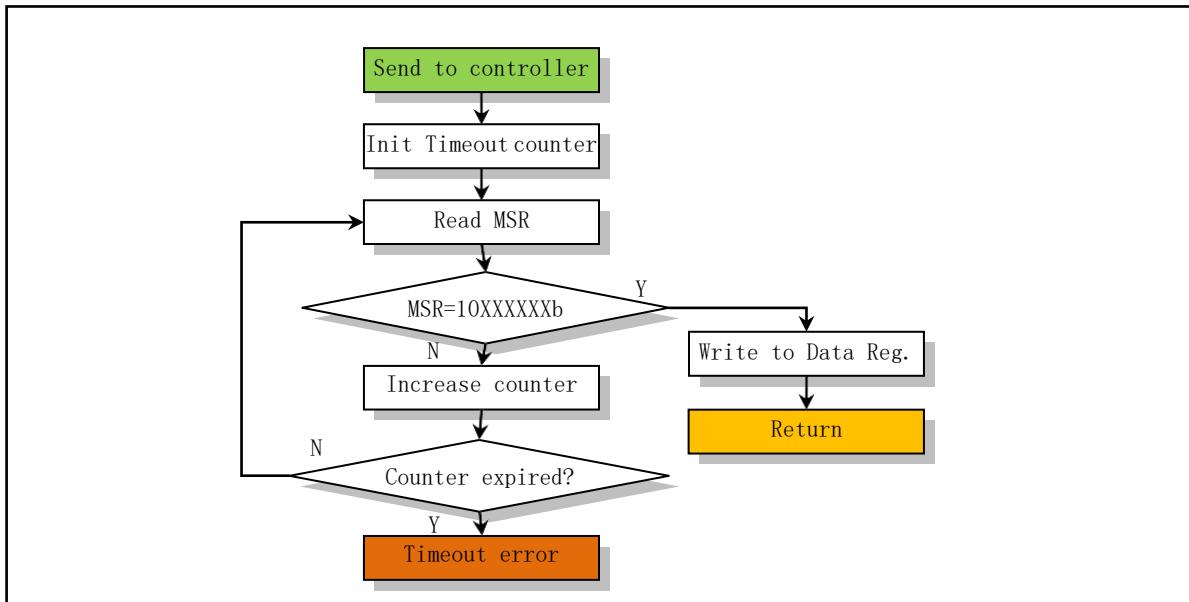


図9-9 フロッピーコントローラへのコマンドまたはパラメータバイトの送信

## 2. Floppy disk controller initialization

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バ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。ただし、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. Drive recalibration and head seek

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。

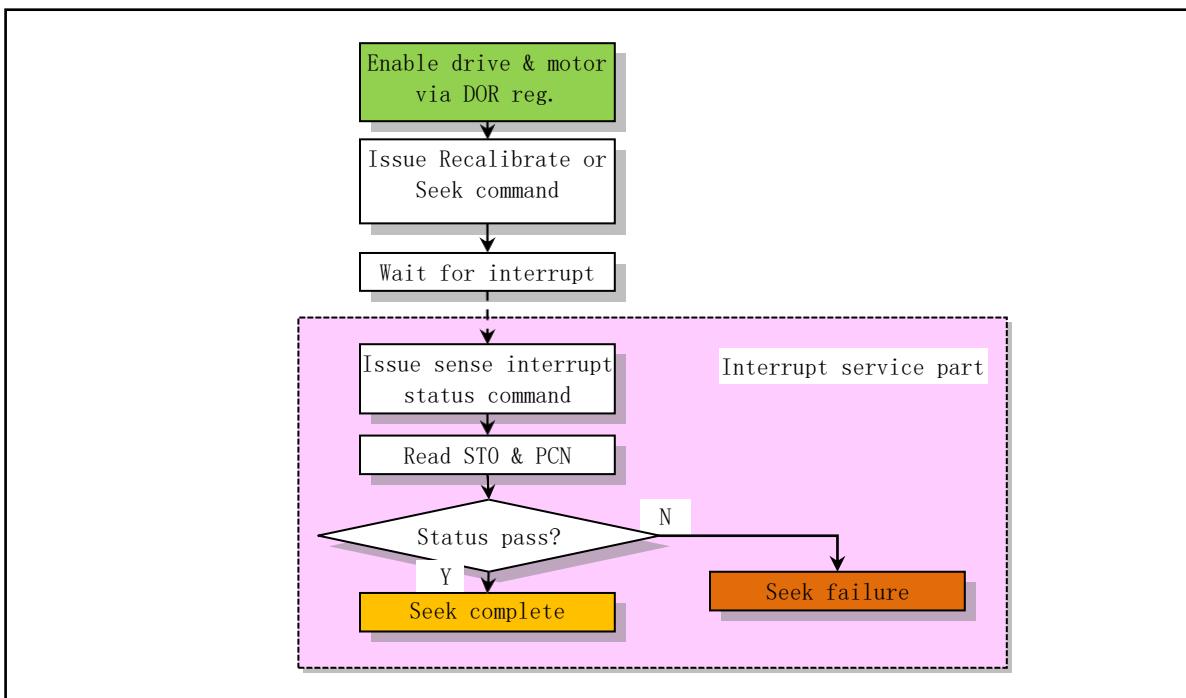


図9-10 再校正とシーク操作

#### 4. Data read/write operations

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、ドライブモーターの電源を入れ、ヘッドを正しいトラックに配置し、次にDMAコントローラーを初期化し、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。

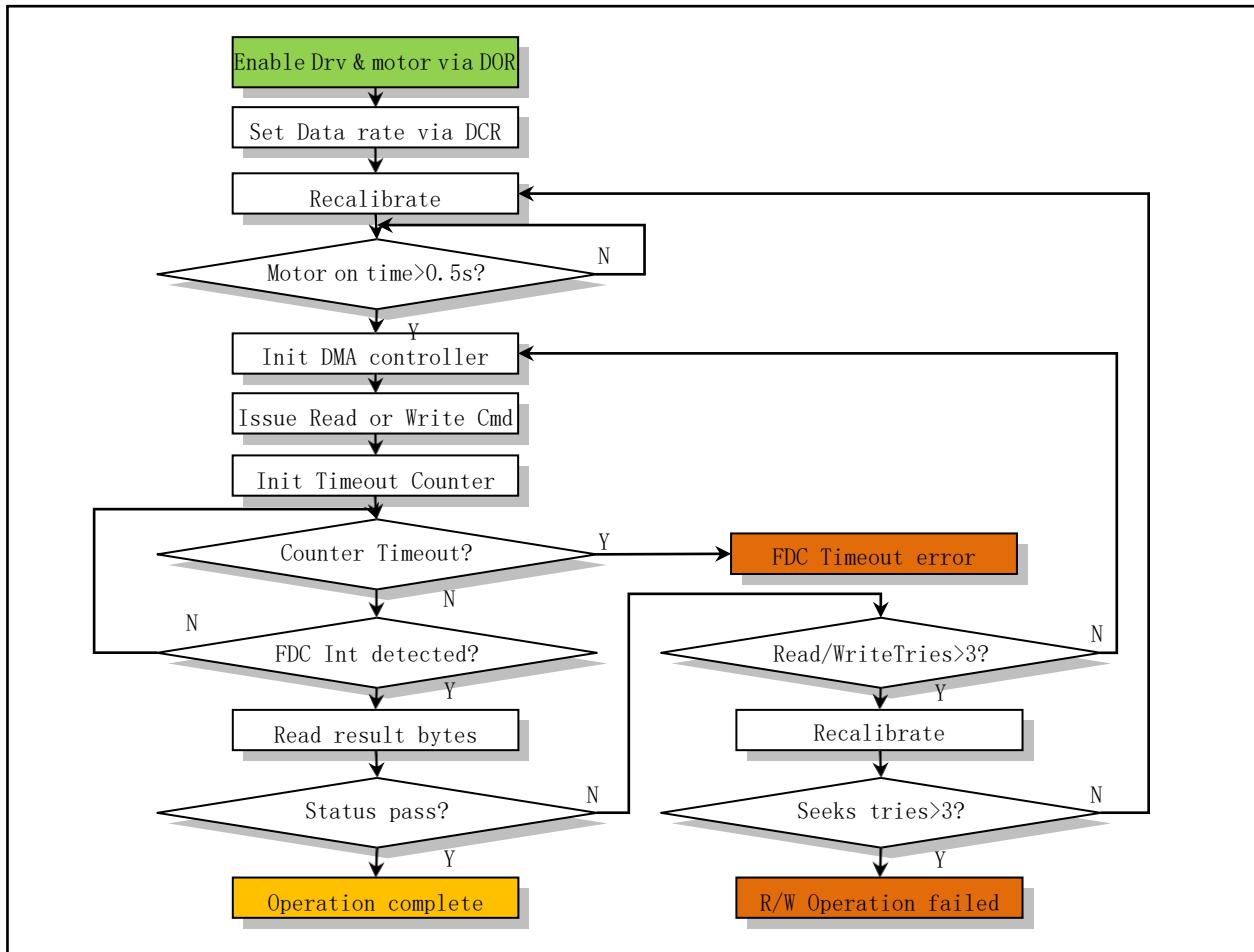


図9-11 データリード/ライトの動作フロー図

The disk drive's motor must first reach normal operating speed before the disk can start transferring data. For most 3 1/2-inch floppy drives, this boot time takes approximately 300ms, while the 5 1/4-inch floppy drive takes approximately 500ms. This startup delay time was set to 500ms in the floppy.c program.

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、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

ヘッドを複数回繰り返し、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. Disk format operations

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る。

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 Controller Programming

- 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コントローラー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。

#### 2. Initialization of the DMA controller.

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、バルク（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）しかないので、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ます、各転送のデータ長は64KBを超えることはできません。

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ぬことがわかります。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 DR AM ペ ー ジ 対 応 メ モ リ ア ド レ ス 範 囲 DMA page	Address range (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

3. Data transmission
4. 初期化完了後、DMAコントローラのマスクレジスタを変更し、DMAチャネル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。

### 5. End of transmission

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ーは、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つのポートの操作が必要です。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- 2 6 各 D M A チ ヤ ネ ル が 使 用 す る ペ ー ジ ,, ア ド レ ス ,, カ	Page register	Base address register	Word count register

ウ ン ト レ ジ ス タ ボ ー ト DMA Channel			
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

For normal DMA applications, there are five common registers that control the operation and state of the DMA controller. These are the Command Register, the Request Register, the Single Mask Register, the Mode Register, and the Clear Pre/Post Pointer Trigger, as shown in Table 9–27. The Linux 0.12 kernel mainly uses three shaded register ports (0x0A, 0x0B, 0x0C) in the table.

Register Name	Read/Write	Port address	
		Channel 0 - 3	Channel 4 - 7
Status register/Command register	Read/Write	0x08	0xD0
Request register	Write	0x09	0xD2
Single Channel mask register	Write	0x0A	0xD4
Mode register	Write	0x0B	0xD6
Clear First/Last Flip-Flop	Write	0x0C	0xD8
Temporary register/Mater Clear	Read/Write	0x0D	0xDA
Clear Mask register	Write	0x0E	0xDC
Full Mask register	Write	0x0F	0xDE

Command Register -- The command register is used to specify the operational requirements of the DMA controller chip and set the overall state of the DMA controller. Usually it does not need to change after boot initialization. In the Linux 0.12 kernel, the floppy driver directly uses the ROM BIOS settings after booting. For reference, the meaning of the bits of the command register is listed here, as shown in Table 9–28. Note that when reading the same port, you will get the information of the DMA controller status register.

表 9 - 2 8  D M A コ マ ン ド	Description

レジスタマップ Bit	
7	DMA response peripheral signal DACK: 0-DACK active low; 1-DACK active high.
6	Peripheral request DMA signal DREQ: 0-DREQ active low; 1-DREQ active high.
5	Write mode selection: 0-select late write; 1-select extended write; X-if bit 3=1.
4	DMA channel priority mode: 0-fixed priority; 1-rotating priority.
3	DMA cycle selection: 0 - normal timing cycle (5); 1- compression timing cycle (3); X - if bit 0 = 1.
2	Start DMA controller: 0 - enable controller; 1 - disables the controller.
1	Channel 0 address hold: 0 - disable channel 0 address hold; 1 - allow channel 0 address to hold; X - if bit 0 = 0.
0	Memory transfer mode: 0 - disable memory to memory; 1 - enable memory to memory.

Request Register -- The 8237A can respond to requests for DMA service which are initiated by software as well as by a DREQ. The request register is used to record the request service signal DREQ of the peripheral to the channel, one bit for each channel. When DREQ is active, it corresponds to position 1, which is set to 0 when

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、リクエストチャネルサービスのバイトフォーマットを表9-29に示す。

Bit	Description
7 -3	Not used.
2	Set flag. 0 - Request bit is set; 1 - Request bit is reset (set to 0).
1	Channel selection. 00-11 selects channel 0-3.
0	

Mask Register -- The port of the single mask register is 0x0A (0xD4 for 16-bit channels). A channel is masked, meaning that the DMA request signal DREQ issued by the peripheral using the channel does not get the response from the DMA controller, therefore, the DMA controller cannot be operated on the channel. Each

channel has associated with it a mask bit which can be set to disable the incoming DREQ. The meaning of each bit of this register is shown in Table 9-30.

表 9 - 3 0  D M A シ ン グ ル マ ス ク レ ジ ス タ の 各 ビ ツ ト の 意 味  Bit	Description
7 -3	Not used.
2	mask flag. 1 - set mask bit; 0 - Clear mask bit.
1	
0	Channel selection. 00-11 selects channel 0-3.

Mode Register -- The mode register is used to specify how a DMA channel operates. The meaning of each bit of this register is shown in Table 9-31. In the Linux 0.12 kernel, two setting modes, read disk (0x46) and write disk (0x4A), are used. According to Table 9-31, use DMA channel 2, read disk (write memory) transfer, disable autoinitialization, select address increment and use single byte mode; 0x4A means set mode: use DMA channel 2, write disk (read memory) transfer, disable autoinitialization, select address increment and use single byte mode.

Bit	Description
9	
8	
7	
6	
5	Select the transfer mode: 00-request mode; 01-single byte mode; 10-block byte mode; 11-cascade mode.
4	Address method. 0 - address increment; 1 - address decrement.
3	Autoinitialization. 0-Autoinitialization disable; 1-Autoinitialization enable.
2	Transmission type: 00-verify transfer; 01-write memory transfer; 10-read memory transfer; 11- illegal; XX - if bits 6-7=11.
1	
0	Channel selection. 00-11 selects channel 0-3 respectively.

Since the channel address and count registers can read and write 16-bit data, you need to perform two write operations, one low byte and one high byte at a time. Which byte is actually written is determined by the state of the software command clear first/last flip-flop. This command must be executed prior to writing or reading new address or word count information to the 8237A. This initializes the flip-flop to a known state so that subsequent accesses to register contents by the CPU will address upper and lower bytes in the correct sequence. Port 0x0C is used to initialize the byte-order flip-flop to the default state before reading or writing the address or count information in the DMA controller. When the clear byte first/last flip-flop is 0, the low byte is accessed; when it is 1, the high byte is accessed. The flip-flop changes once per visit. The 0x0C port can be written to set the clear first/last flip-flop to the 0 state.

1. 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。
  2. Turn off the interrupt to eliminate any interference;
  3. Modify the mask register (port 0x0A) to mask the DMA channel that needs to be used. For the floppy disk driver is channel 2;
  4. Write to the 0x0C port and set "clear first/last flip-flop" to the default state;
  5. Write mode register (port 0x0B) to set the operation mode word of the specified channel;
  6. Write the address register (port 0x04) to set the offset address in the memory page used by the DMA. Write the low byte first, then write the high byte;
  7. Write the page register (port 0x81) to set the memory page used by the DMA;
  8. Write the count register (port 0x05), set the number of bytes for DMA transfer, which should be the transfer length - 1. We also need to write once for the high and low bytes. In this book, the length of the floppy disk driver required by the DMA controller is 1024 bytes, so the length of the write DMA controller should be 1023 (ie 0x3FF);
  9. Modify the mask register (port 0x0A) again to enable the DMA channel;
  10. Finally, enable the interrupt to allow the floppy controller to issue an interrupt request to the system after the transfer is complete.

## 9.7 Summary

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。

