# 13 Memory Management (mm)

Intel 80X86アーキテクチャのシステムでは、Linuxカーネルのメモリ管理プログラムはページング管理を採用しています。ページディレクトリとページテーブル構造を利用して、カーネルの他の部分からのメモリの申請と解放を処理する。メモリ管理はメモリページ単位で行われ、メモリページとは、アドレスが連続する4Kバイトの物理メモリを指す。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により、指定されたメモリページのアドレスと使用状況を管理することができます。リスト13-1に示すように，Linux 0.12のmemory managementディレクトリには3つのファイルがあります。

|  |  |  |
| --- | --- | --- |
| リスト 13-1 メモリ管理サブディレクトリのファイル一覧  Filename | Size Last Modified Time(GMT) | Desc |
| Makefile | 1221 bytes 1992-01-12 19:49:22 |  |
| memory.c | 13464 bytes 1992-01-13 22:57:04 |  |
| page.s | 508 bytes 1991-10-02 14:16:30 |  |
| swap.c | 5193 bytes 1992-01-13 15:46:41 |  |

## / その中でもpage.sアセンブリファイルは比較的短く、メモリページ例外割り込みサービスプログラム（INT14）が含まれているだけで、主にページフォルトとページ書き込み保護の処理を実現しています。Memory.cは、メモリページ管理のコアプログラムです。メモリの初期化操作、ページディレクトリやページテーブルの管理など、カーネルのメモリアプリケーション処理の部分に使用されます。swap.cは、メモリページの交換管理を行うプログラムで、主に交換マッピングのビットマップ管理機能や、スイッチングデバイスのアクセス機能などがあります。

## 13.1 Main Functionalities

インテル80X86のCPUでは、プログラムはアドレッシングの際に、セグメントとセグメント内オフセットからなるアドレスを使用する。このアドレスは、物理メモリのアドレスには直接使用されないため、仮想アドレスと呼ばれます。物理メモリをアドレス指定するためには、仮想アドレスを物理メモリのアドレスにマッピングまたは変換するアドレス変換機構が必要である。このアドレス変換機構は、メモリ管理の主要な機能の1つである（もう1つの主要な機能は、メモリアドレスの保護機構である）。仮想アドレスは、セグメント管理機構によって中間アドレス（CPUの32ビットリニアアドレス）に変換され、このリニアアドレスがページング機構によって物理アドレスにマッピングされる。

Linux カーネルがどのようにメモリ操作を管理しているかを理解するためには、メモリのページング管理の仕組みを理解し、そのアドレッシングメカニズムを理解する必要があります。ページング管理の目的は、物理的なメモリページを直線的なアドレスにマッピングすることです。本章のメモリ管理プログラムを解析する際には、与えられたアドレスがリニアアドレスを指しているのか、実際の物理メモリのアドレスを指しているのかを明確に区別する必要があります。

### Intel 80X86 CPU Protected Modeにおけるメモリ管理の詳細については、第4章を参照してください。読みやすくするために、メモリページング管理機構の関連内容をさらに説明します。

### 13.1.1 Memory paging mechanism

インテル80X86システムでは、図13-1に示すように、メモリページディレクトリテーブルとページテーブルからなる2階層のテーブルでメモリページング管理を行っています。ページディレクトリテーブルとページテーブルは、以下の図13-4に示すように同じ構造をしており、テーブルの項目構造も同じです。ページディレクトリテーブルの各エントリ（ページディレクトリエントリと呼ぶ、4バイト）は、ページテーブルのアドレスに使用され、各ページテーブルエントリ（4バイト）は、物理メモリのページを指定するのに使用される。したがって、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指定されると、対応する物理メモリページを一意に決定することができる。ページディレクトリテーブルは1ページ分のメモリを占有するため、最大1024個のページテーブルを指定することができ、各ページテーブルも1ページ分のメモリを占有するため、1つのページテーブルも最大1024個の物理メモリページを指定することができます。したがって、32ビットの80X86CPUでは、ページディレクトリテーブルでアドレス指定されたすべてのページテーブルは、合計で1024×1024×4096＝4Gのメモリ空間をアドレス指定できることになる。Linux 0.12カーネルでは、すべてのプロセスが1つのページディレクトリテーブルを共有し、各プロセスが独自のページテーブルを持っています。カーネルのコードとデータのセグメント長は16MBと規定されており、4つのページテーブル（つまり4つのページディレクトリエントリ）を使用します。カーネルコードとデータセグメントは、セグメンテーション機構による変換後、リニアアドレス空間の最初の16MBの範囲に配置され、その後、ページング機構によって変換され、16MBの物理メモリに1つずつ直接マッピングされます。つまり、カーネルセグメントの場合、そのリニアアドレスは物理アドレスになります。

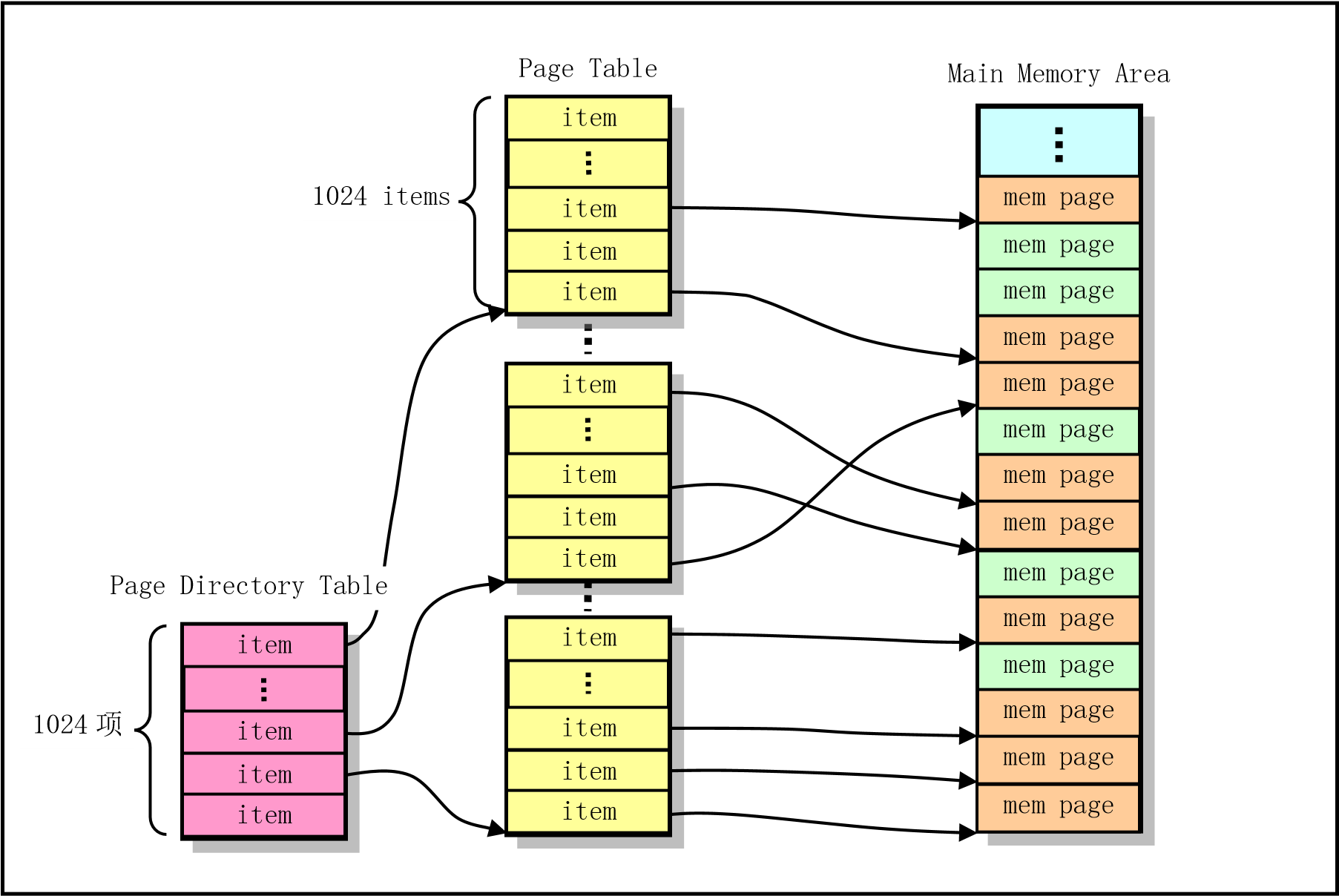


図13-1 ページディレクトリテーブルとページテーブルの構造図

ユーザープロセスやカーネルの他の部分では、メモリを申請する際にリニアアドレスが使用されます。では、リニアアドレスは、この2つのテーブルを使って、どのように物理アドレスにマッピングされるのでしょうか。ページングの仕組みを利用するためには

32ビットリニアアドレスは、3つの部分に分かれており、対応する物理メモリページ上のページディレクトリエントリ、ページテーブルエントリ、オフセットアドレスを指定することで、図13-2のように、リニアアドレスで指定された物理メモリの位置を間接的に指定できるようになっています。

0

12

11

22

21

31

Page Table

Page Frame

Page Dir Table

Dir item

Linear Address:

Dir

Index

Page Index

Offset

Page item

CR3

Physical

Memory

図13-2 リニアアドレス変換の模式図

リニアアドレスのビット31-22は、ページディレクトリのディレクトリエントリを決定するために使用され、ビット21-12は、ページディレクトリエントリで指定されたページテーブルのページテーブルエントリのアドレスとして使用され、最後の12ビットは、ページエントリで指定された物理メモリページのオフセットアドレスとして使用されます。

メモリ管理機能では、リニアアドレスから実際の物理アドレスへの変換計算が大量に行われます。あるプロセスのリニアアドレスに対して、図13-2に示すアドレス変換関係により、リニアアドレスに対応するページディレクトリエントリを簡単に見つけることができます。ディレクトリエントリが有効（使用）であれば、ディレクトリエントリ内のページフレームアドレスは、物理メモリ内のページテーブルのベースアドレスを指定します。そして、リニアアドレスのページテーブルエントリポインタとの組み合わせで、ページテーブルエントリが有効であれば、ページテーブルエントリの指定されたページフレームアドレスに基づいて、指定されたリニアアドレスに対応する実際の物理メモリページのアドレスを最終的に決定することができます。逆に、既知の物理メモリページのアドレスから対応するリニアアドレスを見つける必要がある場合は、ページディレクトリテーブル全体とすべてのページテーブルを検索する必要がある。物理メモリページが共有されている場合は、対応するリニアアドレスが複数見つかることもある。図13-3は、あるリニアアドレスが物理メモリページにマッピングされる様子を図示したものである。最初のプロセス（タスク0）の場合、そのページテーブルは、ページディレクトリテーブルの後にあり、合計4ページになります。アプリケーションのプロセスでは、そのページテーブルが使用するメモリは、プロセスの生成時にメモリマネージャに適用されるため、メインメモリ領域にあります。

2

3

1

Page Directory Table

Page Table of

the p

rocess

Memory page

M

16

0

Dir item

Page item

Offset

32

-

bit Linear Address

Main Memory Area

Physical Memory

1

page

Relevant memory

address

図13-3 リニアアドレスに対応する物理アドレス

1つのシステムには複数のページディレクトリテーブルが同時に存在しますが、一度に利用できるのは1つのページディレクトリテーブルのみです。現在使用されているページディレクトリテーブルは、現在のページディレクトリテーブルの物理メモリアドレスを格納しているCPUのレジスタCR3によって決定されます。しかし、本書で紹介するLinuxカーネルでは、カーネルコードとすべてのプロセスが1つのページディレクトリテーブルを共有しています。

図13-1では、各ページテーブルエントリに対応する物理メモリページは、4Gアドレスの範囲内でランダムであり、メモリマネージャが設定したページテーブルエントリのページフレームアドレスの内容によって決定されることがわかる。各ページテーブルエントリは、図13-4に示すように、ページフレームアドレス、アクセスフラグビット、ダーティ（書き換え）フラグビット、プレゼンスフラグビットで構成されています。



31

12

11

10

9

8

7

6

5

4

3

2

1

0

P

0

0

U

/

S

R

/

W

A

D

Page Fram Address

Bit

31...12

0

0

AVL

図13-4 ページディレクトリとページテーブルのエントリ構造

ページフレームアドレスは、メモリのページの物理的な開始アドレスを指定します。メモリページは4Kアドレス境界上にあるため、その下位12ビットは常に0であり、エントリの下位12ビットは他の目的に使用することができます。ページディレクトリテーブルでは、テーブルエントリのページフレームアドレスは、ページテーブルの開始アドレスです。第2レベルのページテーブルでは、ページテーブルエントリのページフレームアドレスは、目的のメモリ操作の物理的なメモリページアドレスを含んでいます。

図中の存在ビット（P）は、ページテーブルのエントリがアドレス変換処理に使用できるかどうかを決定します。P=1の場合は、そのエントリが使用可能であることを意味します。ディレクトリ・エントリや第2レベル・エントリがP=0の場合、そのエントリは無効であり、アドレス変換処理に使用することはできません。この時点では、エントリの他のすべてのビットがプログラムに使用可能であり、プロセッサはこれらのビットをテストしません。

CPUがアドレス変換のためにページテーブル・エントリを使用しようとしたとき、このときにページテーブル・エントリのいずれか1つのP=0であれば、プロセッサはページ例外割り込み信号を発行します。この時点で、ページ障害割り込み例外ハンドラは、要求されたページを物理メモリにマッピングしてロードすることができ、例外の原因となった命令は再実行されます。

アクセスされた（A）ビットと修正またはダーティ（D）ビットは、ページの使用状況に関する情報を提供するために使用されます。これらのビットはハードウェアによって設定されますが、ページディレクトリエントリのmodifiedビットを除き、リセットされることはありません。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小さな違いは、ページテーブルエントリにはダーティビット（D）があるのに対し、ページディレクトリエントリにはないことです。

メモリのページに対してリード／ライト操作が行われる前に、CPUは関連するディレクトリエントリと二次ページテーブルエントリのアクセス済みビットを設定します。二次ページテーブルエントリがカバーするアドレスに書き込む前に、プロセッサは二次ページテーブルエントリのモディファイドビット（D）を設定し、ページディレクトリエントリのビット（D）は使用されません。必要なメモリが実際の物理メモリの量を超えている場合、メモリマネージャはこれらのビットを使用して、どのページをメモリから取り出してスペースを確保するかを決定することができます。また、メモリマネージャはこれらのビットを検出し、リセットする責任があります。

### Read/Writeビット(R/W)とUser/Supervisorビット(U/S)はアドレス変換には使用されませんが、ページングレベルの保護機構はアドレス変換処理中にCPUが同時に実行します。

### 13.1.2 Physical Memory Allocation and Management

以上の基本的な考え方で、Linux システムがどのようにメモリを管理しているかを説明できますが、まず、Linux カーネルによるメモリ空間の使用方法を理解する必要があります。Linux 0.12カーネルの場合、デフォルトで16Mまでの物理メモリをサポートしています。16MBのメモリを搭載した80X86コンピュータシステムでは、図13-5に示すように、Linuxカーネルが物理メモリの最前部を占めています。図中の「end」というラベルは、カーネルモジュールが終了する位置を示しています。その後に、最大メモリアドレスが4Mのキャッシュバッファが続く。キャッシュバッファは、ディスプレイメモリとROM BIOSによって2つのセクションに分かれています。残りのメモリ部分をメインメモリエリアと呼ぶ。メインメモリ領域は、本章の手順で割り当てられ、管理されます。システム内にRAM仮想ディスクがある場合は、仮想ディスクが占有するメモリ領域をメインメモリ領域の前から差し引く必要があります。主記憶領域を使用する必要がある場合は、本章のメモリ管理プログラムに申請する必要があります。申請の基本単位はメモリページである。図13-5に物理メモリの各部の機能を示す。

end

0

4

M

4.5

M

M

16

M

1

K

640

Kernel Code

Buffer Cache

RAMDisk

Main Memory Area

Video Mem &

BIOS ROM

図13-5 メインメモリ領域の模式図

主な機能

第6章「システムの起動」で、Linuxのページディレクトリとページテーブルは、head.sというプログラムに設定されていることがすでにわかっている。head.sプログラムは、物理アドレス0にページディレクトリテーブルを格納し、その後に4つのページテーブルを格納する。この4つのページテーブルは、カーネルが占有するメモリ領域のマッピング操作に使用される。タスク0のコードとデータはカーネル領域に含まれているので、タスク0もこれらのページテーブルを使用します。他の派生プロセスは、自分のページテーブルを格納するためにメインメモリ領域のメモリページを要求します。この章の2つのプログラムは、これらのテーブルを管理して、主記憶領域のメモリページの割り当てを実現するために使用されます。

物理的なメモリを節約するために、fork()を呼び出して新しいプロセスが生成されると、新しいプロセスは元のプロセスと同じメモリ領域を共有します。一方のプロセスが書き込み操作を開始したときだけ、システムはそのプロセスに追加のメモリページを割り当てます。これがコピーオンライトの概念である。

### page.sプログラムは、ページフォルトや例外処理（INT14）の実装に使用されます。ページフォルトやページ書き込み保護による割り込みの場合、割り込みハンドラはmemory.cのdo\_no\_page()関数とdo\_wp\_page()関数をそれぞれ呼び出します。do\_no\_page()は、必要なページをブロックデバイスからメモリ指定位置に取り込みます。共有メモリページの場合、do\_wp\_page()は書き込まれているページをコピーし(コピーオンライト)、ページの共有をキャンセルします。

### 13.1.3 Linear address space allocation

本章のコードを読む際には、図13-6に示すように、プログラムの論理アドレス空間におけるコードとデータの分布についても理解しておく必要があります。

edata

etext

end

Code

Data

nr\*64M

(

nr+1)\*64M

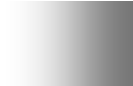
end\_code

end\_data

Stack Pointer

start\_code

bss



brk

\* nr

is task num

ber

Environ & Params block

図13-6 論理アドレス空間におけるプロセスの分布

### 各プロセスが占有する論理アドレス空間は、リニアアドレス空間のnr\*64MBの位置から始まり（nrはタスク番号）、論理アドレス空間は64MBの範囲を占有しています。環境パラメータデータブロックの最後の部分は最大128Kの長さで，その左側は開始スタックポインタである． 另外，図中，bssはプロセスによって初期化されないデータセグメントである．プロセスが生成されると、bssセグメントの最初のページはすべて0に初期化されます。

### 13.1.4 Transform between logical, linear, and physical address

カーネルのメモリ管理コードでは、プログラムの論理アドレス（または仮想アドレス）、CPUのリニアアドレス、物理メモリアドレスの間の変換操作が頻繁に発生します。例えば、ページテーブルをコピーする際には、与えられたページディレクトリエントリ（PDE）を線形変換して、対応するページテーブル（PT）の物理メモリアドレスを取得する必要がある。コピーオンライトの操作に関しては、線形アドレス空間のページを物理アドレスにマッピングする操作があり、ページを共有しようとする際には、プログラム論理アドレスのページをCPUの線形アドレス空間にマッピングする操作がある。以下、両者の変換操作を個別に説明する。カーネルのメモリ管理は、通常、4KBのメモリページ単位で運用されているので、まず、アドレスからページ開始アドレスへの変換方法を示し、その後、これらの異なるアドレス空間のページがどのように変換されるかを説明しよう。

1. Conversion of address to corresponding page address

ページアドレスは4KBのメモリアドレス境界から始まるので（つまり、ページアドレスの下位12ビットは0）、指定されたアドレス「addr」を含むメモリページのアドレス「Page\_addr」は次のようになります。

Page\_addr = addr & 0xfffff000;

この3つのアドレス空間では、そのページアドレスの計算方法は同じです。以下に、ページアドレスの変換計算方法を説明します。

1. Linear address and logical address decomposition

CPUのページング機構によると、32ビットのリニアアドレス「addr」は、図13-2のように、ページディレクトリエントリPDE番号（ビット31-22）、ページテーブルエントリPTE番号（ビット21-12）、ページ内オフセット（ビット11-0）に分解できる。したがって、これらの3つの部分を独立して取得するための一般的な計算式は次のようになります。

/ ページディレクトリエントリ番号。PDE\_No = (addr >> 22)となります。

ページテーブルのエントリ番号。PTE\_No = (addr >> 12) & 0x3ff;

ページ内のオフセット。オフセット = (addr & 0xfff);

同様に、アドレス変換操作を行う際、プロセス論理アドレス空間の論理アドレス「Vaddr」も、これら3つの部分から「論理的に」構成されているとみなすことができます。ただ、これらをCPUのリニアアドレス空間にマッピングする際には、リニアアドレス空間のプログラムコードベースアドレス「Base」の対応部分を追加する必要がある。

Logic\_PDE\_No = (Vaddr >> 22)となります。

Logic\_PTE\_No = (Vaddr >> 12) & 0x3ff です。

Logic\_Offset = (Vaddr & 0xfff)です。

このカーネルでは，各プロセスが64MB単位でリニアアドレス空間を割り当てているため，各プロセスのコードベースアドレスに対応するページテーブルエントリ番号とページ内オフセット値はいずれも0であり，変換処理の際にはコードベースアドレスのページディレクトリエントリ番号のみを考慮すればよいことになります。実際のコードでは、プロセス'p'のコードベースアドレス'Base'は'p->start\_code'なので、プログラム中の論理アドレス'Vaddr'がリニアアドレス空間に対応する場合、アドレスの3つの部分は

/ PDE\_No = (Base >> 22) + Logic\_PDE\_No;

= (p->start\_code >> 22) + (Vaddr >> 22)です。

PTE\_No = Logic\_PTE\_No

= (Vaddr >> 12) & 0x3ff;

オフセット = Logic\_Offset

= (Vaddr & 0xfff) となります。

3. プログラムページの論理アドレスから物理アドレスへの変換

いわゆる論理ページアドレスとは、図13-6に示すように、コードベースアドレスと呼ばれるプログラムプロセス'p'コードの開始アドレスから計算したアドレスをページアドレスとしている。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構造（図13-4参照）と、上記の論理アドレスの分解によれば、各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は4バイトを占めるので、テーブルエントリ番号を2ビット左シフトしてテーブル内のエントリのオフセットを得て、さらにページディレクトリテーブルの物理ベースアドレスPDT\_Baseを加えれば、ディレクトリエントリアドレス（ポインタ）PDEが得られる。Intel 80X86 CPUの場合、その現在のページディレクトリベースアドレスPDT\_BaseはコントロールレジスタCR3に格納されている。このカーネルのすべてのプログラムは1つのページ・ディレクトリ・テーブルしか共有しておらず、ページ・ディレクトリ・テーブルは物理メモリ0の先頭に格納されているので、ページ・ディレクトリ・テーブルのベース・アドレスPDT\_Base=0となり、物理メモリ内のページ・ディレクトリ・エントリのアドレス（ポインタ）PDEが得られる。

PDE = PDT\_Base + (PDE\_No << 2);

= 0 + (PDE\_No << 2)。

= (((p->start\_code >> 22) + (Vaddr >> 22)) << 2)となります。

= ((p->start\_code >> 20) & 0xffc) + ((Vaddr >> 20) & 0xffc) となります。

図13-4を参照すると、ディレクトリエントリの内容から対応するページテーブルの物理アドレスPTを取得し、ページテーブル内のエントリのオフセットを加えて、ページテーブルエントリPTEの物理アドレス（ポインタ）を得ることができます。 。

/ PT = (\*PDE) & 0xfffff000です。

PTE = PT + (PTE\_No << 2)

= PT + (((Vaddr >> 12) & 0x3ff) << 2)。

= ((\*PDE) & 0xfff000) + ((Vaddr >> 10) & 0xffc) となります。

ページテーブルエントリの31-12ビットは、物理ページフレームアドレスです。したがって、最終プログラムの論理アドレス「Vaddr」に対応する物理ページアドレスは

PPaddr = (\*PTE) & 0xfffff000;

4. リニアアドレスから物理アドレスへの変換

上記のリニアアドレスの分解によると、リニアアドレスLaddrについては、そのページディレクトリエントリ番号、ページテーブルエントリ番号、ページ内オフセット値が上記の第2のポイントの最初に与えられている。これに対応して、そのページディレクトリエントリ番号に対応するページディレクトリテーブル内のオフセット値PDEは

PDE = (PDE\_No << 2)。

= ((Laddr >> 22) << 2)となります。

= ((Laddr >> 20) & 0xffc)となります。

ディレクトリエントリの内容から、対応するページテーブルの物理アドレスPTを取得し、ページテーブルのエントリのオフセット値を加えて、ページテーブルエントリの物理アドレスPTE（ポインタ）を得ることができる。

PT = (\*PDE) & 0xfffff000となります。

PTE = PT + (PTE\_No << 2)となります。

= PT + ((Laddr >> 12) & 0x3ff) << 2)となります。

= (\*PDE) & 0xfff000 + ((Laddr >> 10) & 0xffc)となります。

= (\*((Laddr >> 20) & 0xffc) & 0xfff000) + ((Laddr >> 10) & 0xffc)。

したがって、リニアアドレス「Laddr」に対応する実際の物理ページアドレスは「PPaddr」となり、対応する物理アドレス「Paddr」は、以下のように、物理ページアドレスにページ内のオフセットを加えたものとなります。

PPaddr = (\*PTE) & 0xfffff000です。

Paddr = PPaddr + Laddr & 0xfff;

= (\*PTE) & 0xfffff000 + Laddr & 0xfff;

= \*(((Laddr >> 10) & 0xffc) + (\*((Laddr >> 20) & 0xffc) & 0xfff000)) + Laddr & 0xfff;

### 13.1.5 Page-fault exception handling

* ページング機構（PG=1）が有効な状態で、CPUがリニアアドレスから物理アドレスへの変換中に以下の状態を検出すると、ページフォルト例外割り込みINT14が発生します。
* The presence bit (P) in the page directory entry or page table entry used in the address translation process is equal to 0, indicating that the page table or the page containing the operand does not exist in physical memory;
* The current execution code does not have sufficient privileges to access the specified page, or the user mode code writes a read-only page, and so on.
* ページ例外処理手順は、中断されたプログラムやプロセスをページが存在しない状態から回復して再起動することができ、プログラムの実行の継続性に影響を与えません。また、特権侵害を受けたプログラムやタスクを再起動することもできますが、特権侵害の原因となった問題が修正されない場合があります。このとき、CPUはページ・フォルト例外ハンドラに以下の2つの側面を提供し、エラーの診断と修正を支援しています。
* An error code on the stack. The format of the error code is a 32-bit long word, but only the lowest 3 bits are useful. Their names are the same as the last three bits in the page table entry (U/S, W/R, P). Their meanings and roles are:
  + Bit 0 (P), the exception is caused by a not-present page or violating access privileges. P=0, indicating that the page does not exist; P=1 indicates that the page-level protection privilege is violated.
  + Bit 1 (W/R), the exception is due to a memory read or write operation. W/R=0, indicating that it is caused by a read operation; W/R=1, indicating that it is caused by a write operation.
  + Bit 2 (U/S), the code level at which the CPU executes when an exception occurs. U/S=0, the CPU was executing at supervisor mode; U/S=1, CPU was executing at user mode.
* The linear address in control register CR2. The CPU will store the linear address that generated the exception in CR2. The page fault exception handler can use this address to locate the relevant page directory and page table entry. If another page exception is allowed to occur during the execution of the page exception handler, the handler should push CR2 onto the stack.

### 後述するpage.sプログラムでは、これらの情報をもとに、ページフォルト例外と書き込み保護例外を区別し、memory.cプログラムでページフォルト処理関数do\_no\_page()を呼び出すか、書き込み保護関数do\_wp\_page()を呼び出すかを決定しています。

### 13.1.6 Copy-on-write mechanism

Copy-on-writeは、データのコピーを延期または回避する方法です。このとき、カーネルはプロセスのアドレス空間全体のデータをコピーするのではなく、親プロセスと子プロセスが同じコピーを共有できるようにします。プロセスAがfock()を使って子プロセスBを作成した場合、子プロセスBは実際には親プロセスAのコピーなので、親プロセスと同じ物理ページを持つことになります。つまり、fork()関数は、メモリの節約とプロセス作成の高速化のために、子プロセスBに親Aの物理ページをリードオンリーで共有させ、さらに、これらの物理ページに対する親Aのアクセス権もリードモードのみに設定します（memory.cプログラムのcopy\_page\_tables()を参照）。このようにして、親Aまたは子Bがこれらの共有物理ページに対して書き込み操作を行うと、ページフォルト例外（INT14）が発生します。このとき、CPUはシステムが提供する例外ハンドラdo\_wp\_page()を実行して、例外の解決を図ることになる。これがコピーオンライトの仕組みです。

do\_wp\_page()関数は、書き込み例外割り込みの原因となった物理ページの共有を解除して（un\_wp\_page()関数を呼び出して）、書き込み処理のための新しい物理ページをコピーすることで、親Aと子Bはそれぞれ同じ内容の物理ページを持つことになります。そして、書き込み操作を行う物理ページを書き込み可能とマークしてから、実際にコピー操作を行います（この物理ページだけがコピーされます）。最後に、例外ハンドラから戻る際に、CPUは例外の原因となった書き込み操作を再実行し、処理を継続できるようにします。

そのため、プロセスが自身の仮想アドレス範囲内で書き込みを行う場合には、「書き込み操作→ページフォルト例外→書き込み保護例外の処理→書き込み操作命令の再実行」という、上記のパッシブなコピーオンライトの操作が行われます。カーネルコードの場合、プロセスがシステムコールを呼び出すなど、プロセスの仮想アドレス範囲内で書き込み操作が行われた場合、システムコールがプロセスのバッファにデータをコピーすると、カーネルは関数verify\_area()を呼び出します。この関数は、まずメモリページ検証関数 write\_verify()を積極的に呼び出し、ページ共有状態があるかどうかを判断します。ページ共有状態があれば、そのページのコピーオンライト操作が行われます。

また、Linux 0.12カーネルでは、カーネルコードアドレス空間（1MB未満のリニアアドレス）にプロセスを生成するためのfork()を実行する際に、コピーオンライト技術が使用されていないことも注目すべき点です。そのため、プロセス0（すなわちアイドルプロセス）がカーネル空間でプロセス1（initプロセス）を生成する際には、同じコードとデータセグメントを使用します。しかし、プロセス1がコピーしたページテーブルエントリも読み取り専用であるため、プロセス1がスタック（書き込み）操作を行う必要がある場合には、ページ例外も発生するため、この場合、メモリマネージャはメインメモリ領域にプロセス用のメモリを割り当てます。.

コピーオンライトでは、メモリページのコピー動作を実際の動作まで遅らせることができることがわかります。

### 書き込まれていないときには、ページのコピー操作がまったく行われないことがあります。例えば、fork()がプロセスを生成し、すぐにexecve()を呼び出して新しいプログラムを実行するような場合です。そのため、この技術は不必要なメモリのページコピーによるオーバーヘッドを回避することができます。

### 13.1.7 Load on demand mechanism

execve()システムコールでファイルシステム上の実行形式イメージファイルをロードする際、カーネルはCPUの4Gリニアアドレス空間で対応するプロセスに64MBの連続した空間を割り当て、その環境やコマンドラインパラメータのために一定量の物理メモリページを適用・確保します。それ以外に、実行ファイルに割り当てられた物理メモリページは実はありません。もちろん、実行イメージファイルのコードやデータをファイルシステムから読み込むこともできません。したがって、エントリー実行ポイントからプログラムが実行を開始すると、すぐにCPUにページフォルト例外（実行ポインタがあるメモリページが存在しない）が発生する。このとき、カーネルのページ・フォールト例外ハンドラは、ファイル・システムから実行ファイルの該当コード・ページを、ページ・フォールト例外の原因となった特定のリニア・アドレスに応じた物理メモリ・ページにロードし、プロセス論理アドレスで指定されたページ位置にマッピングする。例外ハンドラが戻ってくると、CPUは例外の原因となった命令を再実行し、実行プログラムの実行を継続させます。

プログラムが実行中にロードされていない別のページに実行する必要がある場合や、コード命令がロードされていないデータにアクセスする必要がある場合も、CPUはページフォールト例外割り込みを発生させ、その後、カーネルは対応する別のページ内容をメモリにロードして、再びプログラムを実行します。このようにして、実行ファイルの中で実行される（使用される）コードやデータのページだけが、カーネルによって物理メモリにロードされます。このように、実行ファイル内のページが実際に必要になったときだけロードする方法を、ロード・オン・デマンド技術またはデマンド・ページング技術と呼ぶ。

## デマンドローディング技術を使うことの明らかな利点は、実行プログラムが実行を開始する前に、実行ファイルイメージ全体をメモリにロードするための複数のブロックデバイスのI/O操作を待つ必要がなく、execve()システムコールを呼び出した直後に実行プログラムの実行を開始することができることである。そのため、実行プログラムをロードするシステムの実行速度が大幅に向上する。ただし、この手法では、オブジェクトファイルをロードする形式に一定の要件があります。それは、実行されるファイルのオブジェクトフォーマットがZMAGICタイプであること、つまりデマンドページングフォーマットのオブジェクトファイルフォーマットであることです。このオブジェクトファイル形式では、プログラムのコードセグメントとデータセグメントがページ境界から格納され、カーネルがコードやデータの内容を1ページ単位で読み取れるようになっている。

## 13.2 memory.c

### 13.2.1 Function

メモリのページングを管理するmemory.cプログラムは、主記憶領域のメモリページの動的な割り当てと再利用を実現します。カーネルが占有するメモリ以外の物理メモリ領域（1MBアドレス以上）では、カーネルは物理メモリページの状態を示すためにバイト配列mem\_map[]を使用します。各バイトエントリには、物理メモリページの占有状態が記述されている。値は占有されている回数を示し、0は対応する物理メモリがアイドル状態であることを示す。物理メモリのページを適用する際には、対応するバイトの値が1ずつ増加します。バイトの値が100の場合は、完全に占有されており、これ以上の割り当てができないことを意味する。

メモリ管理の初期化では，図13-7に示すように，カーネルコードはまず，1MB以上のメモリ領域に対応するメモリページ数（PAGING\_PAGES）を計算し，mem\_map[]のすべての項目の値を100（占有）に設定した後，メインメモリ領域に対応するmem\_map[]の項目をすべてクリア（ゼロ）にする。これにより，カーネルが使用する1MBアドレスより上のバッファキャッシュ領域と，仮想ディスク領域がある場合はその領域がフルオキュパシー状態に初期化されたことになる．mem\_map[]のメインメモリ領域に対応する項目は、システムの使用中に設定またはリセットされる。例えば、図13-5のように16MBの物理メモリと512KBの仮想ディスクを持つマシンの場合、mem\_map[]配列には(16MB - 1MB) / 4KB = 3840個のエントリがあり、これは3840ページに相当する。主記憶領域のページ数は(16MB-4.5MB)/4KB=2944で，mem\_map[]配列の最後の2944項目に対応し，最初の896項目は1MBメモリ以上のキャッシュバッファや仮想ディスクが占有する物理メモリに対応する。そのため、メモリ管理の初期化処理では、mem\_map[]の最初の896項目は占有状態（値は100）に設定され、もはや使用するために割り当てることはできない。2944項目の値は0にクリアされ、メモリマネージャによる割り当てが可能となる。

Items occupied by

buffer

cache

a

bove 1MB and virtual disk areas.

(896

items

)

Items occupied by the main memory

area (2944 items)That is, the

value of 'end\_mem' on line 408 of

the program

.

T

otal of PAGING\_PAGES

(3840)

PAGING\_PAGES

-

1

0

100

100

100

0

0

0

0

0

'i' of line 406

in

memory.c

mem\_map[

PAGING

\_PAGES] Array

図13-7 16MBのメモリと512KBのvdiskを使用したMem\_map[]アレイの初期化

プロセスの仮想アドレス（または論理アドレス）の管理については、カーネルはプロセッサのセグメント管理機構を利用して実装し、物理メモリページとリニアアドレスのマッピング関係は、ページディレクトリとページテーブルエントリの内容を変更することで処理しています。以下に、このプログラムで提供されるいくつかの主な機能について詳しく説明します。

get\_free\_page()関数とfree\_page()関数は、各プロセスのリニアアドレスに関係なく、主記憶領域の物理メモリの占有率と空き容量を管理するために特別に設計されています。get\_free\_page()関数は、主記憶領域内の空きメモリのページを要求し、その物理メモリページの開始アドレスを返すために使用されます。まず、メモリページのバイトマップ配列mem\_map[]をスキャンし、値が0（空きページに対応）のバイトエントリを探します。ない場合は0を返し、物理メモリが使い尽くされたことを示します。0の値を持つバイトが見つかった場合は、それを1に設定し、対応する空きページの開始アドレスを計算します。その後、メモリページがクリアされ、最後にフリーページの物理メモリの開始アドレスが返されます。

free\_page()は、指定されたアドレスの物理メモリのページを解放するために使用されます。指定された物理メモリアドレスが実際のメモリ最上位アドレス以上の場合は、エラーメッセージが表示されます。指定された物理メモリアドレスが実際のメモリ最上位アドレスと等しい場合はエラーメッセージが表示され、ページ番号は指定されたメモリアドレスである(addr - 1M) / 4Kで変換され、ページ番号に対応するmem\_map[]バイト項目が0であるかどうかが判断され、0でなければ1だけデクリメントされて返されます。

free\_page\_tables()およびcopy\_page\_tables()関数は、指定されたリニアアドレスと長さ（ページテーブルの数）に対応する物理メモリのページブロックを、1つのページテーブルに対応する物理メモリブロック（4M）を単位として解放またはコピーする。これらの関数は、リニアアドレスのページディレクトリおよびページテーブルの対応するディレクトリエントリの内容を変更するだけでなく、各ページテーブルのすべてのページテーブルアイテムに対応する物理メモリページを解放または占有する。

free\_page\_tables()関数は、指定されたリニアアドレスと長さ（ページテーブルの数）に対応する物理メモリページを解放するために使用されます。まず、指定されたリニアアドレスが4M境界上にあること、指定されたアドレス値がカーネルキャッシュやバッファキャッシュが占める空間よりも上にあることを判断します。次に、ページディレクトリテーブルで占有されるディレクトリエントリ数（すなわち、ページテーブル数）を計算し、対応する開始ディレクトリエントリ番号を計算する。そして、対応する開始ディレクトリエントリから順に、占有されているディレクトリエントリを解放し、対応するディレクトリエントリが指すページテーブル内のすべてのページテーブルエントリと対応する物理メモリページを解放する。最後にページ変換キャッシュをリフレッシュする。

copy\_page\_tables()関数は、指定されたリニアアドレスと長さ（ページテーブル数）のメモリに対応するページディレクトリエントリとページテーブルをコピーし、コピーされたページディレクトリとページテーブルに対応する元の物理メモリ領域を共有するために使用されます。この機能は、まず、指定されたコピー元とコピー先のリニアアドレスがともに4Mbのメモリ境界にあるかどうかを確認し、指定されたリニアアドレスから対応する開始ページディレクトリエントリ（from\_dir, to\_dir）を算出し、コピーするメモリ領域が占有するページテーブル（ページディレクトリエントリ）の数を算出します。そして、元のディレクトリ・エントリとページ・テーブル・エントリを、それぞれ新しい空きディレクトリ・エントリとページ・テーブル・エントリにコピーを開始する。ページ・ディレクトリ・テーブルは1つしかなく、新しいプロセスのページ・テーブルは、格納するための空きメモリ・ページを申請する必要がある。その後、元のページディレクトリと新ページディレクトリ、ページテーブルのエントリはすべて読み取り専用ページに設定されます。書き込み操作があった場合は、ページ例外ハンドラを使用してコピーオンライト操作を行います。最後に、共有物理メモリのページに対応するバイト配列項目が1つずつ増加します。

put\_page()関数は、指定された物理メモリページを指定されたリニアアドレスにマッピングするための関数です。put\_page()関数は、指定された物理メモリページを指定されたリニアアドレスにマッピングするために使用されます。まず、指定されたメモリページアドレスが1Mからシステムの最上位メモリアドレスまでの範囲内にあるかどうかを判断し、ページディレクトリテーブルの指定されたリニアアドレスに対応するディレクトリエントリを計算します。ディレクトリエントリが有効であれば、対応するページテーブルのアドレスが取得され、そうでなければ、フリーページがページテーブルに適用され、ページテーブルエントリの属性が設定される。最後に、やはり指定された物理メモリのページアドレスが返されます。

do\_wp\_page()関数は、ページ例外ハンドラ(mm/page.sに実装)で呼び出されるページ書き込み保護手続きです。まず、そのアドレスがプロセスのコード領域にあるかどうかを判断し、コピーオンライト操作を行います。

do\_no\_page()は、ページ例外発生時に呼び出されるページフォルト関数です。do\_no\_page()は、ページ例外発生時に呼び出されるページフォルト関数で、まず、プロセス空間のプロセスベースアドレスに対する指定されたリニアアドレスのオフセット長値を決定します。それがコード＋データの長さよりも大きい場合や、プロセスの生成が始まったばかりの場合は、物理メモリのページを申請し、プロセスのリニアアドレスにマッピングします。そうでなければ、ページ共有操作を試みます。そうでない場合は、メモリのページを申請し、デバイスからコンテンツのページを読み取る。指定された（リニアアドレス＋1ページ）の長さが、ページのコンテンツを追加したときのプロセスコード＋データの長さを超える場合は、超過分をクリアする。そして、そのページは指定されたリニアアドレスにマッピングされます。

get\_empty\_page()関数は、空いている物理メモリのページを取得し、指定されたリニアアドレスにマッピングするための関数です。この関数の実装には、主にget\_free\_page()関数とput\_page()関数を使用しています。

### 13.2.2 Code annoation

プログラム 13-1 linux/mm/memory.c

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

6

1. /\*
2. \* demand-loading started 01.12.91 - seems it is high on the list of
3. \* things wanted, and it should be easy to implement. - Linus
4. \*/

11

1. /\*
2. \* Ok, demand-loading was easy, shared pages a little bit tricker. Shared 14 \* pages started 02.12.91, seems to work. - Linus.
3. \*
4. \* Tested sharing by executing about 30 /bin/sh: under the old kernel it 17 \* would have taken more than the 6M I have free, but it worked well as 18 \* far as I could see.
5. \*
6. \* Also corrected some "invalidate()"s - I wasn't doing enough of them.
7. \*/

22

1. /\*
2. \* Real VM (paging to/from disk) started 18.12.91. Much more work and
3. \* thought has to go into this. Oh, well..
4. \* 19.12.91 - works, somewhat. Sometimes I get faults, don't know why.
5. \* Found it. Everything seems to work now.
6. \* 20.12.91 - Ok, making the swap-device changeable like the root.
7. \*/ 30

// <signal.h> シグナルのヘッダファイルです。シグナルシンボル定数、シグナル構造体、シグナル操作関数のプロトタイプを定義しています。

// <asm/system.h> システムのヘッダーファイルです。ディスクリプタ／割込みゲートなどを定義／変更する組み込みアセンブリマクロが定義されている。

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

// <linux/head.h> 頭部のヘッダーファイルです。セグメントディスクリプターの簡単な構造が定義されており、 // いくつかのセレクタ定数とともに定義されています。

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

31 #include <signal.h> (英語)

32

33 #include <asm/system.h>.

34

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

38

// CODE\_SPACE(addr) ((((addr)+0xfff)&~0xfff) < current->start\_code + current->end\_code)

1. // このマクロは、与えられたリニアアドレスが、 // 現在のプロセスのコードセクション内にあるかどうかを判断するために使用されます。"(((addr)+4095)&~4095) "は、線形アドレス「addr」が位置するメモリ // ページの終了アドレスを取得するために使用されます。265行目をご覧ください。
2. #define CODE\_SPACE(addr) ((((addr)+4095)&~4095) < \
3. current->start\_code + current->end\_code)

41

// 実際の物理メモリの最高物理アドレスを保持するグローバル変数。

42 unsigned long HIGH\_MEMORY = 0;

43

1. // 1ページ分のメモリをfromからtoにコピーする（4Kバイト）。
2. #define copy\_page(from,to) \
3. \_\_asm\_\_("cld ; rep ; movsl"::"S" (from),"D" (to),"c" (1024):"cx","di","si") 46

// 物理メモリにマッピングされたバイト配列（1バイトはメモリの1ページを表す）。対応する

各ページの // バイトは、そのページが現在参照されている回数を示すために使用されます。

// を占有しています。16MBの物理メモリを持つマシンでは、最大15Mbのメモリをマッピングすることができます。初期化関数mem\_init()では、メインメモリ領域のページとして使用できない位置をあらかじめUSED(100)に設定しておきます。

47 unsigned char mem\_map [ PAGING\_PAGES ] = {0,};

48

1. /\*
2. \* Free a page of memory at physical address 'addr'. Used by
3. \* 'free\_page\_tables()'
4. \*/

//// 物理アドレス「addr」を起点に1ページ分のメモリを解放します。

1. // 物理アドレスの1MB以下のメモリ空間は，カーネルプログラムとバッファに使用され， // アロケーションページのメモリ空間として使用することはできない。そのため，パラメータ'addr'は， // 1MB以上である必要がある。
2. void free\_page(unsigned long addr)
3. {

// この関数は、まず、物理アドレス'addr'の合理性を判断します。

// パラメーターです。物理アドレス'addr'がメモリの下限(1MB)よりも小さい場合は

1. // カーネルのプログラムやキャッシュの範囲内であることを意味し、処理されません。「addr」がシステムの物理メモリの最上限以上である場合は、エラーメッセージが表示され、カーネルの動作が停止します。
2. if (addr < LOW\_MEM) return; // LOW\_MEM = 1MB, defined in include/linux/mm.h, 30.
3. if (addr >= HIGH\_MEMORY)
4. panic("trying to free nonexistent page");

// パラメータ'addr'の検証が渡された場合、この物理アドレスに基づいて、 // メモリの下端から数えたメモリページ番号が計算されます。

// "ページ番号 = (addr - LOW\_MEM) / 4096"

// ページ番号は0から始まり、ページ番号は'addr'に格納されていることがわかります。

// ページ番号に対応するマッピングバイトが0になっていない場合は、次のように返されます。

// この時点で、マッピングされたバイト値は0になるはずで、これは

1. //ページが解放されたことを意味します。しかし、対応するページバイトがもともと0であれば、物理ページがもともとアイドル状態であることを意味し、カーネルコードに不具合があることを示します。そして、エラーメッセージが表示され、カーネルが停止します。
2. addr -= LOW\_MEM;
3. addr >>= 12; // divided by 4096.
4. if (mem\_map[addr]--) return;
5. mem\_map[addr]=0;
6. panic("trying to free free page");
7. }

64

1. /\*
2. \* This function frees a continuos block of page tables, as needed
3. \* by 'exit()'. As does copy\_page\_tables(), this handles only 4Mb blocks.
4. \*/

//// 与えられたリニアアドレスとリミットレングス（ページテーブルの数）に従って、//// メモリブロックを解放し、テーブルエントリを空にします。

// ページディレクトリテーブルは、物理アドレス0の先頭にあります。そこには、1024個の

// のエントリーがあり、それぞれが4バイトで、合わせて4Kバイトになります。各ディレクトリエントリは

// は、1つのページテーブルに対応しています。カーネルのページテーブルは、物理アドレス0x1000から始まる

//（ディレクトリのテーブル空間に続く）で、4つのページテーブルを持っています。各ページテーブルには1024個のエントリがあります。

// それぞれが4バイトであるため、4KB（1ページ）のメモリを占めることになります。ただし

1. // カーネルコードのプロセス0と1の場合、他のプロセスのページテーブルが占有するページは、 // プロセスの生成時にカーネルがメインメモリ領域から要求します。 // ページテーブルの各エントリは物理メモリの1ページに対応しているので，1つのページテーブルは最大4MBの物理メモリをマッピングすることができます。 // パラメータ: from - 線形ベースアドレス, size - 解放されるバイト数.
2. int free\_page\_tables(unsigned long from,unsigned long size)
3. {
4. unsigned long \*pg\_table;
5. unsigned long \* dir, nr;

73

// この関数はこのような状況のみを扱うことができるので、 // 最初にパラメータ 'from' で与えられたリニアベースアドレスが 4MB の境界にあるかどうかをチェックします。from = 0 の場合，エラーが発生します．これは， // この関数がカーネルとバッファによって占有されている領域を解放しようとしていることを示している． // 次に，パラメータ'size'で指定された長さに対応する // ページディレクトリエントリの数を計算します（キャリー付きの 4MB の倍数）。

// 1つのページテーブルで4MBの物理メモリを管理できるので、メモリ長の値は次のようになります。

// コピーされたデータは、22ビットを右にシフトして4MBで分割されます。0x3fffff（つまり4MB -1）を加えて、キャリー付き整数倍の結果を得る。例えば、元のサイズが4.01Mbであれば、結果としてサイズ=2が得られます。

// 次に、与えられたリニアベースアドレスに対応する開始ディレクトリエントリが計算されます。

// 対応するディレクトリエントリ番号は、「from >> 20」と同じです。各エントリが占めるのは

1. // ページディレクトリテーブルは物理アドレス0から格納されているので、実際のディレクトリエントリポインタは「ディレクトリエントリ番号 << 2」、つまり「(from >> 20)」となります。「74 if (from & 0xffc) // ディレクトリエントリポインタが有効範囲内にあることを確認する。 74 if (from & 0x3fffff)
2. panic("free\_page\_tables called with wrong alignment");
3. if (!from)
4. panic("Trying to free up swapper memory space");
5. size = (size + 0x3fffff) >> 22;
6. dir = (unsigned long \*) ((from>>20) & 0xffc); /\* \_pg\_dir = 0 \*/

// この時点で、'size'は解放する必要のあるページテーブルの数、つまりページディレクトリエントリの数、'dir'は開始ディレクトリエントリポインタです。では、スタート

各ページディレクトリエントリに対してループ操作を行い、各ページ // テーブルのエントリを順番に解放していきます。現在のディレクトリ・エントリが無効（Pビット＝0）であることを示すと

// ディレクトリエントリが使用されていない（対応するページテーブルが存在しない）場合は、次のディレクトリ

// エントリーの処理を続けます。それ以外の場合は、ページテーブルのアドレス 'pg\_table' を

// ディレクトリエントリ、およびページテーブルの1024エントリが処理され、有効なページテーブルエントリ(Pビット=1)に対応する物理メモリページが解放されるか、または無効なページが解放されます。

// テーブルエントリ（Pビット＝0）がスワップデバイスから解放される、つまり、対応する

// スワップデバイスのメモリページを削除します（ページがスワップアウトされている可能性があるため）。次に

// ページテーブルのエントリを削除し、次のページエントリの処理を続けます。ページのすべてのエントリが

1. // テーブルの処理が完了すると，ページテーブル自体が占有していたメモリページが解放され， // 次のページディレクトリエントリが処理されます。最後に，ページ変換キャッシュを更新し， // 0 を返します。
2. for ( ; size-->0 ; dir++) {
3. if (!(1 & \*dir))
4. continue;
5. pg\_table = (unsigned long \*) (0xfffff000 & \*dir); // get page table addr.
6. for (nr=0 ; nr<1024 ; nr++) {
7. if (\*pg\_table) {
8. if (1 & \*pg\_table) // free the page if valid.
9. free\_page(0xfffff000 & \*pg\_table);
10. else // free the page in swap device.
11. swap\_free(\*pg\_table >> 1);
12. \*pg\_table = 0; // reset the page content.
13. }
14. pg\_table++; // points to next page.
15. }
16. free\_page(0xfffff000 & \*dir); // free the page of the page table.
17. \*dir = 0; // reset the directory entry.
18. }
19. invalidate(); // refresh CPU page transform cache.
20. return 0;
21. }

100

1. /\*
2. \* Well, here is one of the most complicated functions in mm. It 103 \* copies a range of linerar addresses by copying only the pages.
3. 104 \* バグがないことを祈りましょう。これはデバッグしたくないので :-) 105 \*
4. \* Note! We don't copy just any chunks of memory - addresses have to
5. \* be divisible by 4Mb (one page-directory entry), as this makes the 108 \* function easier. It's used only by fork anyway.
6. \*
7. \* NOTE 2!! When from==0 we are copying kernel space for the first
8. \* fork(). Then we DONT want to copy a full page-directory entry, as
9. \* that would lead to some serious memory waste - we just copy the
10. \* first 160 pages - 640kB. Even that is more than we need, but it
11. \* doesn't take any more memory - we don't copy-on-write in the low
12. \* 1 Mb-range, so the pages can be shared with the kernel. Thus the 116 \* special case for nr=xxxx.
13. \*/

//// 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。

// この関数は、指定されたリニアアドレスとメモリサイズに対応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をコピーして、対応する物理メモリ領域が

は、2組のページテーブルマッピングで共有されます。コピー時には、新しいページテーブルを格納するための新しいページを // 申請する必要がありますが、元の物理メモリ領域は共有されます。

1. // その後、カーネルが書き込み操作を行うまで、2つのプロセス（親プロセスとその子プロセス）は、 // メモリ領域を共有し、カーネルは書き込み操作のプロセスのために新しいメモリページを // 割り当てます（コピーオンライト）。パラメータの'from'と'to'はリニアな // アドレス、'size'はコピー（共有）が必要なメモリの長さをバイト単位で指定します。
2. int copy\_page\_tables(unsigned long from,unsigned long to,long size)
3. {
4. unsigned long \* from\_page\_table;
5. unsigned long \* to\_page\_table;
6. unsigned long this\_page;
7. unsigned long \* from\_dir, \* to\_dir;
8. unsigned long new\_page;
9. unsigned long nr;

126

// コードはまず、ソースアドレス「from」とデスティネーションアドレスの有効性を検出します。

// パラメータで与えられた「to」。送信元と送信先の両方のアドレスが

// 4Mbのメモリ境界アドレス。この要件は、ページテーブルの1024エントリが

//は4Mbのメモリを管理できます。送信元アドレスと送信先アドレスは、この条件を満たすだけで

// ページテーブルのエントリが、ページの最初のエントリからコピーされることを保証する要件

//テーブルを取得し、新しいページテーブルの元のエントリがすべて有効になります。そして、スタートを取得します。

1. コピー元アドレスとコピー先アドレスの // ディレクトリエントリポインタ（from\_dir と to\_dir）を計算します。次に，パラメータで指定された「サイズ」に応じて， // コピーされるメモリブロックが占めるページテーブルの数（すなわち， ディレクトリエントリの数）を計算します。
2. if ((from&0x3fffff) || (to&0x3fffff))
3. panic("copy\_page\_tables called with wrong alignment");
4. from\_dir = (unsigned long \*) ((from>>20) & 0xffc); /\* \_pg\_dir = 0 \*/
5. to\_dir = (unsigned long \*) ((to>>20) & 0xffc);
6. size = ((unsigned) (size+0x3fffff)) >> 22;

// 送信元の開始ディレクトリエントリポインタfrom\_dirを取得した後、送信先の開始

// ディレクトリエントリポインタ to\_dir と、コピーするページテーブルの数を指定すると

// 各ページのディレクトリエントリに1ページ分のメモリを適用して対応する保存を開始

//ページテーブル、およびページテーブルエントリのコピー操作を開始します。で指定されたページテーブルが存在しない場合は

// 宛先ディレクトリエントリが既に存在する場合(P=1)、カーネルはクラッシュします。コピー元のディレクトリエントリが無効な場合，つまり，指定されたページテーブルが存在しない場合（P=0），次のページ

1. // ディレクトリエントリはループし続けます。
2. for( ; size-->0 ; from\_dir++,to\_dir++) {
3. if (1 & \*to\_dir)
4. panic("copy\_page\_tables: already exist");
5. if (!(1 & \*from\_dir))
6. continue;

// 現在のソースディレクトリエントリとデスティネーションエントリが正常であることを確認した後。

// ソースのディレクトリエントリにあるページテーブルアドレス from\_page\_table を取得します。にするために

// 宛先ディレクトリエントリに対応するページテーブルを保存するためには， // メインメモリ領域の1ページ分の空きメモリページを申請する必要がある。get\_free\_page()関数が0を返した場合は、 // 空きメモリページが得られないことを意味し、メモリが不足している可能性がある。

1. // その後、-1の値を返して終了します。
2. from\_page\_table = (unsigned long \*) (0xfffff000 & \*from\_dir);
3. if (!(to\_page\_table = (unsigned long \*) get\_free\_page())) 139 return -1; /\* Out of memory, see freeing \*/ // Next, we set the destination directory entry information: the last 3 bits are set, that is, // the current destination directory entry "OR" 7, indicating that the memory page mapped to

// 対応するページテーブルはユーザーレベルで、読み取り、書き込み、存在することができます（Usr, R/W,

// 現在）。(U/Sビットが0であれば、R/Wは影響しません。もし、U/Sが1で、R/Wが0の場合は

ユーザーレベルで実行されているコードは、 // そのページを読むことしかできません。U/SとR /Wがすべて設定されていれば、 // 読み込みと書き込みの許可があります）。次に、コピーされるページアイテムの数を設定して

// 現在処理されているページディレクトリエントリに対応するページテーブル。それがカーネルの

1. // そうでなければ、4MBの物理メモリをマッピングできるページテーブル(nr=1024)の1024個のエントリをすべてコピーする必要があります。
2. \*to\_dir = ((unsigned long) to\_page\_table) | 7;
3. nr = (from==0)?0xA0:1024; // 0xA0 = 160
4. // この時点で、現在のページテーブルについて、指定された「nr」の // メモリページテーブルエントリの循環的なコピーを開始します。まず、ソースページテーブルエントリの内容を取り出す。 // 現在のソースページが使用されていない（内容が0）場合、テーブルアイテムはコピーされず、 // 次のエントリが処理される。
5. for ( ; nr-- > 0 ; from\_page\_table++,to\_page\_table++) {
6. this\_page = \*from\_page\_table;
7. if (!this\_page)
8. continue;

// エントリーがコンテンツを持っているが、その存在ビットP=0である場合、エントリーに対応するページは

// がスワップデバイスにあることを確認します。そこで、1ページ分のメモリを申請し、スワップデバイスからページを読み込みます

//（スワップデバイスにあれば）、ページテーブルのエントリをコピーして、宛先の

// ページテーブルエントリを変更し、新しいメモリを指すようにソースページテーブルエントリの内容を変更します。

//ページを作成し、テーブルエントリフラグを "page dirty "に7を加えた値に設定します。その後、次の処理を続けます。

1. // ページエントリになります。そうでなければ，ページテーブルエントリのR/Wフラグ（ビット1が0に設定される）がリセットされ， // そのページテーブルエントリに対応するメモリページが読み取り専用に設定され， // そのページテーブルエントリがコピー先のページテーブルにコピーされる。
2. if (!(1 & this\_page)) {
3. if (!(new\_page = get\_free\_page()))
4. return -1;
5. read\_swap\_page(this\_page>>1, (char \*) new\_page);
6. \*to\_page\_table = this\_page;
7. \*from\_page\_table = new\_page | (PAGE\_DIRTY | 7);
8. continue;
9. }
10. this\_page &= ~2;
11. \*to\_page\_table = this\_page;

// ページテーブルエントリーが示す物理ページのアドレスが1MB以上の場合は

// メモリページマップ配列mem\_map[]を設定します。その後、ページ番号を計算し、それを

// ページマッピングの対応する項目の参照数を増やすためのインデックス

// の配列です。1MB以下のページの場合は、カーネルページなので、mem\_map[]を設定する必要はありません。なぜなら

// mem\_map[]は、メインメモリ領域のページ使用量を管理するためにのみ使用されます。そのため、カーネルが

// タスク0に移動し、fork()を呼び出してタスク1を作成（init()の実行に使用）、コピーされたページがあるので

//がまだカーネルコードの領域にある場合、以下の判定の記述は

// を実行しました。タスク0のページは、まだいつでも読み書き可能です。判定文

// は、fork()を呼び出した親プロセスのコードが、 // メインメモリ領域にある（ページ位置が1MBより大きい）場合にのみ実行されます。これは、プロセスがexecve()を呼び出し、 // 新しいプログラムコードをロードしたときにのみ起こります。

// 157行目のステートメントの意味は、ソースページが指すメモリページが

1. 2つのプロセスがメモリ領域を共有しているため， // テーブルエントリも読み取り専用となります。どちらかのプロセスが書き込み操作を行う必要がある場合、書き込み操作のためのページは、ページ例外書き込み保護プロセスによって新しい空きページを割り当てることができます // つまり、書き込み操作時のコピーです。
2. if (this\_page > LOW\_MEM) {
3. \*from\_page\_table = this\_page; // set source read-only too.
4. this\_page -= LOW\_MEM; 159 this\_page >>= 12;
5. mem\_map[this\_page]++;
6. }
7. }
8. }
9. invalidate(); // refresh CPU page transform cache.
10. return 0;
11. }

167

1. /\*
2. \* This function puts a page in memory at the wanted address.
3. \* It returns the physical address of the page gotten, 0 if
4. \* out of memory (either when trying to access page-table or
5. \* page.) 173 \*/

//// 物理的なメモリページを、リニアアドレス空間の指定された位置にマッピングする。

// あるいは、リニアアドレス空間の指定されたアドレスのページがメインメモリにマッピングされる

// エリアページです。この関数を実装する主な仕事は、指定されたページの情報を設定することです。

// 関連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中のページ。この関数が呼び出されるのは

ページが存在しない例外を処理する // do\_no\_page() です。ページが存在しないことで発生する例外については

// 現在では、ページ不在のためにページテーブルが変更された場合には

// CPUのページ変換バッファ（TLB）をリフレッシュします。

// ページテーブルエントリーフラグPが0から1に変更された場合、無効なページエントリーは

// バッファリングされていないので、無効なページテーブルのエントリが変更されても、リフレッシュする必要はありません。

// ここではInvalidate()関数を呼び出さずに示しています。パラメーターの'page'は，割り当てられたメインメモリ領域の // ページ（ページフレーム）へのポインタで，'address'はリニアアドレスである．

174 static unsigned long put\_page(unsigned long page,unsigned long address) 175 {。

176 unsigned long tmp, \*page\_table;

177

178 /\* 注意 !!! これは \_pg\_dir=0 であることを利用しています。

179

// ここではまず、パラメータで指定された物理メモリページ「page」の有効性を判断します。

// ページ位置がLOW\_MEMよりも低い場合や、システムが実際に

// メモリハイエンドHIGH\_MEMORYを含む。LOW\_MEM=1MB」が定義されています。

// include/linux/mm.hファイルでは、メインメモリの最低開始位置である

// の領域になります。システムの物理メモリが6MB以下の場合、メインメモリ領域の

// は直接LOW\_MEMから始まる。そして、そのページが申請されたものであるかどうか、つまり、メモリページマップmem\_map[]の対応するバイトが設定されているかどうかをチェックします。

1. // そうでない場合は、警告が必要です。
2. if (page < LOW\_MEM || page >= HIGH\_MEMORY)
3. printk("Trying to put page %p at %p\n",page,address);
4. if (mem\_map[(page-LOW\_MEM)>>12] != 1)
5. printk("mem\_map disagrees with %p at %p\n",page,address);

次に，パラメータで指定されたリニアアドレスに応じて， // ページディレクトリテーブルの対応するエントリポインタを計算し， // そこからページテーブルアドレスを得る。// ディレクトリエントリが有効な場合（P=1）、つまり指定されたページテーブルがメモリ内にある場合、 // 指定されたページテーブルアドレスがそこから取得され、変数page\_tableに格納される。そうでない場合は

1. // 空きページをページテーブルに適用し、対応するディレクトリエントリに対応するフラグ（7 - User, U/S, R/W）を設定し、ページテーブルアドレスを変数page\_tableに入れています。
2. page\_table = (unsigned long \*) ((address>>20) & 0xffc);
3. if ((\*page\_table)&1)
4. page\_table = (unsigned long \*) (0xfffff000 & \*page\_table);
5. else {
6. if (!(tmp=get\_free\_page()))
7. return 0;
8. \*page\_table = tmp | 7;
9. page\_table = (unsigned long \*) tmp;
10. }

// 最後に、関連エントリのコンテンツは、見つかったページのテーブルpage\_tableに設定される、つまり

物理ページの//アドレスがエントリに記入され、3つのフラグ（U/S、W/R、P）が設定される

1. // を同時に行うことができます。ページテーブルのエントリーのインデックス値は、リニアアドレスのビット21～ビット12で構成される10ビットの値と // 等しくなります。1つのページテーブルには、合計1024個のエントリー（0 -- 0x3ff）があります。
2. page\_table[(address>>12) & 0x3ff] = page | 7;
3. /\* no need for invalidate \*/
4. return page; // Returns the physical page address. 196 }

197

1. /\*
2. \* The previous function doesn't work very well if you also want to mark
3. \* the page dirty: exec.c wants this, as it has earlier changed the page, 201 \* and we want the dirty-status to be correct (for VM). Thus the same 202 \* routine, but this time we mark it dirty too.
4. \*/

//// 内容が変更された物理的なメモリページを特定の場所にマッピングします。

1. // 線形アドレス空間。この関数は、前の関数である // put\_page() とほとんど同じです。223行目のページテーブルエントリの内容を設定することに加えて、この関数は、 // ページ修正フラグ（ビット6、PAGE\_DIRTY）を設定します。
2. unsigned long put\_dirty\_page(unsigned long page, unsigned long address) 205 {

206 unsigned long tmp, \*page\_table;

207

1. 208 /\* 注意 !!! これは \_pg\_dir=0 であることを利用しています \*/ 209
2. if (page < LOW\_MEM || page >= HIGH\_MEMORY)
3. printk("Trying to put page %p at %p\n",page,address);
4. if (mem\_map[(page-LOW\_MEM)>>12] != 1)
5. printk("mem\_map disagrees with %p at %p\n",page,address);
6. page\_table = (unsigned long \*) ((address>>20) & 0xffc);
7. if ((\*page\_table)&1)
8. page\_table = (unsigned long \*) (0xfffff000 & \*page\_table);
9. else {
10. if (!(tmp=get\_free\_page()))
11. return 0;
12. \*page\_table = tmp|7;
13. page\_table = (unsigned long \*) tmp; 222 }
14. page\_table[(address>>12) & 0x3ff] = page | (PAGE\_DIRTY | 7);
15. /\* no need for invalidate \*/
16. return page;
17. }

227

//// 書き込みページ保護をキャンセルする。ページ例外（コピーオンライト）時の書き込み保護例外 // の処理に使用します。

// カーネルがプロセスを生成する際、新しいプロセスと親プロセスが共有するように設定される

// のコードとデータのメモリページがあり、これらのページはすべてリードオンリーに設定されています。新しいプロセスが

// または元のプロセスがメモリページにデータを書き込む必要がある場合、CPUはこれを検出します。

// となり、ページ書き込み保護例外が発生します。そこで、この関数では、コードはまず

// 書き込まれるページが共有されているかどうかを判断します。そうでない場合は、ページを書き込み可能な状態にしてから // 終了します。ページが共有されている状態の場合は、新しいページを再適用して、書き込みプロセスが個別に使用できるように、書き込まれたページの内容を // コピーする必要があります。そのため、共有は解除されました。

1. // 入力パラメータは、ページテーブルエントリポインタです。
2. void un\_wp\_page(unsigned long \* table\_entry)
3. {
4. unsigned long old\_page,new\_page;

231

// まず、パラメータのページテーブルエントリの物理ページの位置（アドレス）を取得します。

// をクリックし、そのページが共有ページかどうかを判断します。元のページのアドレスがより大きい場合

// メモリLOW\_MEM（メインメモリエリア内）の下限よりも、ページ内のその値が

// マッピングバイト配列が1（ページが1回しか参照されないことを示す。

// 共有されている）の場合は、ページテーブルエントリにR/Wフラグ（書き込み可能）が設定され、ページ変換

1. // キャッシュが更新されてから返されます。つまり、メモリページが現時点で1つのプロセスによってのみ使用されており、 // カーネル内のプロセスではない場合、属性を直接書き込み可能に変更することができ、 // 新しいページを再適用する必要はありません。
2. old\_page = 0xfffff000 & \*table\_entry; // get physical page address in the entry.
3. if (old\_page >= LOW\_MEM && mem\_map[MAP\_NR(old\_page)]==1) {
4. \*table\_entry |= 2; // set R/W flag.
5. invalidate();
6. return; 237 }

// そうでない場合は、プロセスのためにメインメモリ領域の空きページを申請する必要があります。

// 書込み操作を行った後に、ページ共有をキャンセルした場合。元のページ位置が

// がメモリの下限よりも大きい場合（mem\_map[] > 1、ページが共有されていることを意味します）には

// 元のページのバイト配列の値を1だけデクリメントし、内容を更新する

1. 指定されたページテーブルエントリの//を新しいページアドレスに設定し、 // 読み書きフラグ（U/S, R/W, P）を設定する。ページ変換キャッシュ(TLB)をリフレッシュした後，元のページ内容が最終的に新しいページにコピーされる。
2. if (!(new\_page=get\_free\_page()))
3. oom(); // Out of Memory.
4. if (old\_page >= LOW\_MEM)
5. mem\_map[MAP\_NR(old\_page)]--;
6. copy\_page(old\_page,new\_page);
7. \*table\_entry = new\_page | 7;
8. invalidate();
9. }

246

1. /\*
2. \* This routine handles present pages, when users try to write
3. \* to a shared page. It is done by copying the page to a new address
4. \* and decrementing the shared-page counter for the old page.
5. \*
6. \* If it's in code space we exit with a segment error.
7. \*/

//// 共有ページの書き込み保護処理を行います。

// これは、ページの例外処理中に呼び出されるC言語の関数で、この関数は

// ページ.sのプログラムです。関数のパラメータ error\_code と address は自動的に

プロセスが書き込み保護されたページを書き込む際に， // CPU によって生成されます．パラメータ error\_code はエラーの種類を示し， address は例外が発生したページのリニアアドレスである。

1. // 共有ページを書き込む際には、ページをコピーする必要があります（コピーオンライト）。
2. void do\_wp\_page(unsigned long error\_code,unsigned long address)
3. {

// この関数は、まず、CPU制御から与えられたリニアアドレスがどの範囲にあるかを判断します。

// ページ例外の原因となったレジスタCR2。アドレスがTASK\_SIZE (0x4000000,

// または64MB）の線形アドレス範囲内にあることを意味します。

// カーネル、タスク0、タスク1のいずれかを選択し、警告メッセージを発行しました; If (address - current process.

1. // コード開始アドレス）が1プロセスのサイズ（64MB）よりも大きい場合、リニアアドレスが例外を引き起こしたプロセスの空間内にないことを示し、エラーメッセージを発行して終了します。
2. if (address < TASK\_SIZE)
3. printk("\n\rBAD! KERNEL MEMORY WP-ERR!\n\r");
4. if (address - current->start\_code > TASK\_SIZE) {
5. printk("Bad things happen: page error in do\_wp\_page\n\r");
6. do\_exit(SIGSEGV);
7. }
8. #if 0
9. /\* we cannot do this yet: the estdio library writes to code space \*/
10. /\* stupid, stupid. I really want the libc.a from GNU \*/
11. // リニアアドレスがプロセスのコード空間にある場合、実行プログラムを // 終了させる必要があります。なぜなら、コードは読み取り専用だからです。
12. if (CODE\_SPACE(address))
13. do\_exit(SIGSEGV);
14. #endif

// 次に、上記の関数 un\_wp\_page() を呼び出して、ページ保護の解除を処理します。しかし、その前に、 // そのためのパラメータを準備する必要があります。そのパラメータは、与えられたリニアアドレス「アドレス」に対するページ // テーブルのエントリーポインタで、次のように計算されます。

// (1) 「((アドレス>>10) & 0xffc)」。のページテーブルエントリのオフセットアドレスを計算します。

// ページテーブルのリニアアドレスを指定します。なぜなら、リニアアドレス構造による。

// "(address>>12) "はページテーブルエントリのインデックスですが、各エントリは4バイトを占めるので

// 4倍した後は、「(アドレス>>12)<<2 = (アドレス>>10)&0xffc」となり、オフセットアドレスを得ることができます。

テーブル内のページテーブルエントリの//。AND "演算 &0xffc は、アドレスを制限するために使用されます。

// 1ページ内の範囲になります。また、10ビットだけがシフトされるので、最後の2ビットはリニアアドレスの下位12ビットのうち、 // 最高位の2ビットとなり、マスクアウトされるべきです。したがって、リニアアドレスのページテーブルエントリのオフセットを求めるには、より // わかりやすい方法があります。

// "(((アドレス>>12) & 0x3ff)<<2) "となります。

// (2) 「(0xfffff000 & \*((アドレス>>20) &0xffc))」。のページテーブルのアドレスを取得するために使用します。

// ディレクトリのエントリーです。ここで、"((address>>20) &0xffc) "は、エントリーのオフセットを得るために使用されます。

// リニアアドレスのディレクトリテーブル内のインデックスです。アドレス>>22」はディレクトリなので

// エントリーインデックスと各アイテムは4バイトなので、4倍します：「(アドレス>>22)<<2 = (アドレス>>20)」。

// これは、ディレクトリテーブルの指定されたエントリのオフセットアドレスです。"&0xffc "が使用される

// ディレクトリエントリのインデックス値の最後の2ビットをマスクアウトするために、"\*((アドレス>>20) &0xffc)"

// 指定されたディレクトリエントリのコンテンツにある、対応するページテーブルの物理アドレスを取得します。最後に、0xffffff000との "AND "は、ページディレクトリエントリのコンテンツ（ディレクトリエントリの下位12ビット）のフラグビットの一部をマスクするために使用されます。これは、より直感的に次のように表すことができます。

// "(0xffffff000 & \*((unsigned long \*) (((アドレス>>22) & 0x3ff)<<2))))".

1. // (3)ページテーブルエントリのポインタ（物理アドレス）は、(1)のページテーブル内のオフセットアドレスに、(2)のディレクトリエントリのコンテンツ内の対応するページテーブルのアドレスを加えたもので得られる。そして、ここではそれを使って共有ページをコピーします。
2. un\_wp\_page((unsigned long \*)
3. (((address>>10) & 0xffc) + (0xfffff000 &
4. \*((unsigned long \*) ((address>>20) &0xffc)))));

271

272 }

273

//// 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は、そのページをコピーする。この関数は， fork.c の 34 行目にある，メモリ検証を行う // 汎用関数 verify\_area() によって呼び出されます．

1. // パラメータの'address'は、指定したページのリニアアドレスです。
2. void write\_verify(unsigned long address)
3. {
4. unsigned long page;

277

// まず、指定されたリニアアドレスに対応するページディレクトリエントリを取得し、判断します。

// ディレクトリエントリに対応するページテーブルが存在するかどうか、既存の

// ディレクトリエントリのビット(P)が(ビットP=1?)、そうでなければ(P=0)、 return . があるからである。

// 存在しないページに対しては、 // 共有もコピーオンライトも行わず、プログラムが存在しないページに対して // 書き込み操作を行うと、システムはページフォルト例外により do\_no\_page() を実行し、 put\_page() 関数を使用してこの場所に物理ページをマッピングします。

// そして、コードはディレクトリエントリからページテーブルアドレスをフェッチし、さらにページテーブルの

1. ページテーブル内の指定されたページの // エントリオフセット値を取得し，そのアドレスに対応するページテーブルエントリ // ポインタを取得する。与えられたリニアアドレスに対応する物理ページがエントリに含まれます。
2. if (!( (page = \*((unsigned long \*) ((address>>20) & 0xffc)) )&1))
3. return;
4. page &= 0xfffff000;
5. page += ((address>>10) & 0xffc);
6. // 次に、ページテーブルエントリのビット1（R/W）とビット0（P）のフラグをチェックします。ページが書き込み可能ではなく（R/W = 0）、かつ存在する場合は、シェアチェックとページコピー操作（コピーオンライト）を行い、そうでない場合は何もせずに直接終了します。
7. if ((3 & \*(unsigned long \*) page) == 1) /\* non-writeable, present \*/
8. un\_wp\_page((unsigned long \*) page);
9. return;
10. }

286

//// 空いているメモリページを取得し、指定したリニアアドレスにマッピングします。

// get\_free\_page()関数は、メインメモリ領域にある物理メモリのページを取得するだけのアプリケーションです。この関数は，物理メモリのページを取得するだけでなく， // メインメモリ領域にある

// put\_page()で物理ページを指定したリニアアドレスにマッピングする。

1. // パラメータの'address'は、指定したページのリニアアドレスです。
2. void get\_empty\_page(unsigned long address)
3. {
4. unsigned long tmp;

290

1. // フリーページが取得できない場合や，取得したページを指定したアドレスに // 置くことができない場合は，メモリ不足を示すメッセージが表示される。292行目のコメントの意味は、「free\_page()関数のパラメータ'tmp'は0でも構いません、問題ありません、関数はそれを無視して正常に戻ります。
2. if (!(tmp=get\_free\_page()) || !put\_page(tmp,address)) {
3. free\_page(tmp); /\* 0 is ok - ignored \*/
4. oom();
5. }
6. }

296

1. /\*
2. \* try\_to\_share() checks the page at address "address" in the task "p", 299 \* to see if it exists, and if it is clean. If so, share it with the current 300 \* task.
3. \*
4. \* NOTE! This assumes we have checked that p != current, and that they 303 \* share the same executable or library.

304 \*/

//// 現在のプロセスの与えられたアドレスのページを、与えられたプロセスと共有しようとする。

// 現在のプロセスは、プロセス'p'と同じ実行コードを持っています。と考えることもできます。

// 現在のプロセスは、'p' プロセスが fork() を実行して生成された子プロセスであるため、 // コードの内容は最初から同じになります。データセグメントの内容が変更されていなければ、 // それらの内容も同じであるはずです。

// パラメータの'address'は、プロセス内の論理的なアドレスであり、論理的なページである

// 現在のプロセスが'p'プロセスと共有したいと考えているアドレス、pはプロセス

//を共有することになります。p' プロセスの 'アドレス' にあるページが存在し、まだ

//修正して、現在のプロセスにpプロセスと共有させます。また、それと同時に

// 指定されたアドレスがページを申請して取得したかどうかを検証する必要があり、 // そうであれば、カーネルがおかしくなってクラッシュします。ページ共有が成功した場合は1を、失敗した場合は0を返します。

1. 305 static int try\_to\_share(unsigned long address, struct task\_struct \* p) 306 {.
2. unsigned long from;
3. unsigned long to;
4. unsigned long from\_page; // page directory entry of process 'p' 310 unsigned long to\_page; // page directory entry of current process.

311 unsigned long physic\_addr;

312

// の論理アドレス'address'に対応するページ・ディレクトリ・エントリ・ロケーションを示します。

// 与えられたプロセスpと現在のプロセスをそれぞれ取得します。便宜上

// 計算の際には、まず、与えられたアドレスの「論理的な」ページ・ディレクトリ・エントリを見つけます。

// 「アドレス」、つまり、プロセス空間で計算されたページ・ディレクトリ・エントリ・ロケーション

// (0 - 64MB)。この論理エントリの位置と、それに対応するページディレクトリのエントリが

1. // CPUの4Gリニアアドレス空間にあるプロセスpとカレントプロセスの開始アドレスから、 // アドレス'address'のページに対応する実際のページディレクトリエントリアドレス'from\_page'と'to\_page'をそれぞれ得ることができます。
2. from\_page = to\_page = ((address>>20) & 0xffc);
3. from\_page += ((p->start\_code>>20) & 0xffc); // page dir entry of process p.
4. to\_page += ((current->start\_code>>20) & 0xffc); // page dir entry of current.

// 2つのプロセスに対応するディレクトリ・エントリを取得した後、それらを処理する

//を別々に行います。pプロセスのエントリーの動作について説明します。その目的は

// pプロセスの'アドレス'に対応する物理ページアドレスを取得して

// 物理ページが存在し、クリーン（変更されていない、ダーティでない）であるかどうかを判断します。このメソッドは

// は、まずディレクトリエントリの内容を取得し、その中のページテーブルの // 物理アドレスを取得することで、論理アドレス「address」に対応するページテーブルエントリポインタを算出し、ページテーブルエントリの内容を抽出し、一時的に

1. physic\_addrに保存されます。
2. /\* is there a page-directory at from? \*/
3. from = \*(unsigned long \*) from\_page; // get content of page dir entry.
4. if (!(from & 1)) // page table not exist ?
5. return 0;
6. from &= 0xfffff000; // page table address.
7. from\_page = from + ((address>>10) & 0xffc); // page table entry pointer.
8. phys\_addr = \*(unsigned long \*) from\_page; // content of the entry.

// 次に、マッピングされたページテーブルエントリの物理ページが存在し、クリーンであるかどうかを調べる。 // 値「0x41」は、ページテーブルエントリのD（Dirty）およびP（Present）フラグに対応する。

// そのページがダーティであるか、存在しないかを返します。次に、物理的なページアドレスを

1. // エントリを保存して phys\_addr に戻します。最後に，この物理ページアドレスの有効性をチェックする必要があります。マシンの物理アドレスの最大値を超えてはいけませんし， // メモリの下限値（1MB）よりも小さくてもいけません。
2. /\* is the page clean and present? \*/
3. if ((phys\_addr & 0x41) != 0x01)
4. return 0;
5. phys\_addr &= 0xfffff000; // physical page address.
6. if (phys\_addr >= HIGH\_MEMORY || phys\_addr < LOW\_MEM)
7. return 0;

// 同様に、次のコードは、現在のプロセスのエントリを操作します。その目的は

// 現在のページテーブルの'address'に対応するエントリのアドレスを取得する。

//プロセスで、ページテーブルエントリがどの物理的な

// まず、現在のプロセスページのディレクトリエントリの内容を

// 'to' です。エントリーが無効（P=0）の場合、つまり、ディレクトリに対応するページテーブルが

1. // エントリが存在しない場合は、新しいページテーブルを格納するために空きページを要求します。そして、ディレクトリエントリ 'to\_page' の内容を更新して、 // 新しいページを指すようにします。
2. to = \*(unsigned long \*) to\_page; // content of the entry of current.
3. if (!(to & 1))
4. if (to = get\_free\_page())
5. \*(unsigned long \*) to\_page = to | 7;
6. else
7. oom();

// 次に、ディレクトリエントリのページテーブルアドレスを'to'に取り、さらにオフセットアドレスを取ります。

テーブルのエントリーの // を調べ、ページテーブルのエントリーアドレスを 'to\_page' に取得します。これに対して

1. // ページテーブルのエントリで、対応する物理ページがすでに存在していることを確認すると (P=1)、 // プロセス p で対応する物理ページを共有したいが、今はそれがあることを意味します。そうすると、カーネルが間違っていて、クラッシュします。
2. to &= 0xfffff000; // page table address.
3. to\_page = to + ((address>>10) & 0xffc); // page table entry pointer.
4. if (1 & \*(unsigned long \*) to\_page)
5. panic("try\_to\_share: to\_page already exists");

// アドレス'address'に対応するクリーンで既存の物理ページを見つけた後

//をプロセスpで使用し、さらに現在のプロセスのページテーブルエントリーアドレスを決定すると

// 今からそれらを共有し始めます。使用する方法は非常に簡単で、まずページテーブルを変更します。

pプロセスの // エントリで、書き込み保護（R/W=0、読み取り専用）フラグを設定してから

1. // 現在のプロセスがpプロセスのエントリをコピーします。この時点で、現在のプロセスの // アドレス 'address' のページは、pプロセスの論理アドレス 'address' // にマッピングされた物理ページにマッピングされます。
2. /\* share them: write-protect \*/
3. \*(unsigned long \*) from\_page &= ~2;
4. \*(unsigned long \*) to\_page = \*(unsigned long \*) from\_page;
5. // その後、ページ変換キャッシュを更新し、操作されている物理ページのページ番号を計算し、マップバイト配列エントリの参照を1つ増やします。最後に、 // 共有処理が成功した場合は1を返します。
6. invalidate();
7. phys\_addr -= LOW\_MEM;
8. phys\_addr >>= 12; // obtain the page numberin main mem area.
9. mem\_map[phys\_addr]++;
10. return 1;
11. }

348

1. /\*
2. \* share\_page() tries to find a process that could share a page with
3. \* the current one. Address is the address of the wanted page relative 352 \* to the current data space.
4. \*
5. \* We first check if it is at all feasible by checking executable->i\_count.
6. \* It should be >1 if there are other tasks sharing this inode.
7. \*/

//// 同じ実行ファイルを実行しているプロセスを探し、そのプロセスとページを共有しようとします。

// ページフォルト例外が発生した場合、まず、同じ実行ファイルを実行している他のプロセスと // ページを共有できるかどうかを確認します。この関数はまず、システム内に現在のプロセスと同じ実行ファイルを実行している // 他のプロセスがあるかどうかをチェックします。

// そうであれば、システムのすべての現在のタスクの中からそのようなタスクを探します。そのようなタスクが見つかった場合は

// 与えられたアドレスのページを共有するようになります。システム内の他のタスクが、現在のプロセスと同じ実行ファイルを // 実行していない場合、共有ページ操作の前提条件が存在しないため、 // この関数は直ちに終了します。

// 他のプロセスがシステム内の同じ実行ファイルを実行しているかどうかを判断する方法

// を指すプロセスタスク構造の実行可能フィールド（またはライブラリフィールド）を使用することです。

メモリ上の実行ファイルのi-nodeへの//。この判断をするには、リファレンスの

// i-nodeのナンバーフィールドi\_count。ノードのi\_countが1より大きければ、それは

// システム内の2つ以上のプロセスが同じ実行ファイルを実行しているので

// タスク構造体の配列にあるすべてのタスクの実行ファイルフィールドを比較して、 // 同じ実行ファイルを実行しているプロセスがあるかどうかを確認します。

1. // パラメータの'inode'は、ページを共有したいと考えているプロセスの実行ファイルの // i-nodeです。'address' は、現在のプロセスがあるプロセスと // 共有したいと考えているページの論理アドレスです。共有操作が成功した場合は1を、失敗した場合は0を返します。
2. static int share\_page(struct m\_inode \* inode, unsigned long address)
3. {
4. struct task\_struct \*\* p;

360

// まず、パラメータで与えられたメモリのi-node参照カウント値をチェックします。もし、i-node

1. // 参照カウント値が1に等しい(executable->i\_count =1)か、i-nodeポインタが空である場合、 // 現在のシステムで1つのプロセスのみが実行ファイルを実行しているか、 // 提供されたi-nodeが無効であることを意味します。そのため、共有は一切行わず、直接この関数を終了させます。
2. if (inode->i\_count < 2 || !inode)
3. return 0;

// それ以外の場合は、タスク配列のすべてのタスクを検索し、ページを共有できるプロセスを見つけます。

// 現在のプロセス、つまり、同じ実行ファイルを実行している別のプロセスに

// 指定されたアドレスのページを共有します。プロセス論理アドレス'address'がプロセスライブラリファイルの開始アドレスLIBRARY\_OFFSETよりも // 小さい場合、共有ページがプロセス実行ファイルに対応する論理アドレス空間内にあることを示します。 // 次に、与えられたi-nodeがプロセスの実行ファイルのi-nodeと同じかどうか（つまり、プロセスの実行可能なフィールド）をチェックします。同じでない場合は、検索を続ける。 // 論理アドレス'address'がプロセスのライブラリファイルの開始アドレスLIBRARY\_OFFSET //以上であれば、共有すべきページがライブラリファイルにあることを示している。 // そこで、指定された'inode'がプロセスのライブラリファイルのi-nodeと同じかどうかをチェックし、 // 同じでなければ、検索を続ける。

1. // inode' と同じ 'execute' または 'library' フィールドを持つプロセス 'p' が見つかると、 // try\_to\_share() 関数が呼び出され、ページの共有が試みられます。この関数は、共有操作が成功した場合は // 1 を、そうでない場合は 0 を返します。
2. for (p = &LAST\_TASK ; p > &FIRST\_TASK ; --p) {
3. if (!\*p) // continue search if the item is null.
4. continue;
5. if (current == \*p) // continue search if it’s the current.
6. continue;
7. if (address < LIBRARY\_OFFSET) {
8. if (inode != (\*p)->executable) // i-node of the executable file.
9. continue;
10. } else {
11. if (inode != (\*p)->library) // i-node of the library file.
12. continue;
13. }
14. if (try\_to\_share(address,\*p))
15. return 1;
16. }
17. return 0;
18. }

380

//// ページノットプレスト処理。

// この関数は、ページ例外割り込み処理中に呼び出され、ページ.sの

// プログラムを作成します。パラメータの'error\_code'と'address'は，プロセスがpage-not-present faultによってページにアクセスする際に， // CPUによって自動的に生成される。error\_code' はエラーの種類を示し、 // 'address' は例外を発生させたページのリニアアドレスです。

// この関数は、まず、見つからないページがスワップデバイスにあるかどうかを確認し、あれば、スワップ

で // 読み込みます。そうでない場合は、すでに読み込まれた同じファイルでページを共有しようとする、または

1. // プロセスが動的にメモリページを要求したからといって、 // 単に物理的なメモリページのページをマッピングするだけです。共有操作が成功しなかった場合、不足しているデータページは、 // 対応するファイルから指定されたリニアアドレスに読み込まれるだけです。
2. void do\_no\_page(unsigned long error\_code,unsigned long address)
3. {
4. int nr[4];
5. unsigned long tmp;
6. unsigned long page;
7. int block,i;
8. struct m\_inode \* inode;

388

// この関数は、まず、CPU制御から与えられたリニアアドレスがどの範囲にあるかを判断します。

// ページ例外の原因となったレジスタCR2。アドレスがTASK\_SIZE (0x4000000,

// または64MB）の線形アドレス範囲内にあることを意味します。

// カーネル、タスク0、タスク1のいずれかを選択し、警告メッセージを発行しました; If (address - current process.

1. // コード開始アドレス）が1プロセスのサイズ（64MB）よりも大きい場合、リニアアドレスが例外を引き起こしたプロセスの空間内にないことを示し、エラーメッセージを発行して終了します。
2. if (address < TASK\_SIZE)
3. printk("\n\rBAD!! KERNEL PAGE MISSING\n\r");
4. if (address - current->start\_code > TASK\_SIZE) {
5. printk("Bad things happen: nonexistent page error in do\_no\_page\n\r");
6. do\_exit(SIGSEGV);
7. }

// そして、指定されたリニアアドレス'address'に応じて、対応するセカンダリページテーブルのエントリポインタを取得し、エントリの内容に応じて、'address'のページがスワップデバイスにあるかどうかを判断する // ことになる。yesの場合、そのページをスワップして終了します。

// この方法では、まず、ページディレクトリのアイテムに対応するコンテンツを取得します。

// 指定されたリニアアドレス「address」、そしてそこにあるページテーブルのアドレスを取り出す。

// そして、ページテーブルエントリのオフセットを追加して、対応するページテーブルエントリポインタを取得します。

// それにより、ページテーブルエントリの内容を取得します。エントリの内容が

1. // ビットP=0であれば，ページテーブルエントリで指定された物理ページが // スワップデバイスにあるべきであることを示します。そして，指定されたページがスワップデバイスから読み込まれた後，この関数は終了する。
2. page = \*(unsigned long \*) ((address >> 20) & 0xffc); // content of page dir entry.
3. if (page & 1) {
4. page &= 0xfffff000; // page table address.
5. page += (address >> 10) & 0xffc; // page table entry pointer.
6. tmp = \*(unsigned long \*) page; // content of page entry.
7. if (tmp && !(1 & tmp)) {
8. swap\_in((unsigned long \*) page); // read in from swap device.
9. return;
10. } 404 }

// そうでなければ、与えられたリニアアドレス'address'のページアドレスを取得して

1. // プロセスベースアドレスに対するプロセス空間内のアドレスの // オフセット「tmp」，すなわち対応する論理アドレス。このようにして，実行ファイルイメージやライブラリファイルの欠落したページの // 特定の開始ブロック番号を計算することができる。
2. address &= 0xfffff000; // page address
3. tmp = address - current->start\_code; // logical address of the page.

// 論理アドレス'tmp'がライブラリイメージファイルの開始位置よりも大きい場合

// プロセスの論理空間では、欠落したページはライブラリのイメージファイルにあります。したがって、その

// 現在のプロセスタスクからライブラリ画像ファイルのi-node 'library'を取得できる

// 構造、およびライブラリファイル内の欠落したページの開始ブロック番号'block'

//が算出されます。論理アドレス'tmp'が実行イメージの終端よりも小さければ

プロセスの//ファイルであれば、欠落しているページはプロセスの実行ファイルイメージにあるので

// 実行ファイルのi-node番号「executable」を現在のプロセスから取得可能

// タスク構造の中で欠落しているページの開始ブロック番号「block」を計算します。

// 実行ファイルイメージ。論理アドレス'tmp'が、実行ファイルイメージのアドレス範囲に含まれていない場合は

// 実行ファイルでもライブラリファイル空間でもなく、ページが存在しないのは、動的に要求されたメモリページデータにプロセスがアクセスしているため、対応するi-nodeとデータブロック番号（両方ともNULLに設定されている）が存在しないためです。

// ブロックデバイスに格納されている画像ファイルの最初のブロックは、プログラムのヘッダーであるため

//構造になっているため、ファイルを読み取る際に最初のデータブロックをスキップする必要があります。なぜなら、各

1. // データブロックの長さはBLOCK\_SIZE = 1KBで、1ページのメモリには4ブロック分のデータが格納できます。プロセスの論理アドレス'tmp'をデータブロックサイズ＋1で割って、実行イメージファイルの欠落ページの開始ブロック番号'block'を求める。
2. if (tmp >= LIBRARY\_OFFSET ) {
3. inode = current->library; // inode & block no. of library file.
4. block = 1 + (tmp-LIBRARY\_OFFSET) / BLOCK\_SIZE;
5. } else if (tmp < current->end\_data) {
6. inode = current->executable; // inode & block no of executable file.
7. block = 1 + tmp / BLOCK\_SIZE;
8. } else {
9. inode = NULL; // for dynamically applied page.
10. block = 0;
11. }

// プロセスがその動的アプリケーションのページにアクセスした場合、またはページフォルト例外が発生した場合

// スタック情報を格納することで、物理メモリのページを直接申請します。

// そしてそれをリニアアドレス「address」にマッピングする。それ以外の場合は、行方不明のページがスコープ内の

プロセス実行ファイルやライブラリファイルの//を共有しようとするので、ページ操作をして終了します。

1. // 成功した場合。失敗した場合は，物理メモリのページを申請し， // 実行ファイルの対応するページをデバイスから読み出し， // プロセスページの論理アドレス'tmp'に配置（マッピング）する必要がある。
2. if (!inode) { // it’s a dynamic applied page.
3. get\_empty\_page(address);
4. return;
5. }
6. if (share\_page(inode,tmp)) // try to share the page at 'tmp'.
7. return;
8. if (!(page = get\_free\_page())) // apply for a free page.
9. oom();
10. /\* remember that 1 block is used for header \*/
11. // このブロック番号と実行ファイルのi-nodeをもとに、マッピングビットマップから対応するブロックデバイスのデバイス論理ブロック番号（nr[]配列に格納）を見つけ、bread\_page()を使って4つの論理ブロックを物理ページに読み込むことができます。
12. for (i=0 ; i<4 ; block++,i++)
13. nr[i] = bmap(inode,block);
14. bread\_page(page,inode->i\_dev,nr);

// デバイス・ロジック・ブロックの読み出し時には、読み出しページ位置が

//は、ファイルの最後から1ページに満たない場合があります。そのため、いくつかの無駄な情報

// を読み取ることができます。以下の操作は、実行を超えてこの部分をクリアするためのものです。

1. // ファイル「end\_data」。もちろん、最後から1ページ以上離れている場合は、実行ファイルイメージのファイルからではなく、ライブラリファイルから読み込まれるので、 // クリア操作を行う必要はありません。
2. i = tmp + 4096 - current->end\_data; // the excess byte length.
3. if (i>4095) // more than 1 page from the end. 431 i = 0;
4. tmp = page + 4096; // points to the end of the page.
5. while (i-- > 0) { // i bytes cleared start from page end.
6. tmp--;
7. \*(char \*)tmp = 0;
8. }

// 最後に、ページフォルト例外の原因となるページが、指定されたリニアアドレスにマッピングされる

1. // のアドレスを返します。操作が成功した場合はリターンし、そうでない場合は、メモリが十分でないことを示すために、メモリページが // リリースされます。
2. if (put\_page(page,address))
3. return;
4. free\_page(page);
5. oom();
6. }

442

//// メモリ管理の初期化。

// 1MBアドレス以上の物理メモリ領域を初期化します。カーネルが管理する

// ページ長4KBのメモリを、ページ単位でアクセスします。この機能は、すべての物理的

// 1MB以上のメモリをページに分割し、そのページをページマップドバイトアレイで管理する。

// メモリが16MBのマシンの場合、配列は3840項目（（16MB - 1MB） / 4KB）となります。

// で、3840の物理ページを管理することができます。メモリページが占有されるたびに、対応する

// mem\_map[]内のバイト項目は1つずつ増加し、ページが解放されると、対応するバイト

// の値が1つデクリメントされます。バイト項目が0の場合は、対応するページがアイドルであることを示し、バイト値が1以上の場合は、そのページが占有されているか、複数のプロセスで共有されていることを示します。

// カーネルのバッファキャッシュや一部のデバイスでは、一定量のメモリを使用する必要があるため

// システムが実際に使用するために割り当てられるメモリの量が減ります。私たちはこのメモリを

// メインメモリエリア」（MMA）としてカーネルが実際に使用するために割り当てることができる領域。

// その開始位置は変数'start\_mem'で表され、終了アドレスは

// end\_mem'で表されます。16MBのメモリを持つPCシステムの場合、'start\_mem'は通常

// 4MB、'end\_mem'は16MBです。したがって，この時点でのメインメモリ領域は [4MB-16MB] であり， // 合計3072の物理ページが割り当て可能である。範囲0～1MBのメモリ領域は、カーネル用に予約されている。

// パラメータの'start\_mem'は、使用可能なメインメモリ領域の開始アドレスです。

1. // ページの割り当てのためのものです（RAMDISKが占有していたメモリ空間は削除されています）。end\_mem」は、 // 実際の物理メモリの最大アドレスです。アドレス範囲[start\_mem, end\_mem]は、 // メインメモリ領域です。
2. void mem\_init(long start\_mem, long end\_mem)
3. {
4. int i; 446
5. // この関数はまず、1MBから16MBの範囲のすべてのページに対応するメモリマップドバイト配列項目を占有状態、つまりすべてのバイトをUSED（100）に設定します。 // PAGING\_PAGESは(PAGING\_MEMORY>>12)と定義され、これは1MB以上のすべての物理メモリページの数(15MB/4KB = 3840)です。
6. HIGH\_MEMORY = end\_mem; // set the memory top (16MB).
7. for (i=0 ; i<PAGING\_PAGES ; i++) // PAGING\_PAGES = 3840.
8. mem\_map[i] = USED;

// 次に、開始アドレスのページに対応するバイト配列のアイテム番号'i'を見つけます。

// 'start\_mem'で、メインメモリ領域のページ数を計算します。この時点では

// mem\_map[]のi番目の項目は、メインメモリ領域の最初のページに対応しています。最後に

1. // メインメモリ領域のページに対応する配列項目がクリアされます（アイドルを示します）。16MBの物理メモリを持つシステムでは、mem\_map[]の4MB～16MBのメインメモリ領域に対応するバイトがクリアされます。
2. i = MAP\_NR(start\_mem); // page number at the beginning of the MMA.
3. end\_mem -= start\_mem;
4. end\_mem >>= 12; // total number of pages in MMA. 453 while (end\_mem-->0)

454 mem\_map[i++]=0; // MMAページのバイトがリセットされる。455 }

456

// システムのメモリ情報を表示します。

// システムで使用されているメモリページ数と物理メモリの総ページ数

メインメモリー領域の//は、mem\_map[]の情報に従ってカウントされており

1. // ページディレクトリとページテーブルの内容を表示します。この関数は、"Shift + Scroll // Lock "キーが押されたときにシステムメモリの統計情報を表示する // chr\_drv/keyboard.S プログラムの 186 行目で呼び出されます。
2. void show\_mem(void)
3. {
4. int i,j,k,free=0,total=0;
5. int shared=0;
6. unsigned long \* pg\_tbl;

462

// まず，バイト配列mem\_map[]に従って，メインメモリ領域のページの総数'total'，空きページの数'free'，共有ページの数をカウントする

1. // 'shared'で、これらの情報を表示します。
2. printk("Mem-info:\n\r");
3. for(i=0 ; i<PAGING\_PAGES ; i++) {
4. if (mem\_map[i] == USED) // pages not for allocation.
5. continue;
6. total++;
7. if (!mem\_map[i])
8. free++; // free pages in the main mem ares.
9. else
10. shared += mem\_map[i]-1; // shared pages (byte value > 1).
11. }
12. printk("%d free pages of %d\n\r",free,total);
13. printk("%d pages shared\n\r",shared);

// 次に、CPUのページング管理ロジックのページ数を数えます。最初の4項目は

ページディレクトリテーブルの // は、カーネルコードが使用するもので、統計的には

// の範囲になります。この方法では、項目5から始まるすべてのページディレクトリエントリをループします。もし

1. // 対応する二次ページテーブルが存在する場合、まず二次ページテーブル自体が占有するメモリページがカウントされ（484行目）、次にページテーブルのすべてのページエントリに対応する物理メモリページがカウントされます。
2. k = 0; // statistics of pages occupied by a process.
3. for(i=4 ; i<1024 ;) {
4. if (1&pg\_dir[i]) {

//(ページディレクトリエントリ内の対応するページテーブルのアドレスが

// マシンの最高物理メモリアドレスであるHIGH\_MEMORYに問題があります。

// ディレクトリ・エントリを表示します。ディレクトリ・エントリの情報が表示され、次のディレクトリ

1. // エントリーの処理を続けます)
2. if (pg\_dir[i]>HIGH\_MEMORY) { // content is abnormal.
3. printk("page directory[%d]: %08X\n\r",
4. i,pg\_dir[i]);
5. continue; // needs "i++;" before it.
6. }

// ページディレクトリエントリのページテーブルの「アドレス」がLOW\_MEMよりも大きい場合（その

//つまり1MB）、1つのプロセスの物理メモリページ統計値'k'がインクリメントされます。

// システムが占有しているすべての物理メモリページの統計値が「free」であること。

//を1つずつ増やしていきます。次に、対応するページテーブルアドレス'pg\_tbl'を取得し、カウント

1. // ページテーブルのすべての項目の内容を示す。現在のページテーブル項目が示す物理ページが存在し、物理ページの「アドレス」がLOW\_MEMよりも大きい場合、 // そのページテーブル項目の対応するページが統計値に含まれる。
2. if (pg\_dir[i]>LOW\_MEM)
3. free++,k++; // pages occupied in the page table.
4. pg\_tbl=(unsigned long \*) (0xfffff000 & pg\_dir[i]);
5. for(j=0 ; j<1024 ; j++)
6. if ((pg\_tbl[j]&1) && pg\_tbl[j]>LOW\_MEM)

// （物理ページのアドレスが本機の最上位の物理メモリアドレスよりも大きい場合は

// マシンのHIGH\_MEMORYは、ページテーブルの内容に問題があることを示しています。

// の項目があるので、ページテーブルの項目の内容が表示されます。それ以外の場合は、対応するページ

13.3ページ.s

1. ページテーブル項目の//は統計値に含まれます)。
2. if (pg\_tbl[j]>HIGH\_MEMORY)
3. printk("page\_dir[%d][%d]: %08X\n\r",
4. i,j, pg\_tbl[j]);
5. else
6. k++,free++; // pages of page table items.
7. }

// 各タスクの線形空間サイズは64MBなので、1つのタスクは16ページのディレクトリエントリを占有します。

1. // したがって、ここでは16個のディレクトリエントリがカウントされるごとに、プロセスの // タスク構造が占有するページテーブルがカウントされることになる。この時、k=0であれば、現在の16ディレクトリエントリに対応するプロセスが // システムに存在しない（作成されていない、終了していない）ことを意味する。 // そして、対応するプロセス番号と物理ページ統計値kが表示された後、 // kをクリアして次のプロセスが占有するページをカウントする。
2. i++;
3. if (!(i&15) && k) { // k !=0 indicates that the process exists.
4. k++,free++; /\* one page/process for task\_struct \*/
5. printk("Process %d: %d pages\n\r",(i>>4)-1,k);
6. k = 0;
7. } 500 }
8. // 最後に、システムで使用されているメモリページと、メインメモリ領域の総ページ数が表示されます。
9. printk("Memory found: %d (%d)\n\r",free-shared,total);
10. }

503

## 13.3 page.s

### 13.3.1 Function

page.sファイルには、ページフォルト例外割り込みハンドラ（割り込み14）が含まれており、主に2つのケースで処理されます。1つ目は、ページが存在しないことによるページ・フォールト例外で、これはdo\_no\_page(error\_code, address)を呼び出して処理する必要があります。2つ目は、ページ・ライト・プロテクションによるページ例外です。このとき、ページ書き込み保護ハンドラdo\_wp\_page(error\_code, address)を呼び出して処理します。関数パラメータのエラーコード(error\_code)は、CPUによって自動的に生成され、スタックにプッシュされます。例外発生時にアクセスされるリニア・アドレス（address）は、コントロール・レジスタCR2から取得されます。CR2は特に、ページが失敗したときのリニアアドレスを格納するために使用されます。

### 13.3.2 Code annotation

プログラム 13-2 linux/mm/page.s

1. /\*
2. \* linux/mm/page.s
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/

6

1. /\*
2. \* page.s contains the low-level page-exception code.
3. \* the real work is done in mm.c // memory.c
4. \*/

11

// この変数は、kernel/traps.c でページ例外記述子を設定するために使用されます。

12 .globl \_page\_fault # グローバル変数として宣言。

13

1. \_page\_fault:
2. xchgl %eax,(%esp) # take the error code to EAX.
3. pushl %ecx
4. pushl %edx
5. push %ds
6. push %es
7. push %fs
8. movl $0x10,%edx # set the kernel data segment selector.
9. mov %dx,%ds
10. mov %dx,%es
11. mov %dx,%fs
12. movl %cr2,%edx # get the linear address that caused the page exception.
13. pushl %edx # the linear address and error code are pushed onto the stack 27 pushl %eax # as arguments to the function to be called.
14. // 「ページ存在」フラグP（ビット0）をテストし、ページフォルトによる例外であれば // do\_no\_page()関数を呼び出す。そうでなければ、ページ書き込み保護関数do\_wp\_page()を呼び出す。
15. testl $1,%eax # Test flag P (bit 0), jump if it is set.
16. jne 1f
17. call \_do\_no\_page # mm/memory.c, line 381. 31 jmp 2f
18. 1: call \_do\_wp\_page # mm/memory.c, line 254.
19. // スタックにプッシュされた2つの引数を破棄し、レジスタをポップして割込みを終了します。
20. 2: addl $8,%esp
21. pop %fs
22. pop %es
23. pop %ds
24. popl %edx
25. popl %ecx
26. popl %eax
27. iret

## 13.4 swap.c

### 13.4.1 Function

Linuxでは、バージョン0.12以降、カーネルに仮想メモリのスワップ機能が追加されました。この機能は主にこのプログラムによって実装されています。このプログラムは、物理メモリの容量が限られていて使用量が逼迫しているときに、一時的に使用していないメモリページの内容をディスク（スワップデバイス）に保存して、緊急に必要なプログラムのためにメモリ領域を確保します。すでにスワップデバイスに保存されているメモリページコンテンツを後になって再び使用する必要が生じた場合には、プログラムがそれらを取り込んでメモリに戻す作業を行う。メモリスワップ管理は、主記憶領域管理と同様のマッピング手法を用いており、ビットマップを用いてスワップされたメモリページの具体的な保存場所やマップの位置を決定する。

カーネルをコンパイルする際に、スワップデバイス番号SWAP\_DEVを定義しておけば、コンパイルされたカーネルはメモリスワップ機能を持つことになります。Linux 0.12では、スワップデバイスは、ファイルシステムを含まないハードディスク上の指定された独立したパーティションを使用しています。スワッププログラムが初期化されると、まずスワップデバイスの0ページを読み込みます。このページはスワップ領域管理ページで、スワッピングページ管理で使用するビットマップが含まれています。4068バイト目から始まる10文字は、スワップデバイスの機能文字列「SWAP-SPACE」です。この特徴的な文字列がパーティションにない場合、与えられたパーティションは有効なスワップデバイスではありません。 swap.c プログラムには、主にスワップマッピングビットマップ管理関数とスワップデバイスアクセス関数が含まれています。get\_swap\_page()関数とswap\_free()関数は、それぞれスワップビットマップに基づいてスワップページを適用し、スワップデバイス内の指定されたページを解放するために使用されます。swap\_out()関数とswap\_in()関数は、それぞれスワップデバイスとの間でメモリページ情報を出力/入力するために使用されます。swap\_out()関数とswap\_in()関数は、それぞれスワップデバイスとの間でメモリページ情報の入出力を行います。この2つの関数は、インクルード/linux/mm.hヘッダーファイルでマクロの形で定義されています。

#define read\_swap\_page(nr,buffer) ll\_rw\_page(READ,SWAP\_DEV,(nr),(buffer))となります。

#define write\_swap\_page(nr,buffer) ll\_rw\_page(WRITE,SWAP\_DEV,(nr),(buffer)) となります。

ll\_rw\_page()関数は、ブロックデバイスの低レベルのページ読み書き関数です。このコードは、kernel/blk\_drv/ll\_rw\_blk.cファイルに実装されています。スワップデバイスアクセス関数は、基本的にデバイス番号を指定したデバイスページアクセス関数であることがわかります。

カーネルの初期化処理において、システムがスワップデバイス番号SWA\_DEVを定義している場合、カーネルはスワップ処理の初期化関数init\_swapping()を実行します。この関数はまず、システム内のブロックの配列に基づいて、システムにスワップデバイスがあるかどうか、デバイスのスワップパーティションが有効かどうかをチェックします。そして、メモリページを要求し、スワップパーティションの最初のスワップ管理ページ（ページ0）をメモリページに読み込みます。ページ0には、スワップページのビットマップマッピング情報が格納されており、各ビットがスワップページを表します。ビットが0の場合は、対応するデバイスのスワップページが使用されている（占有されている）か、利用できないことを示し、ビットが1の場合は、対応するスワップページが利用可能であることを示します。1ページはSWAP\_BITS（4096×8＝32768）ビットの合計であるため、スワップパーティションは最大で32,768ページを管理することができます。 実行中のLinuxでは、ブロックデバイスのパーティショナー（fdiskなど）がスワップパーティションを初期化して「swap\_size」のスワップページを持つようにした場合、デバイス上のスワップパーティションの最初のページ（ページ0）がスワップ管理に使われる（占有される）ので、スワップビットマップの最初のビットも0になるはずです。したがって、デバイス上で実際に利用可能なスワップページの数は「swap\_size - 1」であり、そのページ番号の範囲は[1 -- swap\_size-1]であるため、ビットマップ内の対応するビットはすべて1（アイドル）となります。ビットマップの[swap\_size -- SWAP\_BITS]の範囲のビットは、デバイス上に対応するスワップページがないため、利用できない状態（すべて0）に初期化されます。したがって、初期の正常なケースでは、ビットマップのビット状態は以下のようになるはずです（斜線部分はスワップに利用できません）。

/ ビットマップのビット数。0, 1, 1, 1, ... , 1, 0, ..., 0.

スワップページ番号 : 0, 1, 2, 3, .... , swap\_size-1, swap\_size, ..., SWAP\_BITS-1.

### 13.4.2 Code annotation

プログラム 13-3 linux/mm/swap.c

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

6

1. /\*
2. \* This file should contain most things doing the swapping from/to disk.
3. \* Started 18.12.91
4. \*/

11

// <string.h> 文字列のヘッダファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。

// <linux/mm.h> メモリ管理のヘッダファイルです。ページサイズの定義と、いくつかのページ // リリース関数のプロトタイプが含まれています。

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

// <linux/head.h> 頭部のヘッダーファイルです。セグメントディスクリプターの簡単な構造が定義されており、 // いくつかのセレクタ定数とともに定義されています。

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

12 #include <string.h>

13

1. #include <linux/mm.h>
2. #include <linux/sched.h>
3. #include <linux/head.h>
4. #include <linux/kernel.h>

18

// 各バイトは8ビットなので、1ページ（4096バイト）は合計32768ビットです。もし、1つのビットが

// メモリの1ページ分のビットマップは、メモリ容量の128MBに対応する32,768ページまで管理できます。このビットがセットされている場合、対応するスワップページがアイドル状態であることを示します。

19 #define SWAP\_BITS (4096<<3) // 1ページ内のスワップ総ビット数を定義 (32768).

20

//// bitop()は、ビット操作のマクロです。異なる "op "を与えることで、指定したビットをテスト、設定、クリアする3つの // 操作を定義することができます。

// インライン・アセンブリ関数のパラメータ'addr'はリニア・アドレスを指定し、'nr'は

// 指定されたアドレスのビットオフセットです。このマクロは、指定されたアドレス'addr'の'nr'番目のビットの値を // キャリーフラグに入れ、ビットをセットまたはリセットし、キャリーフラグの値を返します（つまり、元のビット値を返します）。

// 25行目の最初の命令は、"op "と結合して異なる命令を形成する。

// "op" = ""は、bt - Bit testという命令で、キャリーを元の値に設定します。

// "op" = "s", インストラクション bts - ビットテストとセットで，元のビットがキャリーにセットされる。 // "op" = "r", 命令 btr - ビット・テストとリセットを行い、元のビットをキャリーに設定します。 // 入力: %0 - (戻り値), %1 - ビット・オフセット (nr), %2 - ベース・アドレス (addr), %3 - プラス // オペレーション・レジスタの初期値 (0).

// インライン・アセンブリ・コードは、ベース・アドレス(%2)とビット・オフセットで指定されたビットを保存します。

// (%1)をキャリーフラグCFに設定し，ビットをセット（リセット）します。命令ADCLがロードされる

1. キャリービット CF に従ってオペランド(%0)を設定するためのキャリービットを持つ // 。CF＝1の場合、リターンレジスタ値＝1、そうでない場合はリターンレジスタ値＝0となります。
2. #define bitop(name,op) \
3. static inline int name(char \* addr,unsigned int nr) \
4. { \
5. int \_\_res; \
6. \_\_asm\_\_ \_\_volatile\_\_("bt" op " %1,%2; adcl $0,%0" \
7. :"=g" (\_\_res) \
8. :"r" (nr),"m" (\*(addr)),"0" (0)); \
9. return \_\_res; \
10. }

30

1. // ここでは、異なるop文字に応じて3つのインライン関数を定義します。
2. bitop(bit,"") // define bit(char \* addr, unsigned int nr)
3. bitop(setbit,"s") // define setbit(char \* addr, unsigned int nr) 33 bitop(clrbit,"r") // define clrbit(char \* addr, unsigned int nr) 34
4. static char \* swap\_bitmap = NULL;
5. int SWAP\_DEV = 0; // The swap device number set when the kernel is initialized.

37

1. /\*
2. \* We never page the pages in task[0] - kernel memory.
3. \* We page all other pages.
4. \*/
5. // タスク0(64MB)の終わりから始まる仮想メモリページである最初の仮想メモリページです。
6. #define FIRST\_VM\_PAGE (TASK\_SIZE>>12) // = 64MB/4KB = 16384 43 #define LAST\_VM\_PAGE (1024\*1024) // = 4GB/4KB = 1048576 44 #define VM\_PAGES (LAST\_VM\_PAGE - FIRST\_VM\_PAGE) // = 1032192 (count from 0).

45

//// 応募して、スワップのページ番号をゲット。

1. // スワップビットマップ全体（ビットマップページ自体のビット0を除く）をスキャンし、 // 最初に見つかったビットをリセットし、その位置（現在の空きスワップページ番号）を返します。操作が成功した場合はスワップページ番号を、そうでない場合は0を返します。
2. static int get\_swap\_page(void)
3. {
4. int nr;

49

1. if (!swap\_bitmap)
2. return 0;
3. for (nr = 1; nr < 32768 ; nr++)
4. if (clrbit(swap\_bitmap,nr))
5. return nr; // return the current free swap page number.
6. return 0;
7. }

57

// スワップデバイスで指定されたsawpページを解放する。

// パラメータで指定されたページ番号に対応するスワップビットマップのビットを設定します。

1. // スワップビットマップでは、ビットが1の場合、対応するスワップページがアイドル状態であることを意味します。したがって、 // オリジナルのビットが1になった場合は、スワップデバイスのオリジナルページが // 占有されていないか、ビットマップがエラーになっていることを意味します。そして、エラーメッセージが表示され、返されます。
2. void swap\_free(int swap\_nr)
3. {
4. if (!swap\_nr)
5. return;
6. if (swap\_bitmap && swap\_nr < SWAP\_BITS)
7. if (!setbit(swap\_bitmap,swap\_nr))
8. return;
9. printk("Swap-space bad (swap\_free())\n\r");
10. return;
11. }

68

// 指定したページをメモリにスワップします。

// 指定されたページテーブルエントリのページが、スワップデバイスから新たに

1. // 要求されたメモリページを指定します。同時に、スワップビットマップの対応するビットを修正し（セット）、ページテーブルエントリの内容を修正し、メモリページを指すようにし、 // 対応するフラグをセットする。
2. void swap\_in(unsigned long \*table\_ptr)
3. {
4. int swap\_nr;
5. unsigned long page;

73

// この関数は，まずスワップビットマップとパラメータの有効性をチェックする。スワップビットマップが存在しない場合、指定されたページテーブルエントリに対応するページが既にメモリ内に存在する場合、スワップページ番号が0の場合は、警告メッセージを表示して終了します。

1. // スワップデバイスに置かれたメモリページに対して、対応するページテーブルの // エントリはスワップページ番号\*2、すなわち(swap\_nr << 1)でなければなりません。以下の関数 try\_to\_swap\_out() の 111 行目の説明を参照してください。
2. if (!swap\_bitmap) {
3. printk("Trying to swap in without swap bit-map");
4. return;
5. }
6. if (1 & \*table\_ptr) {
7. printk("trying to swap in present page\n\r");
8. return;
9. }
10. swap\_nr = \*table\_ptr >> 1;
11. if (!swap\_nr) {
12. printk("No swap page in swap\_in\n\r");
13. return; 86 }

// その後、メモリページを申請し、ページ番号swap\_nrのページをswapから読み込む

// デバイスになります。read\_swap\_page()を使ってページがスワップされた後、そのページの対応するビットが

// スワップビットマップが設定されます。もともとセットされている場合は、同じページが再度読み込まれることを意味します。

// と表示され、警告メッセージが表示されてしまいます。最後に、ページテーブルに

1. 物理ページへの // エントリポイントを設定し、ページの変更、ユーザーの読み取り可能性と書き込み可能性、 // 存在フラグ (Dirty, U/S, R/W, P) を設定します。
2. if (!(page = get\_free\_page()))
3. oom();
4. read\_swap\_page(swap\_nr, (char \*) page); // defined in file nclude/linux/mm.h
5. if (setbit(swap\_bitmap,swap\_nr))
6. printk("swapping in multiply from same page\n\r");
7. \*table\_ptr = page | (PAGE\_DIRTY | 7);
8. }

94

// ページを入れ替えてみてください。

// メモリページが変更されていない場合、対応するページは対応するイメージファイルからも直接読み取れるため、 // スワップデバイスに保存する必要はありません。 // そのため、対応する物理ページを直接解放することができます。そうでない場合は、スワップを申請する

//のページ番号にしてから、スワップします。このとき、スワップページ番号を保存するために

1. // パラメータ'table\_ptr'はページテーブルエントリへのポインタである。ページのスワップまたはリリースが成功した場合は // 1を返し、そうでない場合は0を返します。
2. int try\_to\_swap\_out(unsigned long \* table\_ptr)
3. {
4. unsigned long page;
5. unsigned long swap\_nr;

99

// この関数は，まず，パラメータの有効性を判断します。を必要とするメモリページが存在しない場合

1. // ページテーブルエントリで指定された物理ページアドレスが、メモリ管理ハイエンドのPAGING\_MEMORY // (15MB)よりも大きい場合も終了します。
2. page = \*table\_ptr;
3. if (!(PAGE\_PRESENT & page))
4. return 0;
5. if (page - LOW\_MEM > PAGING\_MEMORY)
6. return 0;

// メモリページが変更されていても、そのページが共有されている場合は、そのページを改善するために

1. // このようなページはスワップアウトされるべきではないので， // 0を返して存在します。それ以外の場合は，スワップページ番号を取得してページテーブルのエントリに保存し， // そのページをスワップアウトして対応する物理メモリページを解放します。
2. if (PAGE\_DIRTY & page) {
3. page &= 0xfffff000; // get physical page address.
4. if (mem\_map[MAP\_NR(page)] != 1)
5. return 0;
6. if (!(swap\_nr = get\_swap\_page())) // get a swap page number.
7. return 0;

// スワップされるページについては、スワップページ番号（swap\_nr << 1）が、対応する

// ページテーブルエントリ。2をかけるのは、ページテーブルエントリのプレゼンスビット（P、ビット0）を解放するためです。

// ページテーブルエントリ。ビットP=0で、ページテーブルエントリの内容が0以外のページのみが

// がスワップデバイスにあることを示しています。インテルのマニュアルでは、テーブルエントリに

1. // P = 0 (無効なページ)、他のすべてのビット(ビット31 - 1)は自由に使用できます。書き込みページ関数 // write\_swap\_page(nr, buffer)はll\_rw\_page(WRITE, SWAP\_DEV, (nr), (buffer))として定義されており、 // linux/mm.hファイルの12行目を参照。
2. \*table\_ptr = swap\_nr<<1;
3. invalidate(); // refresh the CPU transform cache.
4. write\_swap\_page(swap\_nr, (char \*) page);
5. free\_page(page);
6. return 1;
7. }
8. // それ以外の場合は、そのページが変更されていないことを意味するので、 // 交換する必要はなく、直接リリースすることができます。
9. \*table\_ptr = 0;
10. invalidate();
11. free\_page(page);
12. return 1;
13. }

122

1. /\*
2. \* Ok, this has a rather intricate logic - the idea is to make good
3. \* and fast machine code. If we didn't worry about that, things would
4. \* be easier.
5. \*/

// スワップデバイスにメモリページを入れる。

// リニアに対応するページディレクトリエントリ(FIRST\_VM\_PAGE>>10)から開始します。

// 64MBのアドレスでは、4GBのリニアスペース全体が検索されます。この間に、スワップを試みた

// 対応するメモリページをスワップデバイスに格納します。ページのスワップに成功した場合は 1 を返します。

1. // 2つの静的ローカル変数は、現在の検索ポイントを次の検索の開始のために // 一時的に保存するために使用されます。この関数は get\_free\_page() (172行目で定義)で // 呼び出されます。
2. int swap\_out(void)
3. {
4. static int dir\_entry = FIRST\_VM\_PAGE>>10; // 16, first directory entry of task 1.
5. static int page\_entry = -1;
6. int counter = VM\_PAGES; // 1032192, see line 44.
7. int pg\_table; 134

// まず、ページディレクトリテーブルをループして、ページディレクトリエントリを見つけます pg\_table

// 見つかった場合はループを終了します。

// ディレクトリエントリの残りの2次ページテーブル「カウンタ」の数を計算して

1. // 次のディレクトリ・エントリのチェックを続けます。すべての検索の結果、適切な（既存の）ページ・ディレクトリ・エントリが // 見つからなかった場合、コードは0を返します。
2. while (counter>0) {
3. pg\_table = pg\_dir[dir\_entry]; // content of the directory entry.
4. if (pg\_table & 1) // exit the loop if the table exists.
5. break;
6. counter -= 1024; // One page table has 1024 items.
7. dir\_entry++; // next directory entry.
8. if (dir\_entry >= 1024)
9. dir\_entry = FIRST\_VM\_PAGE>>10;
10. }

// カレントディレクトリエントリのページテーブルポインタを取得した後、swap関数は

1. // try\_to\_swap\_out()は、ページテーブルにある1024ページすべてに対して1つずつ呼び出され、スワップアウトを試みます。ページがスワップデバイスに正常にスワップアウトされた場合は 1 を返します。すべてのディレクトリエントリのページテーブルが // 失敗した場合は、警告メッセージを表示し、0を返します。
2. pg\_table &= 0xfffff000; // page table pointer (address).
3. while (counter-- > 0) {
4. page\_entry++; // Page table entry (initially -1).

// 現在のページテーブルのすべてのアイテムを処理しようとしたが、まだ処理できない場合

1. // ページテーブル項目のインデックス番号が1024以上の場合、前の135〜143行と同じ方法で 次のページディレクトリエントリでセカンダリページテーブルを選択します。
2. if (page\_entry >= 1024) { 148 page\_entry = 0;
3. repeat:
4. dir\_entry++;
5. if (dir\_entry >= 1024)
6. dir\_entry = FIRST\_VM\_PAGE>>10;
7. pg\_table = pg\_dir[dir\_entry]; // content of page directory entry.
8. if (!(pg\_table&1))
9. if ((counter -= 1024) > 0) 156 goto repeat;
10. else
11. break;
12. pg\_table &= 0xfffff000; // get page table ponter.
13. }
14. if (try\_to\_swap\_out(page\_entry + (unsigned long \*) pg\_table))
15. return 1;
16. }
17. printk("Out of swap-memory\n\r");
18. return 0;
19. }

167

1. /\*
2. \* Get physical address of first (actually last :-) free page, and mark it 170 \* used. If no free pages left, return 0.

171 \*/

//// メインメモリ領域の空き物理ページを取得します。

// 利用可能な物理メモリページがない場合は、ページスワップ処理を行い、 // 再度ページの取得を申請し、取得しようとします。

// 入力： %1(ax = 0) ; %2(LOW\_MEM) バイトビットマップで管理されるメモリ開始位置; // %3(cx = PAGING\_PAGES); %4(edi = mem\_map + PAGING\_PAGES - 1)。

// この関数は、新しいページのアドレスを %0(ax = 物理ページの開始アドレス)で返します。

// 上記の%4レジスタは、実際にはメモリバイトビットマップmem\_map[]の最後のバイトを指しています。この関数は、ビットマップの最後から逆方向にすべてのページフラグをスキャンし（総ページ数はPAGING\_PAGES）、ページが空いていればそのページアドレスを返します（ビットマップバイトは0です）。

// 注意！この関数は、単にメインメモリ領域の空き物理ページを指定するだけですが、これは

// は、どのプロセスのアドレス空間にもマッピングされていません。memory.cのput\_page()関数は以下のようになっています。

// プログラムは，指定したページをプロセスのアドレス空間にマッピングするために使用します。もちろん、この関数をカーネルに使用しても、 // カーネルコードとデータ空間（16MB）は物理アドレス空間にピアツーピアでマッピングされるので、 // put\_page()を使ってマッピングする必要はありません。

1. // 174行目では、ローカルレジスタ変数を定義しています。この変数は効率的なアクセスと操作のために // eax レジスタに保存されます。この変数の定義方法は、主にインラインのアセンブリファイルで使用されます。
2. unsigned long get\_free\_page(void)
3. {
4. register unsigned long \_\_res asm("ax");

175

// まず、メモリバイトのビットマップで値が0のバイトを調べて、クリアします。

// 対応する物理メモリページ。結果として得られるページアドレスが、実際の

1. // 物理的なメモリ容量を、もう一度探します。空きページが見つからない場合は、スワップ // 操作を実行してから、もう一度探します。最後に、空き物理ページのアドレスを返します。
2. repeat:
3. // メモリバイトのビットマップを使って空きページを探す mem\_map[].
4. \_\_asm\_\_("std ; repne ; scasb\n\t" // al(0) compared with content of each page (di).
5. "jne 1f\n\t" // jump to label 1 if none of it equals 0.
6. "movb $1,1(%%edi)\n\t" // set byte [1+edi] to 1 of the page.
7. "sall $12,%%ecx\n\t" // pages \* 4K = relative address of the page.
8. "addl %2,%%ecx\n\t" // plus LOW\_MEM to get the absolute page address.
9. // メモリページをゼロにします。
10. "movl %%ecx,%%edx\n\t" // load edx register with page start address.
11. "movl $1024,%%ecx\n\t" // load ecx with counting number 1024.
12. "leal 4092(%%edx),%%edi\n\t" // load edi with page end address (4092 + edx).
13. "rep ; stosl\n\t" // reset (clear) each byte in the page.
14. "movl %%edx,%%eax\n" // load eax with the page start address.
15. "1:"
16. :"=a" (\_\_res)
17. :"0" (0),"i" (LOW\_MEM),"c" (PAGING\_PAGES),
18. "D" (mem\_map+PAGING\_PAGES-1)
19. :"di","cx","dx");
20. if (\_\_res >= HIGH\_MEMORY) // search again if the page is out of main page area.
21. goto repeat;
22. if (!\_\_res && swap\_out()) // do swapping if no free page found and search again.
23. goto repeat;
24. return \_\_res; // return the address of the free page. 197 }

198

// メモリページスワッピングの初期化。

// この関数はまず、デバイスのパーティションに基づいて、デバイスにスワップパーティションがあるかどうかを確認します。

// の配列（ブロックの配列）を取得し、スワップパーティションが有効かどうかをチェックします。その後、ページを取得するために申請

スワップページビットマップ配列 swap\_bitmap[]を格納するメモリの // ために，デバイスのスワップパーティションからスワップ管理ページ // （最初のページ）をスワップビットマップ配列に読み込みます。その後、スワップビットマップ配列の各ビットが有効かどうかを慎重に // 確認し、最後にリターンします。

199 void init\_swapping(void) 200 {。

1. // Blk\_size[]は、メジャー・デバイス番号で指定されたブロック・デバイスのブロック数を示すポインタ配列である。その各項目は、1つのサブデバイスが所有するデータブロックの総数に対応し、各サブデバイスはブロックデバイスの1つのパーティションに対応する。 // システムにスワップデバイス番号が定義されていない場合、コードは戻ります。スワップデバイスにブロックの配列が設定されていない場合、 // 警告メッセージが表示され、戻ります。
2. extern int \*blk\_size[]; // defined in blk\_drv/ll\_rw\_blk.c, line 49.
3. int swap\_size,i,j;

203

1. if (!SWAP\_DEV)
2. return;
3. if (!blk\_size[MAJOR(SWAP\_DEV)]) {
4. printk("Unable to get size of swap device\n\r");
5. return; 209 }
6. // 次に、スワップデバイスのスワップパーティションにあるブロックの総数 'swap\_size' を取得して確認します。それが0であればリターンし、ブロックの総数が100未満であれば、警告 // メッセージを表示してから終了します。
7. swap\_size = blk\_size[MAJOR(SWAP\_DEV)][MINOR(SWAP\_DEV)];
8. if (!swap\_size)
9. return;
10. if (swap\_size < 100) {
11. printk("Swap device too small (%d blocks)\n\r",swap\_size);
12. return; 216 }

// 次に、スワップブロックの合計数を、対応するスワップの合計数に変換します。

1. // ページ(blocks/4)を表します。この値はSWAP\_BITSが表現できるページ数(32768)を超えてはならない。次に，スワップビットマップ配列'swap\_bitmap'を格納するために， // 各ビットが1つのスワップページを表すような空きメモリページを取得する。
2. swap\_size >>= 2;
3. if (swap\_size > SWAP\_BITS)
4. swap\_size = SWAP\_BITS;
5. swap\_bitmap = (char \*) get\_free\_page();
6. if (!swap\_bitmap) {
7. printk("Unable to start swapping: out of memory :-)\n\r"); 223 return;
8. }

// そして、デバイスのスワップパーティションのページ0をswap\_bitmapページに読み込んで、それが

// スワップ領域の管理ページです。その中でも、4086バイト目の先頭には、スワップ

// デバイスの機能文字列「SWAP-SPACE」。特徴文字列が見つからない場合は、有効なスワップデバイスではありません。そこで、警告メッセージを表示し、先ほど取得したメモリページを解放して、 // この関数を終了します。そうでなければ、メモリページ内の機能文字列（10バイト）をクリアします。

1. // マクロread\_swap\_page(nr, buffer)は、ファイルinclude/linux/mm.hの11行目で定義されています。
2. read\_swap\_page(0,swap\_bitmap);
3. if (strncmp("SWAP-SPACE",swap\_bitmap+4086,10)) {
4. printk("Unable to find swap-space signature\n\r");
5. free\_page((long) swap\_bitmap);
6. swap\_bitmap = NULL;
7. return;
8. }
9. memset(swap\_bitmap+4086,0,10);

// 次に、合計32,768ビットを含むリードスワップビットマップを確認します。ビットマップのビットが0であれば、デバイス上の対応するスワップページが使用されている（占有されている）ことを意味します。

// このビットが1の場合、対応するスワップページが利用可能（アイドル）であることを示します。そのため

// デバイスのスワップパーティションでは、最初のページ（ページ0）がスワップ管理に使用され

// はすでに占有されています（ビット0が0）。スワップページ [1 -- swap\_size-1] は利用可能なので、その

// ビットマップの対応するビットは1（アイドル）であるべきです。swap\_size -- SWAP\_BITS] のビット。

// ビットマップ内の // 範囲も、対応するスワップページがないため、0（占有）に初期化されているはずです。以下、ビットマップをチェックする際には、使用不可能な部分と使用可能な部分に基づいて、 // 2段階でチェックを行います。

// まず、利用できないスワップページのビットマップビットをチェックします。これらはすべて0（占有）でなければなりません。

1. // これらのビットのいずれかが1（アイドル）の場合、ビットマップに問題があることを示しています。エラーメッセージが表示され、ビットマップが占有していたページが解放され、関数が終了します。
2. for (i = 0 ; i < SWAP\_BITS ; i++) {
3. if (i == 1)
4. i = swap\_size;
5. if (bit(swap\_bitmap,i)) {
6. printk("Bad swap-space bit-map\n\r");
7. free\_page((long) swap\_bitmap);
8. swap\_bitmap = NULL;
9. return;
10. }
11. }

// 次に、[1～swap\_size-1]の間のすべてのビットが1（アイドル）であるかどうかをチェックし、カウントします。もし

1. // スワップ機能に問題があるということなので，ビットマップが占有していたページを解放し，機能を終了する。 // そうでなければ、スワップデバイスは正常に動作し、スワップページの数とスワップスペースの総バイト数が表示されます。
2. j = 0;
3. for (i = 1 ; i < swap\_size ; i++)
4. if (bit(swap\_bitmap,i))
5. j++;
6. if (!j) {
7. free\_page((long) swap\_bitmap);
8. swap\_bitmap = NULL;
9. return;
10. }
11. printk("Swap device ok: %d pages (%d bytes) swap-space\n\r",j,j\*4096);
12. }

13.5 まとめ

254

## 13.5 Summary

この章では、カーネルがシステム内の物理メモリと仮想メモリをどのように管理し、アクセスするかを説明します。Intel CPUのメモリ割り当て管理機構と、Linuxカーネルによるメモリ空間の分割方法を中心に説明しています。同時に、Copy-on-WriteメカニズムとDemand Loadingメカニズムの原理についても説明しています。最後に、デバイス上のスワップ・パーティションのビットマップ・スワップ・ページの管理と処理のメカニズムを説明しています。

次章では、Linuxカーネルのソースコードに含まれるすべてのヘッダーファイルについて、ほぼすべてのカーネルコードに含まれる重要なデータ構造やマクロの定義を含めて、完全に説明します。

# 13.5 まとめ

# 14 Header Files (include)

プログラムは、関数を使用する前に、まずその関数を宣言する必要があります。使いやすさを考慮して、同じ種類の関数やデータ構造、定数の宣言はヘッダーファイルに入れるのが一般的です。関連する型定義やマクロ定義もすべてヘッダーファイルに含めることができます。プログラムのソースファイルでは、プリプロセッサ指令「#include」を用いて、関連するヘッダーファイルを参照する。

プログラム中に以下のような制御行文を記述すると、その行はファイル「filename」の内容に置き換えられます。

# include

<

filename

>

もちろん、ファイル名「filename」には、>や改行文字、「」「'」「\」「/\*」などの文字は使用できません。コンパイラは、このファイルをあらかじめ設定された場所で検索します。同様に、以下の形式の制御行は、コンパイラがソースプログラムのあるディレクトリの中で、まず'filename'ファイルを検索するようにします。

# include

"

filename

"

ファイルが見つからない場合、コンパイラは上記と同じ検索処理を行います。この形式では、ファイル名「filename」にも、改行文字や ", ', ˶ˆ꒳ˆ˵ ) を含めることはできませんが、「>」文字は使用できます。 一般的なアプリケーションのソースコードでは、開発環境において、ヘッダファイルとライブラリファイルが表裏一体となっています。ライブラリの各関数は、該当するヘッダファイルで宣言する必要があります。アプリケーションの開発環境にあるヘッダーファイル（通常は/usr/include/ディレクトリに置かれます）は、提供するライブラリの関数（例：libc.a）と一体化していると考えることができ、ライブラリの関数に対する指示やインターフェースの記述となります。コンパイラがソースコードのプログラムをオブジェクト・モジュールに変換した後、リンカがプログラムのすべてのオブジェクト・モジュール（使用されているライブラリ・ファイルのモジュールを含む）を結合して、実行可能なプログラムを形成する。

標準的なCライブラリには、約15個の基本的なヘッダファイルが存在する。各ヘッダーファイルは、I/O操作関数や文字処理関数など、特定のクラスの関数の機能記述や構造定義を表している。標準ライブラリとその実装については、プラウガー氏の著書「The Standard C Library」に詳しく書かれている。

本書で紹介するカーネルのソースコードにおいて、関係するヘッダファイルは、カーネルとそのライブラリが提供するサービスの概要と見ることができ、カーネルとその関連プログラムに固有のヘッダファイルとなっています。これらのヘッダーファイルには、主にカーネルが使用するデータ構造、初期化データ、定数、マクロ定義などがすべて記述されており、少量のプログラムコードも含まれています。Linux 0.12カーネルで使用されているヘッダファイルは、いくつかの特殊なヘッダファイル（ブロックデバイスのヘッダファイルblk.hなど）に加えて、カーネルのソースツリーのinclude/ディレクトリに置かれています。したがって，このLinuxカーネルをコンパイルする際には，開発環境で提供されている/usr/include/ディレクトリにあるヘッダファイルを使用する必要はありません。もちろん、tools/build.cというプログラムは別です。なぜなら、このプログラムは、カーネルソースツリーに含まれているものの、カーネルイメージファイルを作成するための単なるユーティリティーまたはアプリケーションであり、以下のようにリンクされることはないからです。

14.1 include/ディレクトリ内のファイル

カーネルコードです。

## カーネルバージョン0.95以降、カーネルをスムーズにコンパイルするためには、カーネルのソースツリーにあるヘッダーファイルを/usr/include/linuxディレクトリにコピーする必要があります。つまり、このバージョンのカーネルからは、Linuxの開発環境で使われているヘッダーファイルがマージされているのです。

## 14.1 Files in the include/ directory

カーネルが使用するヘッダーファイルは，カーネルのソースツリーの include/ ディレクトリに格納されています。このディレクトリ内のファイルはリスト14-1に示されています。ここで注意すべき点は、使いやすさと互換性のために、Linusはカーネルプログラムのヘッダーファイルをコンパイルする際に、標準的なカーネルCのヘッダーファイルに対して同様の命名規則を使用していることです。多くのヘッダーファイルの名前や、ファイルの内容の一部も、基本的には標準Cライブラリのものと同じです。しかし、これらのヘッダーファイルは、やはりカーネル固有のもの、あるいはカーネルと密接に結びついたプログラム固有のものです。Linuxシステムでは、標準ライブラリのヘッダファイルと共存しています。通常、これらのヘッダーファイルは、カーネルのデータ構造や定数を必要とするプログラムが使用するために、標準ライブラリのヘッダーファイルディレクトリのサブディレクトリに配置されます。

また、著作権の問題から、リーナス氏は、著作権制限のある標準Cライブラリのヘッダファイルを置き換えるために、いくつかのヘッダファイルを書き換えようとしました。そのため、これらのカーネルソースに含まれるヘッダファイルは、開発環境のヘッダファイルと一部重複しています。Linuxシステムでは，アプリケーション開発のために，リスト14-1のasm/，linux/，sys/のサブディレクトリにあるカーネルヘッダファイルは，通常，標準Cライブラリのヘッダファイルがあるディレクトリ（/usr/include ）にコピーする必要がありますが，他のファイルは標準ライブラリのヘッダファイルと衝突しないので，直接，標準ライブラリのヘッダファイルのディレクトリに置くか，ここにある3つのサブディレクトリに変更することができます。

asm/ディレクトリは、主に使用しているコンピュータアーキテクチャに密接に関連する関数宣言やデータ構造のヘッダーファイルを格納するために使用されます。例えば、Intel CPUポートIOアセンブリマクロファイルio.h、割り込みディスクリプターセットアセンブリマクロヘッダーファイルsystem.hなどがあります。linux/ ディレクトリには，スケジューラが使用するヘッダファイルsched.h，メモリ管理ヘッダファイルmm.h，端末管理データ構造ファイルtty.hなど，Linuxカーネルプログラムで使用されるいくつかのヘッダファイルが格納されている。 sys/ ディレクトリには，カーネルリソースに関連するいくつかのヘッダファイルが格納されている。sys/ディレクトリには，カーネルリソースに関連するいくつかのヘッダファイルが格納されています。ただし，バージョン0.98からは，カーネルディレクトリツリーの下のsys/ディレクトリにあるヘッダファイルは，すべてlinux/ディレクトリに移動しています。

Linux 0.12カーネルには，asm/サブディレクトリに4個，linux/サブディレクトリに11個，sys/サブディレクトリに8個，合計36個のヘッダファイル(\*.h)が存在する．次節からは，まずinclude/ディレクトリにある13個のヘッダファイルを説明し，その後，各サブディレクトリにあるファイルを順に説明していきます．説明の順番はファイル名でソートされています．

リスト14-1 linux/include/ディレクトリ内のファイル

Filename

Size

Last

Modified Time

Description

asm/

1992

-

01

-

16:46:04

09

linux/

1992

-

01

-

12

19:43:55

sys/

1992

-

01

-

09

16:46:03

a.out.h

6047

bytes

1991

-

09

-

17

15:10:49

const.h

bytes

321

1991

-

09

-

17

15:12:39

ctype.h

1049

bytes

1991

-

11

-

07

17:30:47



|  |  |  |
| --- | --- | --- |
| errno.h | 1364 bytes | 1992-01-03 18:52:20 |
| fcntl.h | 1374 bytes | 1991-09-17 15:12:39 |
| signal.h | 1974 bytes | 1992-01-04 14:54:10 |
| stdarg.h | 780 bytes | 1991-09-17 15:02:23 |
| stddef.h | 285 bytes | 1991-12-28 03:19:05 |
| string.h | 7881 bytes | 1991-09-17 15:04:09 |
| termios.h 5268 bytes | | 1992-01-14 13:53:25 |
| time.h 874 bytes | | 1992-01-04 14:58:17 |
| unistd.h 7300 bytes | | 1992-01-13 22:48:52 |
| utime.h 225 bytes | | 1991-09-17 15:03:38 |

## 14.2 a.out.h

### 14.2.1 Function

Linux カーネルでは、a.out 形式で読み込まれる実行ファイル構造を定義するために a.out.h ファイルが使用されており、主に実行ファイルローダプログラム fs/exec.c で使用されています。このファイルは、標準 C ライブラリの一部ではなく、カーネル固有のヘッダファイルです。しかし、標準ライブラリのヘッダファイル名とは矛盾しないため、一般的にはLinuxシステムの/usr/include/ディレクトリに配置して、関連する内容のプログラムで使用することができるようになっている。このヘッダーファイルは、オブジェクトファイルのa.out（Assembley out）フォーマットを定義しています。このオブジェクトファイル形式は、Linux 0.12システムで使用される.oファイルと実行ファイルに使用されます。

a.out.hファイルには、3つのデータ構造定義と、それに関連するマクロ定義が含まれているので、このファイルは3つの部分に分けられます。

 Lines 1-108 give and describe the object header structure and associated macro definitions;  Lines 109-185 are definitions and descriptions of the structure of symbol entries;  Lines 186-217 define and describe the structure of the relocation table entry.

ファイルの内容が多いため、3つのデータ構造と関連するマクロ定義の詳細な説明をプログラムリストの後に配置しています。ここでは、a.out形式のファイルにあるexec構造を簡単に紹介します。

a.outの実行ファイルは、先頭のexecヘッダー部と、それに続くコード、データなどの部分で構成されています。実行ファイルのヘッダ部には、主にexec構造体があり、マジックナンバーフィールド、コードとデータの長さ、シンボルテーブルの長さ、コード実行開始位置などの情報が含まれている。カーネルはこの情報を使って実行ファイルをメモリにロードして実行し、リンカー（ld）はこれらのパラメータを使っていくつかのモジュールファイルを1つの実行ファイルにまとめます。これがオブジェクトファイルの唯一の必要な構成要素である。

0.96カーネルから、LinuxシステムはGNUのヘッダファイルa.out.hを直接使用するようになりました。そのため，Linux 0.9xでコンパイルされたプログラムは，Linux 0.1xシステムでは実行できません。以下では、2つのa.outヘッダーファイルの違いを分析し、0.9xでコンパイルされた実行ファイルのうち、ダイナミックリンクライブラリを使用しないものを0.1xでも実行できるようにする方法を説明します。

Linux 0.12で使用されているa.out.hファイルと、同名のGNUファイルの主な違いは、exec構造体の最初のフィールドa\_magicです。GNUのファイルのフィールド名はa\_infoで、このフィールドはさらに3つのサブドメインに分かれています。Flags」「Machine Type」「Magic Number」の3つのサブドメインに分かれています。同時に、図14-1に示すように、マシンタイプフィールドには、対応するマクロN\_MACHTYPEとN\_FLAGSが定義されている。

a\_magic

field exec struct, 4 bytes

Flags(00)

Machine Type (100 o

r 0x64)

Magic Number (0413 or 0x10b)

図14-1 exec構造体の最初のフィールド a\_magic(a\_info)

Linux 0.9x系では、スタティック・ライブラリーとリンクされた実行ファイルの場合、図中の（）内の値が各フィールドのデフォルト値となります。このバイナリ実行ファイルの先頭の4バイトは

0

x0b, 0x01, 0x64, 0x

00

このカーネルのa.outヘッダーファイルでは、マジックナンバーフィールドのみを定義しています。したがって、Linux 0.1xシステム上のa.out形式のバイナリ実行ファイルの最初の4バイトは次のようになります。

0

x0b, 0x01, 0x00, 0x00

GNUのa.out形式の実行ファイルと、Linux 0.1xシステムでコンパイルした実行ファイルの違いは、マシンタイプフィールドのみであることがわかります。したがって、Linux 0.9x の a.out 形式の実行ファイルのマシンタイプ・フィールド（3 バイト目）をクリアして、0.1x システムで実行することができます。移植された実行ファイルによって呼び出されるシステムコールが、0.1xシステムですでに実装されていればよいのです。著者は、Linux 0.1xのルート・ファイル・システムにある多くのコマンドの再構築を始める際に、この方法を使用しました。その他の点では、GNUのa.out.hヘッダーファイルは、ここでのa.out.hと変わりません。

### 14.2.2 Code annotation

プログラム14-1 linux/include/a.out.h

1. #ifndef \_A\_OUT\_H
2. #define \_A\_OUT\_H

3

4 #define \_\_GNU\_EXEC\_MACROS\_\_

5

// 6～108行目はドキュメントの最初の部分で、主にオブジェクトファイルの実行構造と関連する操作のマクロを定義しています。

1. // 以下はオブジェクトファイルのヘッダ構造で，プログラムの後に詳細な説明があります。
2. struct exec {
3. unsigned long a\_magic; /\* Use macros N\_MAGIC, etc for access \*/
4. unsigned a\_text; /\* length of text, in bytes \*/
5. unsigned a\_data; /\* length of data, in bytes \*/
6. unsigned a\_bss; /\* length of uninitialized data area for file, in bytes \*/
7. unsigned a\_syms; /\* length of symbol table data in file, in bytes \*/
8. unsigned a\_entry; /\* start address \*/
9. unsigned a\_trsize; /\* length of relocation info for text, in bytes \*/ 14 unsigned a\_drsize; /\* length of relocation info for data, in bytes \*/

15 };

16

1. // 上記のexec構造でマジックナンバーを取得するためのマクロ定義です。
2. #ifndef N\_MAGIC
3. #define N\_MAGIC(exec) ((exec).a\_magic)
4. #endif

20

1. #ifndef OMAGIC
2. /\* Code indicating object file or impure executable. \*/

// 歴史的に、PDP-11コンピュータでは、マジックナンバー（魔法の数字）は8進数の0407 // （0x107）でした。歴史的に、PDP-11コンピュータでは、マジックナンバー（魔法の数字）は8進数の0407（0x107）であり、実行ファイルのヘッダ構造の先頭にあった。

// 元々はPDP-11のジャンプ命令で、先頭にジャンプすることを示していました。

次の7つの単語の後にコードの//を入力します。このようにして，ローダは直接先頭にジャンプすることができます。

1. 実行ファイルをメモリに入れた後の命令の//。この方法を使うプログラムは // ありませんが、この8進数はファイルタイプを識別するためのフラグ（マジックナンバー）として // 保持されています。OMAGIC」は「Old Magic」と考えることができます。
2. #define OMAGIC 0407
3. /\* Code indicating pure executable. \*/
4. // 1975年以降に使用されたNMAGIC（New Magic）のこと。仮想記憶のメカニズムに関わるものです。
5. #define NMAGIC 0410 // 0410 == 0x108
6. /\* Code indicating demand-paged executable. \*/
7. // このタイプのヘッダー構造は、ファイルの先頭に1KBのスペースを占有します。
8. #define ZMAGIC 0413 // 0413 == 0x10b
9. #endif /\* not OMAGIC \*/
10. // There is also a QMAGIC, which is used to save disk capacity and store the header structure // and code of the file on the disk in a compact manner.
11. // 以下のマクロ(Bad Magic)は、マジックナンバー // フィールドの正しさを判定するために使用されます。マジックナンバーが認識できない場合は、trueを返します。
12. #ifndef N\_BADMAG
13. #define N\_BADMAG(x) \
14. (N\_MAGIC(x) != OMAGIC && N\_MAGIC(x) != NMAGIC \
15. && N\_MAGIC(x) != ZMAGIC)
16. #endif

35

1. #define \_N\_BADMAG(x) \
2. (N\_MAGIC(x) != OMAGIC && N\_MAGIC(x) != NMAGIC \
3. && N\_MAGIC(x) != ZMAGIC)

39

// ヘッダー構造の終わりから1KBの位置までの残りの長さを与えるマクロ定義です。

40 #define \_N\_HDROFF(x) (SEGMENT\_SIZE - sizeof (struct exec))

41

// 以下のマクロは、.o // モジュールファイルや実行ファイルを含む、オブジェクトファイルのコンテンツを操作するために使用されます。

// コード部分の開始オフセットです。

1. // ファイルが ZMAGIC タイプ、つまり実行ファイルの場合、コード部分は実行ファイルから // 1024 バイトのオフセットで始まります。そうでない場合、コード部分は実行ヘッダー構造の終わり（32 バイト）で始まります。
2. #ifndef N\_TXTOFF
3. #define N\_TXTOFF(x) \
4. (N\_MAGIC(x) == ZMAGIC ? \_N\_HDROFF((x)) + sizeof (struct exec) : sizeof (struct exec))
5. #endif

46

1. // コードセクションの最後から始まるデータパートの開始オフセットです。
2. #ifndef N\_DATOFF
3. #define N\_DATOFF(x) (N\_TXTOFF(x) + (x).a\_text)
4. #endif

50

1. // データセクションの最後から始まる、コード再配置情報のオフセットです。
2. #ifndef N\_TRELOFF
3. #define N\_TRELOFF(x) (N\_DATOFF(x) + (x).a\_data)
4. #endif

54

1. // データ再配置情報のオフセットで、コード再配置情報の最後から始まる。
2. #ifndef N\_DRELOFF
3. #define N\_DRELOFF(x) (N\_TRELOFF(x) + (x).a\_trsize)
4. #endif

58

1. // シンボルテーブルのオフセットで、データセグメントの再配置テーブルの最後からスタートします。
2. #ifndef N\_SYMOFF
3. #define N\_SYMOFF(x) (N\_DRELOFF(x) + (x).a\_drsize)
4. #endif

62

1. // シンボルテーブルの後の、文字列情報のオフセットです。
2. #ifndef N\_STROFF
3. #define N\_STROFF(x) (N\_SYMOFF(x) + (x).a\_syms)
4. #endif

66

1. // 以下は、実行ファイルを論理空間にロードするロケーション操作です。
2. /\* Address of text segment in memory after it is loaded. \*/
3. #ifndef N\_TXTADDR
4. #define N\_TXTADDR(x) 0 // The code segment begings at address 0.
5. #endif

71

1. /\* Address of data segment in memory after it is loaded.
2. Note that it is up to you to define SEGMENT\_SIZE
3. on machines not listed here. \*/
4. #if defined(vax) || defined(hp300) || defined(pyr)
5. #define SEGMENT\_SIZE PAGE\_SIZE
6. #endif
7. #ifdef hp300
8. #define PAGE\_SIZE 4096
9. #endif
10. #ifdef sony
11. #define SEGMENT\_SIZE 0x2000
12. #endif /\* Sony. \*/
13. #ifdef is68k
14. #define SEGMENT\_SIZE 0x20000

#endif

#if defined(m68k) && defined(PORTAR)

#define PAGE\_SIZE 0x400

#define SEGMENT\_SIZE PAGE\_SIZE

#endif

1. // ここでは、カーネルがメモリページを4KBと定義し、セグメントサイズを1KBと定義しているため、 // 上記の定義は使用されません。
2. #define PAGE\_SIZE 4096
3. #define SEGMENT\_SIZE 1024

94

// セグメントで定義されたサイズ（キャリーを考慮）。

95 #define \_N\_SEGMENT\_ROUND(x) (((x) + SEGMENT\_SIZE - 1) & ~(SEGMENT\_SIZE - 1)) 96

// コードセグメントのエンドアドレス。

97 #define \_N\_TXTENDADDR(x) (N\_TXTADDR(x)+(x).a\_text)

98

// データセグメントのスタートアドレス。

1. // OMAGICタイプのファイルの場合、データセグメントはコードセグメントの直後に位置します。 // ZMAGICタイプのファイルの場合、データセグメントのアドレスは、コードセグメントの後のセグメントバウンダリから始まります。
2. #ifndef N\_DATADDR
3. #define N\_DATADDR(x) \
4. (N\_MAGIC(x)==OMAGIC? (\_N\_TXTENDADDR(x)) \
5. : (\_N\_SEGMENT\_ROUND (\_N\_TXTENDADDR(x))))
6. #endif

104

1. /\* Address of bss segment in memory after it is loaded. \*/
2. // 初期化されていないデータセグメントbbsが配置され、データセグメントの後に続きます。
3. #ifndef N\_BSSADDR
4. #define N\_BSSADDR(x) (N\_DATADDR(x) + (x).a\_data)
5. #endif 109

// 110～185行目がパート2です。オブジェクトファイルのシンボルテーブルエントリを説明し、 // 関連する操作マクロを定義しています。プログラムリストの後にある詳細な説明を参照してください。

1. // a.outオブジェクトファイル内のシンボルテーブルエントリー（レコード）構造です。
2. #ifndef N\_NLIST\_DECLARED
3. struct nlist {
4. union {
5. char \*n\_name;
6. struct nlist \*n\_next;
7. long n\_strx;
8. } n\_un;
9. unsigned char n\_type; // The byte is divided into 3 fields, and
10. char n\_other; // lines 146-154 are the mask code for each field.
11. short n\_desc;
12. unsigned long n\_value;
13. };
14. #endif

123

1. // nlist構造体のn\_typeフィールドの定数は、以下のように定義されています。
2. #ifndef N\_UNDF
3. #define N\_UNDF 0
4. #endif
5. #ifndef N\_ABS
6. #define N\_ABS 2
7. #endif
8. #ifndef N\_TEXT
9. #define N\_TEXT 4
10. #endif
11. #ifndef N\_DATA
12. #define N\_DATA 6
13. #endif
14. #ifndef N\_BSS
15. #define N\_BSS 8
16. #endif
17. #ifndef N\_COMM
18. #define N\_COMM 18
19. #endif
20. #ifndef N\_FN
21. #define N\_FN 15
22. #endif 145
23. // 以下の3つの定数は、nlist構造体のn\_typeフィールドのマスク（8進数）です。
24. #ifndef N\_EXT
25. #define N\_EXT 1 // 0x01 (0b0000,0001) symbol is external (global) ?.
26. #endif
27. #ifndef N\_TYPE
28. #define N\_TYPE 036 // 0x1e (0b0001,1110) The type bits of the symbol.
29. #endif
30. #ifndef N\_STAB // STAB -- Symbol table types.
31. #define N\_STAB 0340 // 0xe0 (0b1110,0000) These bits are used for symbol debugger.
32. #endif

155

1. /\* The following type indicates the definition of a symbol as being
2. an indirect reference to another symbol. The other symbol
3. appears as an undefined reference, immediately following this symbol.

159

1. 160 インダイレクトは非対称です。 他方のシンボルの値は、161 間接的なシンボルに対する要求を満たすために使用されますが、その逆はありません。
2. If the other symbol does not have a definition, libraries will
3. be searched to find a definition. \*/
4. #define N\_INDR 0xa

165

1. /\* The following symbols refer to set elements.
2. All the N\_SET[ATDB] symbols with the same name form one set.
3. Space is allocated for the set in the text section, and each set
4. element's value is stored into one word of the space.
5. The first word of the space is the length of the set (number of elements).

171

1. The address of the set is made into an N\_SETV symbol
2. whose name is the same as the name of the set.
3. This symbol acts like a N\_DATA global symbol
4. in that it can satisfy undefined external references. \*/

176

1. /\* These appear as input to LD, in a .o file. \*/
2. #define N\_SETA 0x14 /\* Absolute set element symbol \*/
3. #define N\_SETT 0x16 /\* Text set element symbol \*/
4. #define N\_SETD 0x18 /\* Data set element symbol \*/
5. #define N\_SETB 0x1A /\* Bss set element symbol \*/

182

1. /\* This is output from LD. \*/
2. #define N\_SETV 0x1C /\* Pointer to set vector in data area. \*/ 185

186 #ifndef N\_RELOCATION\_INFO\_DECLARED

187

1. /\* This structure describes a single relocation to be performed.
2. The text-relocation section of the file is a vector of these structures, 190 all of which apply to the text section.

191 同様に、データ・リロケーション・セクションにも適用されます。 \*/ 192

1. // a.outのオブジェクトファイル内のコードとデータの再配置情報構造。
2. struct relocation\_info
3. {
4. /\* Address (within segment) to be relocated. \*/
5. int r\_address;
6. /\* The meaning of r\_symbolnum depends on r\_extern. \*/
7. unsigned int r\_symbolnum:24;
8. /\* Nonzero means value is a pc-relative offset
9. and it should be relocated for changes in its own address
10. as well as for changes in the symbol or section specified. \*/
11. unsigned int r\_pcrel:1;
12. /\* Length (as exponent of 2) of the field to be relocated.
13. Thus, a value of 2 indicates 1<<2 bytes. \*/
14. unsigned int r\_length:2;
15. /\* 1 => relocate with value of symbol.
16. r\_symbolnum is the index of the symbol
17. in file's the symbol table.
18. 0 => relocate with the address of a segment.
19. r\_symbolnum is N\_TEXT, N\_DATA, N\_BSS or N\_ABS
20. (the N\_EXT bit may be set also, but signifies nothing). \*/
21. unsigned int r\_extern:1;
22. /\* Four bits that aren't used, but when writing an object file
23. it is desirable to clear them. \*/
24. unsigned int r\_pad:4;
25. };
26. #endif /\* no N\_RELOCATION\_INFO\_DECLARED. \*/

218

219

220 #endif /\* \_\_A\_OUT\_GNU\_H\_\_ \*/

221

### 14.2.3 Information

#### 14.2.3.1 a.out executable file format

Linuxカーネルバージョン0.12では、a.out(Assembler output)という実行ファイルやオブジェクトファイルの形式のみをサポートしています。このフォーマットは次第に使われなくなり、ELF（Executable and Link Format）フォーマットが本格的に使われるようになっていますが、そのシンプルさゆえに学習用の教材として適しています。ここでは、a.out形式について総合的に紹介します。

1. ヘッダーファイルa.out.hでは、3つのデータ構造といくつかのマクロが宣言されており、これらのデータ構造は、システム上のオブジェクトファイルの構造を記述しています。Linux 0.12システムでは、リンカーが生成したコンパイル済みのオブジェクト・モジュール・ファイル（モジュール・ファイルと呼ぶ）とバイナリ実行ファイルはa.out形式である。ここでは，これらを総称してオブジェクトファイルと呼ぶことにする．オブジェクトファイルは、最大で7つのパート（セクション）から構成される。順番に並べると
2. **exec header --** This section contains some parameters (exec structure) that the kernel uses to load the executable file into memory and execute it, and the linker (ld) uses these parameters to combine some of the module files into one executable file. This is the only necessary component of the object file.
3. **text segment --** Contains the instruction code (text) and related data that is loaded into memory when the program is executed. It can be loaded in read-only form.
4. **data segment --** This section contains data that has already been initialized and is always loaded into readable and writable memory.
5. **text relocations --** This section contains records data for use by the linker. Used to locate and update a pointer or address in a code segment when combining object module files.
6. **data relocation --** Similar to the role of the code relocation section, but for the relocation of pointers in the data segment.
7. **symbol table --** This section also contains records data for use by the linker to cross-reference named variables and functions (symbols) between binary object module files.
8. **string table --** This part contains character strings corresponding to the symbol names.

各オブジェクトやバイナリファイルは、以下のような形式の実行データ構造（exec構造）で始まります。

struct exec { unsigned long a\_magic // N\_MAGIC などのアクセス用マクロを使用する。 unsigned a\_text // コードの長さ(バイト) unsigned a\_data // データの長さ(バイト) unsigned a\_bss // ファイルの未初期化データ領域の長さ(バイト) unsigned a\_syms // ファイルのシンボルテーブルデータの長さ(バイト) unsigned a\_entry // 実行開始アドレス。 符号なし a\_syms // ファイル内のシンボルテーブルデータの長さ(バイト) 符号なし a\_entry // 実行開始アドレス 符号なし a\_trsize // テキストの再配置情報の長さ(バイト) 符号なし a\_drsize // データの再配置情報の長さ(バイト)

};

* 各フィールドの機能は以下の通りです。
* a\_magic -- This field contains three subfields, the flag field, the machine type id field, and the magic number field, as shown in Figure 14-1. However, for the Linux 0.12 system, its object file only uses the magic number subfield and is accessed using the macro N\_MAGIC(), which uniquely determines the difference between the binary executable file and other loaded files. This subfield must contain one of the following values:
  + OMAGIC -- Indicates that the text and data segments are immediately following the execution header and are stored consecutively. The kernel loads both text and data segments into readable and writable memory. The magic number of the object file compiled by the compiler is OMAGIC (octal 0407).
  + NMAGIC -- Like OMAGIC, text and data segments follow the execution header and are stored continuously. However, the kernel loads the text into read-only memory and loads the data segment into writable memory at the next page boundary after the text.
  + ZMAGIC -- The kernel loads separate pages from the binary executable when necessary. The execution header, text segment, and data segment are all processed by the linker into blocks of multiple page sizes. The text page loaded by the kernel is read-only, and the page of the data segment is writable. The magic number of the executable file generated by the linker is ZMAGIC (0413, ie 0x10b).
* a\_text -- This field contains the size of the text segment, the number of bytes.
* a\_data -- This field contains the size of the data segment, the number of bytes.
* a\_bss -- Contains the length of the 'bss segment' that the kernel uses to set the initial break(brk) after the data segment. When the kernel is loading the program, this writable memory appears to be behind the data segment and is initially all zeros.
* a\_syms -- Contains the size in bytes of the symbol table section.
* a\_entry -- The memory address of the program execution start point after the kernel has loaded the executable file into memory.
* a\_trsize -- This field contains the size of the text relocation table, in bytes.
* a\_drsize -- This field contains the size of the data relocation table, in bytes.

|  |  |
| --- | --- |
| a.out.hヘッダーファイルにはいくつかのマクロが定義されています。これらのマクロはexec構造を利用して整合性をテストしたり、実行ファイル内の様々なセクションオフセットを見つけたりします。これらのマクロは  N\_BADMAG(exec) | Returns a non-zero value if the a\_magic field cannot be recognized. |
| N\_TXTOFF(exec) | The starting byte offset of the code segment. |
| N\_DATOFF(exec) | The starting byte offset of the data segment. |
| N\_DRELOFF(exec) | The starting byte offset of the data relocation table. |
| N\_TRELOFF(exec) | The starting byte offset of the text relocation table. |
| N\_SYMOFF(exec) | The starting byte offset of the symbol table. |
| N\_STROFF(exec) | The starting byte offset of the string table. |

リロケーションレコードは標準的なフォーマットを持っており、以下に示すように、リロケーション情報（relocation\_info）構造体を用いて記述される。

/ struct relocation\_info

{ int r\_address; // セグメント内で再配置する必要のあるアドレスです。

unsigned int r\_symbolnum:24; // 意味は、r\_externに関連する。

// シンボルテーブルのシンボルまたはセグメントを指定します。

unsigned int r\_pcrel:1; // PC関連フラグ。 unsigned int r\_length:2; // 再配置されるフィールドの長さ（2の指数）。

unsigned int r\_pad:4; // 4ビットは使用しませんが、クリアしておいた方が良いでしょう。

};

* 構造体の各フィールドの意味は以下の通りです。
* r\_address -- This field contains the byte offset of the pointer that the linker needs to process (edit). The offset of the text relocation is counted from the beginning of the text segment, and the offset of the data relocation is calculated from the beginning of the data segment. The linker adds the value already stored at the offset to the new value calculated using the relocation record.
* r\_symbolnum -- This field contains the ordinal number (not the byte offset) of a symbol structure in the symbol table. After the linker calculates the absolute address of the symbol, it adds the address to the pointer being relocated. (If the r\_extern bit is 0, then the situation is different, see below.)
* r\_pcrel -- If this bit is set, the linker considers that a pointer is being updated, which uses the pc-related addressing mode and is part of the machine code instruction. When the running program uses this relocated pointer, the address of the pointer is implicitly added to the pointer.
* r\_length -- This field contains the power of 2 of the length of the pointer: 0 means 1 byte long, 1 means 2 bytes long, 2 means 4 bytes long.
* r\_extern -- If set, it indicates that the relocation requires an external reference; the linker must use a symbol address to update the pointer. When the bit is 0, the relocation is "local"; the linker updates the pointer to reflect the changes in the load addresses of the various segments, rather than reflecting changes in the value of a symbol. In this case, the contents of the r\_symbolnum field are an n\_type value; such field tells the linker what segment the relocated pointer points into.
* r\_pad -- These 4 bits are not used in Linux systems. It is best to set all 0 when writing a object file.

シンボルは名前とアドレスを対応させます（より一般的には文字列と値を対応させます）。リンカがアドレスを調整するため、絶対的なアドレス値が割り当てられるまでは、シンボルの名前を使ってアドレスを示す必要があります。シンボルは、シンボルテーブルの固定長のレコードと、文字列テーブルの可変長の名前で構成されています。シンボルテーブルは、以下のようなnlist構造体の配列です。

/ struct nlist { union { char \*n\_name; struct nlist \*n\_next; long n\_strx;

} n\_un; unsigned char n\_type; // 3つのフィールドに分かれていて、146～154行目がそのマスクです。

char n\_other; short n\_desc; unsigned long n\_value;

};

* 各フィールドの意味は
* n\_un.n\_strx -- Contains the byte offset of the symbol name in the string table. When a program accesses a symbol table using the nlist() function, the field is replaced with the n\_un.n\_name field, which is a pointer to a string in memory.
* n\_type -- Used by the linker to determine how to update the value of the symbol. The 8-bit wide n\_type field can be divided into three subfields using the bitmasks code starting at line 146--154, as shown in Figure 14-2. For symbols of N\_EXT type location bits, the linker treats them as "external" symbols and allows other binary object files to reference them. The N\_TYPE mask is used to select the bits of interest to the linker:
  + N\_UNDF -- An undefined symbol. The linker must locate an external symbol with the same name in another binary object file to determine the absolute data value of the symbol. In special cases, if the n\_type field is non-zero and no binary file defines this symbol, the linker resolves the symbol into an address in the BSS segment, reserving bytes of length equal to n\_value. If the symbol is not defined in more than one binary object file and the binary object files do not match their length values, the linker will select the longest value found across all binary object files.
  + N\_ABS -- An absolute symbol. The linker does not update an absolute symbol.
  + N\_TEXT -- A text (code) symbol. The value of this symbol is the text address, and the linker updates its value when it merges the binary object files.
  + N\_DATA -- A data symbol; similar to N\_TEXT, but for data addresses. The value of the corresponding text and data symbol is not the offset of the file but the address; In order to find the offset of the file, it is necessary to determine the address at which the relevant section starts loading and subtract it, and then add the offset of the section.
  + N\_BSS -- A BSS symbol; similar to a text or data symbol, but without a corresponding offset in the binary object file.
  + N\_FN -- A file name symbol. When merging a binary object file, the linker inserts the symbol before the symbol in the binary file. The name of the symbol is the file name given to the linker, and its value is the address of the first text segment in the binary file. File name symbols are not required for linking and loading, but are very useful for debugging programs.
  + N\_STAB -- The mask code is used to select the bits of interest to the symbolic debugger (eg gdb); its value is specified in stab().
* n\_other -- This field provides symbol independence information about the symbol relocation operation according to the segment determined by n\_type. Currently, the lowest 4 bits of the n\_other field contain one of two values: AUX\_FUNC and AUX\_OBJECT. AUX\_FUNC associates symbols with callable functions, and AUX\_OBJECT associates symbols with data, regardless of whether they are in text segments or data segments. This field is mainly used by the linker ld for the creation of dynamic executable programs.
* n\_desc -- Reserved for use by the debugger; the linker does not process it. Different debuggers use this field for different purposes.
* n\_value -- Contains the value of the symbol. For text, data, and BSS symbols, this is an address; for other symbols (such as debugger symbols), the value can be arbitrary.

文字列テーブルは、長さがunsigned longのシンボル文字列と、それに続くnullで構成されています。長さはテーブル全体のバイトサイズを表すので、32ビットマシンでの最小値（つまり、最初の文字列のオフセット）は常に4です。

Bit

7

6

5

4

3

2

1

0

N\_

STAB

N\_

TYPE

N\_EXT

図 14-2 シンボルタイプフィールド n\_type

## 14.3 const.h

## 14.3 const.h

### 14.3.1 Function

### const.hファイルは、ファイルi-nodeのファイルモードとタイプフィールドi\_modeで使用されるいくつかのフラグ定数を定義しています。

### 14.3.2 Code annotation

プログラム 14-2 linux/include/const.h

1. #ifndef \_CONST\_H
2. #define \_CONST\_H

3

4 #define BUFFER\_END 0x200000 // バッファが使用するメモリエンド（使用しない）。

5

1. // i-node構造体のi\_modeフィールドの各フラグビット。
2. #define I\_TYPE 0170000 // Indicate the i-node type (type mask).
3. #define I\_DIRECTORY 0040000 // Is a directory file.
4. #define I\_REGULAR 0100000 // Is a regular file, not a dir or a special file.
5. #define I\_BLOCK\_SPECIAL 0060000 // Is a block device special file.
6. #define I\_CHAR\_SPECIAL 0020000 // Is a character device special file.
7. #define I\_NAMED\_PIPE 0010000 // Is a named pipe node.
8. #define I\_SET\_UID\_BIT 0004000 // Set efficient user ID type at execution time.
9. #define I\_SET\_GID\_BIT 0002000 // Set efficient group ID type at execution time.

14

15 #endif

16

## 14.4 ctype.h

### 14.4.1 Function

ctype.hファイルは、文字の検査・処理を行うためのヘッダファイルで、標準Cライブラリのヘッダファイルの一つです。文字の種類をチェックしたり変換したりするためのマクロを定義しています。文字の型のチェックや変換のためのマクロが定義されています。例えば，文字cが数字であるか(isdigit(c))，空間であるか(isspace(c))を判定する。チェック処理の際には、ASCIIテーブル内のすべての文字のプロパティとタイプを定義した配列またはテーブル（lib/ctype.cで定義）が使用されます。マクロを使用する場合は、文字コードをテーブル \_ctype[] のインデックス値とし、テーブルからバイトを取得することで、該当するビットを取得します。

また、2つのアンダースコアで始まるマクロ名や、アンダースコアの後に大文字が続くマクロ名は、通常、\_\_abcや\_SPのように、ヘッダーライター用に予約されています。

### 14.4.2 Code annotation

プログラム14-3 linux/include/ctype.h

14.4 ctype.h

1. #ifndef \_CTYPE\_H
2. #define \_CTYPE\_H

3

1. #define \_U 0x01 /\* upper \*/ // used for uppercase characters [A-Z].
2. #define \_L 0x02 /\* lower \*/ // used for lowercase characters [a-z].
3. #define \_D 0x04 /\* digit \*/ // used for digitals [0-9].
4. #define \_C 0x08 /\* cntrl \*/ // used for control characters.
5. #define \_P 0x10 /\* punct \*/ // used for punctuation characters.
6. #define \_S 0x20 /\* white space (space/lf/tab) \*/
7. #define \_X 0x40 /\* hex digit \*/ // used for hexadecimal digits.
8. #define \_SP 0x80 /\* hard space (0x20) \*/ // used for the space character (0x20).

12

// 各キャラクターの属性を定義した、キャラクター属性の配列（テーブル）です。

1. // もう一つは、一時的な文字変数です。これらはすべてlib/ctype.cファイルで定義されています。
2. extern unsigned char \_ctype[];
3. extern char \_ctmp;

15

1. // 以下に、文字の種類を決めるマクロを紹介します。
2. #define isalnum(c) ((\_ctype+1)[c]&(\_U|\_L|\_D)) // is a character or digital.
3. #define isalpha(c) ((\_ctype+1)[c]&(\_U|\_L)) // is a character.
4. #define iscntrl(c) ((\_ctype+1)[c]&(\_C)) // is a control character. 19 #define isdigit(c) ((\_ctype+1)[c]&(\_D)) // is a digital.
5. #define isgraph(c) ((\_ctype+1)[c]&(\_P|\_U|\_L|\_D)) // is a graphic character.
6. #define islower(c) ((\_ctype+1)[c]&(\_L)) // is a lowercase character.
7. #define isprint(c) ((\_ctype+1)[c]&(\_P|\_U|\_L|\_D|\_SP)) // is a printable character.
8. #define ispunct(c) ((\_ctype+1)[c]&(\_P)) // is a punctuation mark.
9. #define isspace(c) ((\_ctype+1)[c]&(\_S)) // is a space, \f, \n, \r, \t, \v.
10. #define isupper(c) ((\_ctype+1)[c]&(\_U)) // is an uppercase character.
11. #define isxdigit(c) ((\_ctype+1)[c]&(\_D|\_X)) // is a hexadecimal number.

27

// 以下の2つのマクロ定義では、マクロパラメータはプレフィックス（符号なし）なので、'c'

//が括弧で囲まれているはずなので、(c)であることがわかります。の複雑な表現である可能性があるため、'c'は

1. // プログラムです。例えば、引数が a + b の場合、括弧を付けないと次のようになります。マクロ定義では (unsigned) // a + b となり、明らかに間違っています。括弧をつけると、(unsigned) (a + b) と正しく // 表現できます。
2. #define isascii(c) (((unsigned) c)<=0x7f) // is an ASCII character.
3. #define toascii(c) (((unsigned) c)&0x7f) // convert to ASCII character.

30

// 以下の2つのマクロ定義で一時変数「\_ctmp」を使用する理由は

// は、マクロのパラメータは、マクロ定義の中で一度しか使用できません。しかし、これは

// 2つ以上のスレッドがこのパブリックな一時変数を同時に使用する可能性があるため、 // マルチスレッドでは安全ではありません。そこで，カーネル 2.2.x からは，これらの 2 つのマクロの定義を変更して， // 2 つの関数を使用するようにしました。

31 #define tolower(c) (\_ctmp=c,isupper(\_ctmp)?\_ctmp-('A'-'a'):\_ctmp) // 小文字の文字にします。 32 #define toupper(c) (\_ctmp=c,islower(\_ctmp)?\_ctmp-('a'-'A'):\_ctmp) // 大文字の文字にします。

33

34 #endif

35

## 14.5 errno.h

## 14.5 errno.h

### 14.5.1 Function

UNIXの型システムや標準C言語には「errno」という変数があります。この変数がC言語の標準に必要なのかどうか、C言語の標準化団体では様々な議論が交わされました（X3J11）。しかし、議論の結果、「errno」は削除されず、代わりに「errno.h」というヘッダーファイルが作成されました。なぜなら、標準化団体は、各ライブラリ関数やデータオブジェクトは、対応する標準ヘッダーファイルで宣言されることを望んでいるからです。

議論の主な理由は、カーネル内の各システムコールにおいて、戻り値がシステムコールの結果である場合、エラーを報告することが困難であることです。各関数に真偽の指示値を返させて、その結果の値が別々に返ってくると、システムコールの結果を簡単に知ることができません。一つの解決策は、この二つの方法を組み合わせることです。特定のシステムコールに対して、有効な結果値の範囲とは異なるエラーの戻り値を指定することができます。例えば、ポインタはnull値を取ることができ、pidの場合は-1の値を返すことができます。その他多くの場合、結果の値と矛盾しない限り、エラー値を示すのに'-1'を使用することができます。しかし、標準Cライブラリ関数の戻り値では、エラーが発生したかどうかしかわからず、エラーの種類は他の場所から知る必要があるため、変数'errno'を使用します。

標準Cライブラリの設計機構と互換性を持たせるために、Linuxカーネルのライブラリファイルもこの処理方法を採用しています。そのため，標準Cライブラリのこのヘッダファイルも借用しています。例として，lib/open.cのプログラムとunistd.hのsystem-callマクロの定義を参照してください。返された'-1'の値からプログラムはエラーを知っているが，具体的なエラー番号を知りたい場合には，'errno'の値を読み取ることで，最後に発生したエラーのエラー番号を知ることができる場合があります。

### 14.5.2 Code annotation

プログラム 14-4 linux/include/errno.h

1. #ifndef \_ERRNO\_H
2. #define \_ERRNO\_H

3

1. /\*
2. \* ok, as I hadn't got any other source of information about
3. \* possible error numbers, I was forced to use the same numbers
4. \* as minix.
5. \* Hopefully these are posix or something. I wouldn't know (and posix 9 \* isn't telling me - they want $$$ for their f\*\*\*ing standard).
6. \*
7. \* We don't use the \_SIGN cludge of minix, so kernel returns must 12 \* see to the sign by themselves.
8. \*
9. \* NOTE! Remember to change strerror() if you change this file!
10. \*/

16

// システムコールや多くのライブラリ関数は、操作の失敗やエラーを示すために // 特別な値を返します。この値は通常、'-1'または他の特定の値が選ばれます。 // しかし，この戻り値はエラーが発生したことを示すだけです．エラーの種類を知る必要がある場合は、 // システムエラー番号を表す変数 'errno' を調べる必要があります。

// この変数は errno.h ファイルで宣言され、プログラム開始時に 0 に初期化されます。

14.5 errno.h

// 実行します。

17 extern int errno;

18

1. // エラーが発生した場合、システムコールはエラー番号を // 変数 'errno' (負の値) に格納し、-1 を返します。したがって、プログラムが特定のエラー番号を知る必要がある場合、 // 'errno'の値を確認する必要があります。
2. #define ERROR 99 // General error.
3. #define EPERM 1 // The operation is not permited.
4. #define ENOENT 2 // The file or directory does not exist.
5. #define ESRCH 3 // The specified process does not exist.
6. #define EINTR 4 // Interrupted system-call.
7. #define EIO 5 // Input/output error.
8. #define ENXIO 6 // Specified device or address doesn't exist.
9. #define E2BIG 7 // The parameter list is too long.
10. #define ENOEXEC 8 // The format of executable file is incorrect.
11. #define EBADF 9 // File handle (descriptor) is incorrect.
12. #define ECHILD 10 // The child process does not exist.
13. #define EAGAIN 11 // The resource is temporarily unavailable.
14. #define ENOMEM 12 // No enough memory.
15. #define EACCES 13 // No access permissions.
16. #define EFAULT 14 // The address is wrong. 34 #define ENOTBLK 15 // Not a block device file. 35 #define EBUSY 16 // The resource is busy.
17. #define EEXIST 17 // File already exists.
18. #define EXDEV 18 // Illegal connection to device.
19. #define ENODEV 19 // The device does not exist.
20. #define ENOTDIR 20 // Not a directory file.
21. #define EISDIR 21 // Is a directory file.
22. #define EINVAL 22 // Invalid argument.
23. #define ENFILE 23 // The system has too many open files. 43 #define EMFILE 24 // Too many open files.
24. #define ENOTTY 25 // Inappropriate IO (no tty terminal).
25. #define ETXTBSY 26 // (No longer use).
26. #define EFBIG 27 // File size too big.
27. #define ENOSPC 28 // The device is full (the device has no space).
28. #define ESPIPE 29 // Invalid file pointer relocation.
29. #define EROFS 30 // The file system is read only.
30. #define EMLINK 31 // Too many links. 51 #define EPIPE 32 // The pipe is wrong.
31. #define EDOM 33 // Domain error.
32. #define ERANGE 34 // The result range error.
33. #define EDEADLK 35 // Resource deadlocks.
34. #define ENAMETOOLONG 36 // The filename is too long.
35. #define ENOLCK 37 // No locks are available.
36. #define ENOSYS 38 // Not yet implemented.
37. #define ENOTEMPTY 39 // The directory is not empty.

59

1. /\* Should never be seen by user programs \*/
2. #define ERESTARTSYS 512 // Re-execute the system-call.
3. #define ERESTARTNOINTR 513 // Re-execute the system-call, no interrupt.

63

64 #endif

65

14.6 fcntl.h

## 14.6 fcntl.h

### 14.6.1 Function

fcntl.hファイルは、ファイル制御オプションのヘッダーファイルです。主に、ファイル制御関数であるfcntl()と、ファイル作成やオープン関数で使用されるオプションの一部を定義しています。fcntl()関数は、linux/fs/fcntl.cファイルに実装されており、ファイル記述子(ハンドル)に対して指定された様々な操作を行うために使用されます。具体的な操作は、関数のパラメータcmd (コマンド)で指定します。

### 14.6.2 Code annotation

プログラム 14-5 linux/include/fcntl.h

1. #ifndef \_FCNTL\_H
2. #define \_FCNTL\_H

3

4 #include <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。

5

1. /\* open/fcntl - NOCTTY, NDELAY isn't implemented yet \*/
2. #define O\_ACCMODE 00003 // File access mode mask.
3. // open()やfcntl()で使用されるファイルアクセスモードで、3つのうち1つしか使用できません。
4. #define O\_RDONLY 00 // Open file in read-only mode.
5. #define O\_WRONLY 01 // Open file in write-only mode.
6. #define O\_RDWR 02 // Open file in read-write mode.

// 以下は、open()のファイル作成および操作フラグです。上記の

1. // アクセスモードを「bit or」モードにする。
2. #define O\_CREAT 00100 /\* not fcntl \*/ // Create if file does not exist.
3. #define O\_EXCL 00200 /\* not fcntl \*/ // Exclusive use of file.
4. #define O\_NOCTTY 00400 /\* not fcntl \*/ // No control terminal.
5. #define O\_TRUNC 01000 /\* not fcntl \*/ // truncated to zero if write operation.
6. #define O\_APPEND 02000 // Open file in append mode.
7. #define O\_NONBLOCK 04000 /\* not fcntl \*/ // Open file in non-blocking manner.
8. #define O\_NDELAY O\_NONBLOCK

18

1. /\* Defines for fcntl-commands. Note that currently
2. \* locking isn't supported, and other things aren't really 21 \* tested.
3. \*/
4. // ファイルディスクリプターの操作関数fcntl()で使用するコマンド(cmd)です。
5. #define F\_DUPFD 0 /\* dup \*/ // duplicate a file handle.
6. #define F\_GETFD 1 /\* get f\_flags \*/ // get file handle flags (FD\_CLOEXEC).
7. #define F\_SETFD 2 /\* set f\_flags \*/ // set file handle flags.
8. #define F\_GETFL 3 /\* more flags (cloexec) \*/
9. #define F\_SETFL 4 // get/set file state flag & access mode.
10. // 以下はファイルロックのコマンドです。fcntl()のパラメータ'lock'はflock構造体を指しています。
11. #define F\_GETLK 5 /\* not implemented \*/ // get flock that blocks the lock.
12. #define F\_SETLK 6 // Set (F\_RDLCK or F\_WRLCK) or clear lock (F\_UNLCK). 30 #define F\_SETLKW 7 // Set or clear the lock in wait mode.

31

1. /\* for F\_[GET|SET]FL \*/
2. // exec()関数を実行する際には、ファイルハンドルを閉じる必要があります。
3. #define FD\_CLOEXEC 1 /\* actually anything with low bit set goes \*/ 34
4. 35 /\* OK、これらはロック機能であり、どのレベルでも実装されていません 36 \* 。POSIXはそれを望んでいます。
5. \*/
6. #define F\_RDLCK 0 // Share or read file lock.
7. #define F\_WRLCK 1 // Exclusive or write file lock.
8. #define F\_UNLCK 2 // File unlock.

41

1. /\* Once again - not implemented, but ... \*/
2. // 以下はファイルロックのデータ構造で、影響を受けるファイルセグメントのタイプ（l\_type）、開始オフセット（l\_whence）、相対オフセット（l\_start）、ロック長（l\_len）、ロックを実装するpidが記述されています。
3. struct flock {
4. short l\_type; // Lock type (F\_RDLCK, F\_WRLCK, F\_UNLCK).
5. short l\_whence; // Start offset (SEEK\_SET, SEEK\_CUR or SEEK\_END).
6. off\_t l\_start; // The beginning of the lock. Relative offset (in bytes).
7. off\_t l\_len; // The size of the lock; if 0, it is the end of the file.
8. pid\_t l\_pid; // The process id of the lock.
9. };

50

// 上記のフラグやコマンドを使った関数のプロトタイプを以下に示します。

1. // 新しいファイルを作成したり、既存のファイルを書き換えたりします。パラメータの'filename'は作成されるファイルの // ファイル名、'mode'はプロパティ（include/sys/stat.hを参照）です。
2. extern int creat(const char \* filename,mode\_t mode);

// ファイルハンドル操作関数です。ファイルオープン操作に影響を与えることができます。パラメータの 'fildes' // はファイルハンドル（記述子）、 'cmd' は操作コマンドで、上記23～30行目を参照してください。

// 関数は次のような形になります。

// int fcntl(int fildes, int cmd)。

// int fcntl(int fildes, int cmd, long arg)。

1. // int fcntl(int fildes, int cmd, struct flock \*lock);
2. extern int fcntl(int fildes,int cmd, ...);

// ファイルを開く。ファイルとファイルハンドルの間の接続を確立するために使用されます。

1. // パラメータ'flags'は、上記の7～17行目のフラグを組み合わせたものです。
2. extern int open(const char \* filename, int flags, ...);

54

55 #endif

56

## 14.7 signal.h

### 14.7.1 Functionality

### シグナルは、非同期イベントを処理する方法を提供するもので、シグナルはソフトな割り込みとも呼ばれます。シグナルをプロセスに送ることで、そのプロセスの実行状態（一時停止、再開、終了）を制御することができます。signal.hファイルには、カーネルで使用されるすべてのシグナルの名前と基本的な動作機能が定義されています。最も重要な機能は、シグナルの処理方法を変更する関数signal()とsigaction()です。 ヘッダーファイルを見ればわかるように、LinuxカーネルはPOSIX.1で要求されている20種類のシグナルをすべて実装しています。つまり、Linuxは最初からこの規格との互換性を考慮して完全に設計されていると言えます。この関数の具体的な実装は、kernel/signal.cというプログラムにあります。

### 14.7.2 Code annotation

プログラム 14-6 linux/include/signal.h

1. #ifndef \_SIGNAL\_H
2. #define \_SIGNAL\_H

3

4 #include <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。

5

1. typedef int sig\_atomic\_t; // define signal atomic operation type.
2. typedef unsigned int sigset\_t; /\* 32 bits \*/ // define signal set type.

8

1. #define \_NSIG 32 // number of signals.
2. #define NSIG \_NSIG // NSIG = \_NSIG

11

1. // Linux 0.12カーネルで定義されているシグナルを以下に示します。これにはPOSIX.1で要求されている20個のシグナルがすべて含まれています。
2. #define SIGHUP 1 // Hang Up -- Hang up the control terminal or process.
3. #define SIGINT 2 // Interrupt -- Interrupt from the keyboard.
4. #define SIGQUIT 3 // Quit -- Exit command from the keyboard.
5. #define SIGILL 4 // Illeagle -- Illegal instruction.
6. #define SIGTRAP 5 // Trap -- Track breakpoints.
7. #define SIGABRT 6 // Abort
8. #define SIGIOT 6 // IO Trap
9. #define SIGUNUSED 7 // Unused
10. #define SIGFPE 8 // FPE -- Coprocessor error.
11. #define SIGKILL 9 // Kill -- Force the process to terminate. 22 #define SIGUSR1 10 // User1 -- User defined signal 1.
12. #define SIGSEGV 11 // Segment Violation -- Invalid memory reference.
13. #define SIGUSR2 12 // User2 -- User defined signal 2.
14. #define SIGPIPE 13 // Pipe -- Pipe write error, no reader.
15. #define SIGALRM 14 // Alarm -- Real-time timer alarm.
16. #define SIGTERM 15 // Terminate -- Process terminated.
17. #define SIGSTKFLT 16 // Stack Fault -- Stack error (coprocessor).
18. #define SIGCHLD 17 // Child -- Child process is stopped or terminated.
19. #define SIGCONT 18 // Continue -- Resume the execution of process.
20. #define SIGSTOP 19 // Stop -- Stop the execution of process.
21. #define SIGTSTP 20 // TTY Stop -- Stop sig sent by process, can be ignored.
22. #define SIGTTIN 21 // TTY In -- Background process requests input.
23. #define SIGTTOU 22 // TTY Out -- Background process requests output.

35

1. /\* Ok, I haven't implemented sigactions, but trying to keep headers POSIX \*/

// 0.12カーネルではsigaction()が実装されているので、上記のコメントは廃止されています。 // 以下は，sigaction 構造体のフラグフィールド 'sa\_flags' から取ることができるシンボリック定数である．

// SA\_NOCLDSTOP - 子機が停止状態にあるときは、SIGCHLD シグナルを処理しない。

1. // SA\_INTERRUPT - シグナルによって中断された後、システムコールは再開されません。 // SA\_NOMASK - このシグナルは、そのシグナル ハンドラでの受信がブロックされません。 // SA\_ONESHOT - シグナルハンドラーが呼び出された後、デフォルトのものに復元されます。
2. #define SA\_NOCLDSTOP 1
3. #define SA\_INTERRUPT 0x20000000
4. #define SA\_NOMASK 0x40000000
5. #define SA\_ONESHOT 0x80000000

41

// 以下の定数はsigprocmask(how, )で使用されます -- 与えられたシグナルを追加/削除するために。

ブロッキングシグナルセットへの // の出入りや、ブロッキングシグナルセット（マスクコード）の変更を行います。

1. // この関数の動作を変更するために使用します。
2. #define SIG\_BLOCK 0 /\* for blocking signals \*/
3. #define SIG\_UNBLOCK 1 /\* for unblocking signals \*/

/ #define SIG\_SETMASK 2 /\* シグナルマスクの設定 \*/

45

// 以下の3つの定数記号は、いずれも戻り値のない関数ポインタを表しています。

//の値を持ち、INTの整数パラメータを持つ。これらの3つのポインタは、論理的にはアドレス

実質的に不可能な機能の//。の2番目のパラメータとして使用することができます．

1. // 以下の signal() 関数を使って，カーネルにシグナルの処理をさせるか， // シグナルの処理を無視するか，あるいはシグナルの処理がエラーを返すかを指示します．これらの使い方については、 // kernel/signal.c の 156--158 行目を参照してください。
2. #define SIG\_DFL ((void (\*)(int))0) /\* default signal handling \*/
3. #define SIG\_IGN ((void (\*)(int))1) /\* ignore signal \*/
4. #define SIG\_ERR ((void (\*)(int))-1) /\* error return from signal \*/ 49
5. // sigaction構造のシグナルマスクを初期設定するためのマクロを定義します。
6. #ifdef notdef
7. #define sigemptyset(mask) ((\*(mask) = 0), 1) // Clear mask.
8. #define sigfillset(mask) ((\*(mask) = ~0), 1) // All bits of the mask are set.
9. #endif

54

// 以下はsigactionのデータ構造で、各フィールドの意味を表しています。

// 'sa\_handler'は、シグナルに対して指定されるアクションです。このシグナルは上記の // SIG\_DFL, SIG\_IGN で無視することもできるし、そのシグナルを処理する関数へのポインタを指定することもできる。 // 'sa\_mask'はシグナルのマスキングコードを与え、シグナルプログラムが実行されたときに、これらのシグナルの処理をブロックする。

// 'sa\_flags'は、信号処理を変更する信号のセットを指定するもので、 // 37〜40行目のビットフラグで定義されます。

// 'sa\_restorer'は、関数ライブラリLibcが提供する回復関数ポインタで、 // ユーザスタックのクリーンアップに使用される。signal.c を参照してください．

1. // また、SA\_NOMASKフラグが使用されていない限り、 // トリガー信号処理の原因となる信号もブロックされます。 55 struct sigaction {
2. void (\*sa\_handler)(int);
3. sigset\_t sa\_mask;
4. int sa\_flags;
5. void (\*sa\_restorer)(void);
6. }; 61

// 以下のsignal()関数は、sigaction()と同様に、シグナル\_sigに対する新しいシグナルハンドラをインストールするために使用されます。この関数は2つの引数を取ります。1つはキャプチャするシグナル\_sigを指定するもので、もう1つは1つの引数と戻り値を持たない関数ポインタ\_funcです。 // この関数の戻り値も、引数が1つ(最後の(int))で戻り値のない関数ポインターで、シグナルの元のハンドルです。 62 void (\*signal(int \_sig, void (\*\_func)(int))(int);

14.8 stdarg.h

1. // kill()は任意のプロセスやプロセスグループに // シグナルを送るために使われます。 raise()は現在のプロセス自身に // シグナルを送るために使われ、これは kill(getpid(), sig)と同等です。kernel/exit.c の 205 行目を参照してください。
2. int raise(int sig);
3. int kill(pid\_t pid, int sig);

// プロセスのタスク構造では、現在のプロセスの処理すべき信号を示す32ビットの信号フィールド'signal'に加えて、信号をマスクするための32ビットのブロッキング・シグナル・セット・フィールド'blocked'があり、各ビットが対応するブロッキングされた信号を表しています。

// マスクシグナルセットを変更すると、指定したシグナルをブロックしたり、ブロックを解除したりすることができます。以下の5つの関数は、シグナルセットをマスクするプロセスを操作するために使用されます。非常に // 簡単に実装できますが、本バージョンのカーネルでは実装されていません。

// 関数sigaddset()とsigdelset()は、シグナルの追加、削除、修正に使用されます。

// シグナルセットを指定します。Sigaddset()は，指定されたシグナルsignoを指摘されたシグナルセットに追加するために使用されます．

sigdelset()はその逆で，マスクで // になります．関数sigemptyset()とsigfillset()

// は、プロセスマスキング信号セットの初期化に使用されます。シグナルセットを使用する前に、各

// プログラムは，以下の2つの関数のいずれかを使って，マスク信号セットを初期化する必要があります．シグネプティセット()

// マスクされた信号をすべてクリアする、つまり、すべての信号に反応するために使用されます; sigfillset() puts

1. // すべての信号をシグナルセットに入れる、つまりすべての信号をマスクする。もちろん、SIGINTとSIGSTOPは // ブロックできません。sigismember()関数は、指定したシグナルがシグナルセットに含まれているかどうかを // テストするために使用されます(1 - yes, 0 - no, -1 - error)。
2. int sigaddset(sigset\_t \*mask, int signo);
3. int sigdelset(sigset\_t \*mask, int signo);
4. int sigemptyset(sigset\_t \*mask);
5. int sigfillset(sigset\_t \*mask);
6. int sigismember(sigset\_t \*mask, int signo); /\* 1 - is, 0 - not, -1 error \*/
7. // 'set'では保留中のシグナルがあるかどうかを確認します。そして'set'では、プロセスで設定された現在の // ブロックされているシグナルを返します。
8. int sigpending(sigset\_t \*set);
9. // 次の関数は、プロセスが現在ブロックしているシグナルセットを変更するために使用されます。 // 'oldset' が NULL でない場合は、現在のマスクされたシグナルセットを返します。set' ポインタが // NULL でない場合は、'how' に従ってプロセスのマスクシグナルセットを変更します (42-44 行目)。
10. int sigprocmask(int how, sigset\_t \*set, sigset\_t \*oldset);

// 以下の関数は、プロセスのシグナルマスクを'sigmask'に一時的に置き換えます。

1. // して、シグナルを受け取るまで処理を一時停止します。シグナルが捕捉され，シグナルハンドラから // 返されると，この関数は戻り，シグナルマスクは呼び出し前の値に // 復元されます。
2. int sigsuspend(sigset\_t \*sigmask);
3. // sigaction()関数は，プロセスがシグナルを受信したときの動作を変更するため，つまり，シグナルの処理ハンドラを変更するために使用される。kernel/signal.c プログラムの説明を参照してください。
4. int sigaction(int sig, struct sigaction \*act, struct sigaction \*oldact); 74

75 #endif /\* \_SIGNAL\_H \*/

76

## 14.8 stdarg.h

### 14.8.1 Function

C言語の最大の特徴は、プログラマーが関数をカスタマイズできることです。

14.8 stdarg.h

変数のパラメータ数です。これらの可変パラメータリストのパラメータにアクセスするためには、stdarg.hファイルのマクロを使用する必要があります。stdarg.hファイルは，BSDシステムのvarargs.hファイルにしたがって，C標準化団体によって修正されています。

### Stdarg.hは、可変パラメータのリストをマクロの形で定義した標準パラメータのヘッダファイルです。主にvsprintf, vprintf, vfprintf関数の型(va\_list)と3つのマクロ(va\_start, va\_arg, va\_end)が記述されています。このファイルを読む際には、まず変数パラメータ関数の使い方を理解する必要があります。kernel/vsprintf.cのリストの後の説明を参照してください。

### 14.8.2 Code annotation

プログラム 14-7 linux/include/stdarg.h

1. #ifndef \_STDARG\_H
2. #define \_STDARG\_H

3

1. 4 typedef char \*va\_list; // va\_listの定義は、文字ポインタ型です。 5
2. /\* Amount of space required in an argument list for an arg of type TYPE.
3. TYPE may alternatively be an expression whose type is used. \*/

8

1. // 次の文は、丸みを帯びたTYPE型のバイト長を定義しており、これはint型の長さ（4）の倍数です。
2. #define \_\_va\_rounded\_size(TYPE) \
3. (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) \* sizeof (int)) 11

// 以下のマクロでは，ポインタ AP を初期化して SI に渡された最初の引数を指すようにします．

// 関数の可変引数リストです。va\_arg や va\_end を初めて呼び出す前には

// は最初に va\_start マクロを呼び出さなければなりません。パラメータ LASTARG は，関数定義の中で // 一番右のパラメータの識別子，つまり '...'の左側の識別子です．AP は // 変数パラメータテーブルのパラメータポインタであり， LASTARG は最後に指定されたパラメータです． // &(LASTARG)はそのアドレス(つまりポインタ)を得るために使われ，ポインタは文字型です． // LASTARG の幅の値を加えた後の AP は、 // 変数パラメータリストの最初のパラメータへのポインタです。このマクロは戻り値を持ちません。

1. // 17行目の関数\_\_builtin\_saveregs()は、gccのライブラリプログラムであるlibgcc2.cで定義されており、 // レジスタを保持するために使用されています。手順については、gcc マニュアルの「Target Description Macros」の // 「Implementing the Varargs Macros」セクションを参照してください。
2. #ifndef \_\_sparc\_\_
3. #define va\_start(AP, LASTARG) \
4. (AP = ((char \*) &(LASTARG) + \_\_va\_rounded\_size (LASTARG)))
5. #else
6. #define va\_start(AP, LASTARG) \
7. (\_\_builtin\_saveregs (), \
8. AP = ((char \*) &(LASTARG) + \_\_va\_rounded\_size (LASTARG)))
9. #endif 20
10. // va\_end は、va\_start が再び呼び出されるまで AP を使用しないように変更することができます。
11. void va\_end (va\_list); /\* Defined in gnulib \*/
12. #define va\_end(AP) 23

// 次のマクロは、式を拡張して、同じタイプと値を持つ

// 次に渡されるパラメータ。デフォルトの値としては，va\_arg は，文字，符号なし

14.9 stddef.h

1. // 文字や浮動小数点型を返します。va\_argを最初に使用したとき、テーブルの最初の引数が返され、その後の呼び出しのたびに、テーブルの次の引数が返されます。これは // 最初に AP にアクセスし、その値を増加させて次の項目を指し示すことで行われます。 // va\_arg は TYPE を使用してアクセスを完了し、次のアイテムを見つけます。va\_arg が呼び出されるたびに、 // テーブル内の次のパラメータを示すように AP を変更します。
2. #define va\_arg(AP, TYPE) \
3. (AP += \_\_va\_rounded\_size (TYPE), \
4. \*((TYPE \*) (AP - \_\_va\_rounded\_size (TYPE))))

27

28 #endif /\* \_STDARG\_H \*/

29

## 14.9 stddef.h

### 14.9.1 Functionality

stddef.hというヘッダーファイルは、C言語の標準化団体（X3J11）が作成したもので、ファイル名の意味は「標準（std）の定義（def）」となっている。主に、UNIX系システムでよく使われる定数記号や関数プロトタイプなどの「標準定義」を格納するために使用されます。もう一つの紛らわしいヘッダーファイルはstdlib.hで、これも標準化団体が作成したもので、主に他のヘッダーファイルの種類とは関係のない様々な関数プロトタイプの宣言に使用されます。しかし、この2つのヘッダーファイルの内容を見ると、どの宣言がどのヘッダーファイルにあるのかわからなくなることがよくあります。

* 標準化団体のメンバーの中には、標準Cライブラリを完全にはサポートしていないスタンドアロン環境においても、C言語は有用なプログラミング言語であるべきだと考えている人もいる。スタンドアロン環境では、C言語のすべての属性を提供することがC規格で求められている。標準Cライブラリについては、float.h、limit.h、stdarg.h、stddef.hの4つのヘッダファイルに含まれる関数をサポートするだけでよいとしている。
* float.h Describe the floating point representation feature;
* limits.h Describe the integer representation characteristics;
* stdarg.h Provides macro definitions for accessing variable parameter list.

スタンドアロン環境で使用されるその他の型やマクロの定義は、stddef.hファイルに置くべきですが、後の組織メンバーがこの制限を緩和したため、いくつかの定義が複数のヘッダーファイルに存在することになりました。例えば，NULLマクロの定義は，他の4つのヘッダファイルにも含まれています。そのため、stddef.hファイルでは、NULLを定義する際に、まずundefコマンドを使用して元の定義をキャンセルすることで、衝突を防いでいます（14行目）。また、このファイルで定義されている型やマクロには共通点があり、C言語の機能に含まれるように工夫されていますが、様々なコンパイラが独自の方法でこれらの情報を定義しているため、これらの定義をすべて置き換えるコードを書くことは非常に困難です。このファイルは、Linux 0.12カーネルではほとんど使われていません。

### 14.9.2 Code annotation

プログラム 14-8 linux/include/stddef.h

1. #ifndef \_STDDEF\_H
2. #define \_STDDEF\_H

3

1. #ifndef \_PTRDIFF\_T
2. #define \_PTRDIFF\_T
3. typedef long ptrdiff\_t; // The type of result of subtracting two pointers.
4. #endif

8

1. #ifndef \_SIZE\_T
2. #define \_SIZE\_T
3. typedef unsigned long size\_t; // The type of result returned by sizeof().
4. #endif

13

1. #undef NULL
2. #define NULL ((void \*)0) // null pointer.

16

// 以下は、型の中のメンバーのオフセット位置を計算するマクロを定義しています。

// このマクロを使うと、メンバー（フィールド）の先頭からのバイトオフセットを求めることができます

// を含む構造体タイプの中で、その構造体の このマクロの結果は，整数

// size\_t型の定数表現。ここではトリック的な使い方をします。((TYPE \*)0)は、0型の整数を // データオブジェクトのポインタ型にキャストし、 // その結果に対して演算を行うことを意味します。

17 #define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER)

18

19 #endif

20

## 14.10 string.h

### 14.10.1 Functionality

string.hヘッダファイルでは、すべての文字列操作関数をインライン関数として定義し、実行速度を向上させるためにインラインアセンブリ言語を使用しています。また，ファイルの先頭には，NULLマクロとSIZE\_T型が定義されています。

同名のヘッダーファイルも提供されていますが、関数の実装は標準Cライブラリにあり、その対応するヘッダーファイルには、該当する関数の宣言のみが含まれています。ここに挙げたstring.hファイルでは、Linusが各関数の実装を与えていますが、各関数には「extern」と「inline」のキーワード接頭辞がついており、つまり、いくつかのインライン関数が定義されています。したがって、このヘッダファイルを含むプログラムでは、何らかの理由で使用するインライン関数が呼び出しコードに埋め込めない場合、カーネル関数ライブラリlib/ディレクトリで定義されている同名関数が使用されることになります。lib/string.cプログラムをご覧ください。そのstring.cのプログラムでは、まず「extern」と「inline」を空に定義し、次にstring.hというヘッダーファイルを入れています。したがって、string.cプログラムは、実際にはstring.hヘッダーファイルで宣言された関数の実装コードを含んでいます。

### 一行でコメントを書きやすいように、ここでは混乱を招かないようにいくつかの略語をソースに使用しています。例えば、string - str、character - char、pointer - ptr、address - addr. source - src、destination - dest、length - len.

### 14.10.2 Code annotation

プログラム14-9 linux/include/string.h

1. #ifndef \_STRING\_H\_
2. #define \_STRING\_H\_

3

1. #ifndef NULL
2. #define NULL ((void \*) 0)
3. #endif

7

1. #ifndef \_SIZE\_T
2. #define \_SIZE\_T
3. typedef unsigned int size\_t;
4. #endif

12

13 extern char \* strerror(int errno);

14

1. /\*
2. \* This string-include defines all string functions as inline
3. \* functions. Use gcc. It also assumes ds=es=data space, this should be
4. \* normal. Most of the string-functions are rather heavily hand-optimized,
5. \* see especially strtok,strstr,str[c]spn. They should work, but are not
6. \* very easy to understand. Everything is done entirely within the register
7. \* set, making the functions fast and clean. String instructions have been 22 \* used through-out, making for "slightly" unclear code :-)
8. \*
9. \* (C) 1991 Linus Torvalds
10. \*/

26

//// コピー元の文字列をコピー先の文字列にNULL文字に遭遇するまでコピーします。

// Params: dest - コピー先の str ポインタ, src - コピー元の str ポインタ。

1. // 埋め込みアセンブリコードでは、%0 - ESI(src)を登録、%1 - EDI(dest)を登録します。
2. extern inline char \* strcpy(char \* dest,const char \*src)
3. {
4. \_\_asm\_\_("cld\n" // Clear direction flag.
5. "1:\tlodsb\n\t" // Load 1 byte from DS:[ESI] into AL and update ESI.
6. "stosb\n\t" // Store the byte in AL to ES:[EDI] and update EDI.
7. "testb %%al,%%al\n\t" // The byte just stored is 0?
8. "jne 1b" // If not, jump backward to label 1, else end.
9. ::"S" (src),"D" (dest):"si","di","ax");
10. return dest; // Returns the destination str pointer. 36 }

37

//// コピー元の文字列のカウントバイトをコピー先にコピーします。

1. // ソース str の長さが count バイトよりも小さい場合、NULL 文字が dest 文字列に追加されます。 // パラメータ： dest - コピー先の str ポインタ， src - コピー元の str ポインタ， count - コピーされるバイト数。0 - esi(src), %1 - edi(dest), %2 - ecx(count). 38 エクスターン インライン char \* strncpy(char \* dest,const char \*src,int count)
2. {
3. \_\_asm\_\_("cld\n" // Clear direction flag.
4. "1:\tdecl %2\n\t" // Register ECX-- (count--).
5. "js 2f\n\t" // If count<0 then jump forward to label 2 and end.
6. "lodsb\n\t" // Get 1 byte from DS:[ESI] to AL, and ESI++.
7. "stosb\n\t" // Store the byte to ES:[EDI], and EDI++.
8. "testb %%al,%%al\n\t" // Is this byte 0?
9. "jne 1b\n\t" // If not, jump forward to label 1 to continue copy. 47 "rep\n\t" // Else, store remain number of NULLs into dest str.
10. "stosb\n"
11. "2:"
12. ::"S" (src),"D" (dest),"c" (count):"si","di","ax","cx");
13. return dest; // Returns the dest string pointer. 52 }

53

//// コピー元の文字列をコピー先の文字列の末尾にコピーします。

// パラメータ： dest - 出力先の str ポインタ， src - 出力元の str ポインタ。

1. // アセンブリコードでは、%0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1)。
2. extern inline char \* strcat(char \* dest,const char \* src)
3. {
4. \_\_asm\_\_("cld\n\t"
5. "repne\n\t" // Compare AL and ES:[EDI] byte and update EDI++, 58 "scasb\n\t" // Until byte in dest is 0, the EDI points to last byte.
6. "decl %1\n" // Let ES:[EDI] points to the location of 0.
7. "1:\tlodsb\n\t" // Take source str byte DS:[ESI] to AL, and ESI++.
8. "stosb\n\t" // Store this byte to ES:[EDI] and EDI++.
9. "testb %%al,%%al\n\t" // Is this byte 0?
10. "jne 1b" // If not,jump back to label 1 to continue, else end. 64 ::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff):"si","di","ax","cx");

65 return dest; // destのstrポインタを返します。 66 } 67

//// コピー元strのcountバイトをコピー先strの末尾にコピーし、最後にnull charを追加します。 // dest - コピー先のstr, src - コピー元のstr, count - コピーするバイト数。

1. // アセンブリコードでは、%0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1), %4 - (count)となります。
2. extern inline char \* strncat(char \* dest,const char \* src,int count)
3. {
4. \_\_asm\_\_("cld\n\t"
5. "repne\n\t" // Compare AL and ES:[EDI], and update EDI++, 72 "scasb\n\t" // Until 0 in dest, the EDI points to last byte.
6. "decl %1\n\t" // Let ES:[EDI] points to the location of 0.
7. "movl %4,%3\n" // Place the bytes to be copied into ECX.
8. "1:\tdecl %3\n\t" // ECX-- (counts from 0).
9. "js 2f\n\t" // Ecx < 0 ?, yes, jump forward to the label 2.
10. "lodsb\n\t" // Otherwise take the bytes at DS:[ESI] to AL, ESI++.
11. "stosb\n\t" // Store to ES:[EDI], and EDI++.
12. "testb %%al,%%al\n\t" // The byte value is 0?
13. "jne 1b\n" // If not, jump back to label 1 and continue copying.
14. "2:\txorl %2,%2\n\t" // Clear AL to zero.
15. "stosb" // Save to ES:[EDI].
16. ::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff),"g" (count)
17. :"si","di","ax","cx");
18. return dest; // Returns the dest string pointer. 86 }

87

//// 2つの文字列を比較します。

// パラメータ：cs-文字列1、ct-文字列2。

// %0 - eax(\_\_res) 戻り値、 %1 - edi(cs) str1 ポインタ、 %2 - esi(ct) str2 ポインタ。

1. // str1 > str2の場合は1を、str1 = str2の場合は0を、str1 < str2の場合は-1を返します。
2. extern inline int strcmp(const char \* cs,const char \* ct)
3. {
4. register int \_\_res \_\_asm\_\_("ax"); // \_\_res is a register variable (eax).
5. \_\_asm\_\_("cld\n"
6. "1:\tlodsb\n\t" // Get str2 byte DS:[ESI] to AL, and ESI++.
7. "scasb\n\t" // Compare AL with str1 byte ES:[EDI], and EDI++.
8. "jne 2f\n\t" // If they are not equal, jump forward to label 2.
9. "testb %%al,%%al\n\t" // Is this byte 0 (end of string)?
10. "jne 1b\n\t" // No, jump back to label 1, and continue comparison. 97 "xorl %%eax,%%eax\n\t" // Yes, EAX is cleared, jumps forward to label 3,end.
11. "jmp 3f\n"
12. "2:\tmovl $1,%%eax\n\t" // EAX is set to 1.
13. "jl 3f\n\t" // If str2 < str1, returns a positive value, end.
14. "negl %%eax\n" // Else EAX = -EAX, return a negative value, and end.
15. "3:"
16. :"=a" (\_\_res):"D" (cs),"S" (ct):"si","di");
17. return \_\_res; // Returns the comparison result. 105 }

106

//// 文字列1は、文字列2の最初のカウント文字と比較されます。

// パラメータ：cs - str1, ct - str2, count - 比較する文字数。

// %0 - eax(\_\_res), %1 - edi(cs) str1 ポインタ, %2 - esi(ct) str2 ポインタ, %3 - ecx(count).

1. // str1 > str2の場合は1を、str1 = str2の場合は0を、str1 < str2の場合は-1を返します。
2. extern inline int strncmp(const char \* cs,const char \* ct,int count)
3. {
4. register int \_\_res \_\_asm\_\_("ax"); // \_\_res is a register variable (eax).
5. \_\_asm\_\_("cld\n"
6. "1:\tdecl %3\n\t" // Register ECX-- (count--).
7. "js 2f\n\t" // If count<0, then jump forward to label 2.
8. "lodsb\n\t" // Load str2 char DS:[ESI] to AL, and ESI++.
9. "scasb\n\t" // Compare AL with str1 char ES:[EDI], and EDI++.
10. "jne 3f\n\t" // If they are not equal, jump forward to label 3.
11. "testb %%al,%%al\n\t" // Is it a NULL char?
12. "jne 1b\n" // No, jump back to label 1,continue the comparison.
13. "2:\txorl %%eax,%%eax\n\t" // Yes, then EAX is cleared (return value).
14. "jmp 4f\n" // Jump forward to label 4 and end.
15. "3:\tmovl $1,%%eax\n\t" // Set EAX to 1.
16. "jl 4f\n\t" // If the result is str2 < str1, 1 is returned.
17. "negl %%eax\n" // Else EAX = -EAX, return a negative value and end.
18. "4:"
19. :"=a" (\_\_res):"D" (cs),"S" (ct),"c" (count):"si","di","cx"); 125 return \_\_res; // Returns the comparison result. 126 }

127

//// 文字列の中で最初に一致する文字を探します。

// パラメータ：s - 文字列、c - 検出される文字。

// アセンブリコードでは、%0 - eax(\_\_res), %1 - esi (str pointer s), %2 - eax (char c) です。

// 文字列の中で最初に一致する文字の出現箇所へのポインタを返します。もし、文字列中に

// 一致する文字が見つかった場合は、NULLポインタを返します。

1. 128 extern inline char \* strchr(const char \* s,char c) / 130 register char \* \_\_res \_\_asm\_\_("ax");
2. \_\_asm\_\_("cld\n\t"
3. "movb %%al,%%ah\n" // Move the char to be compared to AH.
4. "1:\tlodsb\n\t" // Get a string char DS:[ESI] to AL, and ESI++.
5. "cmpb %%ah,%%al\n\t" // The char AL is compared with the specified char AH.
6. "je 2f\n\t" // If they are the same, jump forward to the label 2.
7. "testb %%al,%%al\n\t" // Is the char in AL a NULL char? (End of string?) 137 "jne 1b\n\t" // If not, back to label 1 and continue comparison.
8. "movl $1,%1\n" // If yes, no match char is found, and set ESI to 1.
9. "2:\tmovl %1,%0\n\t" // Put pointer at the next byte of match char into EAX.
10. "decl %0" // Adjust the pointer to point to the matching char.
11. :"=a" (\_\_res):"S" (s),"0" (c):"si");
12. return \_\_res; // Returns the pointer.  144

//// 文字列の中で、与えられた文字が最後に現れるかどうかを探します。(逆引き検索文字列) // パラメータ: s - 文字列, c - 見つけるべき文字。

// アセンブリコードでは、%0 - eax(\_\_res), %1 - esi (string pointer s), %2 - eax (char c) となっています。

// 文字列の中で最後にマッチした文字へのポインタを返します。もし、一致する文字が

// 一致する文字が見つかった場合は、NULLポインタを返します。

1. 145 extern インライン char \* strrchr(const char \* s,char c) /.
2. register char \* \_\_res \_\_asm\_\_("dx");
3. \_\_asm\_\_("cld\n\t"
4. "movb %%al,%%ah\n" // Move the char to be compared to AH.
5. "1:\tlodsb\n\t" // Get a string char DS:[ESI] to AL, and ESI++.
6. "cmpb %%ah,%%al\n\t" // The char AL is compared with the specified char AH.
7. "jne 2f\n\t" // If not the same, jump forward to the label 2.
8. "movl %%esi,%0\n\t" // Store the char pointer to EDX.
9. "decl %0\n" // Adjust the pointer to point to the matching char.
10. "2:\ttestb %%al,%%al\n\t" // Is the char in AL a NULL char? (End of string?) 156 "jne 1b" // If not, back to label 1 and continue comparison.

157 :"=d" (\_\_res): "0" (0), "S" (s), "a" (c): "ax", "si"); 158 return \_\_res; // ポインタを返す。

159 } 160

// string1の最初の文字列を見つけ、その中の任意の文字はstring2に含まれます。

// パラメータ：cs - 文字列1のポインタ、ct - 文字列2のポインタ。

// %0 - esi(\_\_res), %1 - eax(0), %2 - ecx(-1), %3 - esi(str1 ptr cs), %4 - (str2 ptr ct)。

1. // str2に含まれるstr1の最初の文字列の長さを返します。161 extern inline int strspn(const char \* cs, const char \* ct) /.
2. register char \* \_\_res \_\_asm\_\_("si");
3. \_\_asm\_\_("cld\n\t"
4. "movl %4,%%edi\n\t" // Calc the len of str2, store its pointer to EDI.
5. "repne\n\t" // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
6. "scasb\n\t" // If they are not equal, continue to compare (ECX--).
7. "notl %%ecx\n\t" // Each bit in ECX is inversed.
8. "decl %%ecx\n\t" // ECX--, The length of str2 is obtained.
9. "movl %%ecx,%%edx\n" // Put the length of str2 into EDX temporarily.
10. "1:\tlodsb\n\t" // Put the str1 char DS:[ESI] to AL, and ESI++.
11. "testb %%al,%%al\n\t" // Is the char equal to 0 (end of str1)?
12. "je 2f\n\t" // If yes, jump forward to label 2 and end.
13. "movl %4,%%edi\n\t" // Get the str2 pointer into EDI.
14. "movl %%edx,%%ecx\n\t" // Put the str2 length into ECX.
15. "repne\n\t" // Compare AL and str2 char ES:[EDI], and EDI++.
16. "scasb\n\t" // If they are equal, continue to compare.
17. "je 1b\n" // If they are equal, jump backward to label 1.
18. "2:\tdecl %0" // ESI--, points to the char in str1 contained in str2.
19. :"=S" (\_\_res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
20. :"ax","cx","dx","di");
21. return \_\_res-cs; // Returns the length of the sequence of chars.  184

//// 文字列1の中で、文字列2の中のどの文字も含まない最初の文字列を検索します。

// パラメータ：cs - 文字列1のポインタ、ct - 文字列2のポインタ。

// %0 - esi(\_\_res), %1 - eax(0), %2 - ecx(-1), %3 - esi (str1 ptr cs), %4 - (str2 ptr ct)。

// str1の最初の文字列で、str2のどの文字も含まないものの長さを返します。

1. 185 extern inline int strcspn(const char \* cs, const char \* ct) /.
2. register char \* \_\_res \_\_asm\_\_("si");
3. \_\_asm\_\_("cld\n\t"
4. "movl %4,%%edi\n\t" // Calc the len of str2, store its pointer to EDI.
5. "repne\n\t" // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
6. "scasb\n\t" // If they are not equal, continue to compare (ECX--). 192 "notl %%ecx\n\t" // Calc the length of str2 by inversing each bits
7. "decl %%ecx\n\t" // in ECX and decreasing ECX by one.
8. "movl %%ecx,%%edx\n" // Put str2 length into EDX temporarily.
9. "1:\tlodsb\n\t" // Take the str1 char DS:[ESI] to AL, and ESI++.
10. "testb %%al,%%al\n\t" // Is the char equal to 0 (end of str1)?
11. "je 2f\n\t" // If yes, jump forward to label 2 and end.
12. "movl %4,%%edi\n\t" // Get the str2 pointer into EDI.
13. "movl %%edx,%%ecx\n\t" // Put the str2 length into ECX.
14. "repne\n\t" // Compare AL and str2 char ES:[EDI], and EDI++.
15. "scasb\n\t" // If they not are equal, continue to compare.
16. "jne 1b\n" // If they not are equal, jump backward to label 1.
17. "2:\tdecl %0" // ESI--, points to the char in str1 contained in str2.
18. :"=S" (\_\_res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
19. :"ax","cx","dx","di");
20. return \_\_res-cs; // Returns the length of the sequence of chars. 207 }

208

//// 文字列1の中で、文字列2に含まれる任意の最初の文字を探します。

// パラメータ：cs - 文字列1のポインタ、ct - 文字列2のポインタ。

// %0 -esi(\_\_res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(str1 ptr cs), %4 -(str2 ptr ct)。

1. // 209 extern inline char \* strpbrk(const char \* cs,const char \* ct) /.
2. register char \* \_\_res \_\_asm\_\_("si");
3. \_\_asm\_\_("cld\n\t"
4. "movl %4,%%edi\n\t" // Calc the len of str2, store its pointer to EDI.
5. "repne\n\t" // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
6. "scasb\n\t" // If they are not equal, continue to compare (ECX--). 216 "notl %%ecx\n\t" // Calc the length of str2 by inversing each bits
7. "decl %%ecx\n\t" // in ECX and decreasing ECX by one.
8. "movl %%ecx,%%edx\n" // Store str2 length into EDX temporarily.
9. "1:\tlodsb\n\t" // Get the str1 char DS:[ESI] to AL, and ESI++.
10. "testb %%al,%%al\n\t" // Is the char equal to 0 (end of str1)?
11. "je 2f\n\t" // If yes, jump forward to label 2.
12. "movl %4,%%edi\n\t" // Get the str2 pointer into EDI.
13. "movl %%edx,%%ecx\n\t" // Put the str2 length into ECX.
14. "repne\n\t" // Compare AL and str2 char ES:[EDI], and EDI++.
15. "scasb\n\t" // If they not are equal, continue to compare.
16. "jne 1b\n\t" // If they not are equal, jump backward to label 1.
17. "decl %0\n\t" // ESI--, points to a char in str1 contained in str2.
18. "jmp 3f\n" // Jump forward to label 3 and end.
19. "2:\txorl %0,%0\n" // If no match is found, it will return NULL.
20. "3:"
21. :"=S" (\_\_res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
22. :"ax","cx","dx","di");
23. return \_\_res; // Returns the pointer in str1. 

235

//// string1では、string2全体にマッチする最初の部分文字列を探します。

// パラメータ：cs - 文字列1のポインタ、ct - 文字列2のポインタ。

// %0 -eax(\_\_res), %1 -eax(0), %2 -ecx(0xffffff), %3 -esi(str1 ptr cs), %4 -(str2 ptr ct)。

1. // 236 extern inline char \* strstr(const char \* cs,const char \* ct) /.
2. register char \* \_\_res \_\_asm\_\_("ax");
3. \_\_asm\_\_("cld\n\t" \
4. "movl %4,%%edi\n\t" // Calc the len of str2, store its pointer to EDI.
5. "repne\n\t" // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
6. "scasb\n\t" // If they are not equal, continue to compare (ECX--). 243 "notl %%ecx\n\t" // Calc str2 len by inversing all bits and dec 1 in ECX.
7. "decl %%ecx\n\t" /\* NOTE! This also sets Z if searchstring='' \*/
8. "movl %%ecx,%%edx\n" // Store str2 length into EDX temporarily.
9. "1:\tmovl %4,%%edi\n\t" // Get the str2 pointer into EDI.
10. "movl %%esi,%%eax\n\t" // Copy the str1 pointer to EAX.
11. "movl %%edx,%%ecx\n\t" // Put the str2 length into ECX.
12. "repe\n\t" // Comp str1,str2 char DS:[ESI],ES:[EDI], ESI++,EDI++.
13. "cmpsb\n\t" // If the chars are equal, they continue to compare.
14. // 全て等しい場合は、ラベル2にジャンプし、終了。
15. "je 2f\n\t" /\* also works for empty string, see above \*/
16. "xchgl %%eax,%%esi\n\t" // str1 ptr => ESI, comparison result str1 ptr => EAX.
17. "incl %%esi\n\t" // str1 pointer points to the next char.
18. "cmpb $0,-1(%%eax)\n\t" // Is the byte pointed to by ptr in str1 (EAX-1) 0?
19. "jne 1b\n\t" // If not, go label1, continue comp from 2nd char of str1.
20. "xorl %%eax,%%eax\n\t" // Clear EAX, indicating that no match was found.
21. "2:"
22. :"=a" (\_\_res):"0" (0),"c" (0xffffffff),"S" (cs),"g" (ct)
23. :"cx","dx","di","si");
24. return \_\_res; // Returns the result.
25. }

262

//// 文字列の長さを計算する。

// パラメータ: s - 文字列。

// %0 - ecx(\_\_res), %1 - edi (string pointer s), %2 - eax(0), %3 - ecx(0xffffff). // 文字列の長さを返します。

1. 263 extern inline int strlen(const char \* s) /.
2. register int \_\_res \_\_asm\_\_("cx"); // \_\_res is a register variable (ECX).
3. \_\_asm\_\_("cld\n\t" // Clear the direction flag.
4. "repne\n\t" // AL(0) is compared with char es:[edi] in the string, 268 "scasb\n\t" // If they are not equal, continue to compare.
5. "notl %0\n\t" // Inverse each bit in ECX.
6. "decl %0" // ECX--, the length of the string is obtained.
7. :"=c" (\_\_res):"D" (s),"a" (0),"0" (0xffffffff):"di"); 272 return \_\_res; // Returns the string length. 273 }

274

// 一時的な文字列ポインタで，以下のように解析された文字列へのポインタを格納するために使用されます．

275 extern char \* \_\_\_strtok; // 文字列のトークンです。

276

//// 文字列2の文字を使って、文字列1をトークン・シーケンスに分割します。

// String1は、0個以上のトークンを含み、1個のトークンで区切られたシーケンスとみなされます。

// またはそれ以上の文字をセパレータのstring2に入れます。strtok()が初めて呼ばれたとき。

// string1 の最初のトークンの最初の文字へのポインタが返され、トークンが返されたときには、 // ヌル文字がセパレータに書き込まれます。最初の引数に null を指定して strtok() を呼び出すと、トークンがなくなるまで、 // この方法で string1 をスキャンし続けます。

// 分割された文字列2は、異なる起動時には異なる可能性があります。

// Params: s - 処理される文字列1, ct - 各セパレータを含む文字列2。

1. // %0-ebx(\_\_res), %1-esi(\_\_strtok); %2-ebx(\_\_strtok), %3-esi(str1 ptr s), %4-(str2 ptr ct)。 // 文字列 s のトークンを返し、トークンが見つからない場合は NULL ポインタを返します。NULL 文字列 s での strtok() の後続の呼び出しは、元の文字列 s 内の次のトークンを検索します。 277 extern inline char \* strtok(char \* s,const char \* ct) /
2. register char \* \_\_res \_\_asm\_\_("si");
3. \_\_asm\_\_("testl %1,%1\n\t" // First test if ESI (str1 pointer s) is NULL.
4. "jne 1f\n\t" // If not, means the first call, and jump label 1.
5. "testl %0,%0\n\t" // If NULL, means subsequent call, test EBX(\_\_strtok).
6. "je 8f\n\t" // If EBX is NULL, it cannot be processed,jump to end.
7. "movl %0,%1\n" // Copy the EBX pointer(current str1 ptr) to ESI.
8. "1:\txorl %0,%0\n\t" // Clear EBX.
9. "movl $-1,%%ecx\n\t" // Set ECX = 0xffffffff. 287 "xorl %%eax,%%eax\n\t" // Clear EAX.
10. "cld\n\t" // Clear direction flag.
11. "movl %4,%%edi\n\t" // Let's find the str2 length. EDI points to str2.
12. "repne\n\t" // Compare AL(0) with char at ES:[EDI], and EDI++.
13. "scasb\n\t" // Until end of str2 (null char), or count ECX = 0.
14. "notl %%ecx\n\t" // Inverse ECX and subtract 1,to get str2 length.
15. "decl %%ecx\n\t" // If str2 len is 0, go forward to label 7.
16. "je 7f\n\t" /\* empty delimeter-string \*/
17. "movl %%ecx,%%edx\n" // Store str2 length to EDX temporarily.
18. "2:\tlodsb\n\t" // Get str1 char DS:[ESI] to AL, and ESI++.
19. "testb %%al,%%al\n\t" // Is the char 0 (end of str1)?
20. "je 7f\n\t" // If yes, jump forward to label 7.
21. "movl %4,%%edi\n\t" // EDI again points to first char of the str2.
22. "movl %%edx,%%ecx\n\t" // Copy str2 length into the counter ECX.
23. "repne\n\t" // Compare str1 char AL with all chars in str2,
24. "scasb\n\t" // to check if the char is a separator.
25. "je 2b\n\t" // If found it in str2, jump backward to label 2.
26. "decl %1\n\t" // Else the str1 pointer ESI points to the char now.
27. "cmpb $0,(%1)\n\t" // Is this a NULL character?
28. "je 7f\n\t" // If yes, jump forward to label 7.
29. "movl %1,%0\n" // The char pointer ESI is stored in EBX.
30. "3:\tlodsb\n\t" // Get next char of str1 to DS:[ESI], and ESI++.
31. "testb %%al,%%al\n\t" // Is the char 0 (end of str1)?
32. "je 5f\n\t" // If yes, means str1 ends and jump forward to label 5.
33. "movl %4,%%edi\n\t" // EDI again points to first char of the str2.
34. "movl %%edx,%%ecx\n\t" // Copy str2 length into the counter ECX.
35. "repne\n\t" // Compare str1 char AL with all chars in str2, 314 "scasb\n\t" // to check if the char is a separator.
36. 315 "jne 3b\\t" // そうでなければ、label3 にジャンプして、str1 の次の char を検出します。316 "decl %1\\t" // 区切り文字であれば、ESI--、区切り文字を指します。
37. "cmpb $0,(%1)\n\t" // Is this a NULL character?
38. "je 5f\n\t" // If yes, jump forward to label 5.
39. "movb $0,(%1)\n\t" // If not, replaced the separator with a NULL char.
40. "incl %1\n\t" // ESI points to next char in str1 (remaining str).
41. "jmp 6f\n" // Jump forward to label 6.
42. "5:\txorl %1,%1\n" // Clear ESI.
43. "6:\tcmpb $0,(%0)\n\t" // Does the EBX pointer point to a NULL char?
44. "jne 7f\n\t" // If not, jump forward to label 7.
45. "xorl %0,%0\n" // If yes, let the EBX = NULL. (end of str1)
46. "7:\ttestl %0,%0\n\t" // EBX is NULL?
47. "jne 8f\n\t" // If not, jump forward to label 8, end return.
48. "movl %0,%1\n" // Set ESI to NULL, means no more token in str1.
49. "8:"
50. :"=b" (\_\_res),"=S" (\_\_\_strtok)
51. :"0" (\_\_\_strtok),"1" (s),"g" (ct)
52. :"ax","cx","dx","di");
53. return \_\_res; // Returns pointer to the new token. 334 }

335

//// メモリのコピーです。ソースアドレスsrcからデスティネーションアドレスdestにnバイトをコピーします。

// Params: dest - デスティネーションアドレス, src - ソースアドレス, n - バイト数。

// %0 - ecx(n), %1 - esi(src), %2 - edi(dest) です。

1. 336 extern inline void \* memcpy(void \* dest,const void \* src, int n) /.
2. \_\_asm\_\_("cld\n\t" // Clear direction flag.
3. "rep\n\t" // Repeatedly copying ECX bytes, 340 "movsb" // From DS:[ESI++] to ES:[EDI++].
4. ::"c" (n),"S" (src),"D" (dest)
5. :"cx","si","di");
6. return dest; // Returns the dest address.
7. } 345

//// メモリの移動です。メモリのコピー操作と同じですが、移動の方向を考慮します。

// Params: dest - デスティネーションアドレス, src - ソースアドレス, n - バイト数。

// If (dest < src) then: %0 - ecx(n), %1 - esi(src), %2 - edi(dest);

// それ以外の場合： %0 - ecx(n), %1 - esi(src + n - 1 ), %2 - edi(dest + n - 1).

1. // この方向性のある操作は、コピー時のデータの重なりを防ぐためのものです。346 extern inline void \* memmove(void \* dest,const void \* src, int n) /.
2. if (dest<src)
3. \_\_asm\_\_("cld\n\t" // Clear direction flag.
4. "rep\n\t" // From DS:[ESI++] to ES:[EDI++],
5. "movsb" // Repeatedly copying ECX bytes.
6. ::"c" (n),"S" (src),"D" (dest)
7. :"cx","si","di");
8. else
9. \_\_asm\_\_("std\n\t" // Set the direction and start copying from the end. 356 "rep\n\t" // From DS:[ESI--] to ES:[EDI--],
10. "movsb" // Repeatedly copying ECX bytes.
11. ::"c" (n),"S" (src+n-1),"D" (dest+n-1)
12. :"cx","si","di");
13. return dest; // Returns the dest address.
14. } 362

//// 2つのメモリブロックのnバイトを比較し、NULL charに遭遇しても比較を中止しません。 // Params: cs - メモリブロック1のアドレス, ct - メモリブロック2のアドレス, count - 比較するバイト数。 // %0 - eax(\_\_res), %1 - eax(0), %2 - edi (mem block1), %3 - esi (mem block2), %4 - ecx(count).

1. // 363 extern inline int memcmp(const void \* cs,const void \* ct,int count) /.
2. register int \_\_res \_\_asm\_\_("ax");
3. \_\_asm\_\_("cld\n\t"
4. "repe\n\t" // Compare DS:[ESI++] with ES:[EDI++].
5. "cmpsb\n\t" // If equal, continue comparing. repeat ECX times.
6. "je 1f\n\t" // If all the same, jump to label 1, return 0 (EAX).
7. "movl $1,%%eax\n\t" // else set EAX to 1.
8. "jl 1f\n\t" // If value of mem block2 <block1, jump to label 1.
9. "negl %%eax\n" // else invert EAX value.
10. "1:"
11. :"=a" (\_\_res):"0" (0),"D" (cs),"S" (ct),"c" (count)
12. :"si","di","cx");
13. return \_\_res; // Returns the compare result (in EAX). 377 }

378

//// nバイトのメモリブロックの中から、指定した文字を探す。

// Params: cs - メモリブロックアドレス、c - 指定されたchar、count - 比較サイズ。 //%0 - edi(\_\_res), %1 - eax (char c), %2 - edi (メモリ・ブロック・アドレス cs), %3 - ecx (num of bytes).

1. // 最初に一致した場所へのポインタを返し、見つからなければNULLを返します。379 extern inline void \* memchr(const void \* cs,char c,int count) /.
2. register void \* \_\_res \_\_asm\_\_("di");
3. if (!count) // If memory block size is 0, then NULL is returned.
4. return NULL;
5. \_\_asm\_\_("cld\n\t"
6. "repne\n\t" // Compare the char in AL with the ES:[EDI++],
7. "scasb\n\t" // Repeat if it is not equal (up to ECX times).
8. "je 1f\n\t" // If equal, jump forward to label 1.
9. "movl $1,%0\n" // else set EDI to 1.
10. "1:\tdecl %0" // Let EDI point to the char searched.(or NULL).
11. :"=D" (\_\_res):"a" (c),"D" (cs),"c" (count)
12. :"cx");
13. return \_\_res; // Return the char pointer.
14. }

394

//// 指定された文字でメモリブロックを埋める。

// 's'が指すメモリ領域をchar 'c'で埋め、'count'バイトを埋めます。

1. //%0 - eax (char c), %1 - edi (メモリアドレス s), %2 - ecx (バイト数 count). 395 extern inline void \* memset(void \* s,char c,int count) / です。
2. \_\_asm\_\_("cld\n\t"
3. "rep\n\t" // Repeat ECX times,
4. "stosb" // Fill the char in AL into ES:[EDI++].
5. ::"a" (c),"D" (s),"c" (count)
6. :"cx","di");
7. return s; // Returns the memory block pointer. 403 }

404

405 #endif

406

## 14.11 termios.h

### 14.11.1 Functionality

termios.hファイルには、termiosデータ構造と一般的なターミナル・インターフェース設定のためのいくつかの関数プロトタイプを含む、ターミナルI/Oインターフェース定義が含まれています。これらの関数は、ターミナルのプロパティの読み取りや設定、ラインコントロール、ボーレートの読み取りや設定、ターミナルのフロントエンドプロセスのグループIDの読み取りや設定に使用されます。これは初期のLinuxヘッダーファイルですが、現在のPOSIX標準に完全に準拠しており、適切に拡張されています。

このファイルで定義されている2つの端末データ構造termioとtermiosは、2種類のUNIX系列（またはクローン）に属しており、termioはAT&T system Vで定義されており、termiosはPOSIX標準で規定されている。この2つの構造体は基本的に同じであるが，termioはモードフラグのセットを定義するために短い整数型を使用し，termiosはモードフラグのセットを定義するために長い整数型を使用する点が異なる。どちらの構造体も現在使用されているため、ほとんどのシステムでは互換性のためにサポートされています。なお、同様のsgtty構造体が以前に使用されたことがありますが、現在は使用されていません。

### 14.11.2 Code annotation

プログラム 14-10 linux/include/termios.h

1. #ifndef \_TERMIOS\_H
2. #define \_TERMIOS\_H

3

4 #include <sys/types.h>.

5

6 #define TTY\_BUF\_SIZE 1024 // tty内のバッファサイズ（バイト単位）。

7

8 /\* 0x54 は、これらを相対的にユニックにするためのマジックナンバー（'T'） \*/ 9

1. // 以下は、ttyのioctlが使用するコマンドセットで、ローバイトにエンコードされています。// 端末のtermios構造体の情報を取得する（tcgetattr()参照）。
2. #define TCGETS 0x5401 // Terminal Command GET Settings.
3. // 端末のtermios構造体に情報を設定する（TCSANOWのtcsetattr()参照）。
4. #define TCSETS 0x5402

// termiosに情報を設定する前に、出力されたすべてのデータを待つ必要がある

// 処理される（使い切られる）キュー。するパラメータを変更するために必要なコマンドです。

1. // 出力に影響を与える（tcsetattr()のTCSADRAINオプション参照）。(TCSETSW - TCSETS Wait)
2. #define TCSETSW 0x5403
3. // termios情報を設定する前に，出力キューのデータがすべて処理されるのを待ち， // 入力キューをフラッシュ（クリア）する必要があります（tcsetattr()のTCSAFLUSHオプションを参照）。
4. #define TCSETSF 0x5404
5. // termio構造体の属性情報を取る（tcgetattr()参照）。
6. #define TCGETA 0x5405
7. // termio構造体の属性情報を設定する（tcsetattr()、TCSANOWオプション参照）。
8. #define TCSETA 0x5406
9. // termioの属性情報を設定する前に、出力キューのデータがすべて処理される（ランアウトする）のを // 待つ必要があります。この種のコマンドは、出力に影響するパラメータを // 修正するために必要です（tcsetattr()、TCSADRAIN オプションを参照）。
10. #define TCSETAW 0x5407
11. // termioの属性を設定する前に、出力キューのデータがすべて処理されるのを待って、 // 入力キューをフラッシュ（使い切る）する必要があります（tcsetattr()のTCSAFLUSHオプションを参照）。
12. #define TCSETAF 0x5408
13. // 出力キューが処理される（空になる）のを待ちます。パラメータ値が0の場合、ブレークを送ります // （tcsendbreak()、tcdrain()を参照してください）。
14. #define TCSBRK 0x5409
15. // スタート／ストップ制御。パラメータ値が 0 の場合は出力が停止し、1 の場合は保留中の出力が再開され、2 の場合は入力が停止し、3 の場合は保留中の入力が再開されます（tcflow() を参照）。
16. #define TCXONC 0x540A
17. // 書き込まれたが送信されていないデータ、または受信されたがまだ // 読まれていないデータをフラッシュします。引数が0の場合は、入力キューがフラッシュ（クリア）され、1の場合は、出力キューがフラッシュされ、2の場合は、入力キューと出力キューがフラッシュされます（tcflush()参照）。
18. #define TCFLSH 0x540B
19. // 端末のシリアルラインの排他モードを設定します。
20. #define TIOCEXCL 0x540C // Terminal I/O Control Exclude.
21. // 端末のシリアルラインの排他モードをリセットします。
22. #define TIOCNXCL 0x540D
23. // ttyを制御端末として設定する。(TIOCNOTTY - ttyを制御端末として禁止する)。
24. #define TIOCSCTTY 0x540E

24 #define TIOCGPGRP 0x540F // 指定された端末デバイスプロセスのグループIDを取得する（tcgetpgrp()参照）。 24 #define TIOCGPGRP 0x540F // 端末IOコントロール取得PGRP。

1. // 指定した端末機器プロセスのグループIDを設定する（tcsetpgrp()参照）。
2. #define TIOCSPGRP 0x5410
3. // 送信されていない出力キューの文字数を返します。
4. #define TIOCOUTQ 0x5411
5. // 端末の入力をシミュレートします．このコマンドは，パラメータとして文字へのポインタを受け取り， // その文字が端末上で入力されたかのように見せかける。ユーザは，制御端末に対して， // スーパーユーザ権限または読み取り権限を持っていなければならない。
6. #define TIOCSTI 0x5412

27 #define TIOCGWINSZ 0x5413 // 端末デバイスのウィンドウサイズ情報を取得します（winsize構造体を参照）。 28 #define TIOCGWINSZ 0x5413

1. // 端末デバイスのウィンドウサイズ情報を設定する（winsize構造体を参照）。
2. #define TIOCSWINSZ 0x5414
3. // MODEMのステータスコントロールピンに設定されている現在のステータスビットフラグを返します (185-196行目参照)。
4. #define TIOCMGET 0x5415
5. // MODEM状態の個別制御ラインの状態（trueまたはfalse）を設定します。
6. #define TIOCMBIS 0x5416
7. // MODEM状態の個別制御ラインの状態をクリア（リセット）します。
8. #define TIOCMBIC 0x5417
9. // MODEMの状態制御ラインの状態を設定します。ビットが設定されると、MODEMに対応するステータスラインが // 有効になるように設定されます。
10. #define TIOCMSET 0x5418

// ソフトウェアのキャリア検出フラグを取得します（1 - オン、0 - オフ）。ローカルに接続された端末や

1. // 他のデバイスではソフトウェアキャリアフラグがオンになり、 // 端末やMODEM回線を使用するデバイスではオフになります。この2つのioctlコールを使えるようにするには、tty // ラインをO\_NDELAYモードでオープンする必要があり、open()がキャリアを待たないようにします。
2. #define TIOCGSOFTCAR 0x5419
3. // ソフトウェアキャリア検出フラグを設定します（1 - オン、0 - オフ）。
4. #define TIOCSSOFTCAR 0x541A
5. // 入力キューの中で、まだフェッチされていない文字の数を返します。
6. #define FIONREAD 0x541B
7. #define TIOCINQ FIONREAD

38

1. // ウィンドウサイズ属性構造体。スクリーンベースのアプリケーションに使用することができます。ioctlsのTIOCGWINSZ // およびTIOCSWINSZコマンドを使って、この情報を読み込んだり設定したりすることができます。
2. struct winsize {
3. unsigned short ws\_row; // window character rows.
4. unsigned short ws\_col; // window character colums.
5. unsigned short ws\_xpixel; // window width in pixels. 43 unsigned short ws\_ypixel; // window hight in pixels.

44 };

45

1. // AT&Tシステムのターミオ構造 V.
2. #define NCC 8 // the size of the control character array in termio.
3. struct termio {
4. unsigned short c\_iflag; /\* input mode flags \*/
5. unsigned short c\_oflag; /\* output mode flags \*/
6. unsigned short c\_cflag; /\* control mode flags \*/
7. unsigned short c\_lflag; /\* local mode flags \*/
8. unsigned char c\_line; /\* line discipline \*/
9. unsigned char c\_cc[NCC]; /\* control characters \*/

 };

55

1. // POSIXで定義されているtermios構造です。
2. #define NCCS 17 // the size of the control character array in termios.
3. struct termios {
4. tcflag\_t c\_iflag; /\* input mode flags \*/
5. tcflag\_t c\_oflag; /\* output mode flags \*/
6. tcflag\_t c\_cflag; /\* control mode flags \*/
7. tcflag\_t c\_lflag; /\* local mode flags \*/
8. cc\_t c\_line; /\* line discipline \*/
9. cc\_t c\_cc[NCCS]; /\* control characters \*/

 };

65

// 制御文字配列c\_cc[]内の項目のインデックスを表します。初期値である

1. // この配列の値は，include/linux/tty.hで定義されており， // プログラムで変更することができます．POSIX\_VDISABLE(0)が定義されている場合、配列の値が // \_POSIX\_VDISABLE と等しいときは、その配列に含まれる特殊文字が禁止されていることを意味します。
2. /\* c\_cc characters \*/
3. #define VINTR 0 // c\_cc[VINTR] = INTR (^C), \003.
4. #define VQUIT 1 // c\_cc[VQUIT] = QUIT (^\), \034.
5. #define VERASE 2 // c\_cc[VERASE] = ERASE (^H), \177.
6. #define VKILL 3 // c\_cc[VKILL] = KILL (^U), \025.
7. #define VEOF 4 // c\_cc[VEOF] = EOF (^D), \004.
8. #define VTIME 5 // c\_cc[VTIME] = TIME (\0), \0. timer value (see below).
9. #define VMIN 6 // c\_cc[VMIN] = MIN (\1), \1. timer value.
10. #define VSWTC 7 // c\_cc[VSWTC] = SWTC (\0), \0. switch char.
11. #define VSTART 8 // c\_cc[VSTART] = START (^Q), \021.
12. #define VSTOP 9 // c\_cc[VSTOP] = STOP (^S), \023.
13. #define VSUSP 10 // c\_cc[VSUSP] = SUSP (^Z), \032. suspend char.
14. #define VEOL 11 // c\_cc[VEOL] = EOL (\0), \0.
15. #define VREPRINT 12 // c\_cc[VREPRINT] = REPRINT (^R), \022.
16. #define VDISCARD 13 // c\_cc[VDISCARD] = DISCARD (^O), \017.
17. #define VWERASE 14 // c\_cc[VWERASE] = WERASE (^W), \027. word erase char.
18. #define VLNEXT 15 // c\_cc[VLNEXT] = LNEXT (^V), \026.
19. #define VEOL2 16 // c\_cc[VEOL2] = EOL2 (\0), \0.

84

1. // termiosの入力モードフィールドc\_iflagで使用される各種フラグの定数です。
2. /\* c\_iflag bits \*/
3. #define IGNBRK 0000001 // Ignore the BREAK condition when input.
4. #define BRKINT 0000002 // Generates a SIGINT signal on BREAK.
5. #define IGNPAR 0000004 // Ignore the character of the parity error.
6. #define PARMRK 0000010 // Mark parity error.
7. #define INPCK 0000020 // Allow input to check parity.
8. #define ISTRIP 0000040 // Mask the 8th bit of the character.
9. #define INLCR 0000100 // Map line feed NL to a carriage return CR.
10. #define IGNCR 0000200 // Ignore the CR.
11. #define ICRNL 0000400 // Map CR to NL.
12. #define IUCLC 0001000 // Convert uppercase chars to lowercase chars.
13. #define IXON 0002000 // Allow start/stop (XON/XOFF) output control.
14. #define IXANY 0004000 // Allow any character to restart the output.
15. #define IXOFF 0010000 // Allow start/stop (XON/XOFF) input control.  #define IMAXBEL 0020000 // Rings when the input queue is full.

100

1. // termiosの出力モードフィールドc\_oflagで使用される各種フラグの定数です。
2. /\* c\_oflag bits \*/
3. #define OPOST 0000001 // Perform output processing.
4. #define OLCUC 0000002 // Convert lowercase chars to uppercase chars.
5. #define ONLCR 0000004 // Map NL to CR-NL.
6. #define OCRNL 0000010 // Map CR to NL.
7. #define ONOCR 0000020 // The CR is not output in the column 0.
8. #define ONLRET 0000040 // The NL performs the function of carriage return.
9. #define OFILL 0000100 // Use fill chars when deferred instead of time delays.
10. #define OFDEL 0000200 // The fill char is DEL. The default is NULL if not set.
11. #define NLDLY 0000400 // Choose a line feed delay.
12. #define NL0 0000000 // NL delay type 0. 112 #define NL1 0000400 // NL delay type 1.

113 #define CRDLY 0003000 // キャリッジリターンディレイを選択します。114 #define CR0 0000000 // CR遅延タイプ0。

1. 115 #define CR1 0001000 // CR 遅延タイプ 1. 116 #define CR2 0002000 // CR 遅延タイプ 2.
2. #define CR3 0003000 // CR delay type 2.
3. #define TABDLY 0014000 // Select horizontal TAB delay.
4. #define TAB0 0000000 // TAB delay type 0.
5. #define TAB1 0004000 // TAB delay type 1.
6. #define TAB2 0010000 // TAB delay type 2.
7. #define TAB3 0014000 // TAB delay type 3.
8. #define XTABS 0014000 // Replace TAB with spaces, it is the number of spaces.
9. #define BSDLY 0020000 // Select the backspace (BS) delay.
10. #define BS0 0000000 // BS delay type 0.
11. #define BS1 0020000 // BS delay type 1.
12. #define VTDLY 0040000 // Select vertical tabulation delay.
13. #define VT0 0000000 // VT delay type 0.
14. #define VT1 0040000 // VT delay type 1.
15. #define FFDLY 0040000 // Select Form feed delay. 131 #define FF0 0000000 // FF delay type 0.

132 #define FF1 0040000 // FFの遅延タイプ1。

133

1. // コントロールモードフィールドc\_flagで使用されるフラグをtermios（8進数）で表したもの。
2. /\* c\_cflag bit meaning \*/
3. #define CBAUD 0000017 // Transmit baud rate bit mask. 136 #define B0 0000000 /\* hang up \*/ 137 #define B50 0000001 // baud rate 50.
4. #define B75 0000002
5. #define B110 0000003
6. #define B134 0000004
7. #define B150 0000005
8. #define B200 0000006
9. #define B300 0000007
10. #define B600 0000010
11. #define B1200 0000011
12. #define B1800 0000012
13. #define B2400 0000013
14. #define B4800 0000014
15. #define B9600 0000015
16. #define B19200 0000016
17. #define B38400 0000017 // baud rate 38400.
18. #define EXTA B19200 // Extended baud rate A.
19. #define EXTB B38400 // Extended baud rate B.

1. #define CSIZE 0000060 // Character bit width mask.
2. #define CS5 0000000 // 5 bits per character.
3. #define CS6 0000020 // 6 bits.
4. #define CS7 0000040 // 7 bits.
5. #define CS8 0000060 // 8 bits.
6. #define CSTOPB 0000100 // Set two stop bits instead of one.
7. #define CREAD 0000200 // Allow to receive.
8. #define PARENB 0000400 // Enable parity operation.
9. #define PARODD 0001000 // The input check is odd check.
10. #define HUPCL 0002000 // Hang up after the last process is closed. 164 #define CLOCAL 0004000 // Ignore the MODEM control lines.

165 #define CIBAUD 03600000 /\* 入力ボーレート(未使用) \*/

/ #define CRTSCTS 020000000000/ /\* フローコントロール \*/ (英語)

167

1. // termiosのローカルモードフィールドc\_lflagで使用されるフラグです。
2. /\* c\_lflag bits \*/
3. #define ISIG 0000001 // signal generated when revice INTR, QUIT, SUSP or DSUSP .
4. #define ICANON 0000002 // Enable canonical mode (cooked mode).
5. #define XCASE 0000004 // If ICANON is set, terminal displays uppercase chars.
6. #define ECHO 0000010 // Echo the input characters.
7. #define ECHOE 0000020 // If ICANON, ERASE/WERASE erase the previous char/word.
8. #define ECHOK 0000040 // If ICANON, KILL char will erase the current line.
9. #define ECHONL 0000100 // If ICANON, NL is echoed even if ECHO is not enabled. // The input/output queues are not flushed when the SIGINT and SIGQUIT signals are generated, // and the input queue is flushed when the SIGSUSP signal is generated.
10. #define NOFLSH 0000200
11. // 自身の制御端末を書き込もうとするバックグラウンドプロセスのプロセスグループにSIGTTOUシグナルを送る。
12. #define TOSTOP 0000400
13. // ECHOが設定されている場合、TAB、NL、START、STOP以外のASCII制御文字は、 // '^X'パターンにエコーバックされ、X値は制御値+0x40となります。
14. #define ECHOCTL 0001000
15. #define ECHOPRT 0002000 // If ICANON & IECHO, chars will be displayed when erasing.
16. #define ECHOKE 0004000 // If ICANON, KILL is echoed by erasing each char on he line. 181 #define FLUSHO 0010000 // Output is flushed. it can be toggled by DISCARD char.
17. #define PENDIN 0040000 // All chars in inqueue are reprinted when next char is read.
18. #define IEXTEN 0100000 // Enable implementation-defined input processing.

184

1. /\* modem lines \*/
2. #define TIOCM\_LE 0x001 // Line Enable.
3. #define TIOCM\_DTR 0x002 // Data Terminal Ready.
4. #define TIOCM\_RTS 0x004 // Request to Send.
5. #define TIOCM\_ST 0x008 // Serial Transfer.
6. #define TIOCM\_SR 0x010 // Serial Receive. 191 #define TIOCM\_CTS 0x020 // Clear To Send.
7. 192 #define TIOCM\_CAR 0x040 // キャリアディテクト。193 #define TIOCM\_RNG 0x080 // Ring indicate.
8. #define TIOCM\_DSR 0x100 // Data Set Ready.
9. #define TIOCM\_CD TIOCM\_CAR
10. #define TIOCM\_RI TIOCM\_RNG

197

1. /\* tcflow() and TCXONC use these \*/
2. #define TCOOFF 0 // Suspend output ("Terminal Control Output OFF").
3. #define TCOON 1 // Restart suspended output.
4. #define TCIOFF 2 // Send a STOP to stop device transmiting data to system.
5. #define TCION 3 // Send a START to start device transmiting to system.

203

1. /\* tcflush() and TCFLSH use these \*/
2. #define TCIFLUSH 0 // Clear the received data but do not read it. 206 #define TCOFLUSH 1 // Clear the output data but not transmit it.

207 #define TCIOFLUSH 2 // データの入出力をクリアするが、読み取り/送信はしない。

208

1. /\* tcsetattr uses these \*/
2. #define TCSANOW 0 // Enable settings now.
3. #define TCSADRAIN 1 // Change occurs after all output has been transmitted. 212 #define TCSAFLUSH 2 // Change occurs after input/output are all flushed.

213

// 以下の関数は、カーネルではなく、ビルド環境の関数ライブラリ libc.a に実装されています。ライブラリの実装では、これらの関数は、システムコールである ioctl()を呼び出すことで // 実装されている。ioctl()については、fs/ioctl.c プログラムを参照されたい。

1. // termios\_p で参照される termios 構造体の受信ボーレートを返す。
2. extern speed\_t cfgetispeed(struct termios \*termios\_p); // Returns the send baud rate in the termios structure.
3. extern speed\_t cfgetospeed(struct termios \*termios\_p);
4. // termios構造体の受信ボーレートを'speed'に設定する。
5. extern int cfsetispeed(struct termios \*termios\_p, speed\_t speed);
6. // termios構造体に送信ボーレートを設定する。
7. extern int cfsetospeed(struct termios \*termios\_p, speed\_t speed);
8. // fildesが指すオブジェクトの書き込まれたデータが送信されるのを待つ。
9. extern int tcdrain(int fildes);
10. // fildesが指すオブジェクトの受信/送信データを一時停止/再開する。
11. extern int tcflow(int fildes, int action);
12. // fildes で指定されたオブジェクトの、書き込まれたがまだ送信されていないすべてのデータと、 // 受信されたがまだ読まれていないすべてのデータを破棄します。
13. extern int tcflush(int fildes, int queue\_selector);
14. // ハンドルfildesに対応するオブジェクトのパラメータを取得し、termiosに保存します。
15. extern int tcgetattr(int fildes, struct termios \*termios\_p);

// 端末が非同期シリアル伝送を使用している場合、一連のゼロ値ビットが

1. // 一定時間連続して送信される。
2. extern int tcsendbreak(int fildes, int duration);
3. // 端末に関連するパラメータを設定するために、termios構造のデータを使用します。
4. extern int tcsetattr(int fildes, int optional\_actions,
5. struct termios \*termios\_p);

225

/ #endif

227

### 14.11.3 Information

#### 14.11.3.1 Control Character TIME、MIN

カノニカルモードフラグICANONが無効の場合、非カノニカルモード（ローモード）での入力となります。非正規モードでは、入力された文字は行に処理されず、入力された文字はすぐに読むことができ、キャリッジリターンやラインフィードなどの行を定義する文字を入力（ユーザーが入力）する必要はありません。そのため、消去や終了処理は行われません。termios構造体のMIN(c\_cc[VMIN])とTIME(c\_cc[VTIME])の設定は、受信した文字をどのように処理するか、関連するread()システムコールで何文字読み取るか、いつユーザープログラムに戻るかを決定するために使用される。

MINは、読み取り操作が満たされたとき（つまり、ユーザーに文字を返すとき）に必要な最小の文字量を表します。TIMEは、バースト転送や短期データ転送のタイムアウト値として、1/10秒単位でカウントされるタイミング値である。これら2つの制御文字の4つの組み合わせとその相互作用を以下に説明する。

◆min＞0、time＞0。

この場合、TIMEは文字間のタイマーとして機能し、最初の文字を受け取った後に機能を開始します。キャラクターを受け取るたびにリセットされ、再起動されます。MIN」と「TIME」の相互作用は

14.12 time.h

TIMEは以下の通りです。文字が受信されると、文字間タイマーが動作を開始します。タイマーが切れる前にMIN文字を受信した場合（文字を受信するたびにタイマーが再スタートすることに注意してください）、読み取り操作が満たされます。MIN文字を受信する前にタイマーが終了した場合、この時点で受信した文字がユーザーに返されます。なお、TIMEがタイムアウトした場合は、文字を受信して初めてタイマーが機能し始めるため、少なくとも1文字が返されます。つまり、この場合（MIN＞0、TIME＞0）は、MIN機構とTIME機構を作動させるために最初の文字を受信するまで、読み取り動作はスリープ状態となります。読み取った文字数が既存の文字数よりも少ない場合、タイマーは再起動されず、後続の読み取り操作は直ちに満足されます。

◆ min > 0, time = 0:

この場合、TIMEの値は0なので、タイマーは動作せず、MINだけが意味を持ちます。MIN文字を受信して初めて、待ち受けていた読み出し動作が満たされることになる（待ち受け動作はMIN文字を受信するまでスリープする）。これを利用してレコードベースのターミナルIOを読み取るプログラムは、読み取り動作が（任意に）無期限にブロックされることになる。

◆min＝0、time＞0。

この場合、MIN=0であるため、TIMEは文字間のタイマーとしてではなく、読み出し動作のタイマーとして機能し、読み出し動作の開始時に機能することになります。読み出し動作は、文字を受信するか、タイマーが切れると同時に成立する。なお、この場合、タイマーがタイムアウトした場合、文字は読み込まれません。タイマーがタイムアウトしない場合は、1文字読んだだけで読み取り動作が成立します。したがって、この場合、文字を待つために読み出し動作が無期限に（不確定に）ブロックされることはありません。読み出し動作の開始後、TIME\*0.10秒以内に1文字も受信しなかった場合、読み出し動作は0文字で戻ります。

◆min＝0、time＝0。

この場合、読み取り操作はすぐに戻ります。読み取りが要求された最小文字数、またはバッファキューに存在する文字数は、バッファに入力される文字数を待たずに返されます。

一般的に、非正規モードでは、この2つの値は、タイムアウトのタイミングの値と、文字数の値になります。MINは、読み出し動作を満足させるために必要な最小文字数を示す。TIMEは、10分の1秒単位でカウントされるタイミング値です。両方が設定されている場合、リードオペレーションは少なくとも1文字が読み込まれるまで待機し、MINの文字を読み込んだ後にリターンするか、TIMEのタイムアウトにより読み込まれた文字をリターンします。MINのみが設定されている場合、読み取り操作はMIN文字を読み取るまでリターンしません。TIMEのみが設定されている場合、リードは少なくとも1文字を読み取った後、すぐにリターンするか、タイムアウトします。どちらも設定されていない場合は、現在読み込まれているバイト数のみを表示して直ちに復帰します。

## 14.12 time.h

### 14.12.1 Functionality

time.hというヘッダーファイルは、時刻や日付を処理する関数を参照しています。MINIXには、「GMT（グリニッジ標準時、現在はUTC時間）とは何か、ローカル時間とは何か、その他の時間とは何かなど、時間の処理はもっと複雑である」という非常に興味深い記述があります。かつてウッシャー司教（1581-1656）が計算したとはいえ、聖書によれば、世界は紀元前4004年10月12日の午前9時に始まるとされている。しかし、UNIXの世界では、時間は空虚で混沌としていた以前の1970年1月1日0時（GMT）から始まっている」という。ここで、UTCとはユニバーサルタイムコードのこと。

14.12 time.h

このファイルは、標準Cライブラリのヘッダファイルの一つである。UNIXオペレーティングシステムの開発者の中には、当時アマチュア天文愛好家がいたため、UNIXシステムの時刻表現には特に厳しく、UNIX系や標準C互換機では、時刻や日付の表現や計算が特に複雑になっています。このファイルでは、1つの定数記号（マクロ）と4つの型、そしていくつかの時刻・日付演算変換関数を定義しています。また、このファイルで宣言されている関数の中には、カーネルには含まれていない標準Cライブラリで提供されている関数も含まれています。

### Linux 0.12カーネルでは、このファイルは主にtm構造体タイプをinit/main.cファイルとkernel/mktime.cファイルに提供しています。この構造体タイプは、カーネルがシステムのCMOSチップからリアルタイムクロック情報（カレンダータイム）を取得し、システムのブートタイムを設定できるようにするために使用されます。ブートタイムは、1970年1月1日0時からの経過時間（秒）であり、グローバル変数startup\_timeに格納され、カーネルが読み取るすべてのコードに適用されます。

### 14.12.2 Code annotation

プログラム 14-11 linux/include/time.h

1. #ifndef \_TIME\_H
2. #define \_TIME\_H 3
3. #ifndef \_TIME\_T
4. #define \_TIME\_T
5. typedef long time\_t; // The time from GMT on January 1, 1970 at midnight.
6. #endif

8

1. #ifndef \_SIZE\_T
2. #define \_SIZE\_T
3. typedef unsigned int size\_t;
4. #endif

13

1. #ifndef NULL
2. #define NULL ((void \*) 0)
3. #endif

17

18 #define CLOCKS\_PER\_SEC 100 // システムクロックの刻みの周波数、100HZ。

19

20 typedef long clock\_t; // プロセスの開始からシステムが渡すティック。

21

1. struct tm {
2. int tm\_sec; // Seconds [0, 59].
3. int tm\_min; // Minutes [0, 59].
4. int tm\_hour; // Hours [0, 59].
5. int tm\_mday; // The number of days in a month [0, 31].
6. int tm\_mon; // The number of months in a year [0, 11]. 28 int tm\_year; // The number of years since 1900.
7. int tm\_wday; // One day of the week [0, 6] (Sunday =0).
8. int tm\_yday; // One day in a year [0, 365].
9. int tm\_isdst; // Summer time sign.> 0 -in use; = 0 -not used; < 0 -invalid.
10. };

33

// うるう年かどうかを確認するマクロです。 34 #define \_\_isleap(year) ＼(^o^)

35 ((year) % 4 == 0 && ((year) % 100 ! = 0 || (year) % 1000 == 0))

36

// ここでは、時間操作のための関数プロトタイプをいくつか紹介します。

1. // プロセッサの使用時間を取得します。プログラムが使用したプロセッサの時間（ティック）を返します。
2. clock\_t clock(void);
3. // 時刻を取得します。1970.1.1:0:0:0（カレンダーの時刻）からの秒数を返します。
4. time\_t time(time\_t \* tp);
5. // 時間差を計算します。time2とtime1の間に経過した秒数を返します。
6. double difftime(time\_t time2, time\_t time1);
7. // tm構造体で表される時間をカレンダー時間に変換する。
8. time\_t mktime(struct tm \* tp);

41

1. // tm構造体の時刻を文字列に変換し、その文字列へのポインタを返します。
2. char \* asctime(const struct tm \* tp);
3. // カレンダーの時間を "Wed Jun 30 21:49:08:1993\n "のように文字列に変換します。
4. char \* ctime(const time\_t \* tp);
5. // カレンダーの時間をtm構造体で表されるUTC時間に変換します。
6. struct tm \* gmtime(const time\_t \*tp);
7. // カレンダーの時刻を、tm構造体で表される指定されたタイムゾーンの時刻に変換します。
8. struct tm \*localtime(const time\_t \* tp);
9. // tm構造体で表された時間は、フォーマット文字列'fmt'を用いて // 最大長'smax'の文字列に変換され、その結果が's'に格納されます。
10. size\_t strftime(char \* s, size\_t smax, const char \* fmt, const struct tm \* tp);
11. // この関数は、タイムゾーンに関連付けられた時間変換関数の中で // 自動的に呼び出されます。
12. void tzset(void);

48

49 #endif

50

## 14.13 unistd.h

### 14.13.1 Functionality

### 標準的なシンボルの定数や型は、unistd.hというヘッダーファイルで定義されています。プログラムの中でシンボル\_\_LIBRARY\_\_が定義されていると、カーネルのシステムコール番号や、インラインアセンブリマクロの\_syscall0()、\_syscall1()なども含まれます。

### 14.13.2 Code annotation

プログラム 14-12 linux/include/unistd.h

1. #ifndef \_UNISTD\_H
2. #define \_UNISTD\_H

3

1. /\* ok, this may be a joke, but I'm working on it \*/

// 以下のシンボル定数は，カーネルが準拠しているIEEE Standard 1003.1実装のバージョンを // 示しており，これは整数値である。

1. \_POSIX\_VERSION 198808L

6

1. \_POSIX\_CHOWN\_RESTRICTED /\* only root can do a chown (I think..) \*/
2. // NAME\_MAXより長いパス名はエラーとなり、自動的には切り捨てられません。
3. #define \_POSIX\_NO\_TRUNC /\* no pathname truncation (but see in kernel) \*/

// \_POSIX\_VDISABLEは、端末の一部の特殊文字の機能を制御するために使用されます。

1. // 端末のtermios構造体のc\_cc[]配列の文字コード値が， // \_POSIX\_VDISABLEの値と等しい場合，対応する特殊文字の機能が // 禁止されていることを意味する。
2. #define \_POSIX\_VDISABLE '\0' /\* character to disable things like ^C \*/

// システムの実装がジョブコントロールをサポートしていることを示す。 10 #define \_POSIX\_JOB\_CONTROL

// 各プロセスは、保存されたset-user-IDと保存されたset-group-IDを持っています。

1. 11 #define \_POSIX\_SAVED\_IDS /\* 何のために実装されたのか \*/ 12
2. #define STDIN\_FILENO 0 // Standard input file handle (descriptor) number.
3. #define STDOUT\_FILENO 1 // Standard output file handle number.
4. #define STDERR\_FILENO 2 // Standard error output file handle number.

16

1. #ifndef NULL
2. #define NULL ((void \*)0) // Define the value of a null pointer.
3. #endif

20

1. // access()関数では、以下のシンボル定数が使用されます。
2. /\* access \*/
3. #define F\_OK 0 // Check if the file exists.
4. #define X\_OK 1 // Check if it is executable (searchable).
5. #define W\_OK 2 // Check if it is writable. 25 #define R\_OK 4 // Check if it is readable.

26

1. // 以下のシンボル定数は、lseek() および fcntl() 関数で使用されます。
2. /\* lseek \*/
3. #define SEEK\_SET 0 // Set file read/write pointer to the offset.
4. #define SEEK\_CUR 1 // Set to the current value plus the offset.
5. #define SEEK\_END 2 // Set to the file length plus the offset.

31

1. // 以下のシンボル定数がsysconf()関数で使用されています。 32 /\* \_SC は System Configuration の略です。あまり使用しません \*/
2. #define \_SC\_ARG\_MAX 1 // The maximum number of arguments.
3. #define \_SC\_CHILD\_MAX 2 // The maximum number of child processes. 35 #define \_SC\_CLOCKS\_PER\_SEC 3 // Ticks per second.
4. #define \_SC\_NGROUPS\_MAX 4 // The maximum number of groups.
5. #define \_SC\_OPEN\_MAX 5 // The maximum number of opened files.
6. #define \_SC\_JOB\_CONTROL 6 // Job control.
7. #define \_SC\_SAVED\_IDS 7 // The saved identifier.
8. #define \_SC\_VERSION 8 // Version.

41

// pathconf()関数では，以下のシンボル定数を使用しています．

42 /\* より多くの(おそらく)設定可能なもの - 今度はパス名 \*/ \*/ 。

\_PC\_LINK\_MAX 1 // リンクの最大数です。

\_PC\_MAX\_CANON 2 // 通常のファイルの最大数です。

\_PC\_MAX\_INPUT 3 // 最大入力の長さ。

\_PC\_NAME\_MAX 4 // 名前の最大長です。

1. \_PC\_PATH\_MAX 5 // The maximum length of a path.
2. \_PC\_PIPE\_BUF 6 // Pipe buffer size.
3. \_PC\_NO\_TRUNC 7 // The file name is not truncated.
4. \_PC\_VDISABLE 8 // The specified control char is disabled.
5. \_PC\_CHOWN\_RESTRICTED 9 // Change the owner is restricted.

52

// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体 stat{} があります。 // と定数が含まれています。

// <sys/time.h> timeval構造体とitimerval構造体が定義されています。

// <sys/times.h> プロセス内の実行時間構造tmsとtimes()関数プロトタイプを定義します。

// <sys/utsname.h> システム名構造体のヘッダファイルです。

// <sys/resource.h> リソースファイル。プロセスが使用するシステムリソースの制限と利用に関する情報が含まれています。

1. // <utime.h> ユーザータイムのヘッダーファイルです。アクセス時間、修正時間の構造体と、utime() // プロトタイプが定義されています。
2. #include <sys/stat.h>
3. #include <sys/time.h>
4. #include <sys/times.h>
5. #include <sys/utsname.h>
6. #include <sys/resource.h>
7. #include <utime.h>



60 #ifdef \_\_LIBRARY\_\_ 61

1. // 以下は、カーネルが実装しているシステムコールシンボル定数で、システムコール関数表（include/linux/sys.h参照）のインデックスとして // 使用されます。
2. #define \_\_NR\_setup 0 /\* used only by init, to get system going \*/
3. #define \_\_NR\_exit 1
4. #define \_\_NR\_fork 2
5. #define \_\_NR\_read 3
6. #define \_\_NR\_write 4
7. #define \_\_NR\_open 5
8. #define \_\_NR\_close 6
9. #define \_\_NR\_waitpid 7
10. #define \_\_NR\_creat 8
11. #define \_\_NR\_link 9
12. #define \_\_NR\_unlink 10
13. #define \_\_NR\_execve 11
14. #define \_\_NR\_chdir 12
15. #define \_\_NR\_time 13
16. #define \_\_NR\_mknod 14
17. #define \_\_NR\_chmod 15
18. #define \_\_NR\_chown 16
19. #define \_\_NR\_break 17
20. #define \_\_NR\_stat 18
21. #define \_\_NR\_lseek 19
22. #define \_\_NR\_getpid 20
23. #define \_\_NR\_mount 21

 \_\_NR\_umount 22

85 \_\_NR\_setuid 23

 \_\_NR\_getuid 24

87 \_\_NR\_stime 25

 \_\_NR\_ptrace 26

\_\_NR\_alarm 27

\_\_NR\_fstat 28

\_\_NR\_pause 29

 \_\_NR\_utime 30 93 #define \_\_NR\_stty 31

1. #define \_\_NR\_gtty 32
2. #define \_\_NR\_access 33
3. #define \_\_NR\_nice 34
4. #define \_\_NR\_ftime 35
5. #define \_\_NR\_sync 36
6. #define \_\_NR\_kill 37
7. #define \_\_NR\_rename 38
8. #define \_\_NR\_mkdir 39
9. #define \_\_NR\_rmdir 40
10. #define \_\_NR\_dup 41
11. #define \_\_NR\_pipe 42
12. #define \_\_NR\_times 43
13. #define \_\_NR\_prof 44
14. #define \_\_NR\_brk 45
15. #define \_\_NR\_setgid 46
16. #define \_\_NR\_getgid 47
17. #define \_\_NR\_signal 48
18. #define \_\_NR\_geteuid 49
19. #define \_\_NR\_getegid 50
20. #define \_\_NR\_acct 51
21. #define \_\_NR\_phys 52
22. #define \_\_NR\_lock 53
23. #define \_\_NR\_ioctl 54
24. #define \_\_NR\_fcntl 55
25. #define \_\_NR\_mpx 56
26. #define \_\_NR\_setpgid 57
27. #define \_\_NR\_ulimit 58
28. #define \_\_NR\_uname 59
29. #define \_\_NR\_umask 60
30. #define \_\_NR\_chroot 61
31. #define \_\_NR\_ustat 62
32. #define \_\_NR\_dup2 63
33. #define \_\_NR\_getppid 64
34. #define \_\_NR\_getpgrp 65
35. #define \_\_NR\_setsid 66
36. #define \_\_NR\_sigaction 67
37. #define \_\_NR\_sgetmask 68
38. #define \_\_NR\_ssetmask 69
39. #define \_\_NR\_setreuid 70
40. #define \_\_NR\_setregid 71
41. #define \_\_NR\_sigsuspend 72
42. #define \_\_NR\_sigpending 73
43. #define \_\_NR\_sethostname 74
44. \_\_NR\_setrlimit 75
45. \_\_NR\_getrlimit 76
46. \_\_NR\_getusage 77
47. \_\_NR\_gettimeofday 78
48. \_\_NR\_settimeofday 79
49. \_\_NR\_getgroups 80
50. \_\_NR\_setgroups 81
51. \_\_NR\_select 82
52. \_\_NR\_symlink 83
53. #define \_\_NR\_lstat 84
54. #define \_\_NR\_readlink 85
55. #define \_\_NR\_uselib 86

149

// 以下は、システムコールの組み込みアセンブリマクロ関数の定義です。

// 引数のないシステムコールマクロ関数：type name(void).

// %0 - eax(\_\_res), %1 - eax(\_\_NR\_##name)。name はシステム・コールの名前で、結合されています。

// を \_\_NR\_ と組み合わせて、上記のシステムコールシンボル定数を形成し、システムコールテーブルの // 関数ポインタのアドレスに使用されます。マクロ定義では、もし2つの

// 2つのシンボルの間に連続した井戸記号「##」があれば、その2つのシンボルが

マクロが置き換えられたときに、 // 連結して新しいシンボルを形成します。例えば、以下の156行目の'\_\_NR\_##name' // パラメータ'name'(例えば'fork')を置き換えた後、プログラム内で最後に出現するのは、 // シンボル'\_\_NR\_fork'となります。

1. // 戻り値：戻り値が0以上の場合は値を返し、そうでない場合は、 // エラー番号errnoを設定して-1を返す。
2. #define \_syscall0(type,name) \
3. type name(void) \
4. { \
5. long \_\_res; \
6. \_\_asm\_\_ volatile ("int $0x80" \ // system interrupt 0x80.
7. : "=a" (\_\_res) \ // The return value is placed in eax(\_\_res).
8. : "0" (\_\_NR\_##name)); \ // The input is system interrupt number \_\_NR\_name.
9. if (\_\_res >= 0) \ // If value >=0, the value is returned directly.
10. return (type) \_\_res; \
11. errno = -\_\_res; \ // Otherwise set error number and return -1.
12. return -1; \
13. }

162

// 1つのパラメータを持つシステムコールのマクロ関数：type name(atype a)

1. // %0 - eax(\_\_res), %1 - eax(\_\_NR\_name), %2 - ebx(a)です。
2. #define \_syscall1(type,name,atype,a) \
3. type name(atype a) \
4. { \
5. long \_\_res; \
6. \_\_asm\_\_ volatile ("int $0x80" \
7. : "=a" (\_\_res) \
8. : "0" (\_\_NR\_##name),"b" ((long)(a))); \
9. if (\_\_res >= 0) \
10. return (type) \_\_res; \
11. errno = -\_\_res; \
12. return -1; \
13. }

175

// 2つのパラメータを持つシステムコールマクロ関数: type name(atype a, btype b)

// %0 - eax(\_\_res), %1 - eax(\_\_NR\_name), %2 - ebx(a), %3 - ecx(b)。

#define \_syscall2(type,name,atype,a,btype,b) ♪ ♪ type name(atype a,btype b) ♪ ♪ ♪

{ \

1. long \_\_res; \
2. \_\_asm\_\_ volatile ("int $0x80" \
3. : "=a" (\_\_res) \
4. : "0" (\_\_NR\_##name),"b" ((long)(a)),"c" ((long)(b))); \
5. if (\_\_res >= 0) \
6. return (type) \_\_res; \
7. errno = -\_\_res; \
8. return -1; \
9. }

188

// 3つのパラメータを持つシステムコールマクロ関数：type name(atype a, btype b, ctype c) // %0 - eax(\_\_res), %1 - eax(\_\_NR\_name), %2 - ebx(a), %3 - ecx(b), %4 - edx(c)...となります。

1. // このカーネルのシステムコールは，最大3つのパラメータを持ちます．それ以上のデータがある場合は， // データをバッファに入れて，バッファポインタをパラメータとしてカーネルに渡すことができます．
2. #define \_syscall3(type,name,atype,a,btype,b,ctype,c) \
3. type name(atype a,btype b,ctype c) \
4. { \
5. long \_\_res; \
6. \_\_asm\_\_ volatile ("int $0x80" \
7. : "=a" (\_\_res) \
8. : "0" (\_\_NR\_##name),"b" ((long)(a)),"c" ((long)(b)),"d" ((long)(c))); \
9. if (\_\_res>=0) \
10. return (type) \_\_res; \
11. errno=-\_\_res; \
12. return -1; \
13. }

201

202 #endif /\* \_\_LIBRARY\_\_ \*/

203

204 extern int errno; // グローバル変数であるエラー番号。

205

// 各システムコールの関数プロトタイプの定義を以下に示します。

1. // 詳細はinclude/linux/sys.hを参照してください。
2. int access(const char \* filename, mode\_t mode);
3. int acct(const char \* filename);
4. int alarm(int sec);
5. int brk(void \* end\_data\_segment);
6. void \* sbrk(ptrdiff\_t increment);
7. int chdir(const char \* filename);
8. int chmod(const char \* filename, mode\_t mode);
9. int chown(const char \* filename, uid\_t owner, gid\_t group);
10. int chroot(const char \* filename);
11. int close(int fildes);
12. int creat(const char \* filename, mode\_t mode);
13. int dup(int fildes);
14. int execve(const char \* filename, char \*\* argv, char \*\* envp);
15. int execv(const char \* pathname, char \*\* argv);
16. int execvp(const char \* file, char \*\* argv);
17. int execl(const char \* pathname, char \* arg0, ...);
18. int execlp(const char \* file, char \* arg0, ...);
19. int execle(const char \* pathname, char \* arg0, ...);

// 関数名の前のキーワード「volatile」は、コンパイラのgccに

// 関数は戻りません。これにより、gcc はより良いコードを生成することができます。さらに重要なことは、このキーワードを使用することで、 // ある種の（初期化されていない変数）に対する誤った警告の発生を避けることができます。これは、 // gcc の関数属性の記述と同じです。

1. // void do\_exit(int error\_code) \_\_attribute\_\_ ((noreturn));
2. volatile void exit(int status);
3. volatile void \_exit(int status);
4. int fcntl(int fildes, int cmd, ...);
5. int fork(void);
6. int getpid(void);
7. int getuid(void);
8. int geteuid(void); 231 int getgid(void);
9. int getegid(void);
10. int ioctl(int fildes, int cmd, ...);
11. int kill(pid\_t pid, int signal);
12. int link(const char \* filename1, const char \* filename2); 236 int lseek(int fildes, off\_t offset, int origin);
13. int mknod(const char \* filename, mode\_t mode, dev\_t dev);
14. int mount(const char \* specialfile, const char \* dir, int rwflag);
15. int nice(int val);
16. int open(const char \* filename, int flag, ...);
17. int pause(void);
18. int pipe(int \* fildes);
19. int read(int fildes, char \* buf, off\_t count);
20. int setpgrp(void);
21. int setpgid(pid\_t pid,pid\_t pgid);
22. int setuid(uid\_t uid);
23. int setgid(gid\_t gid);
24. void (\*signal(int sig, void (\*fn)(int)))(int);
25. int stat(const char \* filename, struct stat \* stat\_buf);
26. int fstat(int fildes, struct stat \* stat\_buf);
27. int stime(time\_t \* tptr);
28. int sync(void);
29. time\_t time(time\_t \* tloc);
30. time\_t times(struct tms \* tbuf);
31. int ulimit(int cmd, long limit);
32. mode\_t umask(mode\_t mask);
33. int umount(const char \* specialfile);
34. int uname(struct utsname \* name);
35. int unlink(const char \* filename);
36. int ustat(dev\_t dev, struct ustat \* ubuf);
37. int utime(const char \* filename, struct utimbuf \* times);
38. pid\_t waitpid(pid\_t pid,int \* wait\_stat,int options); 263 pid\_t wait(int \* wait\_stat);
39. int write(int fildes, const char \* buf, off\_t count);
40. int dup2(int oldfd, int newfd);
41. int getppid(void);
42. pid\_t getpgrp(void);
43. pid\_t setsid(void);
44. int sethostname(char \*name, int len);
45. int setrlimit(int resource, struct rlimit \*rlp);
46. int getrlimit(int resource, struct rlimit \*rlp);
47. int getrusage(int who, struct rusage \*rusage);
48. int gettimeofday(struct timeval \*tv, struct timezone \*tz);
49. 14.14 utime.h
50. int settimeofday(struct timeval \*tv, struct timezone \*tz);
51. int getgroups(int gidsetlen, gid\_t \*gidset);
52. int setgroups(int gidsetlen, gid\_t \*gidset);
53. int select(int width, fd\_set \* readfds, fd\_set \* writefds,
54. fd\_set \* exceptfds, struct timeval \* timeout);

279

280 #endif

281

## 14.14 utime.h

### 14.14.1 Functionality

utime.hファイルでは、ファイルのアクセスと変更の時間構造utimbuf{}と、関数プロトタイプutime()が定義されており、timeは秒単位です。

### 14.14.2 Code annotation

プログラム 14-13 linux/include/utime.h

1. #ifndef \_UTIME\_H
2. #define \_UTIME\_H

3

// <sys/types.h> 型のヘッダファイルです。基本的なシステムデータタイプとファイルシステムパラメータの

// の構造が定義されています。

4 #include <sys/types.h> /\* こんなことしちゃいけないのはわかってるんだけど......。\*/

5

1. struct utimbuf {
2. time\_t actime; // File access time. seconds from 1970.1.1:0:0:0.
3. time\_t modtime; // File modified time. seconds from 1970.1.1:0:0:0.
4. };

10

// ファイルのアクセス時間や修正時間を設定する関数です。

11 extern int utime(const char \*filename, struct utimbuf \*times);

12

13 #endif

14

## 14.15 include/asm/ ディレクトリのファイル

## 14.15 Files in the include/asm/ directory

linux/include/asm/ディレクトリ内のファイル一覧 14-2

Filename

Size

Last Modified Time

(

GMT)

Description

io.h

bytes

477

1991

-

08

-

10:17:51

07

memory.h

bytes

507

1991

-

06

-

15

20:54:44

segment.h

1366

bytes

1991

-

11

-

25

18:48:24

system.h

17

07

bytes

199

2

-

0

1

-

1

3

13:0

2

:

10



## 14.16 io.h

### 14.16.1 Functionality

ハードウェアのIOポートにアクセスする組み込みアセンブリのマクロ関数は、io.hファイルで定義されています： outb(), inb(), outb\_p(), inb\_p().前者2つの関数と後者2つの関数の主な違いは、後者のコードでは時間遅延のためにjmp命令が使用されていることです。

### 14.16.2 Code annotation

プログラム 14-14 linux/include/asm/io.h

/ //// ハードウェアポートのバイト出力機能です。

1. // パラメータ：value - 出力のバイト、port - ポート。
2. #define outb(value,port) \
3. \_\_asm\_\_ ("outb %%al,%%dx"::"a" (value),"d" (port))

3

4

//// ハードウェアポートのバイト入力機能。

1. // 入力バイトを返します。
2. #define inb(port) ({ \
3. unsigned char \_v; \
4. \_\_asm\_\_ volatile ("inb %%dx,%%al":"=a" (\_v):"d" (port)); \
5. \_v; \
6. }) 10

//// ハードウェアポートのバイト出力機能を遅延させる。2つのジャンプ文を使って、しばらくの間、遅延させます。

1. // パラメータ：value - エクスポートされるバイト、port - ポート。
2. #define outb\_p(value,port) \
3. \_\_asm\_\_ ("outb %%al,%%dx\n" \
4. "\tjmp 1f\n" \ // Jump forward to label 1 (the next statement).
5. "1:\tjmp 1f\n" \ // Jump forward to label 1. 15 "1:"::"a" (value),"d" (port)) 16

14.17 memory.h

//// ハードウェアポートのバイト入力機能を遅延させる。2つのジャンプ文を使って、しばらくの間、遅延させます。

1. // 入力バイトを返します。
2. #define inb\_p(port) ({ \
3. unsigned char \_v; \
4. \_\_asm\_\_ volatile ("inb %%dx,%%al\n" \
5. "\tjmp 1f\n" \ // Jump forward to label 1 (the next statement).
6. "1:\tjmp 1f\n" \ // Jump forward to label 1.
7. "1:":"=a" (\_v):"d" (port)); \
8. \_v; \
9. })

25

## 14.17 memory.h

### 14.17.1 Functionality

メモリコピーの組み込みアセンブリマクロ関数memcpy()は、string.hで定義されているmemcpy()と同じですが、後者は組み込みアセンブリC関数の形で定義されています。

### 14.17.2 Code annotation

プログラム 14-15 linux/include/asm/memory.h

1. /\*
2. \* NOTE!!! memcpy(dest,src,n) assumes ds=es=normal data segment. This
3. \* goes for all kernel functions (ds=es=kernel space, fs=local data,
4. \* gs=null), as well as for all well-behaving user programs (ds=es=
5. \* user data space). This is NOT a bug, as any user program that changes 6 \* es deserves to die if it isn't careful.

7 \*/

//// メモリブロックコピー。コピー元アドレスsrcからコピー先destにnバイトをコピーします。

1. // %0 - edi (address dest), %1 - esi (address src), %2 - ecx (number of bytes n). 8 #define memcpy(dest,src,n) ({ ˶ˆ꒳ˆ˵ )
2. void \* \_res = dest; \
3. \_\_asm\_\_ ("cld;rep;movsb" \ // Copy from DS:[ESI++] to ES:[EDI++],total ECX(n) bytes.
4. ::"D" ((long)(\_res)),"S" ((long)(src)),"c" ((long) (n)) \
5. :"di","si","cx"); \
6. \_res; \
7. })

15

## 14.18 segment.h

## 14.18 segment.h

### 14.18.1 Functionality

segment.hファイルには、インテルCPUのセグメントレジスターにアクセスする関数や、セグメントレジスターに関連するメモリ操作関数が定義されています。Linuxシステムでは、ユーザープログラムがシステムコールによってカーネルコードの実行を開始すると、カーネルコードはまず、GDTのカーネルデータセグメント記述子（セグメント値0x10）をレジスタDSとESにロードします（DSとESはカーネルデータセグメントへのアクセスに使用されます）。kernel/sys\_call.sの92～96行目を参照してください。したがって、カーネルコードを実行する際には、ユーザープログラム（タスク）のデータにアクセスするための特別な方法を使用する必要があります。このファイルにあるget\_fs\_byte()やput\_fs\_byte()などの関数は、ユーザープログラムのデータにアクセスするために特別に使用されています。

### 14.18.2 Code annotation

プログラム 14-16 linux/include/asm/segment.h

/ //// FSセグメントの指定されたアドレスのバイトを取得します。

// パラメータ： addr - 指定されたメモリアドレス。

// %0 - (返されたバイト \_v); %1 - (メモリアドレス addr).

1. // メモリFS:[addr]のバイトを返します。
2. extern inline unsigned char get\_fs\_byte(const char \* addr)
3. {
4. unsigned register char \_v; // a register variable for efficient access.

4

1. \_\_asm\_\_ ("movb %%fs:%1,%0":"=r" (\_v):"m" (\*addr));
2. return \_v;
3. }

8

//// FS セグメントの指定されたアドレスのワードを取得します。

// %0 - (返されたワード \_v); %1 - (メモリアドレス addr).

1. // メモリFS:[addr]のワードを返します。 9 extern inline unsigned short get\_fs\_word(const unsigned short \*addr)
2. {
3. unsigned short \_v;

12

1. \_\_asm\_\_ ("movw %%fs:%1,%0":"=r" (\_v):"m" (\*addr));
2. return \_v;
3. }

16

//// FS セグメントの指定されたアドレスのロングワードを取得します。

// %0 - (返されたロングワード \_v); %1 - (メモリアドレス addr).

1. // メモリFS:[addr]のロングワードを返します。
2. extern inline unsigned long get\_fs\_long(const unsigned long \*addr)
3. {
4. unsigned long \_v;

20

1. \_\_asm\_\_ ("movl %%fs:%1,%0":"=r" (\_v):"m" (\*addr)); \
2. 14.18 segment.h
3. return \_v;
4. }

24

//// 指定されたメモリアドレスにバイトをFSセグメントに入れます。

// パラメータ：val - バイト値、addr - メモリアドレス。

1. // %0 - レジスタ (バイト値 val); %1 - (メモリアドレス addr).
2. extern inline void put\_fs\_byte(char val,char \*addr)
3. {
4. \_\_asm\_\_ ("movb %0,%%fs:%1"::"r" (val),"m" (\*addr));
5. }

29

//// FS セグメントの指定されたメモリアドレスにワードを置く。

// パラメータ：val - ワード値、addr - メモリアドレス。

1. // %0 - レジスタ (ワード値 val); %1 - (メモリアドレス addr).
2. extern inline void put\_fs\_word(short val,short \* addr)
3. {
4. \_\_asm\_\_ ("movw %0,%%fs:%1"::"r" (val),"m" (\*addr));
5. }

34

//// FS セグメントの指定されたメモリアドレスにロングワードを置く。

// パラメータ：val - ロングワードの値、addr - メモリアドレス。

1. // %0 - レジスタ (ロングワード値 val); %1 - (メモリアドレス addr).
2. extern inline void put\_fs\_long(unsigned long val,unsigned long \* addr)
3. {
4. \_\_asm\_\_ ("movl %0,%%fs:%1"::"r" (val),"m" (\*addr));
5. }

39

1. /\*
2. \* Someone who knows GNU asm better than I should double check the followig.
3. \* It seems to work, but I don't know if I'm doing something subtly wrong.
4. \* --- TYT, 11/24/91
5. \* [ nothing wrong here, Linus ]
6. \*/

46

1. //// FSセグメントレジスタの値（セレクタ）を取得します。
2. extern inline unsigned long get\_fs()
3. {
4. unsigned short \_v;
5. \_\_asm\_\_("mov %%fs,%%ax":"=a" (\_v):);
6. return \_v;
7. }

53

1. //// DSセグメントレジスタの値を取得します。
2. extern inline unsigned long get\_ds()
3. {
4. unsigned short \_v;
5. \_\_asm\_\_("mov %%ds,%%ax":"=a" (\_v):);
6. return \_v;
7. }

60

//// FSセグメントレジスタの設定

61 extern inline void set\_fs(unsigned long val) 62 {。

1. \_\_asm\_\_("mov %0,%%fs"::"a" ((unsigned short) val));
2. }

65

66

## 14.19 system.h

### 14.19.1 Functionality

system.hファイルには、セグメントディスクリプター/割込みゲートディスクリプターを設定・変更するアセンブリマクロが埋め込まれています。その中で、関数 move\_to\_user\_mode() は、カーネルの初期化が終了したときに、手動で初期プロセス（タスク0）に切り替え（移動）する、つまり、特権レベル0のコードから特権レベル3のコードに実行するために使用されています。その方法は、図14-3に示すように、割込み呼び出しのリターン処理をシミュレートする、つまり、IRET命令を用いて特権レベルの変更とスタックスイッチを実装することで、CPUの実行制御フローを初期タスク0の環境に移動させるというものです。

0

31

Old

SS

Old

CS

Old

ESP

Old

EFLAGS

Old

EIP

SP0

-

SS:ESP in the TSS

SP1

-

Net SS:ESP before executing

IRET instruction

Stack exp

e

nsion

図14-3 IRET命令による特権レベル変更の実行

特権レベルの変更や制御の転送にこの方法が使われるのは、CPUの保護機構が原因です。コールゲート、インタラプト、トラップゲートによって、CPUは低レベル（特権レベル3など）のコードが高レベルのコードを呼び出したり、制御を移すことはできますが、その逆はできません。そこでカーネルは、IRET命令のリターンを高レベルから低レベルコードにシミュレートする方法を採用しています。

タスク0のコードを実行する前に、カーネルはまずスタックを設定して、特権レベルスイッチが割り込み呼び出し手順に入ったばかりのときのスタック内のコンテンツ配置をシミュレートします。その後、IRET命令が実行され、システムがタスク0の実行に移行します。IRET命令が実行されると、スタックの内容は図14-3のようになり、このときのスタックポインタはESP1となります。タスク 0 のスタックは、カーネルのスタックです。IRETが実行された後、タスク0に移動して実行されます。タスク0の記述子の特権レベルは3なので、スタック上のSS:ESPもポップアップされます。つまり、IRETの後、ESPはESP0と同じになります。なお、ここでの割込み復帰命令IRETは、本関数を実行する前にsched\_init()でフラグNTがリセットされているため、CPUにタスク切り替え動作を行わせません。NTがリセットされている状態でIRET命令を実行しても、CPUはタスク切り替え動作を行いません。タスク0の開始は、純粋に手動で行われていることがわかります。

タスク0は、データセグメントとコードセグメントが直接カーネルコードとデータ空間、つまり物理アドレス0から始まる640Kのメモリ空間にマッピングされている特殊なプロセスであり、そのスタックアドレスはカーネルコードが使用するスタックである。したがって、図のスタック内のオリジナルSSとオリジナルESPは、既存のカーネルのスタックポインタを直接スタックに押し込むことで形成されています。

system.hファイルの別の部分では、割り込みディスクリプターテーブルIDTの異なるタイプのディスクリプターエントリを設定するマクロが与えられています。\_set\_gate()は複数のパラメータを持つマクロで、割り込みゲートの記述子を設定するマクロset\_intr\_gate()、トラップゲートの記述子を設定するマクロset\_trap\_gate()、システムゲートの記述子を設定するマクロset\_system\_gate()から呼び出される汎用マクロである。IDTテーブルにおけるインタラプトゲートおよびトラップゲートディスクリプターのエントリのフォーマットを図14-4に示す。



31

12

Interrupt Gate

8

7

6

5

4

0

16

15

1

4

13

31

0

16

15

Procedure entry point offset

31..16

0

1

0

1

P

1

DPL

0

P

rocedure entry point offset

15

..

0

Selector

0

0

31

12

Trap

Gate

8

7

6

5

4

0

16

15

1

4

13

31

0

16

15

Procedure entry point offset

31..16

1

1

0

1

P

1

DPL

0

Procedure entry point offset

15

..

0

Selector

0

0

P

-

Segment present

D

P

L

-

Descriptor Privilege Level

4

0

4

0

図14-4 テーブルIDTにおけるインタラプトゲートおよびトラップゲート記述子のフォーマット

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特権レベルを表しています。割り込みゲートとトラップゲートの違いは、EFLAGSの割り込みイネーブルフラグIFの影響です。割り込みゲートディスクリプターで実行された割り込みはIFフラグをリセットするので、このようにして他の割り込みが現在の割り込み処理を妨害することを防ぎ、その後の割り込み終了命令IRETはIFフラグの元の値をスタックから復元します；トラップゲートで実行された割り込みはIFフラグに影響を与えません。

ディスクリプタを設定するための汎用マクロ\_set\_gate(gate\_addr, type, dpl, addr)では、パラメータgate\_addrは、ディスクリプタが配置されている物理メモリアドレスを指定する。type」は設定するディスクリプターの種類を示し、図14-4のディスクリプターフォーマットの6バイト目の下位4ビットに相当するので、type=14(0x0E)は割り込みゲートディスクリプター、type=15(0x0F)はトラップゲートディスクリプターを示す。パラメータ'dpl'は対応するディスクリプタフォーマットのDPL、'addr'はディスクリプタに対応する割込み処理プロセスの32ビットオフセットアドレスです。割り込み処理はカーネルのセグメントコードの一部であるため、それらのセグメントセレクタの値はすべて0x0008（EAXレジスタのハイワードで指定）となります。 system.hファイルの最後の部分は、一般セグメント記述子の内容を設定し、グローバル記述子テーブルGDTにタスクステートセグメント記述子とローカルテーブルセグメント記述子を設定するために使用されます。これらのマクロのパラメータの意味は、上記と同様です。

### 14.19.2 Code annotation

プログラム 14-17 linux/include/asm/system.h

/ //// ユーザーモードに移行して実行します。

1. // この関数は、IRET命令を使用して、カーネルモードから初期タスク0に移行します。
2. #define move\_to\_user\_mode() \
3. \_\_asm\_\_ ("movl %%esp,%%eax\n\t" \ // Save stack pointer ESP to EAX register.
4. "pushl $0x17\n\t" \ // First push the user stack segment SS,
5. "pushl %%eax\n\t" \ // then push the stack pointer ESP on to stack,
6. "pushfl\n\t" \ // and push the EFLAGS register too.
7. "pushl $0x0f\n\t" \ // Then push the code segment CS of task0,
8. "pushl $1f\n\t" \ // and push the offset (EIP) at label 1.
9. "iret\n" \ // Execute the IRET, causes control ret to label 1. 9 "1:\tmovl $0x17,%%eax\n\t" \ // At this point, kernel begins to execute task 0. 10 "movw %%ax,%%ds\n\t" \ // Initialize segment register points to the data
10. "movw %%ax,%%es\n\t" \ // segment of this local descriptor table.
11. "movw %%ax,%%fs\n\t" \
12. "movw %%ax,%%gs" \
13. :::"ax")

15

1. #define sti() \_\_asm\_\_ ("sti"::) // enable interrupt.
2. #define cli() \_\_asm\_\_ ("cli"::) // disable interrupt.
3. #define nop() \_\_asm\_\_ ("nop"::) // no op.

19

20 #define iret() \_\_asm\_\_ ("iret"::) // 割り込みのリターン。

21

//// ゲートディスクリプターを設定するためのマクロです。

// アドレスgate\_addrに配置されたゲートディスクリプターは、割込みに応じて設定されるか

// 例外処理手続きのアドレス addr、ゲート記述子のタイプ type、パラメータの特権 // レベル情報 dpl。(注：以下の「オフセット」は、カーネルのコードやデータセグメントに対する // 相対的なものです）。

// パラメータ： gate\_addr - ゲートディスクリプターのアドレス，type - ディスクリプタータイプのフィールド値。

// dpl - ディスクリプタの特権レベル; addr - オフセットアドレス。

// %0 - (type flag word combined by 'dpl', 'type'); %1 - (descriptor low 4 byte address); // %2 - (descriptor high 4 byte address); %3 - EDX (program offset address addr)。

1. // %4 - EAX (上位ワードにはセグメントセレクタ 0x0008 が含まれます)。
2. #define \_set\_gate(gate\_addr,type,dpl,addr) \
3. // オフセットアドレスのローワードとセレクタを組み合わせて、ディスクリプタ下位4バイト(EAX)にする。 // タイプフラグワードとオフセットハイワードをまとめてディスクリプターハイ4バイト(EDX)にする。 // 最後に，ゲートディスクリプタの下位4バイトと上位4バイトを別々に設定する。
4. \_\_asm\_\_ ("movw %%dx,%%ax\n\t" \
5. "movw %0,%%dx\n\t" \
6. "movl %%eax,%1\n\t" \
7. "movl %%edx,%2" \
8. : \
9. : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \ // %0
10. "o" (\*((char \*) (gate\_addr))), \ // %1
11. "o" (\*(4+(char \*) (gate\_addr))), \ // %2
12. "d" ((char \*) (addr)),"a" (0x00080000)) // %3, %4 32

//// 割り込みゲートの設定（後続の割り込みを自動的にマスクする）。

// パラメータ：n - 割り込み番号、addr - 割り込みプログラムのオフセットアドレス。

1. // ここで、'&idt[n]'は割込み記述子テーブルIDTの割込み番号nに対応するエントリのオフセット値であり、 // 割込み記述子のタイプは14、特権レベルは0である。
2. #define set\_intr\_gate(n,addr) \
3. \_set\_gate(&idt[n],14,0,addr)

35

//// トラップゲートの設定

1. // パラメータ： n - 割り込み番号、 addr - 割り込みプログラムのオフセットアドレス。 // '&idt[n]'はIDT内の割込み番号'n'の対応するエントリのオフセットである。 // 割込み記述子のタイプは15で、特権レベルは0である。
2. #define set\_trap\_gate(n,addr) \
3. \_set\_gate(&idt[n],15,0,addr)

38

//// システムのトラップゲートを設定します。

// set\_trap\_gate()で設定したディスクリプタの特権レベルは0で、ここでは3です。

1. // パラメータ： n - 割り込み番号、 addr - 割り込みプログラムのオフセットアドレス。 // '&idt[n]'はIDT内の割込み番号nに対応するエントリのオフセットである。 // 割込み記述子のタイプは15で、特権レベルは3である。
2. #define set\_system\_gate(n,addr) \
3. \_set\_gate(&idt[n],15,3,addr)

41

//// セグメントディスクリプターを設定します（カーネルでは使用しません）。

// パラメーター： gate\_addr - ディスクリプターのアドレス、type - ディスクリプターのタイプフィールド値、 // dpl - ディスクリプターの特権レベル、base - セグメントのベースアドレス、limit - セグメントの制限。

// セグメントディスクリプターのフォーマットを参照してください。ここでの割り当てオブジェクトが正しくないことに注意してください

//（逆）になっています。43行目は「\*((gate\_addr)+1)」とし、49行目は「\*(gate\_addr)」とします。ただし。

1. // このマクロはカーネルコードでは使用されていないので、Linusは気付いていません :-)
2. #define \_set\_seg\_desc(gate\_addr,type,dpl,base,limit) {\
3. \*(gate\_addr) = ((base) & 0xff000000) | \ // Descriptor lower 4 bytes
4. (((base) & 0x00ff0000)>>16) | \
5. ((limit) & 0xf0000) | \
6. ((dpl)<<13) | \
7. (0x00408000) | \
8. ((type)<<8); \
9. \*((gate\_addr)+1) = (((base) & 0x0000ffff)<<16) | \ // Descriptor high 4 bytes.
10. ((limit) & 0x0ffff); }

51

//// グローバルテーブルGDTにタスクステータスセグメント／ローカルテーブル記述子を設定。

// ステータスセグメントとローカルテーブルセグメントの長さは、ともに104バイトに設定されています。

// パラメータ： n - グローバルテーブルGDTのディスクリプターアイテムnに対応するアドレス // addr - ステートセグメント／ローカルテーブルが配置されているメモリのベースアドレス。

// type - ディスクリプタ内のフラグタイプバイトです。

// %0 - eax (アドレス addr); %1 - (記述子項目 n のアドレス); %2 - (のオフセット 2)

1. //記述子項目nのオフセット）、 %3 - （記述子項目nのオフセット4）、 %4 - （記述子項目nのオフセット5）、 %5 - （記述子項目nのオフセット6）、 %6 - （記述子項目nのオフセット7）となっている。
2. #define \_set\_tssldt\_desc(n,addr,type) \
3. \_\_asm\_\_ ("movw $104,%1\n\t" \ // The TSS length is stored in length field (0-th byte). 54 "movw %%ax,%2\n\t" \ // Put tne low word of base into the 2-3rd byte.
4. "rorl $16,%%eax\n\t" \ // Rotate base high word into AX (low word to high).
5. "movb %%al,%3\n\t" \ // Move low byte of the base high word to the 4th byte.
6. "movb $" type ",%4\n\t" \ // Move flag type byte into the 5th byte.
7. "movb $0x00,%5\n\t" \ // The sixth byte of the descriptor is set to zero.
8. "movb %%ah,%6\n\t" \ // Move high byte of the base high word to the 7th byte.
9. "rorl $16,%%eax" \ // Loop 16 bits to the right, EAX restores the original.
10. ::"a" (addr), "m" (\*(n)), "m" (\*(n+2)), "m" (\*(n+4)), \
11. "m" (\*(n+5)), "m" (\*(n+6)), "m" (\*(n+7)) \
12. )

64

//// グローバルテーブルGDTにタスクステータスセグメント（TSS）ディスクリプターを設定します。

1. // n - ディスクリプタへのポインタ、addr - ディスクリプタのエントリのセグメントのベースアドレス // です。TSS記述子のタイプは、0x89です。
2. #define set\_tss\_desc(n,addr) \_set\_tssldt\_desc(((char \*) (n)),addr,"0x89") //// Set the local table (LDT) descriptor in the global table.
3. // n - ディスクリプタへのポインタ、addr - ディスクリプタのエントリのセグメントのベースアドレス。ローカルテーブルセグメント記述子のタイプは0x82です。
4. #define set\_ldt\_desc(n,addr) \_set\_tssldt\_desc(((char \*) (n)),addr,"0x82")

67

## 14.20 include/linux/ディレクトリ内のファイル

## 14.20 Files in the directory include/linux/

リスト14-3 linux/include/linux/ディレクトリ内のファイル

Filename

Size

Last Modified Time

GMT)

(

Description

config.h

bytes

1545

1992

-

01

-

11

00:13:18

fdreg.h

2466

bytes

1991

-

11

-

02

10:48:44

fs.h

5754

bytes

1992

-

01

-

12

07:00:20

hdreg.h

1968

bytes

1991

-

10

-

13

15:32:15

head.h

304

bytes

1991

-

06

-

19:24:13

19

kernel.h

bytes

1036

1992

-

01

-

02:17:34

12

math\_emu.h

4924

bytes

1992

-

01

-

17:33:04

01

mm.h

1101

bytes

1992

-

01

-

13

15:46:41

sched.h

7351

bytes

1992

-

01

-

13

22:24:42



|  |  |
| --- | --- |
| sys.h | 3402 bytes 1992-01-13 21:42:37 |
| tty.h | 2801 bytes 1992-01-08 22:51:56 |

## 14.21 config.h

### 14.21.1 Functionality

Config.hはカーネルコンフィグレーションのヘッダーファイルで、unameコマンドが使用するマシンコンフィグレーション情報や、使用するキーボード言語タイプやハードディスクタイプ（HD\_TYPE）のオプションを定義しています。

### 14.21.2 Code annotation

プログラム 14-18 linux/include/linux/config.h

1. #ifndef \_CONFIG\_H
2. #define \_CONFIG\_H

3

1. /\*
2. \* Defines for what uname() should return
3. \*/
4. #define UTS\_SYSNAME "Linux"
5. #define UTS\_NODENAME "(none)" /\* set by sethostname() \*/
6. #define UTS\_RELEASE "" /\* patchlevel \*/
7. #define UTS\_VERSION "0.12"
8. #define UTS\_MACHINE "i386" /\* hardware type \*/

12

1. /\* Don't touch these, unless you really know what your doing. \*/

14.21 config.h

1. // 以下のシンボリック定数は、システムの起動時やカーネルのロード時のメモリの位置を示すために使用され、また、デフォルトの最大カーネルシステムモジュールサイズを示すために使用されます。
2. #define DEF\_INITSEG 0x9000 // The segment to which the boot sector be moved.
3. #define DEF\_SYSSEG 0x1000 // The segment to which the the system module loaded.
4. #define DEF\_SETUPSEG 0x9020 // The segment where the setup program is located.
5. #define DEF\_SYSSIZE 0x3000 // The maximum system module size (in units of 16).

18

1. /\*
2. \* The root-device is no longer hard-coded. You can change the default 21 \* root-device by changing the line ROOT\_DEV = XXX in boot/bootsect.s

22 \*/

23

1. /\*
2. \* The keyboard is now defined in kernel/chr\_dev/keyboard.S
3. \*/

27

1. /\*
2. \* Normally, Linux can get the drive parameters from the BIOS at
3. \* startup, but if this for some unfathomable reason fails, you'd
4. \* be left stranded. For this case, you can define HD\_TYPE, which 32 \* contains all necessary info on your harddisk.
5. \*
6. \* The HD\_TYPE macro should look like this:
7. \*
8. \* #define HD\_TYPE { head, sect, cyl, wpcom, lzone, ctl} 37 \*
9. 38 \* 2つのハードディスクがある場合は、情報をカンマで区切ってください。
10. \*
11. \* #define HD\_TYPE { h,s,c,wpcom,lz,ctl },{ h,s,c,wpcom,lz,ctl }
12. \*/
13. /\*
14. This is an example, two drives, first is type 2, second is type 3:

45

1. 46 #define HD\_TYPE { 4,17,615,300,615,8 }, { 6,17,615,300,615,0 } 47
2. NOTE: ctl is 0 for all drives with heads<=8, and ctl=8 for drives
3. with more than 8 heads.

50

51 BIOSにドライブの種類を知らせたい場合は、HD\_TYPEを未定義にしてください。これは普通のことです。

53 \*/

54

55 #endif

56

## 14.22 fdreg.h

## 14.22 fdreg.h

### 14.22.1 Functionality

fdreg.hというヘッダーファイルを使って、フロッピー装置でよく使われるパラメータや使用するI/Oポートなどを記述しています。フロッピーディスクドライブの制御は煩雑でコマンドも多いので、コードを読む前にマイコンインターフェースの原理に関する本を参照して、フロッピーディスクコントローラ（FDC）の仕組みを理解しておくとよいでしょう。そうすれば、ここでの定義はやはり合理的で整然としていると思うはずです。

フロッピーディスク装置をプログラミングする場合、各レジスタまたは複数のレジスタに1つずつ、計4つのポートにアクセスする必要があります。1.2Mのフロッピーディスクコントローラの場合、表14-1のようなポートがあります。

|  |  |  |
| --- | --- | --- |
| 表14-1 フロッピーディスクコントローラのポート  I/O port | Read/Write | Register name |
| 0x3f2  0x3f4  0x3f5 | 書き込みのみ  読み取り専用  Read/Write | デジタル出力（制御）レジスタ  FDCメインステータスレジスタ  FDC data register |
| 0x3f7 | Read only | Digital input register |
| 0x3f7 | Write only | Floppy disk control register (rate control) |

デジタル出力ポート（デジタルコントロールポート）は、8ビットのレジスタで、駆動モータオン、駆動選択、FDCのスタート／リセット、DMAや割り込み要求の有効／無効を制御します。

また，FDCメインステータスレジスタは，フロッピーディスクコントローラ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する8ビットのレジスタである。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が動作結果を得る前に読み込まれ，現在のFDCデータ・レジスタがレディであるかどうか，またデータ転送の方向を決定するために用いられる。

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場合，メインステートのDIO方向ビットは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場合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。各コマンドの1バイト目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。実行フェーズは，FDCの実行コマンドで指定された動作を行います。実行段階では，CPUは介入しない。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。結果フェーズ

14.22 fdreg.h

は，FDCコマンドの実行結果を得るために，CPUがFDCデータレジスタの戻り値を読み出すことです。返される結果データは、0～7バイトの長さです。リザルトデータが返ってこないコマンドの場合は，割り込みステータスコマンドの取得動作を検出するために，FDCにステータスを送る必要があります。

### 14.22.2 Code annotation

プログラム14-19 linux/include/linux/fdreg.h

1. /\*
2. \* This file contains some defines for the floppy disk controller. 3 \* Various sources. Mostly "IBM Microcomputers: A Programmers 4 \* Handbook", Sanches and Canton.
3. \*/
4. #ifndef \_FDREG\_H // This definition is used to exclude duplicate header files in code.
5. #define \_FDREG\_H

8

1. // いくつかのフロッピーディスク型関数のプロトタイプ宣言。
2. extern int ticks\_to\_floppy\_on(unsigned int nr);
3. extern void floppy\_on(unsigned int nr);
4. extern void floppy\_off(unsigned int nr);
5. extern void floppy\_select(unsigned int nr);
6. extern void floppy\_deselect(unsigned int nr);

14

1. // フロッピーディスクコントローラのためのいくつかのポートとシンボルの定義です。
2. /\* Fd controller regs. S&C, about page 340 \*/
3. #define FD\_STATUS 0x3f4 // Main status register port.
4. #define FD\_DATA 0x3f5 // Data port.
5. #define FD\_DOR 0x3f2 /\* Digital Output Register \*/
6. #define FD\_DIR 0x3f7 /\* Digital Input Register (read) \*/
7. #define FD\_DCR 0x3f7 /\* Diskette Control Register (write)\*/ 21
8. /\* Bits of main status register \*/
9. #define STATUS\_BUSYMASK 0x0F /\* drive busy mask \*/ // (one bit per driver).
10. #define STATUS\_BUSY 0x10 /\* FDC busy \*/
11. #define STATUS\_DMA 0x20 /\* 0- DMA mode \*/
12. #define STATUS\_DIR 0x40 /\* 0- cpu->fdc \*/
13. #define STATUS\_READY 0x80 /\* Data reg ready \*/

28

1. /\* Bits of FD\_ST0 \*/
2. #define ST0\_DS 0x03 /\* drive select mask \*/
3. #define ST0\_HA 0x04 /\* Head (Address) \*/
4. #define ST0\_NR 0x08 /\* Not Ready \*/
5. #define ST0\_ECE 0x10 /\* Equipment chech error \*/
6. #define ST0\_SE 0x20 /\* Seek end \*/ // or recalitrate end.
7. // 割り込みコードビット（割り込み理由）、00 - コマンドが正常に終了、01 - コマンドが異常に終了、10 - コマンドが無効、11 - FDDの準備状態が変化。
8. #define ST0\_INTR 0xC0 /\* Interrupt code mask \*/

36

1. /\* Bits of FD\_ST1 \*/
2. #define ST1\_MAM 0x01 /\* Missing Address Mark \*/
3. #define ST1\_WP 0x02 /\* Write Protect \*/
4. #define ST1\_ND 0x04 /\* No Data - unreadable \*/ // sector not found.
5. #define ST1\_OR 0x10 /\* OverRun \*/ // Data transfer timeout.
6. #define ST1\_CRC 0x20 /\* CRC error in data or addr \*/
7. #define ST1\_EOC 0x80 /\* End Of Cylinder \*/

44

1. /\* Bits of FD\_ST2 \*/
2. #define ST2\_MAM 0x01 /\* Missing Addess Mark (again) \*/
3. #define ST2\_BC 0x02 /\* Bad Cylinder \*/
4. #define ST2\_SNS 0x04 /\* Scan Not Satisfied \*/
5. #define ST2\_SEH 0x08 /\* Scan Equal Hit \*/
6. #define ST2\_WC 0x10 /\* Wrong Cylinder \*/
7. #define ST2\_CRC 0x20 /\* CRC error in data field \*/
8. #define ST2\_CM 0x40 /\* Control Mark = deleted \*/ 53
9. /\* Bits of FD\_ST3 \*/
10. #define ST3\_HA 0x04 /\* Head (Address) \*/
11. #define ST3\_TZ 0x10 /\* Track Zero signal (1=track 0) \*/
12. #define ST3\_WP 0x40 /\* Write Protect \*/

58

1. /\* Values for FD\_COMMAND \*/
2. #define FD\_RECALIBRATE 0x07 /\* move to track 0 \*/ // recalibrate.
3. #define FD\_SEEK 0x0F /\* seek track \*/
4. #define FD\_READ 0xE6 /\* read with MT, MFM, SKip deleted \*/
5. #define FD\_WRITE 0xC5 /\* write with MT, MFM \*/
6. #define FD\_SENSEI 0x08 /\* Sense Interrupt Status \*/ // Set the drive parameters (step rate, head unload time, etc.).
7. #define FD\_SPECIFY 0x03 /\* specify HUT etc \*/

66

1. /\* DMA commands \*/
2. #define DMA\_READ 0x46 // The mode word of DMA read disk (to port 12, 11).
3. #define DMA\_WRITE 0x4A // The mode word of DMA write disk.

70

71 #endif

72

## 14.23 fs.h

### 14.23.1 Functionality

### fs.h ヘッダファイルは、主にファイルシステムに関する定数や構造を定義しており、バッファキャッシュのバッフ ァブロックのデータ構造、MINIX 1.0 ファイルシステムのスーパーブロックや i ノードの構造、ファイルテーブルの 構造や一部のパイプライン演算マクロなどが含まれている。.

### 14.23.2 Code annotation

プログラム 14-20 linux/include/linux/fs.h

1. /\*
2. \* This file has definitions for some important file table 3 \* structures etc. 4 \*/

5

1. #ifndef \_FS\_H
2. #define \_FS\_H

8

9 #include <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。

10

1. /\* devices are as follows: (same as minix, so we can use the minix
2. \* file system. These are major numbers.)
3. \*
4. \* 0 - unused (nodev)
5. \* 1 - /dev/mem // memory device.
6. \* 2 - /dev/fd
7. \* 3 - /dev/hd
8. \* 4 - /dev/ttyx // tty serial terminal device.
9. \* 5 - /dev/tty
10. \* 6 - /dev/lp // printer device.
11. \* 7 - unnamed pipes
12. \*/

23

24 #define IS\_SEEKABLE(x) ((x)>=1 && (x)<=3) // デバイスが位置を検出できるかどうかを判定する。

25

1. #define READ 0
2. #define WRITE 1
3. #define READA 2 /\* read-ahead - don't pause \*/
4. #define WRITEA 3 /\* "write-ahead" - silly, but somewhat useful \*/ 30

31 void buffer\_init(long buffer\_end); // バッファキャッシュの初期化関数。

32

33 #define MAJOR(a) (((unsigned)(a))>>8) // デバイスのメジャー番号を取得します。 34 #define MINOR(a) ((a)&0xff) // デバイスのマイナー番号を取得する。

35

1. #define NAME\_LEN 14 // name length is 14.
2. #define ROOT\_INO 1 // root i-node number.

38

1. #define I\_MAP\_SLOTS 8 // the number of i-node bitmap slots (blocks).
2. #define Z\_MAP\_SLOTS 8 // the number of logical block bitmap slots.
3. #define SUPER\_MAGIC 0x137F // File system magic number.

42

1. #define NR\_OPEN 20 // The maximum number of files opened by process.
2. #define NR\_INODE 32 // The maximum number of I-nodes used by system.
3. #define NR\_FILE 64 // The maximum number of files in the system.
4. #define NR\_SUPER 8 // The maximum number of superblocks in system.
5. #define NR\_HASH 307 // Buffer hash-table array items.
6. #define NR\_BUFFERS nr\_buffers // The number of buffer blocks in the system. 49 #define BLOCK\_SIZE 1024 // Data block size (in bytes).
7. #define BLOCK\_SIZE\_BITS 10 // The number of bits used by the block size.
8. #ifndef NULL
9. #define NULL ((void \*) 0)
10. #endif

54

1. // 各ブロックが格納できるi-nodeの数(1024/32 = 32)です。
2. #define INODES\_PER\_BLOCK ((BLOCK\_SIZE)/(sizeof (struct d\_inode)))
3. // 各ブロックに格納できるディレクトリエントリの数（1024/16=64）です。
4. #define DIR\_ENTRIES\_PER\_BLOCK ((BLOCK\_SIZE)/(sizeof (struct dir\_entry))) 57
5. // パイプの先頭、末尾、サイズ、パイプが空か？ 58 #define PIPE\_READ\_WAIT(inode) ((inode).i\_wait)
6. #define PIPE\_WRITE\_WAIT(inode) ((inode).i\_wait2)
7. #define PIPE\_HEAD(inode) ((inode).i\_zone[0])
8. #define PIPE\_TAIL(inode) ((inode).i\_zone[1])
9. #define PIPE\_SIZE(inode) ((PIPE\_HEAD(inode)-PIPE\_TAIL(inode))&(PAGE\_SIZE-1))
10. #define PIPE\_EMPTY(inode) (PIPE\_HEAD(inode)==PIPE\_TAIL(inode))
11. #define PIPE\_FULL(inode) (PIPE\_SIZE(inode)==(PAGE\_SIZE-1))

65

1. #define NIL\_FILP ((struct file \*)0) // null file structure pointer.
2. #define SEL\_IN 1
3. #define SEL\_OUT 2
4. #define SEL\_EX 4

70

71 typedef char buffer\_block[BLOCK\_SIZE]; // バッファブロックの配列（1024項目）。

72

// バッファブロックヘッダのデータ構造。(とても重要!!)

1. // bhはコード上、buffer\_headの略語を表すことが多いです。
2. struct buffer\_head {
3. char \* b\_data; /\* pointer to data block (1024 bytes) \*/
4. unsigned long b\_blocknr; /\* block number \*/
5. unsigned short b\_dev; /\* device (0 = free) \*/
6. unsigned char b\_uptodate;
7. unsigned char b\_dirt; /\* 0-clean,1-dirty \*/ // Modified flag.
8. unsigned char b\_count; /\* users using this block \*/ 80 unsigned char b\_lock; /\* 0 - ok, 1 -locked \*/ 81 struct task\_struct \* b\_wait; // task wait queue.
9. struct buffer\_head \* b\_prev; // the previous block on the hash queue.
10. struct buffer\_head \* b\_next; // the next block on the hash queue.
11. struct buffer\_head \* b\_prev\_free; // the previous block on the free list.
12. struct buffer\_head \* b\_next\_free; // the next block on the free list.

 };

87

1. // ディスク上のI-nodeデータ構造（32バイト）。
2. struct d\_inode {
3. unsigned short i\_mode; // File type and attribute (rwx bit).
4. unsigned short i\_uid; // User id (file owner identifier).
5. unsigned long i\_size; // File size (in bytes).
6. unsigned long i\_time; // Modified time (from 1970.1.1:0, in seconds).
7. unsigned char i\_gid; // Group id (the group in which file owner belong).
8. unsigned char i\_nlinks; // Number of links (entries pointed to the i-node).
9. unsigned short i\_zone[9]; // logical block number array.

/ }; // 直接（0～6）、間接（7）、二次間接（8）のいずれかになります。

97

1. // これは、メモリ上のi-node構造です。最初の7項目は、d\_inodeと全く同じです。
2. struct m\_inode {
3. unsigned short i\_mode;
4. unsigned short i\_uid;
5. unsigned long i\_size;
6. unsigned long i\_mtime;
7. unsigned char i\_gid;
8. unsigned char i\_nlinks;
9. unsigned short i\_zone[9];
10. /\* these are in memory also \*/
11. struct task\_struct \* i\_wait; // task waiting queue for waiting for the i-node. 108 struct task\_struct \* i\_wait2; /\* for pipes \*/
12. unsigned long i\_atime; // i-node access time.
13. unsigned long i\_ctime; // i-node change time.
14. unsigned short i\_dev; // the device where the i-node is located.
15. unsigned short i\_num; // i-node number.
16. unsigned short i\_count; // i-node used count, 0 indicates it’s idle (free).
17. unsigned char i\_lock; // lock flag.
18. unsigned char i\_dirt; // modified flag
19. unsigned char i\_pipe; // the i-node is used for pipe.
20. unsigned char i\_mount; // mount flag.
21. unsigned char i\_seek; // used for lseek method of the file.
22. unsigned char i\_update; // updated flag.  };

121

1. // ファイル構造（ファイルハンドルとi-nodeとの関係を確立するために使用される） 122 struct file {.
2. unsigned short f\_mode; // file mode (RW bits).
3. unsigned short f\_flags; // file open and control flags.
4. unsigned short f\_count; // file reference count.
5. struct m\_inode \* f\_inode; // file’s i-node.
6. off\_t f\_pos; // read and write position in the file.  };

129

1. // インメモリーディスクのスーパーブロック構造。
2. struct super\_block {
3. unsigned short s\_ninodes; // number of i-nodes in the file system.
4. unsigned short s\_nzones; // number of zones (logica blocks).
5. unsigned short s\_imap\_blocks; // number of data blocks occupied by i-node map.
6. unsigned short s\_zmap\_blocks; // number of blocks occupied by logical block map.
7. unsigned short s\_firstdatazone; // the first block number in the data zone.
8. unsigned short s\_log\_zone\_size; // Log2(number of data blocks / logical block).
9. unsigned long s\_max\_size; // the maximum file size.
10. unsigned short s\_magic; // file system magic number.
11. /\* These are only in memory \*/
12. struct buffer\_head \* s\_imap[8]; // an array of i-node bitmap buffer blocks.
13. struct buffer\_head \* s\_zmap[8]; // an array of logical block bitmap buffer blocks.
14. unsigned short s\_dev; // the device number of the super block.
15. struct m\_inode \* s\_isup; // The root i-node of the mounted file system.
16. struct m\_inode \* s\_imount; // The i-node to which the file system is installed.
17. unsigned long s\_time; // modified time.
18. struct task\_struct \* s\_wait; // the wait queue for processes waiting for it.
19. unsigned char s\_lock; // the lock flag.
20. unsigned char s\_rd\_only; // read only flag.
21. unsigned char s\_dirt; // tirty flag.
22. };

151

1. // ディスク上のスーパーブロック構造。上記131～138行目と全く同じです。
2. struct d\_super\_block {
3. unsigned short s\_ninodes;
4. unsigned short s\_nzones;
5. unsigned short s\_imap\_blocks;
6. unsigned short s\_zmap\_blocks;
7. unsigned short s\_firstdatazone;
8. unsigned short s\_log\_zone\_size;
9. unsigned long s\_max\_size;
10. unsigned short s\_magic;
11. };

162

1. // ファイルディレクトリのエントリ構造（16バイト）。
2. struct dir\_entry {
3. unsigned short inode; // i-node number.
4. char name[NAME\_LEN]; // File name, NAME\_LEN = 14.  };

167

1. extern struct m\_inode inode\_table[NR\_INODE]; // i-node table (32 entries).
2. extern struct file file\_table[NR\_FILE]; // file table (64 items).
3. extern struct super\_block super\_block[NR\_SUPER]; // super block array (8 items).
4. extern struct buffer\_head \* start\_buffer; // starting location of the buffer cache.  extern int nr\_buffers; // the number of buffers.

173

//// ディスク操作機能の試作品を紹介します。

1. // ドライブに入っているフロッピーディスクが変わったかどうかを確認します。
2. extern void check\_disk\_change(int dev);
3. // フロッピーディスクの交換状況を確認します。フロッピーディスクが交換されていれば1を、そうでなければ0を返します。
4. extern int floppy\_change(unsigned int nr);

// ドライブを起動するまでの待ち時間を設定します（待ち時間タイマーを設定）。176 extern int ticks\_to\_floppy\_on(unsigned int dev);

1. // 指定したドライブを起動する。
2. extern void floppy\_on(unsigned int dev);
3. // 指定されたフロッピードライブの電源を切る。
4. extern void floppy\_off(unsigned int dev);

//// ファイルシステムの運用管理に関する機能のプロトタイプを以下に示します。

1. // i-nodeで指定されたファイルのサイズは0に切り捨てられます。
2. extern void truncate(struct m\_inode \* inode); // Refresh (synchronize) the i-node information.
3. extern void sync\_inodes(void); // Waiting for the specified i-node.
4. extern void wait\_on(struct m\_inode \* inode);
5. // ブロックビットマップ操作。デバイス上のデータブロック「block」のブロック番号を取得します。
6. extern int bmap(struct m\_inode \* inode,int block);
7. // block 'block' に対応するデバイス上の論理ブロックを作成し、デバイス上の論理ブロック番号を返す。
8. extern int create\_block(struct m\_inode \* inode,int block); // Get the i-node number of the specified path name.
9. extern struct m\_inode \* namei(const char \* pathname);
10. // シンボリックリンクを辿らずに、指定されたパス名のi-nodeを取得する。
11. extern struct m\_inode \* lnamei(const char \* pathname); // Prepare to open the file according to the path name.
12. extern int open\_namei(const char \* pathname, int flag, int mode,
13. struct m\_inode \*\* res\_inode);

// i-nodeをリリース（戻す）する（デバイスを書き込む）。

1. extern void iput(struct m\_inode \* inode);
2. // デバイスからi-nodeを読み込みます。
3. extern struct m\_inode \* iget(int dev,int nr);
4. // inodeテーブルからidle i-nodeエントリを取得します。
5. extern struct m\_inode \* get\_empty\_inode(void);
6. // パイプのi-nodeを取得（適用）します。i-nodeへのポインタを返します（NULLの場合は失敗）。
7. extern struct m\_inode \* get\_pipe\_inode(void);
8. // ハッシュテーブルの中から，指定されたデータブロックを見つけます．バッファヘッドポインタを返します。
9. extern struct buffer\_head \* get\_hash\_table(int dev, int block);

// 指定されたブロックをデバイスから読み込む（ハッシュテーブルの最初のルック）。193 extern struct buffer\_head \* getblk(int dev, int block);

// 低レベルのリード/ライトブロック機能。

194 extern void ll\_rw\_block(int rw, struct buffer\_head \* bh);

// 低レベルのリード/ライトデータページ、つまり一度に4つのデータブロックを扱うことができます。195 extern void ll\_rw\_page(int rw, int dev, int nr, char \* buffer);

1. // 指定されたバッファブロックを解放します。
2. extern void brelse(struct buffer\_head \* buf);
3. // 指定されたデータブロックを読み込む。
4. extern struct buffer\_head \* bread(int dev,int block);
5. // 指定されたメモリアドレスに1ページ（4つのバッファブロック）を読み込む。
6. extern void bread\_page(unsigned long addr,int dev,int b[4]);
7. // 指定されたデータブロックを読み込み、後で読み込まれるブロックをマークします。
8. extern struct buffer\_head \* breada(int dev,int block,...);
9. // デバイスdevからディスクブロックを要求し、論理ブロック番号を返す
10. extern int new\_block(int dev);
11. // デバイスデータエリアのロジックブロックを解放します。ロジックブロックのビットマップビットをリセットします。
12. extern void free\_block(int dev, int block);
13. // デバイスに新しいi-nodeを作成し、そのi-node番号を返します。
14. extern struct m\_inode \* new\_inode(int dev);

203 extern void free\_inode(struct m\_inode \* inode);; // i-nodeを解放(free)します(ファイルを削除する場合)。203 extern void free\_inode(struct m\_inode \* inode);

1. // 指定されたデバイスバッファを更新します。
2. extern int sync\_dev(int dev);
3. // 指定されたデバイスのスーパーブロックを取得します。
4. extern struct super\_block \* get\_super(int dev); 206 extern int ROOT\_DEV; // root device number.

207

// ルートファイルシステムをマウントします。/ extern void mount\_root(void);

209

/ #endif

211

## 14.24 hdreg.h

### 14.24.1 Functionality

### hdreg.hファイルは、主にハードディスク・コントローラをプログラムするためのいくつかのコマンド定数記号を定義しています。これには、コントローラ・ポート、ハードディスク・ステータス・レジスタの各ビットのステータス、コントローラ・コマンド、エラー・ステータスの定数記号が含まれます。また、ハードディスク・パーティション・テーブルのデータ構造も示されています。

### 14.24.2 Code annotation

プログラム 14-21 linux/include/linux/hdreg.h

1. /\*
2. \* This file contains some defines for the AT-hd-controller.
3. \* Various sources. Check out some definitions (see comments with 4 \* a ques).
4. \*/
5. #ifndef \_HDREG\_H
6. #define \_HDREG\_H

8

1. /\* Hd controller regs. Ref: IBM AT Bios-listing \*/
2. #define HD\_DATA 0x1f0 /\* \_CTL when writing \*/
3. #define HD\_ERROR 0x1f1 /\* see err-bits \*/
4. #define HD\_NSECTOR 0x1f2 /\* nr of sectors to read/write \*/
5. #define HD\_SECTOR 0x1f3 /\* starting sector \*/
6. #define HD\_LCYL 0x1f4 /\* starting cylinder \*/
7. #define HD\_HCYL 0x1f5 /\* high byte of starting cyl \*/
8. #define HD\_CURRENT 0x1f6 /\* 101dhhhh , d=drive, hhhh=head \*/
9. #define HD\_STATUS 0x1f7 /\* see status-bits \*/
10. #define HD\_PRECOMP HD\_ERROR /\* same io address, read=error, write=precomp \*/
11. #define HD\_COMMAND HD\_STATUS /\* same io address, read=status, write=cmd \*/ 20

21 #define HD\_CMD 0x3f6 // コントロールレジスタポート。

22

1. /\* Bits of HD\_STATUS \*/
2. #define ERR\_STAT 0x01 // Command execution error.
3. #define INDEX\_STAT 0x02 // Received the index.
4. #define ECC\_STAT 0x04 /\* Corrected error \*/ // ECC checksum error.
5. #define DRQ\_STAT 0x08 // Request service.
6. #define SEEK\_STAT 0x10 // End of seek.
7. #define WRERR\_STAT 0x20 // Drive error.
8. #define READY\_STAT 0x40 // Drive ready.
9. #define BUSY\_STAT 0x80 // Controller busy.

32

1. /\* Values for HD\_COMMAND \*/
2. #define WIN\_RESTORE 0x10 // Drive reset (recalibration).
3. #define WIN\_READ 0x20 // Read sector.
4. #define WIN\_WRITE 0x30 // Write sector.
5. #define WIN\_VERIFY 0x40 // Sector verify.
6. #define WIN\_FORMAT 0x50 // Format track.
7. #define WIN\_INIT 0x60 // Controller initialize.
8. #define WIN\_SEEK 0x70 // Seek track.
9. #define WIN\_DIAGNOSE 0x90 // Controller diagnose.
10. #define WIN\_SPECIFY 0x91 // Establish drive parameters.

43

44 /\* HD\_ERROR用のビット \*/

// 診断コマンドを実行した場合、その意味は他のコマンドとは異なり、以下のようになります。

// ==============================================================================

// 診断コマンド その他のコマンド

// ------------------------------------------------------------------------------

// 0x01 エラーなし データマーク消失 // 0x02 コントローラエラー トラック0エラー

// 0x03 セクターバッファエラー

// 0x04 ECC パートエラー コマンドアボート // 0x05 制御プロセスエラー

// 0x10 IDが見つかりません。 // 0x40 ECCエラー。

// 0x80 Bad sector

1. //--------------------------------------------------------------------------------
2. #define MARK\_ERR 0x01 /\* Bad address mark ? \*/
3. #define TRK0\_ERR 0x02 /\* couldn't find track 0 \*/
4. #define ABRT\_ERR 0x04 /\* ? \*/
5. #define ID\_ERR 0x10 /\* ? \*/
6. #define ECC\_ERR 0x40 /\* ? \*/
7. #define BBD\_ERR 0x80 /\* ? \*/

51

1. // ハードディスクのパーティションテーブル構造は、以下のリストの後に情報を参照してください。
2. struct partition {
3. unsigned char boot\_ind; /\* 0x80 - active (unused) \*/
4. unsigned char head; /\* ? \*/
5. unsigned char sector; /\* ? \*/
6. unsigned char cyl; /\* ? \*/
7. unsigned char sys\_ind; /\* ? \*/
8. unsigned char end\_head; /\* ? \*/
9. unsigned char end\_sector; /\* ? \*/
10. unsigned char end\_cyl; /\* ? \*/
11. unsigned int start\_sect; /\* starting sector counting from 0 \*/ 62 unsigned int nr\_sects; /\* nr of sectors in partition \*/

63 };

64

65 #endif

66

### 14.24.3 Information

#### 14.24.3.1 Hard disk partition table

データの管理を容易にしたり、複数のOSでハードディスクの資源を共有するために、ハードディスクを論理的に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は4つのエントリで構成されており、各エントリは16バイトで1つのパーティションの情報に対応しています。各エントリは、表14-2に示すように、パーティションサイズ、シリンダ番号、トラック番号、セクタ番号を持っています。パーティションテーブルは、ハードディスクの0シリンダー0ヘッドの第1セクターの0x1BE--0x1FDの位置に格納されています。

|  |  |  |  |
| --- | --- | --- | --- |
| 表14-2 ハードディスク・パーティション・テーブルのエントリ構造  Offset | Name | Size | Description |
| 0x00 | boot\_ind | 1 byte | ブートインデックス。4つのパーティションのうち、一度に起動できるのは1つのパーティションのみです。  0x00 - 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. |

|  |  |  |  |
| --- | --- | --- | --- |
| 14.25ヘッド.H  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. |

## 14.25 head.h

### 14.25.1 Functionality

head.hヘッダーファイルは、インテルCPUにおけるディスクリプターの単純な構造を定義し、ディスクリプターのアイテム番号を指定します。

### 14.25.2 Code annotation

プログラム 14-22 linux/include/linux/head.h

1. #ifndef \_HEAD\_H
2. #define \_HEAD\_H 3
3. // セグメントディスクリプターのデータ構造を以下に定義する。この構造では、各ディスクリプターが8バイトで構成され、各ディスクリプターテーブルが256エントリであることだけが // 記載されています。
4. typedef struct desc\_struct {
5. unsigned long a,b;
6. } desc\_table[256];

7

// ページング管理機構が使用するメモリページのディレクトリテーブルを宣言します。それぞれの

1. // のディレクトリエントリは4バイトです。このカーネルでは、テーブルは物理アドレス0から始まります。
2. extern unsigned long pg\_dir[1024];
3. extern desc\_table idt,gdt; // Interrupt descriptor table, global descriptor table.

10

1. #define GDT\_NUL 0 // The 0th item of the GDT, not used.
2. #define GDT\_CODE 1 // The first item is the kernel code segment descriptor.
3. #define GDT\_DATA 2 // The second item is the kernel data segment descriptor. 14 #define GDT\_TMP 3 // The third item is system segment descriptor, not used.

15

1. #define LDT\_NUL 0 // The 0th item of the LDT, not used.
2. #define LDT\_CODE 1 // The first item is the user code segment descriptor . 18 #define LDT\_DATA 2 // The second item is the user data segment descriptor.

19

20 #endif

21

## 14.26 kernel.h

## 14.26 kernel.h

14.26.1 **Functionality**

### kernel.hファイルには、カーネルで使用される関数のプロトタイプが定義されています。

### 14.26.2 Code annotation

プログラム 14-23 linux/include/linux/kernel.h

1. /\*
2. \* 'kernel.h' contains some often-used function prototypes etc
3. \*/
4. // メモリブロックがオーバーランしていることを確認します。(kernel/fork.c, 24)を参照してください。
5. void verify\_area(void \* addr,int count);
6. // カーネルのエラーメッセージを表示してから、無限ループに入る。(kernel/panic.c, 16)を参照してください。
7. volatile void panic(const char \* str);
8. // プロセス終了のための関数です。(kernel/exit.c, 262)があります。
9. volatile void do\_exit(long error\_code);
10. // 標準的なプリント（表示）関数です。(init/main.c, 179)を参照してください。
11. int printf(const char \* fmt, ...);
12. // printf()(kernel/printk.c)と同じ機能を持つ、カーネル固有の印刷関数です。
13. int printk(const char \* fmt, ...);
14. // コンソールの表示機能です。(kernel/chr\_drv/console.c, 995)となっています。
15. void console\_print(const char \* str);
16. // 指定された長さの文字列をttyに書き込みます。(kernel/chr\_drv/tty\_io.c, 339)となっています。
17. int tty\_write(unsigned ch,char \* buf,int count);
18. // 汎用的なカーネルのメモリ割り当て関数です。( lib/malloc.c, 117)。
19. void \* malloc(unsigned int size);
20. // 指定されたオブジェクトが占有していたメモリを解放します。( lib/malloc.c, 182) を参照してください。
21. void free\_s(void \* obj, int size);
22. // ハードディスクの処理がタイムアウトしました。(kernel/blk\_drv/hd.c, 318)となります。
23. extern void hd\_times\_out(void);
24. // ビープ音を止めます。(kernel/chr\_drv/console.c, 944)。
25. extern void sysbeepstop(void);
26. // ブラックスクリーン処理。(kernel/chr\_drv/console.c, 981)となっています。
27. extern void blank\_screen(void);
28. // ブラックアウトされている画面を元に戻す。(kernel/chr\_drv/console.c, 988)となっています。
29. extern void unblank\_screen(void);

17

1. extern int beepcount; // The beep time tick counts (chr\_drv/console.c, 950).
2. extern int hd\_timeout; // Hard disk timeout ticks (kernel/blk\_drv/blk.h).
3. extern int blankinterval; // Screen black screen interval(chr\_drv/console.c,138).
4. extern int blankcount; // Black screen time count (chr\_drv/console.c, 139).

22

23 #define free(x) free\_s((x), 0)

24

1. /\*
2. \* This is defined as a macro, but at some point this might become a 27 \* real subroutine that sets a flag if it returns true (to do
3. \* BSD-style accounting where the process is flagged if it uses root
4. \* privs). The implication of this is that you should do normal
5. \* permissions checks first, and check suser() last.
6. \*/
7. #define suser() (current->euid == 0) // Check if it is a super user.

33

## 14.27 math\_emu.h

### 14.27.1 Functionality

math\_emu.hファイルには、第11章（数学的コプロセッサ）に関連する定数定義および構造が含まれており、数学的コプロセッサをシミュレートする際にカーネルコードが様々な種類のデータをシミュレートするために使用されるデータ構造の一部も含まれています。

### 14.27.2 Code annotation

プログラム 14-24 linux/include/linux/math\_emu.h

1. /\*
2. \* linux/include/linux/math\_emu.h
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/
6. #ifndef \_LINUX\_MATH\_EMU\_H
7. #define \_LINUX\_MATH\_EMU\_H 8

// スケジューラのヘッダファイルは、タスクの構造、初期タスク0、およびディスクリプタのパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数のマクロステートメントを定義します。

9 #include <linux/sched.h> 10

1. // CPUが例外INT7(device not exist)を発生させたときのスタック上のデータの構造は、 // システムコールが呼び出されたときのカーネルスタック内のデータ分布に似ています。 11 struct info {
2. long \_\_\_math\_ret; // The return address of the math\_emulate() caller (INT 7).
3. long \_\_\_orig\_eip; // A place to temporarily save the original EIP.
4. long \_\_\_edi; // The registers that the exception handler pushed.
5. long \_\_\_esi;
6. long \_\_\_ebp;

// 割り込み7が戻ると、システムコールの戻り処理コードを実行します。

1. // 以下（18～30行目）は、system-callが呼び出されたときのスタック内の構造と同じです。
2. long \_\_\_sys\_call\_ret;
3. long \_\_\_eax;
4. long \_\_\_ebx;
5. long \_\_\_ecx;
6. long \_\_\_edx;
7. long \_\_\_orig\_eax; // If it's not a sys-call but other interrupts, it is -1.
8. long \_\_\_fs;
9. long \_\_\_es;
10. long \_\_\_ds;
11. long \_\_\_eip; // Lines 26 -- 30 are pushed by the CPU automatically.
12. long \_\_\_cs;
13. long \_\_\_eflags;
14. long \_\_\_esp;
15. long \_\_\_ss;
16. }; 32
17. // 情報構造（スタック内のデータ）のフィールドの参照を容易にするために定義された定数です。
18. #define EAX (info->\_\_\_eax)
19. #define EBX (info->\_\_\_ebx)
20. #define ECX (info->\_\_\_ecx)
21. #define EDX (info->\_\_\_edx)
22. #define ESI (info->\_\_\_esi)
23. #define EDI (info->\_\_\_edi)
24. #define EBP (info->\_\_\_ebp)
25. #define ESP (info->\_\_\_esp)
26. #define EIP (info->\_\_\_eip)
27. #define ORIG\_EIP (info->\_\_\_orig\_eip)
28. #define EFLAGS (info->\_\_\_eflags)
29. #define DS (\*(unsigned short \*) &(info->\_\_\_ds))
30. #define ES (\*(unsigned short \*) &(info->\_\_\_es))
31. #define FS (\*(unsigned short \*) &(info->\_\_\_fs))
32. #define CS (\*(unsigned short \*) &(info->\_\_\_cs))
33. #define SS (\*(unsigned short \*) &(info->\_\_\_ss))

49

// 演算コプロセッサのエミュレーション操作を終了します（ファイルmath\_emulation.cの488行目）。

// 以下の 52-53 行目のマクロ定義の実際の効果は、 \_\_math\_abort // を返さない関数として再定義することです (つまり、前に volatile を追加します)。マクロの最初の部分です。

// '(volatile void (\*)(struct info \*, unsigned int))'は、使用される関数型定義です。

// で \_\_math\_abort 関数の定義を再指定します。これに続いて、対応する

// パラメータです。キーワード volatile を関数名の前に置いて関数を装飾するのは， // その関数がリターンしないことを gcc コンパイラに伝えるためであり， // gcc がよりよいコードを生成できるようにするためである．

50 void \_\_math\_abort(struct info \*, unsigned int);

51

1. #define math\_abort(x,y) \
2. (((volatile void (\*)(struct info \*,unsigned int)) \_\_math\_abort)((x),(y)))

54

1. /\*
2. \* Gcc forces this stupid alignment problem: I want to use only two longs
3. \* for the temporary real 64-bit mantissa, but then gcc aligns out the 58 \* structure to 12 bytes which breaks things in math\_emulate.c. Shit. I 59 \* want some kind of "no-alignt" pragma or something.

60 \*/

61

// 一時的な本物の構造

1. // 合計64ビットのマンタを持っています。ここで，'a'は下位32ビット，'b'は上位32ビット // （1つの固定ビットを含む），'exponent'は指数の値である．
2. typedef struct {
3. long a,b;
4. short exponent;
5. } temp\_real;

66

1. // 上記のオリジナルノートで述べられているアライメント問題を解決するために設計された構造体で、 // 上記の temp\_real 構造体のように動作します。
2. typedef struct {
3. short m0,m1,m2,m3;
4. short exponent;
5. } temp\_real\_unaligned;

71

1. // temp\_real型の値「a」を80387のスタックレジスタ「b」に割り当てる（ST（i））。
2. #define real\_to\_real(a,b) \
3. ((\*(long long \*) (b) = \*(long long \*) (a)),((b)->exponent = (a)->exponent)) 74
4. // 長い実数（倍精度）の構造体です。
5. typedef struct {
6. long a,b; // 'a' is the lower 32 bits; 'b' is the upper 32 bits.
7. } long\_real;

78

79 typedef long short\_real; // 短い実数型を定義します。

80

1. // 一時的な整数構造。
2. typedef struct {
3. long a,b; // 'a' is the lower 32 bits; 'b' is the upper 32 bits.
4. short sign;
5. } temp\_int;

85

// 80387内部のステータス・ワード・レジスタの内容に対応する構造体

1. //のコプロセッサです（図11-6参照）。
2. struct swd {
3. int ie:1; // Invalid operation exception.
4. int de:1; // Denormalized exception.
5. int ze:1; // Divide by zero exception.
6. int oe:1; // Overflow exception.
7. int ue:1; // Underflow exception.
8. int pe:1; // Precision exception.
9. int sf:1; // Stack error flag, caused by overflow of the accumulator.
10. int ir:1; // Ir, b: Set if any of the above 6 unmasked exceptions occur.
11. int c0:1; // c0--c3: Condition code bits.
12. int c1:1;
13. int c2:1;
14. int top:3; // Indicates the 80-bit register currently at the top of the stack.
15. int c3:1;
16. int b:1;
17. };

102

1. // 80387内部レジスタ制御モード定数。
2. #define I387 (current->tss.i387) // 80387 status information of the process.
3. #define SWD (\*(struct swd \*) &I387.swd) // Status control word in 80387.
4. #define ROUNDING ((I387.cwd >> 10) & 3) // Get the rounding mode in the control word.
5. #define PRECISION ((I387.cwd >> 8) & 3) // Get the precision mode in the control word.

107

1. // 精度の有効桁数を定義する定数。
2. #define BITS24 0 // Precision Effective bits: 24 bits.
3. #define BITS53 2 // 53 bits.
4. #define BITS64 3 // 64 bits.

111

1. // 丸めモードの定数を定義します。
2. #define ROUND\_NEAREST 0 // Round to the nearest or even.
3. #define ROUND\_DOWN 1 // Trend to negative infinite.
4. #define ROUND\_UP 2 // Trend to positive infinite.
5. #define ROUND\_0 3 // Trend to cut to zero.

116

1. // 定数の定義。
2. #define CONSTZ (temp\_real\_unaligned) {0x0000,0x0000,0x0000,0x0000,0x0000} // 0
3. #define CONST1 (temp\_real\_unaligned) {0x0000,0x0000,0x0000,0x8000,0x3FFF} // 1.0
4. #define CONSTPI (temp\_real\_unaligned) {0xC235,0x2168,0xDAA2,0xC90F,0x4000} // Pi
5. #define CONSTLN2 (temp\_real\_unaligned) {0x79AC,0xD1CF,0x17F7,0xB172,0x3FFE} // Loge(2)
6. #define CONSTLG2 (temp\_real\_unaligned) {0xF799,0xFBCF,0x9A84,0x9A20,0x3FFD} // Log10(2) 122 #define CONSTL2E (temp\_real\_unaligned) {0xF0BC,0x5C17,0x3B29,0xB8AA,0x3FFF} // Log2(e)

123 #define CONSTL2T (temp\_real\_unaligned) {0x8AFE,0xCD1B,0x784B,0xD49A,0x4000} // Log2(10)

124

1. // 80387の状態を設定します。
2. #define set\_IE() (I387.swd |= 1)
3. #define set\_DE() (I387.swd |= 2)
4. #define set\_ZE() (I387.swd |= 4)
5. #define set\_OE() (I387.swd |= 8)
6. #define set\_UE() (I387.swd |= 16)
7. #define set\_PE() (I387.swd |= 32)

131

1. // 80387の制御条件を設定
2. #define set\_C0() (I387.swd |= 0x0100)
3. #define set\_C1() (I387.swd |= 0x0200)
4. #define set\_C2() (I387.swd |= 0x0400)
5. #define set\_C3() (I387.swd |= 0x4000)

136

1. /\* ea.c \*/
2. // Calculates the effective address used by the operand in the emulation instruction, that is, // calculates the effective address according to the addressing mode byte in the instruction. // Params: \_\_info - the contents of the stack at time of interrupt; \_\_code - instruction code.
3. // 有効なアドレスを返します。
4. char \* ea(struct info \* \_\_info, unsigned short \_\_code);

140

1. /\* convert.c \*/
2. // Various data type conversion functions implemented in the convert.c file.
3. void short\_to\_temp(const short\_real \* \_\_a, temp\_real \* \_\_b); 144 void long\_to\_temp(const long\_real \* \_\_a, temp\_real \* \_\_b);
4. void temp\_to\_short(const temp\_real \* \_\_a, short\_real \* \_\_b);
5. void temp\_to\_long(const temp\_real \* \_\_a, long\_real \* \_\_b);
6. void real\_to\_int(const temp\_real \* \_\_a, temp\_int \* \_\_b); 148 void int\_to\_real(const temp\_int \* \_\_a, temp\_real \* \_\_b);

149

1. /\* get\_put.c \*/
2. // Access functions of various types.
3. void get\_short\_real(temp\_real \*, struct info \*, unsigned short);
4. void get\_long\_real(temp\_real \*, struct info \*, unsigned short);
5. void get\_temp\_real(temp\_real \*, struct info \*, unsigned short);
6. void get\_short\_int(temp\_real \*, struct info \*, unsigned short);
7. void get\_long\_int(temp\_real \*, struct info \*, unsigned short);
8. 14.28 mm.h
9. void get\_longlong\_int(temp\_real \*, struct info \*, unsigned short);
10. void get\_BCD(temp\_real \*, struct info \*, unsigned short);
11. void put\_short\_real(const temp\_real \*, struct info \*, unsigned short);
12. void put\_long\_real(const temp\_real \*, struct info \*, unsigned short);
13. void put\_temp\_real(const temp\_real \*, struct info \*, unsigned short);
14. void put\_short\_int(const temp\_real \*, struct info \*, unsigned short);
15. void put\_long\_int(const temp\_real \*, struct info \*, unsigned short);
16. void put\_longlong\_int(const temp\_real \*, struct info \*, unsigned short);
17. void put\_BCD(const temp\_real \*, struct info \*, unsigned short);

166

1. /\* add.c \*/
2. // A function that simulates a floating-point addition instruction.
3. void fadd(const temp\_real \*, const temp\_real \*, temp\_real \*);

170

1. /\* mul.c \*/
2. // Simulate floating point multiply instructions.
3. void fmul(const temp\_real \*, const temp\_real \*, temp\_real \*);

174

1. /\* div.c \*/
2. // Simulate floating point division instructions.
3. void fdiv(const temp\_real \*, const temp\_real \*, temp\_real \*);

178

1. /\* compare.c \*/
2. // Simulate floating point comparison instructions.
3. void fcom(const temp\_real \*, const temp\_real \*); // FCOM, compare two numbers.
4. void fucom(const temp\_real \*, const temp\_real \*); // FUCOM, no order comparison.
5. void ftst(const temp\_real \*); // FTST, top stack accumulator compared to 0.

184

185 #endif

186

## 14.28 mm.h

### 14.28.1 Functionality

### mm.hファイルは、メモリ管理用のヘッダーファイルです。主に、メモリページのサイズと、いくつかのページリリース関数のプロトタイプを定義しています。

### 14.28.2 Code annotation

プログラム 14-25 linux/include/linux/mm.h

1. #ifndef \_MM\_H
2. #define \_MM\_H

3

// メモリページサイズ（バイト）を定義します。なお、キャッシュブロックサイズは1024バイトです。

4 #define PAGE\_SIZE 4096 5

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

14.28 mm.h

1. // <signal.h> シグナルのヘッダファイルです。シグナルシンボル定数、シグナル構造体、シグナル操作関数のプロトタイプを定義しています。
2. #include <linux/kernel.h>
3. #include <signal.h>

8

9 extern int SWAP\_DEV; // メモリページのスワップデバイス番号 (mm/memory.c, line 36).

10

1. // スワップされたメモリページがスワップデバイスに読み込まれたり、書き込まれたりします。ll\_rw\_page()関数は、ファイルblk\_drv/ll\_rw\_block.cで定義されています。パラメータの'nr'はメインメモリ領域のページ番号、'buffer'はリード/ライトバッファです。
2. #define read\_swap\_page(nr,buffer) ll\_rw\_page(READ,SWAP\_DEV,(nr),(buffer));
3. #define write\_swap\_page(nr,buffer) ll\_rw\_page(WRITE,SWAP\_DEV,(nr),(buffer)); 13
4. // メインメモリ領域の空き物理ページを取得します。利用可能なメモリがない場合は 0 を返します。
5. extern unsigned long get\_free\_page(void);
6. // 内容が変更された物理メモリページを， // リニアアドレス空間の指定された場所にマッピングする．put\_page()とほぼ同じです。
7. extern unsigned long put\_dirty\_page(unsigned long page,unsigned long address);
8. // アドレス「addr」で始まる物理メモリのページを解放する。
9. extern void free\_page(unsigned long addr);
10. // スワップデバイスで指定されたスワップページを解放する（mm/swap.c, line 58）。
11. void swap\_free(int page\_nr);
12. // 指定されたページをデバイスからメモリにスワップする (mm/swap.c, line 69).
13. void swap\_in(unsigned long \*table\_ptr);

19

1. //// アウトオブメモリー（oom）処理機能。
2. extern inline volatile void oom(void)
3. {
4. // do\_exit()は、終了コードを使用する必要があります。シグナル値 // SIGSEGV(11)と同じ値のエラーコードは「リソースが一時的に利用できません」であり、これは同義です。
5. printk("out of memory\n\r");
6. do\_exit(SIGSEGV);
7. }

25

// オンチップのTLB(Translation Look-aside Buffer)を無効にする。

// アドレス変換の効率を上げるために、CPUは直近の

// トランスレーション・ルックサイド・バッファーと呼ばれるチップの内部キャッシュにあるページテーブルデータを使用する

//（TLB）を使用しています。ページテーブルの情報を変更した後、バッファをリフレッシュする必要があります。これは、ページディレクトリベースレジスタ（PDBR）CR3をリロードすることで // 行われます。以下では，ページディレクトリのベースアドレスである EAX = 0 を示しています． 26 #define invalidate()

27 \_\_asm\_\_("movl %%eax,%%cr3":: "a" (0))

28

1. /\* these are not to be changed without changing head.s etc \*/
2. // Linux 0.12カーネルがデフォルトでサポートしている最大メモリ容量は16MBですが、これらの定義を変更することで、より多くのメモリに対応することができます。
3. #define LOW\_MEM 0x100000 // Low end of the physical memory (1MB).
4. extern unsigned long HIGH\_MEMORY; // The highest address of physical memory.
5. #define PAGING\_MEMORY (15\*1024\*1024) // The size of the paged memory (15MB).
6. #define PAGING\_PAGES (PAGING\_MEMORY>>12) // The number of pages after paging (3840).
7. #define MAP\_NR(addr) (((addr)-LOW\_MEM)>>12) // Map the memory address to the page number.
8. #define USED 100 // Page used flag, see memory.c, line 449.

36

// メモリマッピングのバイトマップで、1バイトが1ページを表します。各ページの対応するバイトは

// ページが現在参照（使用）されている回数を示すために使用されます。最大で15Mbのメモリ空間をマッピングすることができます。memory.cプログラムの関数mem\_init()では、主記憶領域のページとして // 使用できない位置をあらかじめUSED(100)に設定しています。

37 extern unsigned char mem\_map [ PAGING\_PAGES ];

38

1. // 以下のシンボル定数は，ページディレクトリやページテーブル（2次ページテーブル）エントリのフラグビットに対応しています。
2. #define PAGE\_DIRTY 0x40 // Bit 6, the page is dirty (modified).
3. #define PAGE\_ACCESSED 0x20 // Bit 5, the page was accessed.
4. #define PAGE\_USER 0x04 // Bit 2, the page belongs to: 1-user; 0-superuser. 42 #define PAGE\_RW 0x02 // Bit 1, read/write rights: 1 – write; 0 – read.

43 #define PAGE\_PRESENT 0x01 // ビット0、ページが存在する。1 - 存在する; 0 - 存在しない。

44

45 #endif

46

## 14.29 sched.h

### 14.29.1 Functionality

Sched.hは、スケジューラのヘッダファイルで、タスク構造体task\_struct、タスク0の初期データ、メモリディスクリプタのパラメータ設定・取得やタスクコンテキストスイッチマクロswitch\_to()などの組み込みアセンブリ関数マクロが定義されています。 タスクスイッチマクロswitch\_to(n)(222行目)では、まず構造体「struct {long a,b;} \_\_tmp」を宣言します。これは、カーネルのステートスタック上に8バイトのスペースを確保するためのものです。この空間には、これから切り替わる新しいタスクのタスク・ステータス・セグメントTSSのセレクタが格納されます。そして、現在のタスクに切り替える操作を行っているかどうかをテストし、そうであれば何もする必要はなく、そのまま終了します。そうでなければ、新しいタスクTSSのセレクタを一時構造体\_\_tmpのオフセット位置4に保存し、その時点で\_\_tmp内のデータを以下のように設定します。

\_\_tmp+0: 未定義 (long)

\_\_tmp+4: 新しいタスクのTSSセレクタ(ワード)

\_\_tmp+6: 未定義 (word)

次に、ECXレジスタの新しいタスクのポインタをグローバル変数'current'の現在のタスクのポインタと交換し、'current'にこれから切り替える新しいタスクのポインタ値を入れ、ECXは現在のタスクを保存します。その後、\_\_tmpに間接的にジャンプするLJMP命令を実行します。新しいタスクのTSSセレクタにジャンプする命令は、\_\_tmpの未定義値の部分を無視して、CPUは自動的にTSSセグメントで指定された新しいタスクにジャンプして実行し、タスク（現在のタスク）は中断されます。このため、構造体変数\_\_tmpの他の未定義値部分を設定する必要はありません。タスク切り替え動作の模式図は、4.7節の図4-37を参照してください。

一定期間が経過すると、タスクのLJMP命令がタスクのTSSセグメントセレクタにジャンプするため、CPUはタスクに切り替わり、LJMPの次の命令から実行を開始します。この時点でECXには現在のタスクへのポインタが含まれているので、このポインタを使って、最後（直近）の

### タスクが数学コプロセッサを使用していたことを示します。そのタスクがコプロセッサを使用していない場合は、直ちに終了します。そうでなければ、CLTS命令を実行して、コントロールレジスタCR0のタスク切り替えフラグTSをリセットします。CPUはタスクが切り替わるたびにこのフラグをセットし、コプロセッサ命令を実行する前にこのフラグをテストします。このようなLinuxシステムにおけるTSフラグの処理方法により、カーネルはコプロセッサの状態に対する不要な保存・回復動作を回避することができ、コプロセッサの実行性能を向上させることができます。

### 14.29.2 Code annotation

プログラム 14-26 linux/include/linux/sched.h

1. #ifndef \_SCHED\_H
2. #define \_SCHED\_H

3

1. 4 #define HZ 100 // システムクロックの刻みの周波数を定義 (10ms/tick) 5
2. #define NR\_TASKS 64 // The max number of tasks in the system.
3. #define TASK\_SIZE 0x04000000 // The size of each task (64MB).
4. #define LIBRARY\_SIZE 0x00400000 // The size of the loaded library (4MB).

9

1. #if (TASK\_SIZE & 0x3fffff)
2. #error "TASK\_SIZE must be multiple of 4M"
3. #endif

13

1. #if (LIBRARY\_SIZE & 0x3fffff)
2. #error "LIBRARY\_SIZE must be a multiple of 4M"
3. #endif

17

1. #if (LIBRARY\_SIZE >= (TASK\_SIZE/2))
2. #error "LIBRARY\_SIZE too damn big!"
3. #endif

21

22 #if (((TASK\_SIZE>>16)\*NR\_TASKS) !=0x10000) 23 #error "TASK\_SIZE\*NR\_TASKS must be 4GB"

24 #endif 25

// プロセスの論理アドレス空間でライブラリがロードされる場所（60MBの場合）。

26 #define LIBRARY\_OFFSET (TASK\_SIZE - LIBRARY\_SIZE)

27

1. // 次のマクロ CT\_TO\_SECS と CT\_TO\_USECS は、現在のシステムのティックを // 秒とマイクロ秒に変換するために使用されます。
2. #define CT\_TO\_SECS(x) ((x) / HZ)
3. #define CT\_TO\_USECS(x) (((x) % HZ) \* 1000000/HZ)

30

1. #define FIRST\_TASK task[0] // Task 0 is special, so we define a symbol for it.
2. #define LAST\_TASK task[NR\_TASKS-1] // The last task in the task array.

33

// <linux/head.h> 頭部のヘッダーファイルです。セグメントディスクリプターの簡単な構造が定義されており、 // いくつかのセレクタ定数とともに定義されています。

// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する（file,buffer\_head,

// m\_inode など）を使用しています。

// <linux/mm.h> メモリ管理用のヘッダーファイルです。ページサイズの定義と、いくつかのページ

// 関数のプロトタイプをリリースします。

// <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/time.h> timeval構造体とitimerval構造体が定義されています。

// <sys/resource.h> リソースファイル。プロセスが使用するシステムリソースの制限と利用に関する情報が含まれています。

1. // <signal.h> シグナルのヘッダファイルです。シグナルシンボル定数、シグナル構造体、シグナル操作関数のプロトタイプを定義しています。
2. #include <linux/head.h>
3. #include <linux/fs.h>
4. #include <linux/mm.h>
5. #include <sys/param.h>
6. #include <sys/time.h>
7. #include <sys/resource.h>
8. #include <signal.h>

41

1. #if (NR\_OPEN > 32)
2. #error "Currently the close-on-exec-flags and select masks are in one long, max 32 files/proc"
3. #endif

45

1. // プロセスの動作状態を定義します。
2. #define TASK\_RUNNING 0 // process is running or is ready to run.
3. #define TASK\_INTERRUPTIBLE 1 // in an interruptible wait state.
4. #define TASK\_UNINTERRUPTIBLE 2 // in an uninterruptible wait state (mainly I/O op).
5. #define TASK\_ZOMBIE 3 // in a dead state and the father has not yet signaled.
6. #define TASK\_STOPPED 4 // process has stopped.

51

1. #ifndef NULL
2. #define NULL ((void \*) 0)
3. #endif

55

// プロセスのページディレクトリテーブルをコピーします。( mm/memory.c, 118 )

1. // Linusは、これをカーネルの中で最も複雑な関数の一つと考えています。
2. extern int copy\_page\_tables(unsigned long from, unsigned long to, long size);
3. // ページテーブルで指定されたメモリとページテーブル自体を解放します。( mm/memory.c, 69 )
4. extern int free\_page\_tables(unsigned long from, unsigned long size); 58
5. // スケジューラの初期化関数です。(kernel/sched.c, 417 )
6. extern void sched\_init(void);
7. // プロセスのスケジューリング機能。(kernel/sched.c, 119)
8. extern void schedule(void);

// 例外（トラップ）処理の初期化関数です。(kernel/traps.c, 185) 61 extern void trap\_init(void);

// カーネルのエラーメッセージを表示し、その後、無限ループに入ります。(kernel/panic.c, 16) 62 extern void panic(const char \* str);

// 指定された長さの文字列をttyに書き込みます。(kernel/chr\_drv/tty\_io.c, 339)

63 extern int tty\_write(unsigned minor,char \* buf,int count);

64

65 typedef int (\*fn\_ptr)(); // 関数ポインタ型を定義する。

66

1. // 以下は数学コプロセッサの構造で、主にプロセス切り替え時にi387の実行状況情報を // 保存するために使用されます。
2. struct i387\_struct {
3. long cwd; // Control word.
4. long swd; // Status word. 70 long twd; // Tag word.
5. long fip; // Coprocessor code ip pointer.
6. long fcs; // Coprocessor code segment register. 73 long foo; // The offset of the memory operand.
7. long fos; // The segment of the memory operand.
8. long st\_space[20]; /\* 8\*10 bytes for each FP-reg = 80 bytes \*/
9. };

77

1. // TSS（Task Status Segment）データ構造。
2. struct tss\_struct {
3. long back\_link; /\* 16 high bits zero \*/
4. long esp0;
5. long ss0; /\* 16 high bits zero \*/
6. long esp1;
7. long ss1; /\* 16 high bits zero \*/
8. long esp2;
9. long ss2; /\* 16 high bits zero \*/
10. long cr3;
11. long eip;
12. long eflags;
13. long eax,ecx,edx,ebx;
14. long esp;
15. long ebp;
16. long esi;
17. long edi;
18. long es; /\* 16 high bits zero \*/
19. long cs; /\* 16 high bits zero \*/
20. long ss; /\* 16 high bits zero \*/
21. long ds; /\* 16 high bits zero \*/
22. long fs; /\* 16 high bits zero \*/
23. long gs; /\* 16 high bits zero \*/
24. long ldt; /\* 16 high bits zero \*/
25. long trace\_bitmap; /\* bits: trace 0, bitmap 16-31 \*/
26. struct i387\_struct i387;
27. }; 104

// 以下は、タスク（プロセス）データ構造、またはプロセス制御ブロック、またはプロセス記述子です。

// 詳細な説明は5.7節を参照してください。

//構造体 task\_struct {.

// long state; // -1 unrunnable, 0 runnable (ready), > 0 stopped.

// long counter; // タスクの実行時間の刻み（デクリメント）、実行時間のスライス。

// long priority; // 優先度。タスクの実行を開始すると、counter=priorityとなります。

// long signal; // シグナルのビットマップ、各ビットがシグナル( = bit offset + 1)となります。

// struct sigaction sigaction[32]; // シグナル属性構造体。シグナルの操作とフラグです。

// long blocked; // シグナルマスクの処理（マスクされたシグナルのビットマップ）。

// int exit\_code; // タスクが停止した後の終了コード、親がそれを取得します。

// unsigned long start\_code; // リニアアドレス空間でのコード開始位置。

// unsigned long end\_code; // コードの長さまたはサイズ（バイト）。

// unsigned long end\_data; // コードサイズ＋データのシース（バイト）。

// unsigned long brk; // トータルサイズ（バイト数）。

// unsigned long start\_stack; // スタックの底の位置。// long pid; // プロセスの識別子。// long pgrp; // プロセスグループ番号。

// long session; // プロセスのセッション番号。

// long leader; // リーダーのセッション番号。

// int groups[NGROUPS]; // グループ番号。1つのプロセスは、より多くのグループに所属することができます。

// task\_struct \*p\_pptr; // 親プロセスへのポインタを指定します。

// task\_struct \*p\_cptr; // 最年少の子プロセスへのポインタ。

// task\_struct \*p\_ysptr; // その後に作成された弟分プロセスへのポインタ。

// task\_struct \*p\_osptr; // 先に作成された古い兄弟プロセスへのポインタ。

// unsigned short uid; // ユーザーID。

// unsigned short euid; // 有効なユーザーID。

// unsigned short suid; // 保存されたユーザーID。

// unsigned short gid; // グループIDです。

// unsigned short egid; // 有効なグループID。

// unsigned short sgid; // 保存されたグループID。

// long timeout; // カーネルのタイミングのタイムアウト値。

// long alarm; // アラームのタイミング値（tick）。

// long utime; // ユーザーステートの実行時間（ティック）。

// long stime; // システム状態のランタイム（tick）。

// long cutime; // 子プロセスのユーザー状態のランタイム。

// long cstime; // 子プロセスのシステム状態のランタイム。

// long start\_time; // プロセスの実行開始時刻。

// struct rlimit rlim[RLIM\_NLIMITS]; // リソース使用量統計の配列。

// unsigned int flags; // プロセスごとのフラグ。

// unsigned short used\_math; // フラグ。コプロセッサを使用するかどうか。

// int tty; // 使用する tty サブデバイスの番号。-1は使用しないことを意味します。

// unsigned short umask; // ファイル作成属性のマスクビットです。

// struct m\_inode \* pwd; // 現在の作業ディレクトリ i ノード構造体ポインタ。

// struct m\_inode \* root; // ルートのi-node構造体ポインタ。

// struct m\_inode \* executable; // 実行ファイルのi-node構造へのポインタです。

// struct m\_inode \* library; // ロードされたライブラリのi-node構造体のポインタです。

// unsigned long close\_on\_exec; // 実行時にファイルハンドルを閉じるビットマップフラグです。

// struct file \* filp[NR\_OPEN]; // ファイル構造体ポインタテーブル、最大32項目。

// // インデックスは、ファイルディスクリプターの値です。

// struct desc\_struct ldt[3]; // LDT. 0-empty, 1-code seg cs, 2-data & stack seg ds & ss.

// struct tss\_struct tss; // プロセスのタスクステータスセグメント構造体TSSです。

//};

1. struct task\_struct {
2. /\* these are hardcoded - don't touch \*/
3. long state; /\* -1 unrunnable, 0 runnable, >0 stopped \*/
4. long counter;
5. long priority;
6. long signal;
7. struct sigaction sigaction[32];
8. long blocked; /\* bitmap of masked signals \*/
9. /\* various fields \*/
10. int exit\_code;
11. unsigned long start\_code,end\_code,end\_data,brk,start\_stack;
12. long pid,pgrp,session,leader;
13. int groups[NGROUPS];
14. /\*
15. \* pointers to parent process, youngest child, younger sibling,
16. \* older sibling, respectively. (p->father can be replaced with
17. \* p->p\_pptr->pid)
18. \*/
19. struct task\_struct \*p\_pptr, \*p\_cptr, \*p\_ysptr, \*p\_osptr;
20. unsigned short uid,euid,suid;
21. unsigned short gid,egid,sgid;
22. unsigned long timeout,alarm;
23. long utime,stime,cutime,cstime,start\_time;
24. struct rlimit rlim[RLIM\_NLIMITS];
25. unsigned int flags; /\* per process flags, defined below \*/
26. unsigned short used\_math;
27. /\* file system info \*/
28. int tty; /\* -1 if no tty, so it must be signed \*/
29. unsigned short umask;
30. struct m\_inode \* pwd;
31. struct m\_inode \* root;
32. struct m\_inode \* executable; 137 struct m\_inode \* library;
33. unsigned long close\_on\_exec;
34. struct file \* filp[NR\_OPEN];
35. /\* ldt for this task 0 - zero 1 - cs 2 - ds&ss \*/
36. struct desc\_struct ldt[3];
37. /\* tss for this task \*/
38. struct tss\_struct tss;
39. };

145

1. /\*
2. \* Per process flags
3. \*/
4. #define PF\_ALIGNWARN 0x00000001 /\* Print alignment warning msgs \*/
5. /\* Not implemented yet, only for 486\*/

151

1. /\*
2. \* INIT\_TASK is used to set up the first task table, touch at
3. \* your own risk!. Base=0, limit=0x9ffff (=640kB)
4. \*/
5. // 上記タスク構造の第1タスクに対応するハードコードされた情報。
6. #define INIT\_TASK \
7. /\* state etc \*/ { 0,15,15, \ // state, counter, priority
8. /\* signals \*/ 0,{{},},0, \ // signal, sigaction[32], blocked
9. /\* ec,brk... \*/ 0,0,0,0,0,0, \ // exit\_code,start\_code,end\_code,end\_data,brk,start\_stack
10. /\* pid etc.. \*/ 0,0,0,0, \ // pid, pgrp, session, leader
11. /\* suppl grps\*/ {NOGROUP,}, \ // groups[]
12. /\* proc links\*/ &init\_task.task,0,0,0, \ // p\_pptr, p\_cptr, p\_ysptr, p\_osptr
13. /\* uid etc \*/ 0,0,0,0,0,0, \ // uid, euid, suid, gid, egid, sgid
14. /\* timeout \*/ 0,0,0,0,0,0,0, \ // alarm,utime,stime,cutime,cstime,start\_time,used\_math
15. /\* rlimits \*/ { {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}, \
16. {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}, \
17. {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}}, \
18. /\* flags \*/ 0, \ // flags
19. /\* math \*/ 0, \ // used\_math, tty,umask,pwd,root,executable,close\_on\_exec
20. /\* fs info \*/ -1,0022,NULL,NULL,NULL,NULL,0, \
21. /\* filp \*/ {NULL,}, \ // filp[20]
22. { \ // ldt[3]
23. {0,0}, \
24. /\* ldt \*/ {0x9f,0xc0fa00}, \ // code size 640K,base 0x0,G=1,D=1,DPL=3,P=1 TYPE=0xa
25. {0x9f,0xc0f200}, \ // data size 640K,base 0x0,G=1,D=1,DPL=3,P=1 TYPE=0x2
26. }, \
27. /\*tss\*/ {0,PAGE\_SIZE+(long)&init\_task,0x10,0,0,0,0,(long)&pg\_dir,\ // tss
28. 0,0,0,0,0,0,0,0, \
29. 0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
30. \_LDT(0),0x80000000, \
31. {} \
32. }, \



184

1. extern struct task\_struct \*task[NR\_TASKS]; // An array of task pointers.
2. extern struct task\_struct \*last\_task\_used\_math;
3. extern struct task\_struct \*current; // The current process pointer.
4. extern unsigned long volatile jiffies; // Ticks (10ms/tick) from start of boot.
5. extern unsigned long startup\_time; // Boot time. seconds since 1970:0:0:0.
6. extern int jiffies\_offset; // The number of ticks that need to be adjusted.

191

192 #define CURRENT\_TIME (startup\_time+(jiffies+jiffies\_offset)/HZ) // 現在の時刻(秒)です。

193

1. // タイマーを追加する（ティック、タイミングが来たら関数\*fn()を呼ぶ）。(kernel/sched.c )
2. extern void add\_timer(long jiffies, void (\*fn)(void));
3. // 断続的にスリープを待つ。(kernel/sched.c)
4. extern void sleep\_on(struct task\_struct \*\* p);
5. // スリープ待ちで中断。(kernel/sched.c )
6. extern void interruptible\_sleep\_on(struct task\_struct \*\* p);
7. // 睡眠のプロセスを明確に起こす。(kernel/sched.c )
8. extern void wake\_up(struct task\_struct \*\* p);
9. // 現在のプロセスが指定されたユーザーグループgrpに属しているかどうかをチェックします。
10. extern int in\_group\_p(gid\_t grp);

199

1. /\*
2. \* Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall 202 \* 4-TSS0, 5-LDT0, 6-TSS1 etc ...
3. \*/

// グローバルテーブルで最初のTSSのエントリを探します。0-nul（使用しない）、1-コードセグメント

// cs、2-dataセグメントds、3-systemセグメントsyscall、4-task stateセグメントTSS0、5-local table // LTD0、6-task stateセグメントTSS1、というようになっています。

// 元のコメントから推測されるように、Linusはシステムコールのコードを // GDTテーブルの4番目の独立したセグメントに入れたかったのです。しかし、それをしなかったので、 // GDTテーブルの4番目の記述子項目（syscall項目）をアイドル状態にしていました。

1. // GDTテーブルの最初のTSSおよびLDT記述子のセレクタ・インデックスを以下のように定義する。
2. #define FIRST\_TSS\_ENTRY 4
3. #define FIRST\_LDT\_ENTRY (FIRST\_TSS\_ENTRY+1)

// このマクロは、n番目のタスクのTSSディスクリプターのセレクタ（オフセット）を計算するために使用されます

// をGDTの中に入れます。各ディスクリプターは8バイトで構成されているため，FIRST\_TSS\_ENTRY<<3>は各ディスクリプターの開始位置を示す。

// GDTテーブル内の記述子のオフセット位置。各タスクは1つのTSSと1つのLDTを使用するので

1. // 合計16バイトを占めるディスクリプターには、対応するTSSの開始位置を示すためにn<<4が必要です。このマクロで得られる値は，TSSのセレクタの // インデックス値でもある。後者のマクロは，GDT内のLDT記述子のセレクタ（オフセット）を定義するものである。
2. #define \_TSS(n) ((((unsigned long) n)<<4)+(FIRST\_TSS\_ENTRY<<3)) 207 #define \_LDT(n) ((((unsigned long) n)<<4)+(FIRST\_LDT\_ENTRY<<3))

組込みのアセンブリマクロでは、n番目のタスクのTSSセグメントセレクタを

1. // タスクレジスタTRにロードする。後者のマクロは、n番目のタスクのLDTセグメントセレクタをローカルディスクリプタテーブルレジスタLDTRにロードするために使用されます。
2. #define ltr(n) \_\_asm\_\_("ltr %%ax"::"a" (\_TSS(n)))
3. #define lldt(n) \_\_asm\_\_("lldt %%ax"::"a" (\_LDT(n)))

プログラム kernel/traps.c の 78 行目で使用されている、 // 現在実行中のタスクのタスク番号を取得します（タスク配列のインデックスであり、 // プロセス番号 pid とは異なります）。

1. // 返却： n - 現在のタスク番号。
2. #define str(n) \
3. \_\_asm\_\_("str %%ax\n\t" \ // Save the TSS selector in the task register to EAX.
4. "subl %2,%%eax\n\t" \ // (EAX - FIRST\_TSS\_ENTRY \* 8) => EAX 213 "shrl $4,%%eax" \ // (EAX / 16) => EAX = current task number.
5. :"=a" (n) \
6. :"a" (0),"i" (FIRST\_TSS\_ENTRY<<3))
7. /\*
8. \* switch\_to(n) should switch tasks to task nr n, first
9. \* checking that n isn't the current task, in which case it does nothing. 219 \* This also clears the TS-flag if the task we switched to has used 220 \* tha math co-processor latest.

221 \*/

// TSSセレクタで構成されるアドレスにジャンプすると、CPUがタスクを切り替えます。 // 入力: %0 - \_\_tmp を指す; %1 - 新しい TSS セレクタの \_\_tmp.b を指す; DX - 新しいタスク n の TSS セグメント // セレクタ; ECX - 新しいタスク n のタスク構造体ポインタ task[ n].

// 一時データ構造 \_\_tmp は、228 行目のファージャンプ命令のオペランドを構築するために使用されます。オペランドは、4 バイトのオフセットアドレスと 2 バイトのセグメントセレクタで構成されます。

// したがって、\_\_tmpの'a'の値は32ビットのオフセットであり、'b'の下位2バイトは

// 新しいTSSセグメントのセレクタ(上位2バイトは使用しない) TSSセグメントへのジャンプ

// セレクタは、TSSに対応するプロセスにタスクを切り替えるようにします。タスク切り替えを引き起こすロングジャンプでは、 // 'a' 値は役に立たない。228行目の間接ジャンプ命令は、6バイトのオペランドをジャンプ先のロングポインタとして使用します。そのフォーマットは

// JMP 16ビットセグメントセレクタ。32ビットオフセット

// タスクが切り替わった後は、元のタスクポインタと比較して判断する

// 元のタスクが前回実行されたかどうかを判断する際に、 // last\_task\_used\_math 変数に保存されている最後に使用されたコプロセッサタスクポインタを使用する。の説明を参照してください。

1. // kernel/sched.cのmath\_state\_restore()関数。
2. #define switch\_to(n) {\
3. struct {long a,b;} \_\_tmp; \
4. \_\_asm\_\_("cmpl %%ecx,\_current\n\t" \ // Is task n the current task? (current ==task[n]?) 225 "je 1f\n\t" \ // If so, do nothing and return.
5. 226 "movw %%dx,%1\\t" ‥‥ // 新しいタスクの TSS セレクタが \_\_tmp.b に格納される 227 "xchgl %%ecx,\_current\t" ‥‥ // Current = task[n]; ECX =切り替えられたタスク。
6. "ljmp %0\n\t" \ // Long jump to \*&\_\_tmp, causing task switching. // The following statement will not continue until the task is switched back. First check if //the original task used the coprocessor last time. If yes, clear the TS flag in CR0.
7. "cmpl %%ecx,\_last\_task\_used\_math\n\t" \ // original task used math last time?
8. "jne 1f\n\t" \ // If not, jump forward and exit.
9. "clts\n" \ // If yes, then clear the TS flag in CR0.
10. "1:" \ // exit.
11. ::"m" (\*&\_\_tmp.a),"m" (\*&\_\_tmp.b), \
12. "d" (\_TSS(n)),"c" ((long) task[n])); \
13. }

236

ページの配置（カーネルのどこにも参考になるものがありません）

237 #define PAGE\_ALIGN(n) (((n)+0xfff)&0xfffff000)

238

// ディスクリプタの各ベースアドレスフィールドをアドレス addr に設定します。

1. // %0 - addr offset 2; %1 - addr offset 4; %2 - addr offset 7; EDX - base address。
2. #define \_set\_base(addr,base) \
3. \_\_asm\_\_("movw %%dx,%0\n\t" \ // Lower 16 bits(15-0) of the base=>[addr+2].
4. "rorl $16,%%edx\n\t" \ // Upper 16 bits(31-16) of the base in EDX => DX.
5. "movb %%dl,%1\n\t" \ // Lower 8 bits(23-16) of the upper 16 bits =>[addr+4] 243 "movb %%dh,%2" \ // Upper 8 bits(31-24) of the upper 16 bits =>[addr+7]
6. ::"m" (\*((addr)+2)), \
7. "m" (\*((addr)+4)), \
8. "m" (\*((addr)+7)), \
9. "d" (base) \
10. :"dx") // Tell gcc that the value in EDX has been changed.

249

// アドレス addr のディスクリプターにセグメントリミットフィールドを設定します。

1. // %0 - アドレスの addr; %1 - アドレスのオフセット 6; EDX - セグメントの長さの制限。
2. #define \_set\_limit(addr,limit) \
3. \_\_asm\_\_("movw %%dx,%0\n\t" \ // Lower 16 bits(15-0) of the segment limit =>[addr].
4. "rorl $16,%%edx\n\t" \ // Upper 4 bits(19-16) of the limit in EDX => DL.
5. "movb %1,%%dh\n\t" \ // Original [addr+6] => DH, the upper 4 bits are flags. 254 "andb $0xf0,%%dh\n\t" \ // Clear lower 4 bits of DH (will be stored at 19-16). 255 "orb %%dh,%%dl\n\t" \ // Combine original high 4-bit flags and upper 4 bits 256 "movb %%dl,%1" \ // (19-16)of the limit into 1 byte,stored at [addr+6].
6. ::"m" (\*(addr)), \
7. "m" (\*((addr)+6)), \
8. "d" (limit) \
9. :"dx")

261

// ローカルディスクリプターテーブルLDTのディスクリプターのベースアドレスフィールドを設定します。

1. // LDTのディスクリプターのセグメントリミットフィールドを設定します。
2. #define set\_base(ldt,base) \_set\_base( ((char \*)&(ldt)) , base )
3. #define set\_limit(ldt,limit) \_set\_limit( ((char \*)&(ldt)) , (limit-1)>>12 ) 264

// アドレス addr のディスクリプターから base を取得します。これは、\_set\_base()の逆です。

1. // EDX - store base (\_\_base); %1 - addr offset 2; %2 - addr offset 4; %3 - addr offset 7。
2. #define \_get\_base(addr) ({\
3. unsigned long \_\_base; \
4. \_\_asm\_\_("movb %3,%%dh\n\t" \ // Upper 8 bits(31-24) of upper 16 bits [addr+7] =>DH.
5. "movb %2,%%dl\n\t" \ // Lower 8 bits(23-16) of upper 16 bits [addr+4] =>DL.
6. "shll $16,%%edx\n\t" \ // Upper 16 bits are moved to the upper 16 bits of EDX.
7. "movw %1,%%dx" \ // Lower 16 (15-0) bits of the base at [addr+2] => DX.
8. :"=d" (\_\_base) \ // Thus EDX contains a 32-bit segment base address.
9. :"m" (\*((addr)+2)), \
10. "m" (\*((addr)+4)), \
11. "m" (\*((addr)+7))); \
12. \_\_base;}) 276

// LDTの'ldt'が指すセグメントディスクリプターのベースアドレスを取る。

277 #define get\_base(ldt) \_get\_base( ((char \*)&(ldt)) )

278

// セグメントセレクタ'segment'で指定されたディスクリプタのセグメント制限を取る。

LSLという命令は、Load Segment Limitの略である。これは、散乱した制限を

14.30 sys.h

// 指定されたセグメントのディスクリプターから // 長さのビットを取り出し、完全なセグメント制限値を指定されたレジスタに入れます。結果として得られるセグメントリミット値は、実際のバイト数から1を引いたものなので、1を加えて返す必要があります。

1. // %0 - セグメントの長さ(バイト); %1 - セグメントセレクタ「segment」です。
2. #define get\_limit(segment) ({ \
3. unsigned long \_\_limit; \
4. \_\_asm\_\_("lsll %1,%0\n\tincl %0":"=r" (\_\_limit):"r" (segment)); \
5. \_\_limit;})

283

284 #endif

285

## 14.30 sys.h

### 14.30.1 Functionality

### sys.hヘッダファイルには，カーネル内のすべてのシステムコール関数のプロトタイプと，システムコール関数ポインタテーブルが記載されています。

### 14.30.2 Code annotation

プログラム 14-27 linux/include/linux/sys.h

1. /\*
2. \* Why isn't this a .c file? Enquiring minds....
3. \*/

4

1. extern int sys\_setup(); // 0 - System initializations. (kernel/blk\_drv/hd.c,74)
2. extern int sys\_exit(); // 1 - Program exit. (kernel/exit.c, 365)
3. extern int sys\_fork(); // 2 - Create a new process. (kernel/sys\_call.s,222)
4. extern int sys\_read(); // 3 - Read file. (fs/read\_write.c,55)
5. extern int sys\_write(); // 4 - Write file. (fs/read\_write.c,83)
6. extern int sys\_open(); // 5 - Open file. (fs/open.c,171)
7. extern int sys\_close(); // 6 - Close file. (fs/open.c,219)
8. extern int sys\_waitpid(); // 7 - Wait process to terminate. (kernel/exit.c,370)
9. extern int sys\_creat(); // 8 - Create file. (fs/open.c,214)
10. extern int sys\_link(); // 9 - Ceate hard linke to a file.(fs/namei.c,837)
11. extern int sys\_unlink(); // 10 - Delete a filename(or file)(fs/namei.c,709)
12. extern int sys\_execve(); // 11 - Execute a program. (kernel/sys\_call.s,214)
13. extern int sys\_chdir(); // 12 - Change current directory. (fs/open.c,76)
14. extern int sys\_time(); // 13 - Get current time. (kernel/sys.c,134)
15. extern int sys\_mknod(); // 14 - Create block/char special file. (fs/namei.c,464)
16. extern int sys\_chmod(); // 15 - Change file mode. (fs/open.c,106)
17. extern int sys\_chown(); // 16 - Chane file owner or group.(fs/open.c,122)
18. extern int sys\_break(); // 17 - (kernel/sys.c,33)\*
19. extern int sys\_stat(); // 18 - Get file status using path name.(fs/stat.c,36)
20. extern int sys\_lseek(); // 19 - Reposite r/w file offset. (fs/read\_write.c,25)
21. extern int sys\_getpid(); // 20 - Get process id. (kernel/sched.c,380)
22. extern int sys\_mount(); // 21 - Mount a file-system. (fs/super.c,199)
23. extern int sys\_umount(); // 22 - Unmount a file-system. (fs/super.c,166)
24. 14.30 sys.h
25. extern int sys\_setuid(); // 23 - Set process user id. (kernel/sys.c,196)
26. extern int sys\_getuid(); // 24 - Get process user id. (kernel/sched.c,390)
27. extern int sys\_stime(); // 25 - Set system time. (kernel/sys.c,207)
28. extern int sys\_ptrace(); // 26 - Process tracing. (kernel/sys.c,38)\*
29. extern int sys\_alarm(); // 27 - Set alarm time. (kernel/sched.c,370)
30. extern int sys\_fstat(); // 28 - Get file status by using handle.(fs/stat.c,58)
31. extern int sys\_pause(); // 29 - Pause the process running.(kernel/sched.c,164)
32. extern int sys\_utime(); // 30 - Set file access and modified time.(fs/open.c,25)
33. extern int sys\_stty(); // 31 - Modify terminal settings. (kernel/sys.c,43)\*
34. extern int sys\_gtty(); // 32 - Get terminal settings. (kernel/sys.c,48)\*
35. extern int sys\_access(); // 33 - Check file access permission.(fs/open.c,48)
36. extern int sys\_nice(); // 34 - Set execution priority. (kernel/sched.c,410)
37. extern int sys\_ftime(); // 35 - Get date and time. (kernel/sys.c,28)\*
38. extern int sys\_sync(); // 36 - Synchronous data with dev.(fs/buffer.c,44)
39. extern int sys\_kill(); // 37 - Terminate a process. (kernel/exit.c,205)
40. extern int sys\_rename(); // 38 - Change filename. (kernel/sys.c,53)\*
41. extern int sys\_mkdir(); // 39 - Make a directory. (fs/namei.c,515)
42. extern int sys\_rmdir(); // 40 - Remove a directory. (fs/namei.c,635)
43. extern int sys\_dup(); // 41 - Duplicate a file handle. (fs/fcntl.c,42)
44. extern int sys\_pipe(); // 42 - Create a pipe. (fs/pipe.c,76)
45. extern int sys\_times(); // 43 - Get running time. (kernel/sys.c,216)
46. extern int sys\_prof(); // 44 - Execution time zone. (kernel/sys.c,58)\*
47. extern int sys\_brk(); // 45 - Change data segment len. (kernel/sys.c,228)
48. extern int sys\_setgid(); // 46 - Set process group id. (kernel/sys.c,98)
49. extern int sys\_getgid(); // 47 - Set process group id. (kernel/sched.c,400)
50. extern int sys\_signal(); // 48 - Signal processing. (kernel/signal.c,85)
51. extern int sys\_geteuid(); // 49 - Get efficient user id. (kenrl/sched.c,395)
52. extern int sys\_getegid(); // 50 - Get efficient group id. (kenrl/sched.c,405)
53. extern int sys\_acct(); // 51 - Process accounting. (kernel/sys.c,109)\*
54. extern int sys\_phys(); // 52 - Map phy mem to process space. (kernel/sys.c,114)\*
55. extern int sys\_lock(); // 53 - (kernel/sys.c,119)\*
56. extern int sys\_ioctl(); // 54 - Device i/o control. (fs/ioctl.c,31)
57. extern int sys\_fcntl(); // 55 - File operation control. (fs/fcntl.c,47)
58. extern int sys\_mpx(); // 56 - (kernel/sys.c,124)\* 62 extern int sys\_setpgid(); // 57 - Set process group id. (kernel/sys.c,245)
59. extern int sys\_ulimit(); // 58 - Statistical resource usage.(kernel/sys.c,129)
60. extern int sys\_uname(); // 59 - Show system info. (kernel/sys.c,343)
61. extern int sys\_umask(); // 60 - Get default file creation mode. (kernel/sys.c,515)
62. extern int sys\_chroot(); // 61 - Change root directory. (fs/open.c,91)
63. extern int sys\_ustat(); // 62 - Get file system states. (fs/open.c,20)
64. extern int sys\_dup2(); // 63 - Duplicate file handle. (fs/fcntl.c,36)
65. extern int sys\_getppid(); // 64 - Get parenet process id. (kernel/sched.c,385)
66. extern int sys\_getpgrp(); // 65 - Get pid (getpgid(0)) (kernel/sys.c,271)
67. extern int sys\_setsid(); // 66 - Set new session id. (kernel/sys.c,276)
68. extern int sys\_sigaction(); // 67 - Set signal operation. (kernel/signal.c,100)
69. extern int sys\_sgetmask(); // 68 - Get signal mask code. (kernel/signal.c,14)
70. extern int sys\_ssetmask(); // 69 - Set signal mask code. (kernel/signal.c,19)
71. extern int sys\_setreuid(); // 70 - Set real/efficient uid. (kernel/sys.c,159)
72. extern int sys\_setregid(); // 71 - Set real/efficient pid. (kernel/sys.c,74)
73. extern int sys\_sigpending(); // 73 - Check pending signals. (kernel/signal.c,27)
74. extern int sys\_sigsuspend(); // 72 - Suspending a process. (kernel/signal.c,48)
75. extern int sys\_sethostname(); // 74 - Set host name. (kernel/sys.c,357)
76. extern int sys\_setrlimit(); // 75 - Set resource using limit. (kernel/sys.c,387)
77. extern int sys\_getrlimit(); // 76 - Get resource using limit. (kernel/sys.c,375)
78. extern int sys\_getrusage(); // 77 - Get resource usage. (kernel/sys.c,412)
79. extern int sys\_gettimeofday(); // 78 - Get time of the day. (kernel/sys.c,440)
80. extern int sys\_settimeofday(); // 79 - Set time of the day. (kernel/sys.c,466)
81. extern int sys\_getgroups(); // 80 - Get process all group ids.(kernel/sys.c,289)
82. extern int sys\_setgroups(); // 81 - Set process group array. (kernel/sys.c,307)
83. extern int sys\_select(); // 82 - wait for file change state.(fs/select.c,216)
84. extern int sys\_symlink(); // 83 - Create file sysmbol link. (fs/namei.c，767)
85. extern int sys\_lstat(); // 84 - Get link file state. (fs/stat.c，47)
86. extern int sys\_readlink(); // 85 - Read link file contents. (fs/stat.c，69)
87. extern int sys\_uselib(); // 86 - Select shared lib. (fs/exec.c, 42) 92
88. // システムコールの関数ポインタテーブルで、システムコールの割り込みハンドラ(int 0x80)がジャンプテーブルとして使用している。
89. fn\_ptr sys\_call\_table[] = { sys\_setup, sys\_exit, sys\_fork, sys\_read,
90. sys\_write, sys\_open, sys\_close, sys\_waitpid, sys\_creat, sys\_link,
91. sys\_unlink, sys\_execve, sys\_chdir, sys\_time, sys\_mknod, sys\_chmod,
92. sys\_chown, sys\_break, sys\_stat, sys\_lseek, sys\_getpid, sys\_mount,
93. sys\_umount, sys\_setuid, sys\_getuid, sys\_stime, sys\_ptrace, sys\_alarm,
94. sys\_fstat, sys\_pause, sys\_utime, sys\_stty, sys\_gtty, sys\_access,
95. sys\_nice, sys\_ftime, sys\_sync, sys\_kill, sys\_rename, sys\_mkdir,
96. sys\_rmdir, sys\_dup, sys\_pipe, sys\_times, sys\_prof, sys\_brk, sys\_setgid,
97. sys\_getgid, sys\_signal, sys\_geteuid, sys\_getegid, sys\_acct, sys\_phys,
98. sys\_lock, sys\_ioctl, sys\_fcntl, sys\_mpx, sys\_setpgid, sys\_ulimit,
99. sys\_uname, sys\_umask, sys\_chroot, sys\_ustat, sys\_dup2, sys\_getppid,
100. sys\_getpgrp, sys\_setsid, sys\_sigaction, sys\_sgetmask, sys\_ssetmask,
101. sys\_setreuid,sys\_setregid, sys\_sigsuspend, sys\_sigpending, sys\_sethostname,
102. sys\_setrlimit, sys\_getrlimit, sys\_getrusage, sys\_gettimeofday,
103. sys\_settimeofday, sys\_getgroups, sys\_setgroups, sys\_select, sys\_symlink,
104. sys\_lstat, sys\_readlink, sys\_uselib };

109

1. /\* So we don't have to do any more manual updating.... \*/
2. int NR\_syscalls = sizeof(sys\_call\_table)/sizeof(fn\_ptr);

112

## 14.31 tty.h

### 14.31.1 Functionality

### tty.hファイルでは、端末のデータ構造といくつかの定数、およびttyのキュー・バッファ操作で使用されるマクロが定義されています。

### 14.31.2 Code annotation

プログラム 14-28 linux/include/linux/tty.h

1. /\*
2. \* 'tty.h' defines some structures used by tty\_io.c and some defines.
3. \*
4. \* NOTE! Don't touch this without checking that nothing in rs\_io.s or
5. \* con\_io.s breaks. Some constants are hardwired into the system (mainly
6. \* offsets into 'tty\_queue'
7. \*/

8

1. #ifndef \_TTY\_H
2. #define \_TTY\_H

11

1. #define MAX\_CONSOLES 8 // The maximum number of virtual consoles.
2. #define NR\_SERIALS 2 // The number of serial terminals.
3. #define NR\_PTYS 4 // The number of psesudo terminals.

15

16 extern int NR\_CONSOLES; // バーチャルコンソールの数です。

17

// <termios.h> 端末入出力機能のヘッダファイルです。主に非同期通信ポートを制御するターミナル // インターフェースを定義している。

18 #include <termios.h>

19

20 #define TTY\_BUF\_SIZE 1024 // ttyキューバッファのサイズです。

21

// Tty character buffer queueデータ構造。tty\_struc 構造体の read, write, auxiliary (canonical) buffer queue // に使用されます。

1. // 最初のフィールド「data」には、 // キューバッファ内の文字列値（文字数ではない）が格納されています。シリアル端末の場合は、シリアルポートのアドレスが格納されています。
2. struct tty\_queue {
3. unsigned long data; // Char lines in the buffer or the serial port.
4. unsigned long head; // The data header in the buffer.
5. unsigned long tail; // The data tail in the buffer.
6. struct task\_struct \* proc\_list; // process list waiting for this queue buffer.
7. char buf[TTY\_BUF\_SIZE]; // The buffer.
8. };

29

1. // これらのマクロは、ターミナルタイプをチェックするために使用されます。
2. #define IS\_A\_CONSOLE(min) (((min) & 0xC0) == 0x00) // console.
3. #define IS\_A\_SERIAL(min) (((min) & 0xC0) == 0x40) // serial terminal.
4. #define IS\_A\_PTY(min) ((min) & 0x80) // psesudo terminal.
5. #define IS\_A\_PTY\_MASTER(min) (((min) & 0xC0) == 0x80) // master pty.
6. #define IS\_A\_PTY\_SLAVE(min) (((min) & 0xC0) == 0xC0) // slate pty.
7. #define PTY\_OTHER(min) ((min) ^ 0x40) // other type terminal.

36

// ttyキューでのバッファ操作マクロは以下のように定義されています。(tty\_io.cの図を参照）。

1. // バッファのポインタ'a'は1バイトずつ前後にシフトされ， // バッファの左右を超えた場合，ポインタは循環的に移動する。
2. #define INC(a) ((a) = ((a)+1) & (TTY\_BUF\_SIZE-1))
3. #define DEC(a) ((a) = ((a)-1) & (TTY\_BUF\_SIZE-1))

// バッファを空にします。 // バッファに残っている空き領域のサイズです。

1. // バッファの最後の位置。// バッファは満杯です。 // バッファ内の文字数です。
2. #define EMPTY(a) ((a)->head == (a)->tail)
3. #define LEFT(a) (((a)->tail-(a)->head-1)&(TTY\_BUF\_SIZE-1))
4. #define LAST(a) ((a)->buf[(TTY\_BUF\_SIZE-1)&((a)->head-1)])
5. #define FULL(a) (!LEFT(a))
6. #define CHARS(a) (((a)->head-(a)->tail)&(TTY\_BUF\_SIZE-1))

// 'queue' バッファの 'tail' から文字を取得し、tail++とします。

1. // 'queue'キューバッファの'head'に文字を配置し、head++。
2. #define GETCH(queue,c) \
3. (void)({c=(queue)->buf[(queue)->tail];INC((queue)->tail);})
4. #define PUTCH(c,queue) \
5. (void)({(queue)->buf[(queue)->head]=(c);INC((queue)->head);})

48

1. // 端末のキーボードに入力されている文字の種類を確認します。
2. #define INTR\_CHAR(tty) ((tty)->termios.c\_cc[VINTR]) // Send signal SIGINT.
3. #define QUIT\_CHAR(tty) ((tty)->termios.c\_cc[VQUIT]) // Send signal SIGQUIT.
4. #define ERASE\_CHAR(tty) ((tty)->termios.c\_cc[VERASE]) // Erase a char.
5. #define KILL\_CHAR(tty) ((tty)->termios.c\_cc[VKILL]) // Kill a line of chars.
6. #define EOF\_CHAR(tty) ((tty)->termios.c\_cc[VEOF]) // End of file char.
7. #define START\_CHAR(tty) ((tty)->termios.c\_cc[VSTART]) // Start output.
8. #define STOP\_CHAR(tty) ((tty)->termios.c\_cc[VSTOP]) // Stop output.
9. #define SUSPEND\_CHAR(tty) ((tty)->termios.c\_cc[VSUSP]) // Send signal SIGTSTP.

57

1. // 端末のデータ構造。
2. struct tty\_struct {
3. struct termios termios; // Terminal io mode & control structure.
4. int pgrp; // The pgroup the terminal belongs to.
5. int session; // The session.
6. int stopped; // Terminal stopped flag.
7. void (\*write)(struct tty\_struct \* tty); // tty write function pointer.
8. struct tty\_queue \*read\_q; // tty read queue.
9. struct tty\_queue \*write\_q; // tty write queue.
10. struct tty\_queue \*secondary; // tty aux or canonical queue.
11. };

68

1. extern struct tty\_struct tty\_table[];
2. extern int fg\_console; // Front console number.

71

// 以下のマクロは，端末の種類に応じて，tty\_table[]内の端末番号「nr」に対応する // tty構造体へのポインタを取得する。

// 73行目の後半は、tty\_table[]内の対応するtty構造を選択するために使用されます。

// サブデバイス番号'dev'に基づいてテーブルを作成します。dev = 0の場合は、フォアグラウンドの端末が

// が使われているので、端末番号「fg\_console」を tty\_table[] エントリーのインデックスとして // 使うことで tty 構造を取得することができます。devが0より大きい場合は、次の2つのケースを考慮する必要があります。

// (1)devは仮想端末番号，(2)devはシリアル端末番号または疑似端末番号 // です．仮想端末の場合，tty\_table[]のtty構造は， dev-1(0 -- // 63)でインデックスされる．他の種類の端末の場合，tty構造体のインデックスエントリはdevである．

// たとえば，dev = 64，つまりシリアル端末 1 であれば，その tty 構造は // ttb\_table[dev]. dev = 1 の場合、対応する端末の tty 構造体は tty\_table[0] である。

1. // tty\_io.cプログラムの70～73行目をご覧ください。
2. #define TTY\_TABLE(nr) \
3. (tty\_table + ((nr) ? (((nr) < 64)? (nr)-1:(nr)) : fg\_console))

74

// ここでは、特殊文字の配列c\_cc[]の初期値を変更できるようにしています。

// 端末のtermios構造。POSIX.1では11種類の特殊文字が定義されていますが、Linuxシステムでは

// さらに、SVR4で使用される6つの特殊文字を定義しています。\_POSIX\_VDISABLE(0)が

1. // 定義されている場合は、項目の値が \_POSIX\_VDISABLE になると、対応する特殊文字が禁止されます。81行目の初期値では8進数で表現されています。
2. /\* intr=^C quit=^| erase=del kill=^U
3. eof=^D vtime=\0 vmin=\1 sxtc=\0
4. start=^Q stop=^S susp=^Z eol=\0
5. reprint=^R discard=^U werase=^W lnext=^V
6. eol2=\0
7. \*/
8. #define INIT\_C\_CC "\003\034\177\025\004\0\1\0\021\023\032\0\022\017\027\026\0" 82
9. void rs\_init(void); // serial terminal init.(kernel/chr\_drv/serial.c)
10. void con\_init(void); // Control terminal init.(kernel/chr\_drv/console.c)
11. void tty\_init(void); // (kernel/chr\_drv/tty\_io.c)

86

1. 87 int tty\_read(unsigned c, char \* buf, int n); // (kernel/chr\_drv/tty\_io.c) 88 int tty\_write(unsigned c, char \* buf, int n); // (kernel/chr\_drv/tty\_io.c) 89
2. void con\_write(struct tty\_struct \* tty); // (kernel/chr\_drv/console.c)
3. void rs\_write(struct tty\_struct \* tty); // (kernel/chr\_drv/serial.c)
4. void mpty\_write(struct tty\_struct \* tty); // (kernel/chr\_drv/pty.c) 93 void spty\_write(struct tty\_struct \* tty); // (kernel/chr\_drv/pty.c) 94

95 void copy\_to\_cooked(struct tty\_struct \* tty); // (kernel/chr\_drv/tty\_io.c) 96

97 void update\_screen(void); // (kernel/chr\_drv/console.c)

98

99 #endif

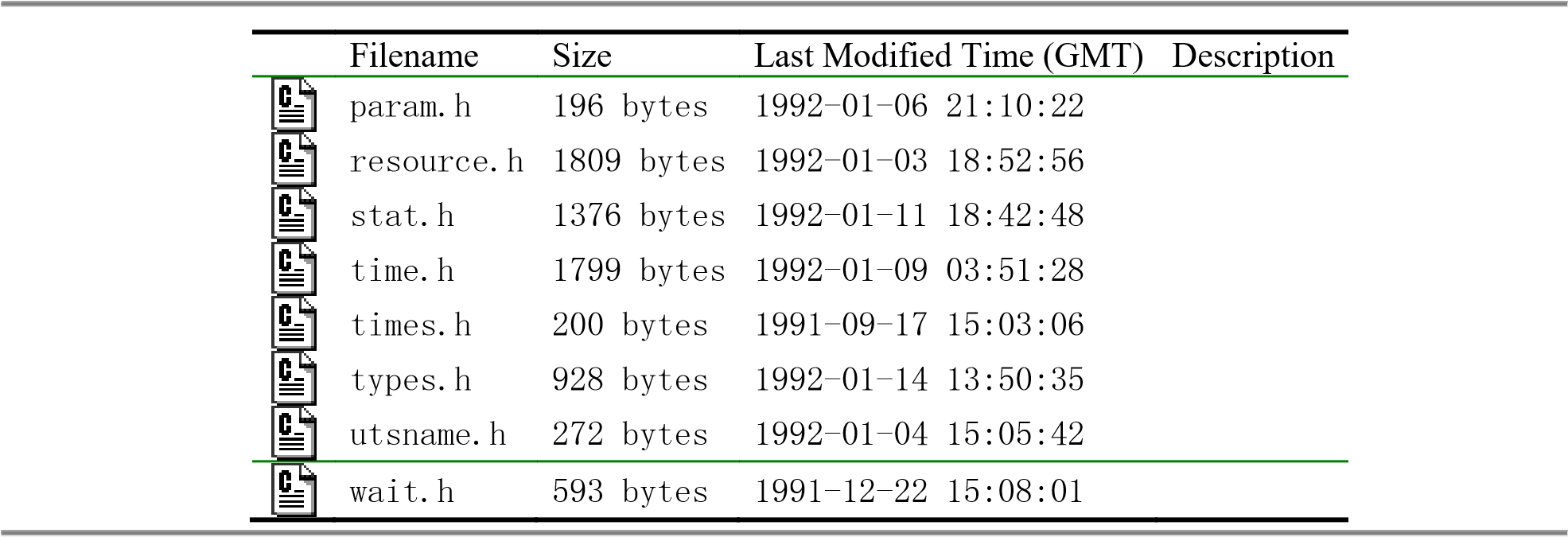
100

## 14.32 include/sys/ディレクトリ内のヘッダーファイル

## 14.32 Header files in the include/sys/ directory

include/sys/ディレクトリには，リスト14-4に示すように，システムのハードウェアリソースとその設定に密接に関連する8つのヘッダーファイルが含まれています。

リスト14-4 linux/include/sys/ディレクトリ内のファイル



## 14.33 param.h

14.33.1 **Functionality**

### param.hファイルには、システムのハードウェアに関連するいくつかのパラメータ値が含まれ、定義されています。

### 14.33.2 Code annotation

プログラム 14-29 linux/include/sys/param.h

|  |  |
| --- | --- |
| 1. #ifndef \_SYS\_PARAM\_H 2. #define \_SYS\_PARAM\_H   3  4 #define HZ 100 | // The system clock frequency, 100 times per second. |
| 5 #define EXEC\_PAGESIZE 4096 | // Executable page size. |
| 6  7 #define NGROUPS 32 | /\* Max number of groups per user \*/ |
| 8 #define NOGROUP -1  9  10 #define MAXHOSTNAMELEN 8  11  12 #endif  13 | // The maximum length of the host name, 8 bytes. |

## 14.34 resource.h

## 14.34 resource.h

### 14.34.1 Functionality

resource.hヘッダーファイルには、プロセスで使用されるシステムリソースの制限と利用に関する情報が含まれています。システムコール（またはライブラリ関数）のgetrusage()で使用されるrusage構造とシンボリック定数 RUSAGE\_SELF、RUSAGE\_CHILDRENを定義しています。また、システムコールまたは関数 getrlimit() および setrlimit() で使用される rlimit 構造体と、その引数で使用される記号定数を定義している。

getrlimit()やsetrlimit()でアクセスする情報は、プロセスタスク構造体のrlim[]配列にあります。この配列には合計でRLIM\_NLIMITS項目があり、各項目は図14-5に示すようにリソースの使用制限を定義するrlimit構造体である。図のように，Linux 0.12カーネルのプロセスには6つのリソース制限が定義されており，本ヘッダファイルの41～46行目で定義されている．

0

1

2

3

4

5

rlim[]

array in the process structure

rlimit structure

rlim\_cur

rlim\_max

RLIMIT\_CPU

RLIMIT\_FSIZE

RLIMIT\_DATA

RLIMIT\_STACK

RLIMIT\_CORE

RLIMIT\_RSS

図14-5 プロセス記述子でのrlim[]配列の使い方

### 14.34.2 Code annotation

プログラム 14-30 linux/include/sys/resource.h

1. /\*
2. \* Resource control/accounting header file for linux
3. \*/

4

1. #ifndef \_SYS\_RESOURCE\_H
2. #define \_SYS\_RESOURCE\_H 7
3. // 以下のシンボル定数および構造体がgetrusage()で使用されます。kernel/sys.c ファイルの 412 行目を参照してください。
4. /\*
5. \* Definition of struct rusage taken from BSD 4.3 Reno
6. \*
7. \* We don't support all of these yet, but we might as well have them.... 12 \* Otherwise, each time we add new items, programs which depend on this 13 \* structure will lose. This reduces the chances of that happening.
8. \*/

// 以下は、getrusage()のパラメータ 'who' で使用されるシンボル定数です。 // 現在のプロセスのリソース利用情報を返します。

14.34 resource.h

1. // 終了して待機している子プロセスのリソース使用率を返します。
2. #define RUSAGE\_SELF 0 // Returns resource utilization of the current process. 16 #define RUSAGE\_CHILDREN -1 // Returns resource utilization of the child processes.

17

// Rusage は、プロセスのリソース使用率の統計構造であり、以下のように使用されます。

// getrusage() で、指定されたリソースの使用率の統計値を返します。

// プロセスです。Linux 0.12カーネルでは，最初の2つのフィールドのみを使用しており，これらはすべてtimeval構造体です．

1. // (include/sys/time.h)を参照してください。ru\_utimeフィールドは、ユーザー状態でのユーザーの実行時間統計を // 保存するために使用されます。ru\_stimeフィールドは、カーネル状態でのプロセスの実行時間統計を // 保存するために使用されます。
2. struct rusage {
3. struct timeval ru\_utime; /\* user time used \*/
4. struct timeval ru\_stime; /\* system time used \*/
5. long ru\_maxrss; /\* maximum resident set size \*/
6. long ru\_ixrss; /\* integral shared memory size \*/
7. long ru\_idrss; /\* integral unshared data size \*/
8. long ru\_isrss; /\* integral unshared stack size \*/
9. long ru\_minflt; /\* page reclaims \*/
10. long ru\_majflt; /\* page faults \*/
11. long ru\_nswap; /\* swaps \*/
12. long ru\_inblock; /\* block input operations \*/
13. long ru\_oublock; /\* block output operations \*/
14. long ru\_msgsnd; /\* messages sent \*/
15. long ru\_msgrcv; /\* messages received \*/
16. long ru\_nsignals; /\* signals received \*/
17. long ru\_nvcsw; /\* voluntary context switches \*/
18. long ru\_nivcsw; /\* involuntary " \*/
19. }; 36

// getrlimit()とsetrlimit()で使用するシンボル定数と構造体を以下に示す。 37 /\*

38 \* リソースの制限 39 \*/

// Linux 0.12カーネルで定義されているリソースの種類は以下のとおりです。それらは、範囲

// getrlimit()およびsetrlimit()の最初のパラメータリソースの値のこと。実際には、 // これらのシンボリック定数は、プロセスタスクの rlim[] 配列の項目のインデックスです。

// 構造体です。rlim[]配列の各項目はrlimit構造体で、以下の58行目を参照してください。

40

1. #define RLIMIT\_CPU 0 /\* CPU time in ms \*/
2. #define RLIMIT\_FSIZE 1 /\* Maximum filesize \*/
3. #define RLIMIT\_DATA 2 /\* max data size \*/
4. #define RLIMIT\_STACK 3 /\* max stack size \*/
5. #define RLIMIT\_CORE 4 /\* max core file size \*/
6. #define RLIMIT\_RSS 5 /\* max resident set size \*/

47

1. #ifdef notdef
2. #define RLIMIT\_MEMLOCK 6 /\* max locked-in-memory address space\*/
3. #define RLIMIT\_NPROC 7 /\* max number of processes \*/
4. #define RLIMIT\_OFILE 8 /\* max number of open files \*/
5. #endif 53

// このシンボリック定数は、Linuxで制限されているリソースの種類を定義します。ここで定義されているリソースタイプの数は // 6つなので、最初の6つの項目だけが有効です。

14.35 stat.h

54 #define RLIM\_NLIMITS 6

55

56 #define RLIM\_INFINITY 0x7fffff // リソースは無制限であるか、変更できない。

57

1. // リソースリミット構造
2. struct rlimit {
3. int rlim\_cur; // Current resource limit, or soft limit.
4. int rlim\_max; // Hard limit.
5. };

62

63 #endif /\* \_SYS\_RESOURCE\_H \*/

64

## 14.35 stat.h

### 14.35.1 Functionality

### stat.hヘッダファイルには、ファイル関数stat()が返すデータとその構造型、およびプロパティ操作テスト用のマクロと関数プロトタイプが記載されています。

### 14.35.2 Code annotation

プログラム 14-31 linux/include/sys/stat.h

1. #ifndef \_SYS\_STAT\_H
2. #define \_SYS\_STAT\_H

3

4 #include <sys/types.h> 5

1. // ファイルステータスデータ構造。すべてのフィールド値は、ファイルのinode構造から得られます。
2. struct stat {
3. dev\_t st\_dev; // The device number that contains the file.
4. ino\_t st\_ino; // File i-node number.
5. umode\_t st\_mode; // File type and modes (see below).
6. nlink\_t st\_nlink; // The number of links to the file.
7. uid\_t st\_uid; // The user ID of the file.
8. gid\_t st\_gid; // The group ID of the file.
9. dev\_t st\_rdev; // Device number (if it is a special char or block file).
10. off\_t st\_size; // File size (in bytes).
11. time\_t st\_atime; // Last access time.
12. time\_t st\_mtime; // Last modify time.
13. time\_t st\_ctime; // The time the inode was last changed.
14. };

19

//

// st\_modeで使用する値に定義されているシンボル定数の一部をご紹介します。

//のフィールドになります。これらの値はすべて8進法で表されます。覚えやすくするために、これらの記号の

// 名前は、いくつかの英単語の頭文字や略語を組み合わせたものです。例えば、以下のようになります。

// S\_IFMTという名前の各大文字は、State、Inode、File、Mask、を表しています。

// また、S\_IFREGという名前は、State、Inode、Fileの頭文字を組み合わせたものです。

// S\_IRWXUという名前は、State、Inode、Read、Write、eXecute、Userを表しています。その他の名称

14.35 stat.h

// は、類推して推し量ることができます。

1. // ファイルタイプ
2. #define S\_IFMT 00170000 // File type bit mask (in octal).
3. #define S\_IFLNK 0120000 // Symbolic link.
4. #define S\_IFREG 0100000 // Regular file.
5. #define S\_IFBLK 0060000 // Block special device files, such as harddisk dev/hd0.
6. #define S\_IFDIR 0040000 // Directory.
7. #define S\_IFCHR 0020000 // Char device file.
8. #define S\_IFIFO 0010000 // FIFO special file.

// ファイルモードのビットです。

1. // S\_ISUID は，ファイルの set-user-ID フラグが設定されているかどうかをテストするために使用されます。フラグが設定されている場合， // ファイルが実行されたときに，プロセスの効率的なユーザIDは， // ファイル所有者のユーザIDに設定されます．S\_ISGID は，グループ ID に対しても同様の処理を行う．
2. #define S\_ISUID 0004000 // Set the user ID (set-user-ID) at execution time.
3. #define S\_ISGID 0002000 // Set the group ID (set-group-ID) at execution time.

/ #define S\_ISVTX 0001000 // ディレクトリの場合、削除制限フラグ。

30

1. #define S\_ISLNK(m) (((m) & S\_IFMT) == S\_IFLNK) // Test if it's a symbolic link file.
2. #define S\_ISREG(m) (((m) & S\_IFMT) == S\_IFREG) // a regular file.
3. #define S\_ISDIR(m) (((m) & S\_IFMT) == S\_IFDIR) // a directory.
4. #define S\_ISCHR(m) (((m) & S\_IFMT) == S\_IFCHR) // a char device file.
5. #define S\_ISBLK(m) (((m) & S\_IFMT) == S\_IFBLK) // a block device file. 36 #define S\_ISFIFO(m) (((m) & S\_IFMT) == S\_IFIFO) // a FIFO special file.

37

1. // ファイルのアクセス許可。
2. #define S\_IRWXU 00700 // The owner can read, write, execute/search (U for User).
3. #define S\_IRUSR 00400 // The owner can read.
4. #define S\_IWUSR 00200 // The owner can write.
5. #define S\_IXUSR 00100 // The owner can execute/search.



1. #define S\_IRWXG 00070 // Group members can read, write, execute/search (G for Group).
2. #define S\_IRGRP 00040 // Group members can read.
3. #define S\_IWGRP 00020 // Group members can write.

/ #define S\_IXGRP 00010 // グループメンバーが実行/検索できる。

47

1. #define S\_IRWXO 00007 // Others can read, write, execute/search (O stands for Other).
2. #define S\_IROTH 00004 // Others can read (the last 3 letters represent Other).
3. #define S\_IWOTH 00002 // Others can write.
4. #define S\_IXOTH 00001 // Others can execute/search.



1. extern int chmod(const char \*\_path, mode\_t mode); // Change file modes.
2. extern int fstat(int fildes, struct stat \*stat\_buf); // Get file state info by fhandle.
3. extern int mkdir(const char \*\_path, mode\_t mode); // Make a directory.
4. extern int mkfifo(const char \*\_path, mode\_t mode); // Make a pipe file. 57 extern int stat(const char \*filename, struct stat \*stat\_buf); // Get file state info.

/ extern mode\_t umask(mode\_t mask); // モードマスクを設定する。

59

60 #endif

61

## 14.36 time.h

## 14.36 time.h

### 14.36.1 Functionality

### time.hヘッダーファイルでは、timeval構造体と内部で使用されるitimerval構造体、およびタイムゾーン定数が定義されています。

### 14.36.2 Code annotation

プログラム 14-32 linux/include/sys/time.h

1. #ifndef \_SYS\_TIME\_H
2. #define \_SYS\_TIME\_H

3

1. /\* gettimofday returns this \*/
2. struct timeval {
3. long tv\_sec; /\* seconds \*/
4. long tv\_usec; /\* microseconds \*/
5. };

9

// タイムゾーンの構造

1. // TZはTime Zoneの略で、DSTはDaylight Saving Timeの略です。
2. struct timezone {
3. int tz\_minuteswest; /\* minutes west of Greenwich \*/
4. int tz\_dsttime; /\* type of dst correction \*/
5. };

14

1. #define DST\_NONE 0 /\* not on dst \*/
2. #define DST\_USA 1 /\* USA style dst \*/
3. #define DST\_AUST 2 /\* Australian style dst \*/
4. #define DST\_WET 3 /\* Western European dst \*/ 19 #define DST\_MET 4 /\* Middle European dst \*/
5. #define DST\_EET 5 /\* Eastern European dst \*/
6. #define DST\_CAN 6 /\* Canada \*/
7. #define DST\_GB 7 /\* Great Britain and Eire \*/
8. #define DST\_RUM 8 /\* Rumania \*/
9. #define DST\_TUR 9 /\* Turkey \*/
10. #define DST\_AUSTALT 10 /\* Australian style with shift in 1986 \*/ 26
11. // select()関数で使用する、ファイルディスクリプターセットの設定マクロです。
12. #define FD\_SET(fd,fdsetp) (\*(fdsetp) |= (1 << (fd))) // Set the fd in the fd set.
13. #define FD\_CLR(fd,fdsetp) (\*(fdsetp) &= ~(1 << (fd))) // Clear the fd in the set.
14. #define FD\_ISSET(fd,fdsetp) ((\*(fdsetp) >> fd) & 1) // Is the fd in the set ? 30 #define FD\_ZERO(fdsetp) (\*(fdsetp) = 0) // Clear all fds in the set.

31

1. /\*
2. \* Operations on timevals.
3. \*
4. \* NB: timercmp does not work for >= or <=.
5. \*/
6. // timeval時間構造体の操作マクロです。
7. #define timerisset(tvp) ((tvp)->tv\_sec || (tvp)->tv\_usec)
8. 14.37倍.h
9. #define timercmp(tvp, uvp, cmp) \
10. ((tvp)->tv\_sec cmp (uvp)->tv\_sec || \
11. (tvp)->tv\_sec == (uvp)->tv\_sec && (tvp)->tv\_usec cmp (uvp)->tv\_usec)
12. #define timerclear(tvp) ((tvp)->tv\_sec = (tvp)->tv\_usec = 0) 42
13. /\*
14. \* Names of the interval timers, and structure 45 \* defining a timer setting.
15. \*/
16. #define ITIMER\_REAL 0 // Decrease in real time.
17. #define ITIMER\_VIRTUAL 1 // Decrease in the virtual time of the process. 49 #define ITIMER\_PROF 2 // Decrease in process virtual time or runtime.

50

1. // 内部の時間構造
2. struct itimerval {
3. struct timeval it\_interval; /\* timer interval \*/
4. struct timeval it\_value; /\* current value \*/
5. };

55

1. #include <time.h>
2. #include <sys/types.h>

58

1. int gettimeofday(struct timeval \* tp, struct timezone \* tz);
2. int select(int width, fd\_set \* readfds, fd\_set \* writefds,
3. fd\_set \* exceptfds, struct timeval \* timeout);

62

63 #endif /\*\_SYS\_TIME\_H\*/

64

## 14.37 times.h

### 14.37.1 Functionality

times.hヘッダーファイルは、主にファイルのアクセスと変更の時間構造tmsを定義しています。これは、times()関数によって返されます。time\_t は sys/types.h で定義されています。また、関数プロトタイプ times() も定義されています。

### 14.37.2 Code annotation

プログラム 14-33 linux/include/sys/times.h

1. #ifndef \_TIMES\_H
2. #define \_TIMES\_H

3

4 #include <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。

5

1. struct tms {
2. time\_t tms\_utime; // CPU time used by the user.
3. time\_t tms\_stime; // CPU time used by the system (kernel).
4. 14.38タイプ.h
5. time\_t tms\_cutime; // User CPU time used by the terminated child process.
6. time\_t tms\_cstime; // The system CPU time used by the terminated child.
7. };

12

13 extern time\_t times(struct tms \* tp);

14

15 #endif

16

## 14.38 types.h

### 14.38.1 Functionality

types.hヘッダーファイルでは、基本的なデータ型を定義しています。すべての型は、適切な数学的型の長さで定義されています。また、size\_tは符号なし整数型、off\_tは拡張符号付き整数型、pid\_tは符号付き整数型となっています。

### 14.38.2 Code annotation

プログラム 14-34 linux/include/sys/types.h

1. #ifndef \_SYS\_TYPES\_H
2. #define \_SYS\_TYPES\_H

3

1. #ifndef \_SIZE\_T
2. #define \_SIZE\_T
3. typedef unsigned int size\_t; // Used for the size (length) of the object.
4. #endif

8

1. #ifndef \_TIME\_T
2. #define \_TIME\_T
3. typedef long time\_t; // Used for time (in seconds).
4. #endif

13

1. #ifndef \_PTRDIFF\_T
2. #define \_PTRDIFF\_T
3. typedef long ptrdiff\_t;
4. #endif

18

1. #ifndef NULL
2. #define NULL ((void \*) 0)
3. #endif

22

1. typedef int pid\_t; // Used for process id and process group id.
2. typedef unsigned short uid\_t; // Used for the user id.
3. typedef unsigned char gid\_t; // Used for the group id.
4. typedef unsigned short dev\_t; // Used for the device number.
5. typedef unsigned short ino\_t; // Used for the inode number. 28 typedef unsigned short mode\_t; // Used for some file modes.
6. 14.39 utsname.h
7. typedef unsigned short umode\_t; //
8. typedef unsigned char nlink\_t; // Used for the file links counting.
9. typedef int daddr\_t;
10. typedef long off\_t; // Used for the offset in a file.
11. typedef unsigned char u\_char; // unsigned char.
12. typedef unsigned short ushort; // unsigned short.

35

1. typedef unsigned char cc\_t;
2. typedef unsigned int speed\_t;
3. typedef unsigned long tcflag\_t;

39

40 typedef unsigned long fd\_set; // ファイルディスクリプターのセット。各ビットが1つのディスクリプターを表します。

41

1. typedef struct { int quot,rem; } div\_t; // Used for DIV operation.
2. typedef struct { long quot,rem; } ldiv\_t; // Used for long DIV operation.

44

1. // ustat()関数のファイルシステムパラメータ構造。最後の2つのフィールドは未使用で， // 常にNULLを返します。
2. struct ustat {
3. daddr\_t f\_tfree; // Total free blocks in the system.
4. ino\_t f\_tinode; // Total free inodes.
5. char f\_fname[6]; // File system name.
6. char f\_fpack[6]; // The packed file system name.
7. };

51

52 #endif

53

## 14.39 utsname.h

### 14.39.1 Functionality

### utsname.hは、システム名構造体のヘッダファイルで、utsname構造体と関数プロトタイプuname()を定義している。utsname構造体と関数プロトタイプuname()が定義されている。この関数はutsname構造体の情報を用いて、システム識別子、バージョン番号、ハードウェアタイプなどの情報を与える。POSIXでは、文字配列のサイズは不定ですが、文字配列に格納されるデータはNULL終端でなければなりません。そのため、カーネルのutsname構造体の定義はPOSIXの要件を満たしていません（文字列配列のサイズは9と定義されています）。また、utsnameという名前は、Unix Timesharing System nameの略語である。

### 14.39.2 Code annotation

プログラム 14-35 linux/include/sys/utsname.h

1. #ifndef \_SYS\_UTSNAME\_H
2. #define \_SYS\_UTSNAME\_H

3

1. #include <sys/types.h> // The basic system data types are defined.
2. #include <sys/param.h> // Some hardware-related parameter values are given. 5
3. 14.40 wait.h
4. struct utsname {
5. char sysname[9]; // system name.
6. char nodename[MAXHOSTNAMELEN+1]; // The node name in the network.
7. char release[9]; // release level. 10 char version[9]; // version. 11 char machine[9]; // hardware type.

12 };

13

14 extern int uname(struct utsname \* utsbuf);

15

16 #endif

17

## 14.40 wait.h

### 14.40.1 Functionality

このヘッダファイルは、プロセスが待機しているときの情報を記述したもので、いくつかのシンボル定数やwait()、waitpid()関数のプロトタイプ宣言が含まれています。

### 14.40.2 Code annotation

プログラム 14-36 linux/include/sys/wait.h

1. #ifndef \_SYS\_WAIT\_H
2. #define \_SYS\_WAIT\_H

3

4 #include <sys/types.h>.

5

1. #define \_LOW(v) ( (v) & 0377) // Get the low byte (in octal).
2. #define \_HIGH(v) ( ((v) >> 8) & 0377) // Get the high byte.

8

1. /\* options for waitpid, WUNTRACED not supported \*/

// [ 注：実際には、0.12のカーネルはすでにWUNTRACEDオプションをサポートしています].

1. // 以下の定数記号は，関数waitpid()で使用されるオプションです。
2. #define WNOHANG 1 // Don't hang and return immediately.
3. #define WUNTRACED 2 // Reports the child status that was stopped.

12

1. // マクロは、waitpid()が返すステータスワードの意味を確認するために使用されます。
2. #define WIFEXITED(s) (!((s)&0xFF) // True if the child exits normally.
3. #define WIFSTOPPED(s) (((s)&0xFF)==0x7F) // True if the child is stopping.
4. #define WEXITSTATUS(s) (((s)>>8)&0xFF) // The exit status.
5. #define WTERMSIG(s) ((s)&0x7F) // Signal that caused process to terminate.
6. #define WCOREDUMP(s) ((s)&0x80) // Check if a core dump has been performed.
7. #define WSTOPSIG(s) (((s)>>8)&0xFF) // Signal that caused process to stop. // True if the child process exited due to an uncaptured signal.

/ #define WIFSIGNALED(s) (((unsigned int)(s)-1 & 0xFFFF) < 0xFF)

20

// wait()およびwaitpit()関数は、プロセスが自分の

14.41 まとめ

// 子プロセス。この関数の様々なオプションにより、 // 終了または停止した子プロセスのステータス情報を取得することができます。2つ以上の子プロセスのステータス情報が // ある場合は、報告の順序は指定されません。

1. // wait()は、子プロセスの1つが終了(ターミネイト)するか、 // プロセスの終了を要求するシグナルを受け取るか、シグナルハンドラを呼び出す必要があるまで、現在のプロセスを一時停止します。 // Waitpid() は、pid で指定された子が終了するか、 // プロセスの終了を要求するシグナルを受け取るか、 // シグナルハンドラを呼び出す必要があるまで、現在のプロセスを一時停止します。 // pid= -1, options=0 の場合，waitpid() は wait() 関数と同じ動作をしますが，そうでない場合は， // pid と options のパラメータによって動作が異なります（kernel/exit.c, 142 を参照）． // パラメータの'pid'はプロセスID、'\*stat\_loc'はステータス情報の // 位置へのポインタ、'options'はwaitオプションで、上記10,11行目を参照してください。
2. pid\_t wait(int \*stat\_loc);
3. pid\_t waitpid(pid\_t pid, int \*stat\_loc, int options);



24 #endif

25

## 14.41 Summary

本章では，カーネルが使用するすべてのヘッダファイルについて説明します。次の章からは，カーネルが使用するライブラリファイルのコードを紹介します。これらのライブラリファイルのコードは，カーネルのコンパイル時にカーネルコードにリンクされます。

# 14.41 まとめ

# 15 Library files (lib)

C言語ライブラリは、再利用可能なプログラムモジュールの集まりで、Linuxカーネルライブラリファイルは、カーネルで使用するためにコンパイルされた、よく使われるいくつかの関数の組み合わせです。リスト15-1のC言語ファイルは，カーネルライブラリファイルのモジュールを構成するプログラムです。主に，プロセスの実行・終了関数，ファイルアクセス操作関数，メモリ確保関数，文字列操作関数などが含まれています。

具体的には、終了関数\_exit()、ファイルクローズ関数close()、ファイルディスクリプターコピー関数dup()、ファイルオープン関数open()、ファイルライト関数write()、プログラム実行関数execve()、メモリ確保関数malloc()、子プロセスの状態待ち関数wait()、セッション作成システムコールsetsid()、include/string.hで実装されているすべての文字列操作関数が実装されている。

タイツォさんが書いたmalloc.cのプログラムを除けば、プログラムのサイズは非常に短く、中には1〜2行のコードしかないものもあります。それらは基本的にシステムコールを直接呼び出して機能を実現している。

|  |  |  |
| --- | --- | --- |
| リスト15-1 /linux/lib/ディレクトリのファイル  Filename | Size Last Modified Time(GMT) Description | |
| Makefile | 2602 bytes 1991-12-02 03:16:05 | |
| \_exit.c | 198 bytes 1991-10-02 14:16:29 | |
| close.c | 131 bytes 1991-10-02 14:16:29 | |
| ctype.c | 1202 bytes 1991-10-02 14:16:29 | |
| dup.c | 127 bytes 1991-10-02 14:16:29 | |
| errno.c | 73 bytes 1991-10-02 14:16:29 | |
| execve.c | 170 bytes 1991-10-02 14:16:29 | |
| malloc.c | 7469 bytes 1991-12-02 03:15:20 | |
| open.c | 389 bytes | 1991-10-02 14:16:29 |
| setsid.c | 128 bytes | 1991-10-02 14:16:29 |
| string.c | 177 bytes | 1991-10-02 14:16:29 |
| wait.c | 253 bytes | 1991-10-02 14:16:29 |
| write.c | 160 bytes | 1991-10-02 14:16:29 |

カーネルのコンパイル段階では、カーネルMakefileの関連命令によって、これらのプログラムが.oモジュールにコンパイルされた後、lib.aライブラリにビルドされ、カーネルモジュールにリンクされます。このライブラリは、通常のコンパイル環境で提供される様々なライブラリファイル（gccが提供するlibc.aやlibufc.aなど）とは異なり、主にカーネルの初期化段階のinit/main.cプログラムで、ユーザモードでinit()関数を実行するために使用される関数です。そのため、含まれる関数は少なく、非常にシンプルなものとなっています。

しかし、一般的なライブラリと全く同じ方法で実装されています。

関数ライブラリの作成には、通常、ar（archive abbreviation）というコマンドを使用します。例えば、3つのモジュールa.o、b.o、c.oを持つ関数ライブラリlibmine.aを作成するには、以下のコマンドを実行する必要があります。

15.1 \_exit.c

ar -rc libmine.a a.o b.o c.o d.o

このライブラリファイルに関数モジュールdup.oを追加するには、以下のコマンドを実行します。

ar

-

rs dup.o

## 15.1 \_exit.c

15.1.1 機能性

### \_exit.cファイルは、プログラムがカーネルのexitシステムコール関数を呼び出すために使用されます。

### 15.1.2 Code annotation

プログラム 15-1 linux/lib/\_exit.c

1. /\*
2. \* linux/lib/\_exit.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/ 6

// <unistd.h> Linux標準のヘッダファイルです。様々なシンボル定数や型が定義され、 // 様々な関数が宣言されています。また、'\_\_LIBRARY\_\_'が定義されている場合、その中には

1. // システムコール番号とインラインアセンブリ\_syscall0()を指定します。
2. #define \_\_LIBRARY\_\_
3. #include <unistd.h>

9

//// プログラムが終了（ターミネイト）する機能です。

// ライブラリ関数は、システム割り込みint 0x80、関数番号 \_\_NR\_exit を直接呼び出します。 // パラメータ：exit\_code - 終了コード。

1. // 関数名の前のキーワード 'volatile' 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用します。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、いくつかの誤った警告を避けることができます。
2. volatile void \_exit(int exit\_code)
3. {
4. // %0 - eax(\_\_NR\_exit)；%1 - ebx(exit\_code)。
5. \_\_asm\_\_("int $0x80"::"a" (\_\_NR\_exit),"b" (exit\_code));
6. }

14

### 15.1.3 Information

システムコールの割り込み番号の説明は、include/unistd.hファイルの記述を参照してください。

## 15.2 close.c

## 15.2 close.c

15.2.1 機能性

### ファイルクローズ関数close()は、close.cファイルで定義されています。

### 15.2.2 Code annotation

プログラム 15-2 linux/lib/close.c

1. /\*
2. \* linux/lib/close.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/ 6

// <unistd.h> Linux標準のヘッダファイルです。様々なシンボル定数や型が定義され、 // 様々な関数が宣言されています。また、'\_\_LIBRARY\_\_'が定義されている場合、その中には

1. // システムコール番号とインラインアセンブリ\_syscall0()を指定します。
2. #define \_\_LIBRARY\_\_
3. #include <unistd.h>

9

// ファイル機能を閉じる。

// 次のマクロは、関数プロトタイプに対応しています: int close(int fd). これは、パラメータ \_\_NR\_close で、 // システムの int 0x80 を直接呼び出します。fd はファイル記述子です。

10 \_syscall1(int,close,int,fd)

11

## 15.3 ctype.c

### 15.3.1 Functionality

### ctype.cプログラムは、文字の種類を決定するための補助的な配列構造データを、ctype.hに提供するために使用されます。

### 15.3.2 Code annotation

プログラム 15-3 linux/lib/ctype.c

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

6

// <ctype.h> 文字型ファイルです。文字型変換のためのマクロを定義しています。

1. 7 #include <ctype.h> 8
2. char \_ctmp; // a tem variable for macros that convert characters in the ctype.h file.

15.4 dup.c

// 以下は、対応する属性を定義する文字属性の配列です。

1. // を各文字に設定します。これらの属性タイプ（ \_C など）は ctype.h で定義されており、 // その文字が制御文字（ \_C ）、大文字（ \_U ）、 // 小文字（ \_L ）などであるかどうかをチェックするために使用されます。
2. unsigned char \_ctype[] = {0x00, /\* EOF \*/
3. \_C,\_C,\_C,\_C,\_C,\_C,\_C,\_C, /\* 0-7 \*/
4. \_C,\_C|\_S,\_C|\_S,\_C|\_S,\_C|\_S,\_C|\_S,\_C,\_C, /\* 8-15 \*/
5. \_C,\_C,\_C,\_C,\_C,\_C,\_C,\_C, /\* 16-23 \*/
6. \_C,\_C,\_C,\_C,\_C,\_C,\_C,\_C, /\* 24-31 \*/
7. \_S|\_SP,\_P,\_P,\_P,\_P,\_P,\_P,\_P, /\* 32-39 \*/
8. \_P,\_P,\_P,\_P,\_P,\_P,\_P,\_P, /\* 40-47 \*/
9. \_D,\_D,\_D,\_D,\_D,\_D,\_D,\_D, /\* 48-55 \*/
10. \_D,\_D,\_P,\_P,\_P,\_P,\_P,\_P, /\* 56-63 \*/
11. \_P,\_U|\_X,\_U|\_X,\_U|\_X,\_U|\_X,\_U|\_X,\_U|\_X,\_U, /\* 64-71 \*/
12. \_U,\_U,\_U,\_U,\_U,\_U,\_U,\_U, /\* 72-79 \*/
13. \_U,\_U,\_U,\_U,\_U,\_U,\_U,\_U, /\* 80-87 \*/
14. \_U,\_U,\_U,\_P,\_P,\_P,\_P,\_P, /\* 88-95 \*/
15. \_P,\_L|\_X,\_L|\_X,\_L|\_X,\_L|\_X,\_L|\_X,\_L|\_X,\_L, /\* 96-103 \*/
16. \_L,\_L,\_L,\_L,\_L,\_L,\_L,\_L, /\* 104-111 \*/
17. \_L,\_L,\_L,\_L,\_L,\_L,\_L,\_L, /\* 112-119 \*/
18. \_L,\_L,\_L,\_P,\_P,\_P,\_P,\_C, /\* 120-127 \*/
19. 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /\* 128-143 \*/
20. 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /\* 144-159 \*/
21. 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /\* 160-175 \*/
22. 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /\* 176-191 \*/
23. 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /\* 192-207 \*/
24. 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /\* 208-223 \*/
25. 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, /\* 224-239 \*/ 34 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; /\* 240-255 \*/

35

36

## 15.4 dup.c

### 15.4.1 Functionality

dup.cプログラムには、ファイル記述子のコピーを作成する関数dup()が含まれています。成功して戻ってくると，新しいディスクリプターと元のディスクリプターを入れ替えて使うことができる。両者は，ロック，ファイル読み書きポインタ，ファイルフラグを共有している。例えば、lseek()を使って一方のディスクリプターでファイルの読み書き位置のポインタが変更された場合、もう一方のディスクリプターでもファイルの読み書きポインタが変更されます。この関数は、使用されていない最小のディスクリプターを使用して新しいディスクリプターを作成しますが、2つのディスクリプターはclose-on-execフラグを共有しません。

### 15.4.2 Code annotation

プログラム 15-4 linux/lib/dup.c

1. /\*
2. \* linux/lib/dup.c
3. 15.5 errno.c
4. \*
5. \* (C) 1991 Linus Torvalds
6. \*/ 6

// <unistd.h> Linux標準のヘッダファイルです。様々なシンボル定数や型が定義され、 // 様々な関数が宣言されています。また、'\_\_LIBRARY\_\_'が定義されている場合、その中には

1. // システムコール番号とインラインアセンブリ\_syscall0()を指定します。
2. #define \_\_LIBRARY\_\_
3. #include <unistd.h>

9

//// ファイル記述子（ハンドル）機能の重複。

// 以下のマクロは、関数プロトタイプであるint dup(int fd)に対応しています。このマクロは、システムのint 0x80を直接呼び出し、パラメータは\_\_NR\_dupです。fd はファイル記述子です。

10 \_syscall1(int,dup,int,fd)

11

## 15.5 errno.c

### 15.5.1 Functionality

### プログラムでは、関数呼び出しに失敗したときのエラー番号を格納するための変数errnoのみを定義しています。include/errno.hファイルの記述を参照してください。

### 15.5.2 Code annotation

プログラム 15-5 linux/lib/errno.c

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

6

7 int errno;

8

## 15.6 execve.c

15.6.1 機能性

### execve.cプログラムには、実行ファイルを実行するシステムコール関数が含まれています。

### 15.6.2 Code annotation

プログラム 15-6 linux/lib/execve.c

/ 1 /\*

1. \* linux/lib/execve.c
2. \*
3. \* (C) 1991 Linus Torvalds
4. \*/ 6

// <unistd.h> Linux標準のヘッダファイルです。様々なシンボル定数や型が定義され、 // 様々な関数が宣言されています。また、'\_\_LIBRARY\_\_'が定義されている場合、その中には

1. // システムコール番号とインラインアセンブリ\_syscall0()を指定します。
2. #define \_\_LIBRARY\_\_
3. #include <unistd.h>

9

//// 子プロセス（他のプログラム）機能をロードして実行します。

// このマクロは，関数int execve(const char \* file, char \*\* argv, char \*\* envp)に対応しています．

// パラメータ： file - 実行ファイル名、 argv - コマンドライン引数のポインタの配列、 // envp - 環境変数のポインタの配列。システムを直接呼び出します int 0x80, // パラメータは \_\_NR\_execve. include/unistd.h および fs/exec.c を参照してください。

10 \_syscall3(int,execve,const char \*,file,char \*\*,argv,char \*\*,envp)

11

## 15.7 malloc.c

### 15.7.1 Functionality

malloc.cプログラムには、主にメモリ割り当て関数malloc()が含まれています。ユーザープログラムが使用するmalloc()関数と混同しないように、カーネルバージョン0.98からはkmalloc()と呼ばれ、free\_s()関数はkfree\_s()に改称されています。

なお、アプリケーションが使用する同名のメモリ確保関数は、一般的にGCC環境のlibc.aライブラリのように、開発環境のライブラリファイルに実装されています。開発環境のライブラリ関数は、それ自体がユーザープログラムにリンクされているため、カーネルのget\_free\_page()などの関数を直接使用してメモリ確保関数を実装することはできません。もちろん、libc.aライブラリのメモリ割り当て機能は、プログラムの要求に応じてプロセスデータセグメントの末尾の設定値を動的に調整すればよく、エンドスタックや環境パラメータ領域まではカバーしていないので、メモリページを直接管理する必要もない。それ以外の具体的なメモリマッピングなどの操作は、カーネルが行う。このプロセスデータセグメントの終了位置を調整する操作が、ライブラリのメモリ割り当て関数の主目的であり、カーネルのシステムコールであるbrk()が呼び出される。kernel/sys.cプログラムの228行目を参照。つまり、開発環境のライブラリ関数実装のソースコードを見ることができれば、malloc()やcalloc()などのメモリ割り当て関数は、動的なアプリケーションメモリ領域の管理に加えて、カーネルのシステムコールであるbrk()を呼び出しているだけであることがわかります。開発環境ライブラリのメモリ確保関数は、確保したメモリを動的に管理するという点では、ここで紹介した関数と同じです。その管理方法も基本的には同じです。

malloc()関数は、割り当てられたメモリを管理するために、バケットの原理を利用しています。基本的な考え方は、要求されたメモリブロックのサイズが異なる場合に、バケットディレクトリ（以下、ディレクトリ）を使用するというものです。例えば，要求されたメモリブロックのサイズが32バイト以下で16バイト以上の場合，バケットディレクトリの2番目の項目に対応するバケットディスクリプタリストを使用してメモリブロックが割り当てられる。基本構造を図15-1に示す。本機能が一度に割り当てることのできる最大のメモリサイズは1メモリページ（4096バイト）である。

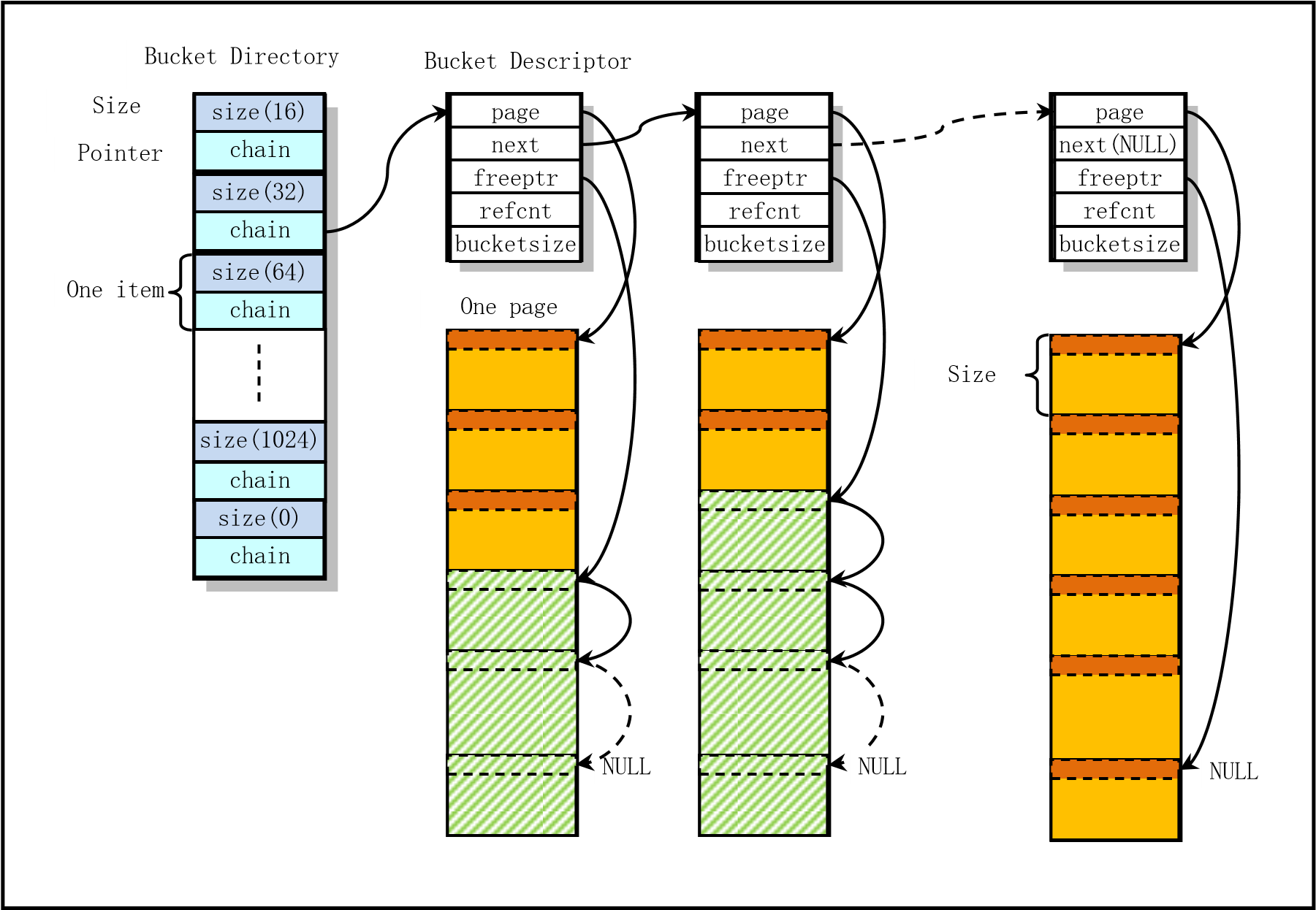


図15-1 バケットの原理を利用したメモリの割り当て管理

malloc()関数を初めて呼び出したときには、まず、そのページのフリーバケットディスクリプターリストを作成する必要があります。このリストには、未使用または再利用されたディスクリプターが格納されています。リンクリストの構造を図15-2に示しますが、free\_bucket\_descはリンクリストのヘッダーへのポインタです。リンクリストからの記述子の取り出し/入れは、リンクリストの先頭から始まる。記述子が取り出されると，ヘッダポインタが指す最初の記述子が取り出され，アイドル状態の記述子が解放されると，その記述子もリストの先頭に置かれる。

page

next

freeptr

refcnt

bucketsize

page

next

freeptr

refcnt

bucketsize

page

next

freeptr

refcnt

bucketsize

page

next(NULL)

freeptr

refcnt

bucketsize

free\_bucket\_desc

Bucket Descriptor

図15-2 フリーバケット記述子のチェーンテーブル構造

システムの実行中、ある時刻にすべてのバケット記述子が占有されていた場合、free\_bucket\_descはNULLとなる。そのため、バケットディスクリプターが解放されないまま、次にフリーのバケットディスクリプターを使用する必要が生じた場合、プログラムは図のように再びページを申請し、その上に新しいフリーのバケットディスクリプターリストを作成します。

1. malloc()関数の基本的な手順は以下の通りです。
2. First search the directory and look for the descriptor list corresponding to the directory entry that matches the size of the requested memory block. When the memory size of the directory entry is larger than the requested byte size, the corresponding directory entry is found. If the search for the entire directory does not find a suitable directory entry, then the memory block requested by the user is too large.
3. Find a descriptor with free space in the descriptor list corresponding to the directory entry. If the free memory pointer freeptr of a descriptor is not NULL, it means that the corresponding descriptor is found. If we don't find a descriptor with free space, then we need to create a new descriptor. The steps to create a new descriptor are as follows:
   1. If the idle descriptor list header pointer is still NULL, it means that the malloc() function is called for the first time, or all empty bucket descriptors are used up. In this case, you need to use the function init\_bucket\_desc() to create a list of idle descriptors.
   2. Then get a descriptor from the header of the idle descriptor chain, initialize the descriptor, make its object reference count 0, the object size is equal that of the directory entry, and apply for a memory page, let the descriptor page pointer points to the memory page, and the free memory pointer of the descriptor also points to the beginning of the page.
   3. c. Initialize the page for the memory page according to the size of the object used in this directory entry, and establish a linked list of all objects. That is, each object's head stores a pointer to the next object, and the last object stores a NULL at the beginning.
   4. Then insert the descriptor into the beginning of the descriptor list for the corresponding directory entry.
4. Copy the free memory pointer freeptr of the descriptor to the memory pointer returned to the user, and then adjust the freeptr to point to the next free object location in the memory page corresponding to the descriptor, and increment the descriptor reference count by one.

free\_s()関数は、ユーザーが解放したメモリブロックを再利用するために使用されます。基本的な方法は、まずメモリブロックのアドレスに応じて対応するページのアドレスを変換し、ディレクトリ内のすべてのディスクリプタを検索して、そのページに対応するディスクリプタを見つけるというものである。解放されたメモリブロックは、freeptrが指すフリーオブジェクトリストにチェーンされ、ディスクリプターのオブジェクト参照カウント値が1つデクリメントされる。このとき、参照カウント値が0になっていれば、ディスクリプターに対応するページが完全にフリーになったことを意味し、メモリページを解放してディスクリプターをアイドルディスクリプターリストに格納することができる。

### 15.7.2 Code annotation

プログラム 15-7 linux/lib/malloc.c

1. /\*
2. \* malloc.c --- a general purpose kernel memory allocator for Linux.
3. \*
4. \* Written by Theodore Ts'o (tytso@mit.edu), 11/29/91
5. \*
6. \* This routine is written to be as fast as possible, so that it
7. \* can be called from the interrupt level.
8. \*
9. \* Limitations: maximum size of memory we can allocate using this routine 10 \* is 4k, the size of a page in Linux.
10. \*
11. \* The general game plan is that each page (called a bucket) will only hold
12. \* objects of a given size. When all of the object on a page are released,
13. \* the page can be returned to the general free pool. When malloc() is
14. \* called, it looks for the smallest bucket size which will fulfill its 16 \* request, and allocate a piece of memory from that bucket pool.
15. \*
16. \* Each bucket has as its control block a bucket descriptor which keeps
17. \* track of how many objects are in use on that page, and the free list
18. \* for that page. Like the buckets themselves, bucket descriptors are
19. \* stored on pages requested from get\_free\_page(). However, unlike buckets,
20. \* pages devoted to bucket descriptor pages are never released back to the
21. \* system. Fortunately, a system should probably only need 1 or 2 bucket
22. \* descriptor pages, since a page can hold 256 bucket descriptors (which
23. \* corresponds to 1 megabyte worth of bucket pages.) If the kernel is using 26 \* that much allocated memory, it's probably doing something wrong. :-) 27 \*
24. \* Note: malloc() and free() both call get\_free\_page() and free\_page()
25. \* in sections of code where interrupts are turned off, to allow 30 \* malloc() and free() to be safely called from an interrupt routine.
26. \* (We will probably need this functionality when networking code,
27. \* particularily things like NFS, is added to Linux.) However, this
28. \* presumes that get\_free\_page() and free\_page() are interrupt-level
29. \* safe, which they may not be once paging is added. If this is the
30. \* case, we will need to modify malloc() to keep a few unused pages 36 \* "pre-allocated" so that it can safely draw upon those pages if 37 \* it is called from an interrupt routine.
31. \*
32. \* Another concern is that get\_free\_page() should not sleep; if it
33. \* does, the code is carefully ordered so as to avoid any race
34. \* conditions. The catch is that if malloc() is called re-entrantly,
35. \* there is a chance that unecessary pages will be grabbed from the
36. \* system. Except for the pages for the bucket descriptor page, the
37. \* extra pages will eventually get released back to the system, though, 45 \* so it isn't all that bad.

46 \*/ 47

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

// <linux/mm.h> メモリ管理用のヘッダーファイルです。ページサイズの定義と、いくつかのページ

// 関数のプロトタイプをリリースします。

// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。

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

51

1. // バケットディスクリプター構造
2. struct bucket\_desc { /\* 16 bytes \*/
3. void \*page; // Memory page pointer.
4. struct bucket\_desc \*next; // The next descriptor pointer.
5. void \*freeptr; // A pointer to the free memory.
6. unsigned short refcnt; // Refernce count.
7. unsigned short bucket\_size; // The size of the bucket.
8. };

59

1. // バケットディスクリプターのディレクトリ構造
2. struct \_bucket\_dir { /\* 8 bytes \*/
3. int size; // The size (in bytes) of this bucket.
4. struct bucket\_desc \*chain; // Bucket descriptor list pointer.
5. }; 64
6. /\*
7. \* The following is the where we store a pointer to the first bucket 67 \* descriptor for a given size.
8. \*
9. \* If it turns out that the Linux kernel allocates a lot of objects of a
10. \* specific size, then we may want to add that specific size to this list,
11. \* since that will allow the memory to be allocated more efficiently.
12. \* However, since an entire page must be dedicated to each specific size 73 \* on this list, some amount of temperance must be exercised here.
13. \*
14. \* Note that this list \*must\* be kept in order.
15. \*/
16. // バケットのディレクトリリストです。
17. struct \_bucket\_dir bucket\_dir[] = {
18. { 16, (struct bucket\_desc \*) 0}, // A 16-byte memory block.
19. { 32, (struct bucket\_desc \*) 0}, // A 32-byte memory block.
20. { 64, (struct bucket\_desc \*) 0},
21. { 128, (struct bucket\_desc \*) 0},
22. { 256, (struct bucket\_desc \*) 0},
23. { 512, (struct bucket\_desc \*) 0},
24. { 1024, (struct bucket\_desc \*) 0},
25. { 2048, (struct bucket\_desc \*) 0},
26. { 4096, (struct bucket\_desc \*) 0}, // A 4096-byte memory block.
27. { 0, (struct bucket\_desc \*) 0}}; /\* End of list marker \*/

88

1. /\*
2. \* This contains a linked list of free bucket descriptor blocks 91 \*/

92 struct bucket\_desc \*free\_bucket\_desc = (struct bucket\_desc \*) 0;

93

1. /\*
2. \* This routine initializes a bucket description page.
3. \*/

//// バケットディスクリプターを初期化します。

1. // フリーバケットディスクリプタのリストを作成し、'free\_bucket\_desc' に最初のフリーバケットの // ディスクリプタを指定させます。
2. static inline void init\_bucket\_desc()
3. {
4. struct bucket\_desc \*bdesc, \*first;
5. int i;

101

1. // まず，バケットディスクリプタを格納するためのメモリのページを申請する．そして，メモリのページに格納できるバケットディスクリプタの // 数を計算し，それに対する一方通行のリンクポインタを確立する．
2. first = bdesc = (struct bucket\_desc \*) get\_free\_page();
3. if (!bdesc)
4. panic("Out of memory in init\_bucket\_desc()");
5. for (i = PAGE\_SIZE/sizeof(struct bucket\_desc); i > 1; i--) { 106 bdesc->next = bdesc+1;
6. bdesc++;
7. }
8. /\*
9. \* This is done last, to avoid race conditions in case
10. \* get\_free\_page() sleeps and this routine gets called again....
11. \*/
12. // フリーのバケット記述子のポインタ'first'をリストの先頭に追加する。
13. bdesc->next = free\_bucket\_desc;
14. free\_bucket\_desc = first;
15. }

116

//// メモリの割り当て機能。

// パラメータ： len - 要求されたメモリブロックのサイズ。

1. // 戻り値: 割り当てられたメモリへのポインタ。失敗した場合は NULL を返します。
2. void \*malloc(unsigned int len)
3. {
4. struct \_bucket\_dir \*bdir;
5. struct bucket\_desc \*bdesc;
6. void \*retval;

122

1. /\*
2. \* First we search the bucket\_dir to find the right bucket change 125 \* for this request.
3. \*/

// メモリブロックのサイズを適用するのに適したバケットディスクリプターリストを、バケットディレクトリから検索します。ディレクトリエントリのバケットサイズが

1. // 要求されたバイト数に応じて、対応するバケットディレクトリのエントリを探します。
2. for (bdir = bucket\_dir; bdir->size; bdir++)
3. if (bdir->size >= len)
4. break;
5. // ディレクトリ全体を検索しても適切なサイズのディレクトリエントリが見つからない場合は、要求されたメモリブロックのサイズが大きすぎて、プログラムの割り当て制限（最大1ページ）を超えていることを示しています。その後、エラーメッセージが表示され、クラッシュが発生します。
6. if (!bdir->size) { 131 printk("malloc called with impossibly large argument (%d)\n",
7. len);
8. panic("malloc: bad arg");
9. }
10. /\*
11. \* Now we search for a bucket descriptor which has free space
12. \*/
13. cli(); /\* Avoid race conditions \*/

// 対応するバケットディレクトリエントリの記述子リストを検索して、バケットを見つける

1. // 空き容量のあるディスクリプター。バケットディスクリプタの空きメモリポインタ 'freeptr' が // 空ではない場合、対応するバケットディスクリプタが見つかったことを示します。
2. for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)
3. if (bdesc->freeptr)
4. break;
5. /\*
6. \* If we didn't find a bucket with free space, then we'll 144 \* allocate a new one.
7. \*/
8. if (!bdesc) {
9. char \*cp;
10. int i;

149

// free\_bucket\_desc がまだ空であれば、この関数が初めて呼び出されたことを意味します。

// またはリンクリスト内のすべての空のバケット記述子が使い尽くされます。この時点で、あなたは

// free\_bucket\_desc は、 // 最初のフリーバケットディスクリプタを指します。

150 if (!free\_bucket\_desc) 151 init\_bucket\_desc() です。

// free\_bucket\_desc が指すフリーバケット記述子を取り、free\_bucket\_desc が指す

// 次の空きバケットディスクリプターに移動します。その後、新しいバケットディスクリプターを初期化します。

1. // 参照数は 0 に，バケットサイズは対応するバケットディレクトリのサイズに // なります．メモリページを適用し，ページポインタはそのページを指します．
2. bdesc = free\_bucket\_desc;
3. free\_bucket\_desc = bdesc->next;
4. bdesc->refcnt = 0;
5. bdesc->bucket\_size = bdir->size;
6. bdesc->page = bdesc->freeptr = (void \*) cp = get\_free\_page();

// メモリページ操作の申請に失敗した場合、エラーが発生してマシンがクラッシュします。

1. // それ以外の場合は，ページサイズをバケットディレクトリの // エントリで指定されたバケットサイズで割って，各オブジェクトの最初の 4 バイトを次のオブジェクトへのポインタに設定する．最後のオブジェクトのポインタには， // 0（NULL）が設定されます．
2. if (!cp)
3. panic("Out of memory in kernel malloc()");
4. /\* Set up the chain of free objects \*/
5. for (i=PAGE\_SIZE/bdir->size; i > 1; i--) {
6. \*((char \*\*) cp) = cp + bdir->size;
7. cp += bdir->size;
8. }
9. \*((char \*\*) cp) = 0;

// そして，バケットディスクリプタの次のディスクリプタポインタフィールドを指して，ディスクリプタの

1. // もともとバケットディレクトリのエントリポインタが指していたもので、バケットディレクトリのチェーンがバケットディスクリプタを指している、つまりディスクリプタがディスクリプタチェーンのヘッダに挿入されているということです。
2. bdesc->next = bdir->chain; /\* OK, link it in! \*/
3. bdir->chain = bdesc;
4. }

// リターンポインタは，ページの現在の空き領域ポインタと同じです。自由空間のポインタは

// のオブジェクト参照カウントが調整され、次の空きオブジェクトを指すようになります。

1. // ディスクリプタの対応するページが1つ増加します。最後に、割り込みを有効にして、 // フリーメモリオブジェクトへのポインタを返します。168 retval = (void \*) bdesc->freeptr;
2. bdesc->freeptr = \*((void \*\*) retval);
3. bdesc->refcnt++;
4. sti(); /\* OK, we're safe again \*/
5. return(retval);
6. }

174

1. /\*
2. \* Here is the free routine. If you know the size of the object that you 177 \* are freeing, then free\_s() will use that information to speed up the 178 \* search for the bucket descriptor.
3. \*
4. \* We will #define a macro so that "free(x)" is becomes "free\_s(x, 0)"
5. \*/

//// バケットオブジェクトをリリースします。

1. // パラメータ： obj - 対応するオブジェクトポインタ， size - オブジェクトのサイズ．
2. void free\_s(void \*obj, int size)
3. {
4. void \*page;
5. struct \_bucket\_dir \*bdir;
6. struct bucket\_desc \*bdesc, \*prev;

187

1. /\* Calculate what page this object lives in \*/
2. page = (void \*) ((unsigned long) obj & 0xfffff000);
3. /\* Now search the buckets looking for that page \*/
4. for (bdir = bucket\_dir; bdir->size; bdir++) {
5. prev = 0;
6. /\* If size is zero then this conditional is always false \*/
7. if (bdir->size < size)
8. continue;

// ディレクトリエントリ内のすべての記述子を検索して、対応するページを見つけます。もし、そのような

// ディスクリプタのページポインタが'page'と等しい場合は， // 対応するディスクリプタが見つかったことを意味するので， // ラベル'found'にジャンプする。ディスクリプタに対応するページが存在しない場合は， // ディスクリプタポインタ 'prev' を指す．

// 対応するディレクトリエントリを検索しているすべてのディスクリプターが、そのディレクトリエントリを見つけられなかった場合

1. // 指定したページにアクセスすると、エラーメッセージが表示され、コンピュータがクラッシュします。
2. for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
3. if (bdesc->page == page)
4. goto found;
5. prev = bdesc;
6. }
7. }
8. panic("Bad address passed to kernel free\_s()"); 203 found:

// 対応するバケットディスクリプタを見つけたら、まず割り込みをオフにします。オブジェクトの

1. // メモリブロックはフリーブロックのオブジェクトリストにチェーンされ、そのディスクリプターのオブジェクト参照カウントは // 1つずつ減らされていきます。
2. cli(); /\* To avoid race conditions \*/
3. \*((void \*\*)obj) = bdesc->freeptr;
4. bdesc->freeptr = obj;
5. bdesc->refcnt--;

// 参照カウントが0になった場合、対応するメモリページと // バケットディスクリプタを解放することができます。

1. 15.8 open.c
2. if (bdesc->refcnt == 0) {
3. /\*
4. \* We need to make sure that prev is still accurate. It 211 \* may not be, if someone rudely interrupted us....
5. \*/

// 'prev'が検索されたディスクリプターの前のディスクリプターでない場合は、前の

1. // 現在のディスクリプターを再検索します。
2. if ((prev && (prev->next != bdesc)) ||
3. (!prev && (bdir->chain != bdesc)))
4. for (prev = bdir->chain; prev; prev = prev->next)
5. if (prev->next == bdesc)
6. break;

// 前の記述子が見つかった場合、現在の記述子は記述子から削除されます。

1. // チェーンを指す。prev==NULLの場合は、現在のディスクリプタが // ディレクトリエントリの最初のディスクリプタであることを意味し、 つまり、ディレクトリエントリのチェーンは、現在のディスクリプタの // 'bdesc' を直接指すべきであり、そうでない場合は、リンクリストに問題があることを示す。 // したがって，リンクリストから現在の記述子を削除するためには， // 'chain' が次の記述子を指している必要がある。
2. if (prev)
3. prev->next = bdesc->next; 220 else {
4. if (bdir->chain != bdesc)
5. panic("malloc bucket chains corrupted");
6. bdir->chain = bdesc->next;
7. }
8. // 最後に，現在のディスクリプターが操作するメモリページが解放され，ディスクリプターは // アイドルディスクリプターリストの先頭に挿入される．
9. free\_page((unsigned long) bdesc->page);
10. bdesc->next = free\_bucket\_desc;
11. free\_bucket\_desc = bdesc;
12. }
13. sti(); // Enable interrupt and return.
14. return;
15. }

232

233

## 15.8 open.c

### 15.8.1 Functionality

open.cファイル内のopen()ライブラリ関数は，指定されたファイル名のファイルを開くために使用されます。open()の呼び出しに成功すると，そのファイルのファイルディスクリプターが返される。この呼び出しにより、新しいオープンファイルが作成され、他のプロセスと共有されることはありません。exec()関数が実行されると、新しいファイルディスクリプターは常にオープン状態になります。ファイルの読み書きポインタは、ファイルの先頭に設定されます。

関数のパラメータ'flag'には、O\_RDONLY, O\_WRONLY, O\_RDWRのいずれかを指定することができます。これらはそれぞれ、ファイルが読み取り専用オープン、書き込み専用オープン、読み書き可能オープンであることを意味し、他のいくつかのフラグと一緒に使用することができます。fs/open.cプログラムのsys\_open()関数の実装(171行目)も参照してください。

15.8 open.c

### 15.8.2 Code annotation

プログラム 15-8 linux/lib/open.c

1. /\*
2. \* linux/lib/open.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/ 6

// <unistd.h> Linux標準のヘッダファイルです。様々なシンボル定数や型が定義され、 // 様々な関数が宣言されています。LIBRARY\_\_」が定義されている場合は、 // システムコール番号とインラインアセンブリの \_syscall0() も含まれます。

1. // <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マクロの形で // 定義する。主に vsprintf, vprintf, vfprintf 関数のための 1 つの型 (va\_list) と 3 つのマクロ (va\_start, va\_arg, // va\_end) が記述されている。
2. #define \_\_LIBRARY\_\_
3. #include <unistd.h>
4. #include <stdarg.h>

10

//// ファイルライブラリを開く機能です。

// ファイルを開いたり、ファイルが存在しない場合にファイルを作成したりします。

// パラメータ： filename - ファイル名， flag - ファイルオープンフラグ， ...

1. // ファイルディスクリプターを返します。エラーが発生した場合は、エラーコードが設定され、-1が返されます。
2. int open(const char \* filename, int flag, ...)
3. {
4. register int res;
5. va\_list arg; 15
6. // va\_start()マクロ関数を使って、フラグの後のパラメータのポインタを取得します。その後、 // システム割り込み int 0x80 を関数番号 \_\_NR\_open で呼び出し、ファイルを開きます。 // %0 - eax (返されたディスクリプタまたはエラーコード); %1 - eax (システムコール関数 \_\_NR\_open); // %2 - ebx (ファイル名); %3 - ecx (オープンファイルフラグ); %4 - edx (ファイルモードに続く).
7. va\_start(arg,flag);
8. \_\_asm\_\_("int $0x80"
9. :"=a" (res)
10. :"" (\_\_NR\_open),"b" (filename),"c" (flag),
11. "d" (va\_arg(arg,int)));
12. // システムインタラプトコールがファイルディスクリプタであることを示す0以上の値を返した場合は、 // 直接戻ります。そうでない場合、戻り値が0よりも小さい場合は、 // エラーコードになります。そこで、エラーコードを設定し、-1を返します。
13. if (res>=0)
14. return res;
15. errno = -res;
16. return -1;
17. }

26

## 15.9 setsid.c

## 15.9 setsid.c

### 15.9.1 Functionality

setsid.cプログラムには、setsid()システムコール関数が含まれています。この関数は、呼び出したプロセスがグループのリーダーでない場合に、新しいセッションを作成するために使用されます。呼び出したプロセスが新しいセッションのリーダーとなり、新しいプロセス・グループのグループ・リーダーとなり、制御する端末はありません。呼び出したプロセスのグループIDとセッションIDには、プロセスのPIDが設定されます。呼び出したプロセスは、新しいプロセス・グループと新しいセッションの唯一のプロセスになります。

### 15.9.2 Code annotation

プログラム 15-9 linux/lib/setsid.c

1. / 1 /\*
2. \* linux/lib/setsid.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/ 6

// <unistd.h> Linux標準のヘッダファイルです。様々なシンボル定数や型が定義され、 // 様々な関数が宣言されています。また、'\_\_LIBRARY\_\_'が定義されている場合、その中には

1. // システムコール番号とインラインアセンブリ\_syscall0()を指定します。
2. #define \_\_LIBRARY\_\_
3. #include <unistd.h>

9

//// セッションを作成し、プロセスグループ番号を設定します。

// 以下のシステムコールマクロは、pid\_t setsid()という関数に対応しています。

// 呼び出したプロセスのセッション識別子（セッションID）を返します。

10 \_syscall0(pid\_t,setsid)

11

## 15.10 string.c

### 15.10.1 Functionality

### すべての文字列操作関数は、すでにstring.hヘッダーファイルに存在していますが、インラインコードとして表示されています。ここでは，string.cの文字列関数を含む実装コードとして，まず「extern」と「inline」の接頭辞を空にして宣言し，string.hヘッダファイルをインクルードしています。include/string.hヘッダファイルの前の説明を参照してください。

### 15.10.2 Code annotation

プログラム 15-10 linux/lib/string.c

1. / 1 /\*
2. \* linux/lib/string.c
3. 15.11 wait.c
4. \*
5. \* (C) 1991 Linus Torvalds
6. \*/

6

7 #ifndef \_\_GNUC\_\_ 8 #error gccが欲しい!

9 #endif

10

1. #define extern
2. #define inline
3. #define \_\_LIBRARY\_\_
4. #include <string.h>

15

## 15.11 wait.c

### 15.11.1 Functionality

wait.cプログラムには、関数waitpid()とwait()が含まれています。この2つの関数により、プロセスはその子プロセスの1つの状態情報を取得することができます。様々なオプションにより、終了または停止した子プロセスの状態情報を取得することができます。2つ以上の子プロセスの状態情報がある場合、報告の順序は指定されません。

wait()は、子プロセスの1つが終了（ターミネイト）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します。

waitpid()は、pidで指定された子プロセスが終了（ターミネイト）するか、プロセスの終了を要求するシグナルを受け取るか、シグナルハンドラを呼び出す必要があるまで、現在のプロセスを一時停止します。

pid=-1、options=0の場合、waitpid()はwait()関数と同じ動作をしますが、そうでない場合はpidとoptionsのパラメータに応じて動作が変わります（kernel/exit.c, 370参照）。

### 15.11.2 Code annotation

プログラム 15-11 linux/lib/wait.c

1. /\*
2. \* linux/lib/wait.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/ 6

// <unistd.h> Linux標準のヘッダファイルです。様々なシンボル定数や型が定義され、 // 様々な関数が宣言されています。また、'\_\_LIBRARY\_\_'が定義されている場合、その中には

// システムコール番号とインラインアセンブリ\_syscall0()を指定します。

// <sys/wait.h> wait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

1. // 定数記号。
2. #define \_\_LIBRARY\_\_
3. #include <unistd.h>
4. #include <sys/wait.h>

15.12 write.c

10

//// プロセスが終了するのを待ちます。

// pid\_t waitpid(pid\_t pid, int \* wait\_stat, int options)

// パラメータ: pid - 終了を待つプロセスのプロセスID、または特殊なケースを指定するために使用されるその他の特定の値 // wait\_stat - 状態情報を保存するために使用されます。

// オプション - WNOHANG or WUNTRACED or 0.

11 \_syscall3(pid\_t,waitpid,pid\_t,pid,int \*,wait\_stat,int,options)

12

1. //// wait() システムコールです。waitpid()関数を直接呼び出します。
2. pid\_t wait(int \* wait\_stat)
3. {
4. return waitpid(-1,wait\_stat,0);
5. }

17

## 15.12 write.c

### 15.12.1 Functionality

write.cプログラムには、ファイルディスクリプタへの書き込み関数write()が含まれています。この関数は、ファイルディスクリプタで指定されたファイルの'buf'に'count'バイトのデータを書き込む。

### 15.12.2 Code annotation

プログラム 15-12 linux/lib/write.c

1. /\*
2. \* linux/lib/write.c
3. \*
4. \* (C) 1991 Linus Torvalds
5. \*/ 6

// <unistd.h> Linux標準のヘッダファイルです。様々なシンボル定数や型が定義され、 // 様々な関数が宣言されています。また、'\_\_LIBRARY\_\_'が定義されている場合、その中には

1. // システムコール番号とインラインアセンブリ\_syscall0()を指定します。
2. #define \_\_LIBRARY\_\_
3. #include <unistd.h>

9

//// ファイルの書き込みを行います。

// int write(int fd, const char \* buf, off\_t count) // Parameters: fd - ファイルディスクリプタ; buf - 書き込みバッファポインタ; count - 書き込みバイト数。 count - 書き込みバイト数 // 戻り値: 成功時に書き戻されたバイト数 (0 は 0 バイトを書き込むことを意味する); エラー時には // -1 が返され、エラーコードが設定される。

10 \_syscall3(int,write,int,fd,const char \*,buf,off\_t,count)

11

## 15.13 まとめ

## 15.13 Summary

この章では、カーネルが初期化時にユーザーモードで実行するいくつかのタスクが使用するいくつかのライブラリ関数ファイルについて説明します。これらのライブラリ関数は、開発環境で使用されている一般的なライブラリ関数とまったく同じ方法で実装されています。

次の章では、カーネルコードツリーに含まれるtools/build.cというツールについて説明します。このツールは、すべてのカーネルモジュールを組み合わせてカーネルイメージファイルを生成するためのものです。

# 16 Building Kernel (tools)

Linuxカーネルソースコードの「tools」ディレクトリには、カーネルのディスクイメージファイルを生成するユーティリティプログラムbuild.cが含まれています。このプログラムは個別に実行ファイルにコンパイルされ、Makefileの中で呼び出されて、カーネルのコンパイル済みモジュールをすべて接続して作業用イメージファイルImageにマージするために使用されます。Makefileの内容によると、Makeプログラムはまず8086アセンブラを使ってboot/bootsect.sとboot/setup.sをコンパイルし、MINIX形式の2つのオブジェクトモジュールファイル「bootsect」と「setup」を生成し、次にGNU.Gcc/gasでコンパイルします。Gcc/gasは、ソースコードに含まれる他のすべてのプログラムをコンパイルしてリンクし、a.out形式のオブジェクトモジュール'system'を生成します。最後に、ビルドツールを使って、3つのモジュールから余分なヘッダーデータを取り除き、カーネルイメージファイル「Image」につなぎ合わせます。基本的なコンパイル・リンク・結合の仕組みを図16-1に示します。

head

main

kernel

mm

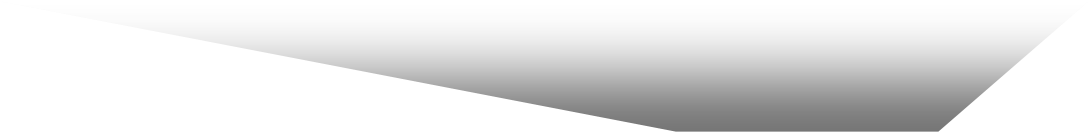
fs

lib

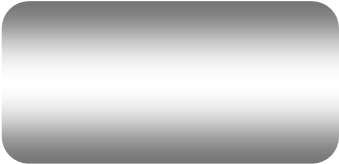
bootsect

setup

system



Build Tool



Kernel

Image file

図16-1 カーネルのコンパイル・リンク・結合構造

## 16.1 build.c

### 16.1.1 Functionality

linux/Makefileの42--44行目で、ビルドプログラムを実行するためのコマンドライン形式は以下の通りです。

tools/build boot/bootsect boot/setup tools/system $(ROOT\_DEV) $(SWAP\_DEV) > Image

ビルドプログラムでは、bootsect、setup、system、オプションのroot fsデバイス名ROOT\_DEV、オプションのswapデバイスSWAP\_DEVの5つのパラメータを使用しています。bootsectとsetupモジュールはas86でコンパイルされており、MINIXの実行ファイル形式になっています（プログラムリストの説明を参照）。systemモジュールは他のソースコードからコンパイルされたモジュールがリンクされており、GNU a.outの実行ファイル形式になっています。ビルドプログラムの主な仕事は、bootsectとsetupのMINIX実行ファイルのヘッダ情報を削除し、システムモジュールのa.outヘッダ情報を削除して、それらのコード部分とデータ部分だけを残し、それらを順番に結合して、「Image」という名前のファイルに順次書き込むことです。

プログラムはまず、コマンドラインの最後のオプションパラメーターであるルートデバイスファイル名をチェックします。存在する場合は、デバイスファイルのステータス情報構造（stat）が読み込まれ、デバイス番号が抽出されます。このパラメータがコマンドラインに存在しない場合は、デフォルト値が使用されます。次に bootsect ファイルを処理し、ファイルの minix 実行ヘッダ情報を読み込み、有効性を確認した後、続く 512 バイトのブートコードを読み込み、ブータブルフラグ 0xAA55 を持っているかどうかを判断し、先に取得したルートデバイス番号を 508, 509 のオフセットに書き込み、最後に 512 バイトのコードデータを stdout の標準出力に書き込み、Make ファイルによって Image ファイルにリダイレクトします。次に、セットアップファイルも同様に処理します。ファイルの長さが4セクタに満たない場合は、4セクタの長さまで0で埋められ、標準出力のstdoutに書き込まれます。

* 最後にシステムファイルを処理します。このファイルはGCCコンパイラを使用して生成されているため、実行ヘッダの形式はGCC型であり、linuxで定義されているa.out形式と同じです。実行エントリポイントが0であることを確認した後、標準出力のstdoutにデータを書き込みます。コードとデータのサイズが128KBを超える場合は、エラーメッセージが表示されます。結果として得られるカーネルイメージファイルのフォーマットは図16-2のようになります。
* The first sector stores the bootsect code, which is exactly 512 bytes long;
* The 4 sectors (2 - 5 sectors) starting from the 2nd sector store the setup code, and the size is no more than 4 sectors;
* The system module is stored starting from the sixth sector, and its length does not exceed the size (128KB) defined on line 37 of build.c.

system module

boot

sector

setup module

図 16-2 ディスク上の Linux カーネルのフォーマット

### 16.1.2 Code annotation

プログラム 16-1 linux/tools/build.c

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

6

1. /\*
2. \* This file builds a disk-image from three different files:
3. \*
4. \* - bootsect: max 510 bytes of 8086 machine code, loads the rest
5. \* - setup: max 4 sectors of 8086 machine code, sets up system parm
6. \* - system: 80386 code for actual system 13 \*
7. \* It does some checking that all files are of the correct type, and
8. \* just writes the result to stdout, removing headers and padding to 16 \* the right amount. It also writes some system data to stderr.

17 \*/

18

1. /\*
2. \* Changes by tytso to allow root device specification 21 \*
3. \* Added swap-device specification: Linux 20.12.91
4. \*/

24

1. #include <stdio.h> /\* fprintf \*/
2. #include <string.h>
3. #include <stdlib.h> /\* contains exit \*/
4. #include <sys/types.h> /\* unistd.h needs this \*/
5. #include <sys/stat.h> // file state information structure
6. #include <linux/fs.h>
7. #include <unistd.h> /\* contains read/write \*/
8. #include <fcntl.h> // file operation mode constants

33

34 #define MINIX\_HEADER 32 // MINIXオブジェクトファイルのヘッダサイズは32バイトです。 35 #define GCC\_HEADER 1024 // GCCのヘッダー情報のサイズは1024バイトです。

36

37 #define SYS\_SIZE 0x3000 // システムモジュールの最大サイズ(SYS\_SIZE\*16=128KB)です。

38

// デフォルトでは、Linuxのルートfsデバイスは、2番目のハードディスクの第1パーティション（デバイス

1. // 番号0x0306）になっています。これは、リーナス氏がLinuxを開発した際に、1台目のハードディスクをMINIXのシステムディスクとして、2台目のハードディスクをLinuxのルートファイルシステムのディスクとして使用したためです。
2. #define DEFAULT\_MAJOR\_ROOT 3 // major device number - 3, hard disk.
3. #define DEFAULT\_MINOR\_ROOT 6 // minor device number - 6, 1st partition of the 2nd disk.

41

1. #define DEFAULT\_MAJOR\_SWAP 0 // swap device number.
2. #define DEFAULT\_MINOR\_SWAP 0

44

1. /\* max nr of sectors of setup: don't change unless you also change
2. \* bootsect etc \*/
3. #define SETUP\_SECTS 4 // The maximum size of setup is 4 sectors (2KB).

48

49 #define STRINGIFY(x) #x // エラー表示に使用するために、xを文字列型に変換します。

50

1. //// エラーメッセージを表示して、プログラムを終了します。
2. void die(char \* str)
3. {
4. fprintf(stderr,"%s\n",str);
5. exit(1);
6. }

56

1. //// プログラムの使用状況を表示して終了します。
2. void usage(void)
3. {
4. die("Usage: build bootsect setup system [rootdev] [> image]");
5. }

61

//// メインプログラムが始まる...。

// プログラムはまず、コマンドラインのパラメータが遵守されているかどうかをチェックし、ルート

// その後、bootsect、setup、および // system module ファイルをそれぞれ読み込んで処理し、それらのファイルを以前の標準出力に書き出します。

1. // Imageファイルにリダイレクトされます。.
2. int main(int argc, char \*\* argv)
3. {
4. int i,c,id;
5. char buf[1024];
6. char major\_root, minor\_root;
7. char major\_swap, minor\_swap;
8. struct stat sb; 69

// (1) まず、ビルドプログラム実行時の実際のコマンドラインパラメータを確認して

// パラメータの数に応じて適宜設定してください。ビルドプログラムでは4～6個のパラメータが必要です。コマンドラインのパラメータ数が条件を満たさない場合、 // プログラムの使用状況を表示して終了します。

// プログラムのコマンドラインに4つ以上のパラメータがある場合、ルートデバイス名が

// が "FLOPPY "でない場合は、デバイスファイルのステータス情報を取得し、メジャー、マイナー

1. // デバイス番号は、ルートデバイス番号として扱われます。ルートデバイスが FLOPPY デバイスの場合は、 // メジャーデバイス番号とマイナーデバイス番号を 0 にして、ルートデバイスが現在の // ブートデバイスであることを示します。
2. if ((argc < 4) || (argc > 6))
3. usage();
4. if (argc > 4) {
5. if (strcmp(argv[4], "FLOPPY")) {
6. if (stat(argv[4], &sb)) {
7. perror(argv[4]);
8. die("Couldn't stat root device.");
9. }
10. major\_root = MAJOR(sb.st\_rdev); // Get the device number.
11. minor\_root = MINOR(sb.st\_rdev);
12. } else {
13. major\_root = 0;
14. minor\_root = 0;
15. }
16. // パラメータが4つしかない場合は、メジャーデバイス番号とマイナーデバイス番号を、システムの // デフォルトのルートデバイス番号と同じにします。
17. } else {
18. major\_root = DEFAULT\_MAJOR\_ROOT;
19. minor\_root = DEFAULT\_MINOR\_ROOT;
20. }

// コマンドラインに6つのパラメータがある場合、最後のパラメータが示しているのが

// スイッチングデバイスが "NONE "ではない場合、デバイスファイルのステータス情報を取得して

1. // メジャーデバイス番号とマイナーデバイス番号がスワップデバイスの番号として扱われます。最後のパラメータが "NONE "の場合、スワップデバイスのメジャーとマイナーのデバイス番号は0として扱われ、スワップデバイスが現在のブートデバイスであることを示します。
2. if (argc == 6) {
3. if (strcmp(argv[5], "NONE")) {
4. if (stat(argv[5], &sb)) {
5. perror(argv[5]);
6. die("Couldn't stat root device."); 93 }
7. major\_swap = MAJOR(sb.st\_rdev);
8. minor\_swap = MINOR(sb.st\_rdev);
9. } else {
10. major\_swap = 0;
11. minor\_swap = 0;
12. }
13. // 6つのパラメータがなく、5つのパラメータがある場合は、コマンド // ラインにスワップデバイス名がないことを意味します。そこで、スワップデバイスのメジャーとマイナーのデバイス番号を、システムのデフォルトのスワップ // デバイス番号と同じにします。
14. } else {
15. major\_swap = DEFAULT\_MAJOR\_SWAP;
16. minor\_swap = DEFAULT\_MINOR\_SWAP;
17. }

// 次に、選択されたルートデバイスのメジャーとマイナーのデバイス番号と、メジャーとマイナーの

// スワップデバイスのデバイス番号が標準エラー端子に表示されます。もし、メジャー

// デバイス番号は、2（フロッピーディスク）でも3（ハードディスク）でもなく、0（システムのデフォルトデバイス）でもありません。

1. // エラーメッセージを表示して終了します。ターミナルの標準出力は，ファイル 'Image' にリダイレクトされるので， // 保存されたカーネルコードとデータを出力して， // カーネルイメージファイルを生成するのに使用される。
2. fprintf(stderr, "Root device is (%d, %d)\n", major\_root, minor\_root);
3. fprintf(stderr, "Swap device is (%d, %d)\n", major\_swap, minor\_swap);
4. if ((major\_root != 2) && (major\_root != 3) &&
5. (major\_root != 0)) {
6. fprintf(stderr, "Illegal root device (major = %d)\n",
7. major\_root);
8. die("Bad root device --- major #");
9. }
10. if (major\_swap && major\_swap != 3) {
11. fprintf(stderr, "Illegal swap device (major = %d)\n",
12. major\_swap);
13. die("Bad root device --- major #");
14. }

// (2) 以下は、各ファイルの内容を読み込んで、対応する実行を開始します。

1. // コピー処理を行います。まず1KBのバッファを初期化し、パラメータ1（bootsect）で指定されたファイルを読み込み専用モードでオープンし、32バイトのMINIX実行ヘッダ構造体（以下のリストを参照）をバッファbufに読み込みます。
2. for (i=0;i<sizeof buf; i++) buf[i]=0; 118 if ((id=open(argv[1],O\_RDONLY,0))<0)
3. die("Unable to open 'boot'");
4. if (read(id,buf,MINIX\_HEADER) != MINIX\_HEADER)
5. die("Unable to read header of 'boot'");

// 次に、ブートセクトが有効なMINIX実行ファイルであるかどうかを、MINIXヘッダに基づいてチェックします。

// 構造になっています。その場合、512バイトのブートセクターコードとデータがファイルから読み込まれます。0x0301 - MINIXヘッドマジック値 a\_magic; 0x10 - 実行可能フラグ a\_flag; // 0x04 - マシンタイプ a\_cpu, Intel 8086のマシンコードを意味します。

// その後、ヘッダー情報に対して一連のチェックが行われます。ヘッダーサイズが

// フィールド a\_hdrlen は正しい (32 バイト) (最後の 3 バイトは役に立たないので 0); confirm

1. // データセグメントのサイズ a\_data フィールド（long）の内容が 0 であるかどうか、ヒープ // a\_bss フィールド（long）の内容が 0 であるかどうか、実行ポイント a\_entry フィールド（long）の内容が 0 であるかどうか、 // シンボルテーブルサイズフィールド a\_sym の内容が 0 であるかどうかを確認します。
2. if (((long \*) buf)[0]!=0x04100301)
3. die("Non-Minix header of 'boot'"); 124 if (((long \*) buf)[1]!=MINIX\_HEADER)
4. die("Non-Minix header of 'boot'");
5. if (((long \*) buf)[3]!=0)
6. die("Illegal data segment in 'boot'");
7. if (((long \*) buf)[4]!=0)
8. die("Illegal bss in 'boot'");
9. if (((long \*) buf)[5] != 0)
10. die("Non-Minix header of 'boot'");
11. if (((long \*) buf)[7] != 0)
12. die("Illegal symbol table in 'boot'");

// 上記の判定を条件に、実際にファイル内のコードやデータを読み込んだ結果

1. // が正しい場合、読み取りバイト数は512バイトになるはずです。ブートセクトファイルには、1セクタ分のブートセクタコードとデータが // 含まれており、最後の2バイトはブート可能フラグ0xAA55であるべきだからです。
2. i=read(id,buf,sizeof buf);
3. fprintf(stderr,"Boot sector %d bytes.\n",i);
4. if (i != 512)
5. die("Boot block must be exactly 512 bytes");
6. if ((\*(unsigned short \*)(buf+510)) != 0xAA55)
7. die("Boot block hasn't got boot flag (0xAA55)");
8. // その後、バッファの内容を変更し、スワップデバイス番号をオフセット506, // 507に格納し、ルートデバイス番号をオフセット508, 509に格納する。
9. buf[506] = (char) minor\_swap;
10. buf[507] = (char) major\_swap;
11. buf[508] = (char) minor\_root;
12. buf[509] = (char) major\_root;

// 次に、512バイトのデータを標準出力のstdoutに書き込み、bootsectファイルを閉じます。

1. // linux/Makefileでは、ビルドプログラムが標準出力を「>」インジケータを使ってカーネルイメージ // ファイル名Imageにリダイレクトするので、ブートセクタのコードとデータはImageの // 最初の512バイトに書き込まれます。
2. i=write(1,buf,512);
3. if (i!=512)
4. die("Write call failed");
5. close (id); 148
6. // (3) パラメータ2で指定されたファイル(setup)をリードオンリーモードでオープンし， // 32バイトのMINIX実行ファイルヘッダの内容をバッファbufに読み込む。次に、MINIXヘッダ構造に従って、setupが有効なMINIX実行ファイルであるかどうかがチェックされる。 // そうであれば、ヘッダ情報に対して一連のチェックが行われますが、上記と同様に処理されます。
7. if ((id=open(argv[2],O\_RDONLY,0))<0)
8. die("Unable to open 'setup'");
9. if (read(id,buf,MINIX\_HEADER) != MINIX\_HEADER)
10. die("Unable to read header of 'setup'");
11. if (((long \*) buf)[0]!=0x04100301)
12. die("Non-Minix header of 'setup'");
13. if (((long \*) buf)[1]!=MINIX\_HEADER) // header size (32 bytes) 156 die("Non-Minix header of 'setup'");
14. if (((long \*) buf)[3]!=0) // data size field a\_data
15. die("Illegal data segment in 'setup'");
16. if (((long \*) buf)[4]!=0) // a\_bss field.
17. die("Illegal bss in 'setup'");
18. if (((long \*) buf)[5] != 0) // a\_entry point.
19. die("Non-Minix header of 'setup'");
20. if (((long \*) buf)[7] != 0)
21. die("Illegal symbol table in 'setup'");

// ファイル内の後続の実際のコードやデータは、上記の条件で読み込まれます。

// のチェックは正しく、端末の標準出力に書き込まれます。それと同時に、長さ

書き込みの//をカウントし、動作終了後に設定ファイルを閉じます。その後、チェック

// 書き込み操作のコードとデータのサイズが、(SETUP\_SECTS

1. // そうでなければ，ビルド，ブートセクト，セットアッププログラムで設定したセットアップが占めるセクタ数を再編集して，カーネルを再コンパイルしなければなりません。問題がなければ、 // セットアップの実際の長さが表示されます。
2. for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
3. if (write(1,buf,c)!=c)
4. die("Write call failed");
5. close (id); // close setup file.
6. if (i > SETUP\_SECTS\*512)
7. die("Setup exceeds " STRINGIFY(SETUP\_SECTS)
8. " sectors - rewrite build/boot/setup");
9. fprintf(stderr,"Setup is %d bytes.\n",i);

// バッファbufをクリアした後、実際の書き込みセットアップ長と // (SETUP\_SECTS \* 512)の差をチェックします。セットアップサイズがサイズ(4 \* 512バイト)より小さい場合は、バッファの中の

1. // NULL文字で4 \*512バイトになるように設定。
2. for (c=0 ; c<sizeof(buf) ; c++)
3. buf[c] = '\0';
4. while (i<SETUP\_SECTS\*512) {
5. c = SETUP\_SECTS\*512-i;
6. if (c > sizeof(buf))
7. c = sizeof(buf);
8. if (write(1,buf,c) != c)
9. die("Write call failed");
10. i += c;
11. }

183

// (4) 以下のシステムモジュールファイルの処理を開始します。このファイルはgas/gccでコンパイルされているため， // GNU a.outオブジェクトファイル形式になっています。

// まず、システムモジュールファイルを読み取り専用で開き、a.out形式のヘッダー構造を読みます。

// の情報（1KBサイズ）を入力します。システムが有効なa.out形式のファイルであることを確認した後、すべての

// ファイルの後続データを標準出力（イメージファイル）に出力し、ファイルを閉じます。その後、表示

1. // システムのサイズです。システムコードとデータのサイズがSYS\_SIZEセクション（128KB // バイト）を超えた場合は、エラーメッセージを表示して終了する。エラーがない場合は0を返し、 // 正常終了を示します。
2. if ((id=open(argv[3],O\_RDONLY,0))<0)
3. die("Unable to open 'system'");
4. if (read(id,buf,GCC\_HEADER) != GCC\_HEADER)
5. die("Unable to read header of 'system'");
6. if (((long \*) buf)[5] != 0) // entry location should be 0.
7. die("Non-GCC header of 'system'");
8. for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
9. if (write(1,buf,c)!=c)
10. die("Write call failed");
11. close(id);
12. fprintf(stderr,"System is %d bytes.\n",i);
13. if (i > SYS\_SIZE\*16)
14. die("System is too big");
15. return(0);
16. }

199

### 16.1.3 Information

#### 16.1.3.1 MINIX module and executable header data structure

MINIXのコンパイラとリンカが生成するモジュールと実行ファイルのヘッダ構造は以下の通りです。

/ struct exec { unsigned char a\_magic[2]; // マジックナンバー、0x0301でなければならない。 unsigned char a\_flags; // フラグ（後述）。

unsigned char a\_hdrlen; // 予約されたヘッダーサイズ（32または48バイト）。

long a\_text; // コードセクションのサイズ(バイト) long a\_data; // データセクションのサイズ(バイト) long a\_bss; // スタックサイズ(バイト) long a\_entry; // 実行エントリポイント。

long a\_total; // 割り当てられたメモリの総量。 long a\_syms; // シンボルテーブルのサイズ。

// long a\_trsize; // コード部再配置テーブルサイズ long a\_drsize; // データ部再配置テーブルサイズ long a\_tbase; // コード部再配置ベースアドレス long a\_dbase; // データ部再配置ベースアドレス。

};

その中でも、MINIX実行ファイルのヘッダにあるフラグフィールドa\_flagsは以下のように定義されています。

A\_UZP 0x01 // Unmapped 0 pages (pages).

A\_PAL 0x02 // ページ境界でアジャストされる。

A\_NSYM 0x04 // 新しいタイプのシンボルテーブル。

A\_EXEC 0x10 // 実行可能ファイル。

A\_SEP 0x20 // コードとデータが別々になっている（IとDが独立している）。

CPUの識別番号フィールドa\_cpuには

|  |  |
| --- | --- |
| A\_NONE 0x00 | // Unknown. |
| A\_I8086 0x04 | // Intel i8086/8088. |
| A\_M68K 0x0B | // Motorola m68000. |
| A\_NS16K 0x0C | // National Semiconductor Co. 16032. |
| A\_I80386 0x10 | // Intel i80386. |
| A\_SPARC 0x17 | // Sun SPARC. |

16.2 まとめ

上記のMINIX実行ヘッダー構造execは、Linux 0.12システムで使用されているa.out形式のヘッダー構造に似ています。Linuxのa.out形式実行ファイルのヘッダ構造および関連情報については、linux/include/a.out.hファイルを参照してください。

## 16.2 Summary

この章では、カーネルソースツリーにあるビルドツール build.c について説明します。このプログラムは、主にカーネルモジュールを修正・結合して、起動可能なカーネルブートイメージファイルを生成するために使用します。これまでに、カーネル内のすべてのソースコードファイルについて、詳細な説明とコメントを終えています。

カーネルの動作メカニズムをより深く理解するために、次の章では、Bochsシミュレーションプログラムを使って、Linux 0.12オペレーティングシステムをセットアップして動作させるテスト方法を詳しく説明し、いくつかの実験について具体的なテスト手順を示します。

16.2 まとめ

**17 実験環境の設定と使用方法**

本章では、Linux 0.1x カーネルの動作原理を学ぶために、PC シミュレーション・ソフトウェアを使用して、実際のコンピュータ上で Linux 0.1x システムを動作させる実験方法を紹介します。具体的には、カーネルのコンパイルプロセス、シミュレーション環境でのファイルアクセスとコピー、起動ディスクとルートファイルシステムの作成方法、Linux 0.1xシステムの使用方法などです。最後に、既存のRedHatシステム（gcc 3.x）でのコンパイルプロセスを成功させ、対応するカーネル・イメージ・ファイルを作成するために、カーネル・コードに少数の構文変更を加える方法も紹介しました。

* 実験を始める前に、まず便利なツールを用意する必要があります。Windowsプラットフォームで実験を行う場合は、以下のソフトウェアを準備する必要があります。
* Bochs 2.6.x open source PC simulation package (<https://sourceforge.net/projects/bochs/>);
* Notepad++ Editor. Used to edit binary files (<https://sourceforge.net/projects/notepad-plus/>);  HxD hex editor. Used to edit binary or disk files, even in-memory data ([https://mh-nexus.de/en/hxd/)](https://mh-nexus.de/en/hxd/);  WinImage DOS format floppy image file editing software ( <http://www.winimage.com/>).

最新のLinuxシステム（Redhat、Ubuntuなど）で実験を行う場合は、通常、Bochsパッケージをインストールするだけでシミュレーションを行うことができます。実験におけるその他の操作は、Linuxシステムの一般的なツールを使って行うことができます。

Linux 0.1xのシステムを動かすには、PCのエミュレーションソフトを使うのが一番です。現在、世界で人気のあるPCシミュレーションソフトは4つあります。VMware社のソフトウェア「VMware Workstation」、Oracle社のオープンソースソフトウェア「VirtualBox」、Microsoft社の「Virtual PC」、そしてオープンソースソフトウェア「Bochs」（発音は「ボックス」と同じ）です。これらは、PCの操作を完全に仮想化し、シミュレートすることができる。これらのタイプのソフトウェアは、インテルx86ハードウェア環境を仮想化またはエミュレートすることができ、ソフトウェアが動作しているプラットフォーム上で、他のさまざまな「お客様」のオペレーティングシステムを実行することができます。

使用範囲や操作性の面で、4つのシミュレーションソフトウェアにはまだいくつかの違いがあります。Bochsは、x86CPU搭載PCのハードウェア環境（CPU命令）とその周辺機器をすべてソフトウェアでシミュレーションしているため、多くのOSやアーキテクチャの異なるプラットフォームへの移植が容易である。主にソフトウェアのシミュレーション技術を利用しているため、他のシミュレーションソフトウェアに比べて実行性能や速度が大幅に低下します。Virtual PCソフトウェアの性能は、BochsとVMware Workstation（またはVirtualBox）の間です。ほとんどのx86命令をエミュレートし、その他の部分は仮想技術を使って実装しています。VMware WorkstationやVirtualBoxは一部のI/O機能をシミュレートしているだけで、それ以外の部分はx86リアルタイムハードウェア上で直接実行されます。つまり、ゲストOSがある命令を実行する必要があるとき、VMwareやVirtualBoxはその命令をシミュレーションで実行するのではなく、その命令を実際のシステムのハードウェアに直接「渡す」だけなのです。ですから、VMwareやVirtualBoxは、これらのソフトウェアの中では速度と性能の面で最高のものです。もちろん、Qemuのような他のシミュレーションソフトでも、高いシミュレーション性能を発揮することができます。

## アプリケーションの観点から見ると、シミュレーション環境で主にアプリケーションを実行する場合は、VMware WorkstationやVirtualBoxが良い選択となるはずです。しかし、何らかの低レベルのシステムソフトウェアを開発する必要がある場合（オペレーティングシステムの開発やデバッグ、コンパイラシステムの開発など）は、Bochsが良い選択となります。Bochsを使えば、実際のハードウェアシステムの実行ではなく、シミュレーションされたハードウェア環境の中で、実行されたプログラムの具体的な状態や正確なタイミングを知ることができます。これが、多くのOS開発者がBochsを使いたがる理由です。この章では、Bochsシミュレーション環境を使ってLinux 0.1xを実行する方法を説明します。現在、Bochsのウェブサイトの名前は、http://sourceforge.net/projects/bochs/。上記からBochsソフトウェアの最新リリースをダウンロードすることができますし、多くのready-run systemのイメージファイルをダウンロードすることもできます。

## 17.1 Bochs Simulation Software

Bochsは、Intel 80X86コンピュータを完全にエミュレートするプログラムです。インテルの80386、486、Pentium以上の新しいCPUプロセッサをエミュレートするように設定することができます。実行段階を通じて、Bochsは、標準的なPC周辺機器のすべてのデバイスモジュールのエミュレーションを含む、すべての実行命令をシミュレートします。BochsはPC環境全体をシミュレートしているため、そこで実行されるソフトウェアは、あたかも本物のマシン上で実行されているかのように「考える」ことができます。このように完全にシミュレートされたアプローチにより、多数のソフトウェアシステムをBochs上で変更することなく実行することができます。

Bochsは、1994年にKevin Lawtonによって開発されたC++言語によるソフトウェアシステムです。このシステムは、Intel 80X86、PPC、Alpha、Sun、MIPSなどのハードウェア環境で動作するように設計されています。ホストが実行されているハードウェアプラットフォームにかかわらず、BochsはIntel 80X86 CPUのIntelハードウェアプラットフォームをシミュレートすることができます。この機能は、他のいくつかのシミュレーションソフトウェアでは利用できません。シミュレーションされているマシン上で何らかの活動を行うためには、Bochsはホストのオペレーティングシステムと対話する必要があります。Bochsのディスプレイ・ウィンドウでキーが押されると、キーストローク・イベントがキーボード・デバイス・プロセッシング・モジュールに送られます。シミュレーションされているマシンが、シミュレーションされているハードディスクからの読み取り操作を行う必要がある場合、Bochsはホスト上のハードディスク・イメージ・ファイルに対して読み取り操作を行います。

Bochsソフトウェアのインストールはとても便利です。http://bochs.sourceforge.net から直接Bochsのインストールパッケージをダウンロードすることができます。お使いのコンピュータのOSがWindowsの場合、インストール方法は通常のソフトウェアと全く同じです。Bochsソフトウェアがインストールされると、C:Program Files\Bochs-2.6\というディレクトリが生成されます（バージョンによって番号が異なります）。お使いのシステムがRedHatなどのLinuxであれば、BochsのRPMパッケージをダウンロードして、以下のようにインストールすることができます。

/ user$ su Password: root# rpm -i bochs-2.6.i386.rpm root# exit user$ \_.

Bochsをインストールする際には、root権限が必要です。そうしないと、Bochsシステムを自分のディレクトリで再コンパイルしなければなりません。また、BochsはX11環境で動作する必要がありますので、Bochsを使用するには、LinuxシステムにX Window Systemがインストールされている必要があります。Bochsをインストールした後は、Bochsパッケージに含まれているLinux dlxデモシステムを使って、Bochsシステムをテストし、慣れ親しんでおくことをお勧めします。また、Bochsのウェブサイトから他のLinuxイメージファイルをダウンロードして、実験を行うこともできます。Linux 0.1xのエミュレーションシステムを作成するための補助的なプラットフォームとして、BochsのWebサイトでSLS Linuxエミュレーションシステムパッケージ（sls-0.99pl.tar.bz2）をダウンロードすることをお勧めします。ハードディスクのイメージファイルを新たに作成する際に、これらのシステムを利用して、ハードディスクのイメージファイルのパーティションやフォーマットを行うことができます。今回のSLSのイメージファイル

### Linuxシステムは、oldlinux.orgから直接ダウンロードすることもできます。http://oldlinux.org/Linux.old/bochs/sls-1.0.zip。ダウンロードしたファイルを解凍した後、そのディレクトリに移動し、設定ファイル名bochsrc.bxrcをダブルクリックすると、BochsにSLS Linuxシステムを実行させることができます。設定ファイル名に接尾辞.bxrcが付いていない場合は、ご自分で修正してください。例えば、元の名前bochsrcをbochsrc.bxrcに修正します。 Bochsシステムの再コンパイルや、他のハードウェアプラットフォームへのBochsのインストール方法については、Bochsユーザーマニュアルに記載されている手順を参照してください。

### 17.1.1 Setting up the Bochs system

BochsでOSを動かすためには、以下のリソースや情報のうち、少なくともいくつかが必要です。

* Bochs and bochsdbg executable files;
* BIOS image files (commonly referred to as 'BIOS-bochs-latest');
* VGABIOS image files (eg 'VGABIOS-lgpl-latest');
* at least one boot image file (floppy, hard disk or CDROM image) file).

### Bochs.exeは、Bochsシステムの実行ファイルです。Bochsでプログラムを追跡・デバッグする必要がある場合は、プログラムを実行するためのbochsdbg.exeも必要です。BIOSとVGABIOSは、それぞれPCのROM BIOSとディスプレイカード内のBIOSソフトをエミュレートしたイメージファイルです。加えて、エミュレーションに使用するシステムのブートイメージファイルも必要です。これらのファイルは連携して動作する必要があるので、Bochsプログラムを実行する前に、シミュレーションのための環境パラメータをいくつか設定する必要があります。これらのパラメータはコマンドラインでBochs実行ファイルに渡すこともできますが、通常はテキスト形式の設定ファイル（ファイルのサフィックスが.bxrcで、Sample.bxrcなど）を使って、特定のアプリケーションの実行パラメータを設定します。以下では、Bochsコンフィギュレーションファイルの設定方法について説明します。

### 17.1.2 \*.bxrc Configuration File

1. Bochsは、設定ファイルの情報をもとに、使用したディスクイメージファイル、動作環境の周辺機器の設定など、仮想マシンの設定情報を見つけ出します。模擬システムごとに、対応する設定ファイルを設定する必要があります。インストールされているBochsシステムが2.1以降であれば、拡張子が「.bxrc」の設定ファイルを自動的に認識し、ファイルのアイコンをダブルクリックするとBochsシステムが自動的に起動します。例えば、設定ファイル名を「bochsrc-0.12.bxrc」とします。Bochsインストールのホームディレクトリ（通常はC:\Program Files\Bochs-2.6\）には、'bochsrc-sample.txt'というテンプレートの設定ファイルがあり、利用可能なすべてのパラメータが詳細な説明とともにリストアップされています。ここでは、私たちの実験でよく変更されるパラメータをいくつか紹介します。
2. megs

模擬システムのメモリ容量を設定します。デフォルトでは32MBです。例えば、シミュレーション・マシンに128MBのシステムを設定したい場合は、設定ファイルに次の行を記述する必要があります。

meg

s

: 128

1. floppya（floppyb）

floppyaは1台目のフロッピーディスクドライブ、floppybは2台目のフロッピーディスクドライブを表しています。フロッピーディスクからシステムを起動する必要がある場合は、floppyaは起動可能なディスクを指す必要があります。ディスクイメージファイルを使用したい場合は、このオプションの後にディスクイメージファイルの名前を書きます。多くのOSでは、Bochはホストシステムのフロッピーディスクドライブを直接読み書きすることができます。これらの実際のドライブのディスクにアクセスするには、デバイス名（Linuxシステム）またはドライブレター（Windowsシステム）を使用します。また、ディスクの挿入状態を示すためにステータスを使用することができます。「ejected」は挿入されていないことを意味し、「inserted」はディスクが挿入されていることを意味します。ここでは、すべてのディスクが挿入されている例をいくつか紹介します。設定ファイルに同じ名前のパラメータが複数列ある場合、最後の列のパラメータのみが動作します。

/ floppya: 1\_44=/dev/fd0, status=inserted # Linuxでは1.44MBのAドライブにアクセス。 floppya: 1\_44=b:, status=inserted # Winでは1.44MBのBドライブにアクセス。

floppya: 1\_44=bootimage.img, status=inserted # イメージファイル bootimage.img を使う。

floppyb: 1\_44=...Linux\rootimage.img, status=inserted # Use image ...Linux\rootimage.img.

1. ata0、ata1、ata2、ata3

これらの4つのパラメータ名は、シミュレーションされたシステムで最大4つのATAチャネルを起動するために使用されます。有効なチャンネルごとに、2つのIOベースアドレスと1つの割り込み要求番号を指定する必要があります。デフォルトでは、ata0のみが有効で、パラメータは以下の値に設定されています。

/ ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14 ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15 ata2: enabled=1, ioaddr1=0x1e8, ioaddr2=0x3e0, irq=11 ata3: enabled=1, ioaddr1=0x168, ioaddr2=0x360, irq=9

1. ata0-master（ata0-slave）

ata0-masterは、シミュレーション・システムの最初のATAチャンネル（0チャンネル）に接続された最初のATAデバイス（ハードディスクやCDROMなど）を示し、ata0-slaveは、最初のチャンネルに接続された2番目のATAデバイスを示しています。以下に例を示しますが、デバイス構成のオプションは表の通りです。

17-1.

/ ata0-master: type=disk, path=hd.img, mode=flat, cylinders=306, heads=4, spt=17, translation=none ata1-master: type=disk, path=2G.cow, mode=vmware3, cylinders=5242, heads=16, spt=50, translation=echs ata1-slave: type=disk, path=3G. img, mode=sparse, cylinders=6541, heads=16, spt=63, translation=auto ata2-master: type=disk, path=7G.img, mode=undoable, cylinders=14563, heads=16, spt=63, translation=lba ata2-slave: type=cdrom, path=iso.sample, status=inserted

ata0-master: type=disk, path="hdc-large.img", mode=flat, cylinders=487, heads=16, spt=63

ata0-slave: type=disk, path="...hdc-large.img", mode=flat, cylinders=121, heads=16, spt=63

|  |  |  |
| --- | --- | --- |
| 表17-1 デバイスの構成オプション  Options | Description | Available value |
| type | Connected device type | [disk | cdrom] |
| path | Image file path name |  |
| mode | Image file type, valid only for disk | [flat | concat | external | dll | sparse | vmware3 | undoable | growing | volatile ] |
| cylinders | Valid only for disk |  |
| heads | Valid only for disk |  |
| spt | Valid only for disk |  |
| status | Valid only for disk | [inserted | ejected] |
| biosdetect | Bios detect type | [none | auto], Only valid for disk on ata0 [cmos] |
| translation | The type of bios convertion (int13), valid only for disk | [none | lba | large | rechs | auto] |
| mode | デバイスから返された文字列を確認する  ATA command |  |

ATAデバイスを設定する際には、接続されているデバイスのタイプを指定する必要があり、「disk」または「cdrom」を指定します。また、デバイスのパス名を指定する必要があります。パス名」には、ハードディスクのイメージファイル、CDROMのisoファイル、またはシステムを直接指し示すCDROMドライブを指定することができます。Linuxシステムでは、システムデバイスをBochsのハードディスクとして使用することができますが、セキュリティ上の理由から、Windowsではシステム上の物理的なハードディスクを直接使用することは推奨されていません。

disk」タイプのデバイスでは、「path」、「cylinders」、「head」、「spt」のオプションが必要です。cdrom」タイプのデバイスでは、「path」オプションが必要です。

ディスク変換方式（従来のint13 bios関数で実装され、DOSなどの古いOSで使用されていた）は次のように定義できます。

* none：No need to translate, suitable for hard disks with a capacity less than 528MB (1032192 sectors);
* large：Standard bit shift algorithm for hard disks with a capacity of less than 4.2 GB (8257536 sectors);
* rechs：The modified shift algorithm uses a pseudo-physical hard disk parameter of 15 heads for hard disks with a capacity less than 7.9 GB (15482880 sectors);
* lba：Standard lba-assisted algorithm. Suitable for hard drives with a capacity less than 8.4GB (16,450,560 sectors);
* auto：Automatically select the best conversion scheme (should be changed if system does not start).

モードオプションは、ハードディスクイメージファイルの使用方法を説明するために使用されます。以下のモードのいずれかになります。

* flat：a flat sequential file;
* concat：Multiple files;
* external：Dedicated by the developer, specified by the C++ class;
* dll：Dedicated by the developer, used by the DLL;
* sparse：Stackable, identifiable, retractable;
* vmware3：Support vmware3 hard disk format;
* undoable：a flat file with a confirmed redo log;
* growing：Capacity scalable image file;  volatile：A flat file with a variable redo log.

上記オプションのデフォルト値は

mode=flat, biosdetect=auto, translation=auto, model=”Generic 1234”

1. boot

boot」は、エミュレートされたマシンのブート用ドライブを定義するために使用します。指定できるのは、フロッピーディスク、ハードディスク、CDROM、ドライブレター「c」「a」などです。例は以下の通りです。

/ boot: a boot: c boot: floppy boot: disk boot: cdrom

1. cpu

cpu」は、シミュレーション・システムでシミュレートされるCPUのパラメータを定義するために使用します。このオプションは4つのパラメータを取ることができます。COUNT」、「QUANTUM」、「RESET\_ON\_TRIPLE\_FAULT」、「IPS」です。

ここで「COUNT」は、システムにエミュレートされているプロセッサの数を示すために使用されます。BochsパッケージがSMPサポートオプション付きでコンパイルされている場合、Bochsは現在最大8個の同時スレッドをサポートしています。しかし、コンパイルされたBochsがSMPをサポートしていない場合、COUNTは1にしか設定できません。

QUANTUM」は、あるプロセッサから別のプロセッサに切り替わる前に実行できる最大の命令数を指定するために使用します。また、このオプションはSMPをサポートするBochsプログラムでのみ利用可能です。

RESET\_ON\_TRIPLE\_FAULT」は、プロセッサにトリプルエラーが発生したときに、CPUが単なるパニックではなくリセット操作を行う必要があることを指定するために使用されます。

IPSは、シミュレーションする1秒あたりの命令数を指定します。これは、Bochsがホストシステム上で実行するIPSの値です。この値は、シミュレーションシステムの時間に関する多くのイベントに影響します。例えば、IPS値を変更すると、VGAの更新速度やその他のシミュレーションシステムの評価に影響を与えます。そのため、使用するホスト性能に応じてこの値を設定する必要があります。設定方法は表17-2を参照してください。例えば

cpu: count=1, ips=

5

0000000

, reset\_on\_triple\_fault

=1

|  |  |  |  |
| --- | --- | --- | --- |
| 表17-2 IPSの設定例  **Bochs version** | **Speed** | **Machine / Compiler** | **Typical IPS** |
| 2.4.6 | 3.4Ghz | Intel Core i7 2600 with Win7x64/g++ 4.5.2 | 85 to 95 MIPS |
| 2.3.7 | 3.2Ghz | Intel Core 2 Q9770 with WinXP/g++ 3.4 | 50 to 55 MIPS |
| 2.3.7 | 2.6Ghz | Intel Core 2 Due with WinXP/g++ 3.4 | 38 to 43 MIPS |
| 2.2.6 | 2.6Ghz | Intel Core 2 Due with WinXP/g++ 3.4 | 21 – 25 MIPS |
| 2.2.6 | 2.1Ghz | Athlon XP with Linux 2.6/g++ 3.4 | 12 – 15 MIPS |

7.ログ

log」のパス名を指定することで、Bochsは実行中にいくつかのログ情報を記録することができます。Bochsで動作しているシステムが正常に動作しない場合、その情報を参照することで基本的な原因を探ることができます。ログは通常、以下のように設定します。

log: bochsout.txt

## 17.2 Running Linux 0.1x system in Bochs

Linuxオペレーティングシステムを実行するには、カーネルのほかにルートファイルシステム（ルートfs）が必要である。ルートファイルシステムは、通常、Linuxシステムの実行に必要なファイル（システム構成ファイルやデバイスファイルなど）やデータファイルを格納する外部デバイスである。最近のLinux OSでは、カーネルイメージファイル（ブートイメージ）がルートファイルシステムに格納されている。システムブートイニシエータは、このルートファイルシステムのデバイスからカーネルの実行コードをメモリにロードして実行する。

ただし，カーネルイメージファイルとルートファイルシステムは，必ずしも同一のデバイスに格納されている必要はなく，フロッピーディスクやハードディスクの同じパーティションに格納されている必要もない。フロッピーディスクのみを使用する場合は、フロッピーディスクの容量に制限があるため、通常、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別々のディスクに配置します。ブート可能なカーネルイメージファイルを格納したフロッピーディスクをカーネルブートディスクと呼ぶ。(bootimage); ルートファイルシステムを格納するフロッピーディスクをルートファイルシステムイメージファイル(rootimage)と呼びます。もちろん、フロッピーディスクからカーネルイメージファイルを読み込み、同時にハードディスク内のルートファイルシステムを使用することもできますし、ハードディスクから直接システムを起動させる、つまり、ハードディスクのルートファイルシステムからカーネルイメージファイルを読み込み、ハードディスク内のルートファイルシステムを使用することもできます。

このセクションでは、Bochs でセットアップされた複数の Linux 0.1x システムの実行方法と、関連する設定ファイルのいくつかの主要なパラメータの設定について説明します。まず、Web サイトから以下の Linux 0.1x システムパッケージをコンピュータのデスクトップにダウンロードします。

http://oldlinux.org/Linux.old/bochs/linux

-

0.1

2

-

0

8

0

324

.zip

### パッケージ名の下6桁は日付情報です。通常は、ダウンロード日が最新のパッケージをお選びください。ダウンロードが完了したら、unzip、7-zip、rarなどの一般的な解凍プログラムを使って解凍してください。なお、このファイルの解凍には250MB程度のディスク容量が必要です。

### 17.2.1 Description of the files in the package

linux-0.12-080324.zipファイルを解凍すると、linux-0.12-080324という名前のディレクトリが生成されます。そのディレクトリに入ってみると、以下のように20個ほどのファイルがあることがわかります。

[root@www linux-0.12-080324]# ls -o -g total 256916

-rw-r--r-- 1 3078642 Mar 24 10:49 bochs-2.3.6-1.i586.rpm

-rw-r--r-- 1 3549736 Mar 24 10:48 Bochs-2.3.6.exe

-rw-r--r-- 1 15533 Mar 24 18:04 bochsout.txt

-rw-r--r-- 1 1774 Mar 24 20:13 bochsrc-0.12-fd.bxrc

-rw-r--r-- 1 5903 Mar 24 17:56 bochsrc-0.12-hd.bxrc

-rw-r--r-- 1 35732 Dec 24 20:15 bochsrc-sample.txt

-rw-r--r-- 1 150016 Mar 6 2004 bootimage-0.12-fd

-rw-r--r-- 1 154624 Aug 27 2006 bootimage-0.12-hd

-rw-r--r-- 1 68 Mar 24 12:21 debug.bat

-rw-r--r-- 1 1474560 Mar 24 15:27 diska.img

-rw-r--r-- 1 1474432 Aug 27 2006 diskb.img

-rw-r--r-- 1 7917 Mar 24 11:32 linux-0.12-README

-rw-r--r-- 1 1474560 Mar 24 17:03 rootimage-0.12-fd

-rw-r--r-- 1 251338752 Mar 24 18:04 rootimage-0.12-hd

-rw-r--r-- 1 21253 Mar 13 2004 SYSTEM.MAP

[root@www linux-0.12-080324]#.

* 本パッケージには、2つのBochsインストーラ、2つのBochs .bxrc設定ファイル、カーネルコードを含む2つのbootimageファイル、フロッピーディスクとハードディスクのルートファイルシステム（rootimage）ファイル、その他のファイルが含まれています。その中のREADMEファイルでは、各ファイルの目的を簡単に説明しています。ここでは，各ファイルの目的をより詳しく説明します。
* Bochs-2.3.6-1.i586.rpm is the Bochs installer for the Linux operating system. You can re-download the latest program.
* Bochs-2.3.6.exe is the Bochs installer for the Windows operating system platform. The latest version of the Bochs software can be downloaded from http://sourceforge.net/projects/bochs/. As Bochs continues to improve, some newer versions may cause compatibility issues. This needs to be resolved by modifying the .bxrc configuration file, and some issues need to be resolved by modifying the Linux 0.1x kernel code.
* bochsout.txt is a log file that is automatically generated when the Bochs system is running. It contains various status information for the Bochs runtime. When running Bochs has problems, you can check the contents of this file to preliminarily determine the cause of the problem.
* bochsrc-0.12-fd.bxrc is the configuration file that allows the system to boot from a floppy disk. This configuration file is used to boot the Linux 0.12 system from the Bochs Virtual A drive (/dev/fd0), ie the kernel image file is set in virtual disk A and the subsequent root file system is required to be inserted into the current virtual boot drive. During the boot process it will ask us to "insert" the root filesystem disk (rootimage-0.12-fd) in the virtual A drive. The kernel image and boot file used by this configuration file is bootimage-0.12-fd. After Bochs is properly installed, double-click this configuration file to run the configured Linux 0.12 system.
* bochsrc-0.12-hd.bxrc is also a configuration file set to boot from drive A, but will use the root file system in the hard drive image file (rootimage-0.12-hd). This configuration file is booted using bootimage-0.12-hd. Similarly, after properly installing Bochs, double-click on this configuration file to run the configured Linux 0.12 system.
* bootimage-0.12-fd is an image file generated by the compiled kernel. It contains the code and data for the entire kernel, including the code for the floppy boot sector. You can run the configured Linux 0.12 system by double-clicking on the relevant configuration file.
* bootimage-0.12-hd is the kernel image file used to use the root file system on the virtual hard disk, that is, the root file system device number of the 509th and 510th bytes of the file has been set to the 1st partition of the C hard disk (/dev /hd1), the device number is 0x0301.
* debug.bat is a batch program that starts the Bochs debugging function on the Windows platform.
* なお、Bochsがインストールされているディレクトリによっては、パス名を変更する必要がありますのでご注意ください。また、LinuxシステムにインストールされているBochsシステムには、デフォルトではデバッグ機能が含まれていません。Linuxではgdbプログラムを使って直接デバッグすることができます。それでもBochsのデバッグ機能を利用したいのであれば、Bochsのソースコードをダウンロードして、自分でカスタマイズする必要があります。
* diska.img and diskb.img are two floppy image files in DOS format. It contains some utilities. In Linux 0.12 you can use the command mcopy and other commands to access these two image files. Of course, you need to dynamically "insert" the corresponding "floppy disk" before accessing. When you double-click the bochsrc-0.12-fd.bxrc or bochsrc-0.12--hd.bxrc configuration file to run the Linux 0.12 system, the B drive is configured to be "inserted" with the diskb.img disk.
* rootimage-0.12-hd is the virtual hard disk image file mentioned above, which contains 3 partitions. The first partition is a MINIX file system type 1.0 root file system, and the other two partitions are also MINIX 1.0 file system types, and some source code files for testing are stored. You can load and use these spaces by using the mount command.
* rootimage-0.12-fd is the root file system on the floppy disk. This root file system disk is used when running the Linux 0.12 system using the bochsrc-0.12-fd configuration file.
* The SYSTEM.MAP file is the kernel memory storage location information file generated when the Linux 0.12 kernel is compiled. The contents of this file are very useful when debugging the kernel.

### 17.2.2 Installing the Bochs

パッケージに含まれるbochs-2.3.6-1.i586.rpmファイルは、Linuxで使用するBochsのインストーラです。Bochs-2.3.6.exe は、Windows オペレーティングシステムで使用する Bochs インストーラです。Bochsソフトウェアの最新版は、以下のウェブサイトの場所で常に入手可能です。

http://sourceforge.net/projects/bochs/

Linuxシステムで実験している場合、コマンドラインでrpmコマンドを実行するか、Xウィンドウで上記パッケージの最初のファイルをダブルクリックすることで、Bochsソフトウェアをインストールすることができます。

rpm

-

i bochs

-

2.

3

.

6

-

1

.i586.rpm

### Windowsシステムをお使いの場合、Bochs-2.3.6.exeファイルのアイコンをダブルクリックするだけで、Bochsシステムをインストールすることができます。インストール後、バッチファイルdebug.batの内容を、インストール先のディレクトリに合わせて変更してください。なお、以下の実験手順・例では、主にWindowsプラットフォーム上でのBochsの使用方法を紹介しています。

### 17.2.3 Running the Linux 0.1x System

BochsでLinux 0.1xシステムを動かすのはとても簡単です。Bochsソフトウェアが正しくインストールされたら、適切なBochs設定ファイル(\*.bxrc)をダブルクリックするだけで開始できます。ランタイムシミュレーションのためのPC環境は、各設定ファイルで設定されています。これらのファイルは、任意のテキストエディタで変更することができます。Linux 0.12システムを実行するためには、対応する設定ファイルは通常、以下の行を含むだけで十分です。

/ romimage: file=$BXSHARE/BIOS-bochs-latest vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest megs: 16 floppya: 1\_44="bootimage-0.12-hd", status=inserted ata0-master: type=disk, path="rootimage-0.12-hd", mode=flat, cylinders=487, heads=16, spt=63 boot: a

最初の2行は、模擬PCのROM BIOSとVGAディスプレイカードのROMプログラムを示しており、通常は変更する必要はありません。3行目は、PCの物理的なメモリ容量を示しており、16MBに設定されています。デフォルトのLinux 0.12カーネルは16MBまでのメモリしかサポートしていないため、大きな設定をしても動作しません。floppyaは，模擬PCのフロッピーディスクドライブAが，1.44MBのディスクタイプで，bootimage-0.12-fdのフロッピーイメージファイルを使用するように設定されており，挿入状態であることを示しています．対応するfloppybは、Bドライブに使用または挿入されているフロッピーイメージファイルを示すために使用することができます。パラメータata-masterは，シミュレーションされたPCに装着されている仮想ハードディスクの容量やハードディスクのパラメータを指定するためのものである。これらのハードディスク・パラメーターの具体的な意味については、前の説明を参照してください。また、ata0-slaveは、2つ目の仮想ハードディスクが使用するイメージファイルとパラメータを指定するために使用できます。最後の「boot」は、ブートドライブを指定するためのもので、Aドライブから起動するか、Cドライブ（ハードディスク）から起動するかを設定できます。ここでは、Aドライブ（a）から起動するように設定しています。

1. bochsrc-0.12-fd.bxrcファイルを使って、Linux 0.12システムを起動する。

つまり、フロッピーディスクからLinux 0.12システムを起動し、現在のドライブにあるルートファイルシステムを使用します。

この方法では、Linux 0.12システムを起動するのに、bootimage-0.12-fdとrootimage-0.12-fdの2つのフロッピーディスクしか使いません。上に挙げた数行の設定ファイルの内容は、bochsrc-0.12-fd.bxrcの基本的な設定で、ブートイメージファイルがbootimage-0.12-fdに置き換えられているだけです。この設定ファイルをダブルクリックしてLinux 0.12システムを実行すると、図17-1に示すように、Bochsディスプレイのメインウィンドウにメッセージが表示されます。

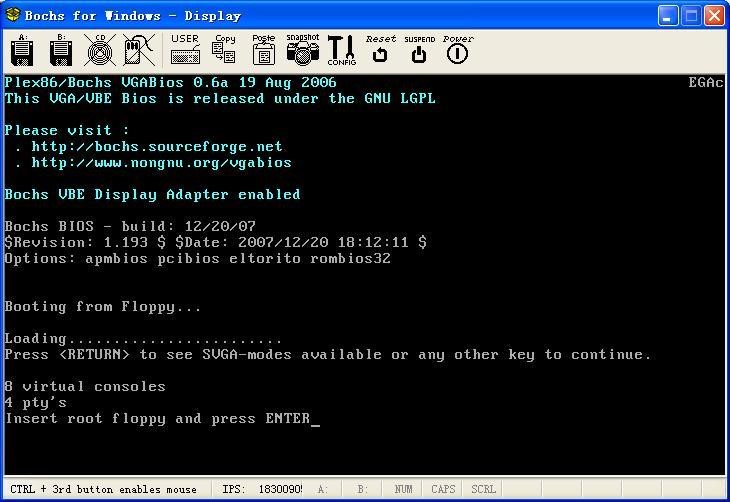


図17-1 フロッピーディスクからの起動とフロッピーディスク内のルートfsの使用

bochsrc-0.12-fd.bxrcは、Linux 0.12ランタイムがAドライブから起動するように設定されており、カーネルイメージファイルbootimage-0.12-fdは、ルートファイルシステムが現在起動しているドライブ(Aディスク)にあることを必要とするからです。そのため、カーネルはカーネルブートイメージファイルbootiamge-0.12-fdを「remove」し、ルートファイルシステムを「insert in」するよう求めるメッセージを表示します。この時点で、ウィンドウの左上にあるAディスクのアイコンを使って、Aディスクを「交換」することができます。このアイコンをクリックして、元のイメージファイル名(bootimage-0.12-fd)をrootimage-0.12-fdに変更すれば、フロッピーディスクの交換作業は完了です。OK」ボタンをクリックしてダイアログウィンドウを閉じた後、Enterキーを押してカーネルにフロッピーディスク上のルートファイルシステムを読み込ませると、最後に図17-2に示すようなコマンドプロンプトラインが表示されます。

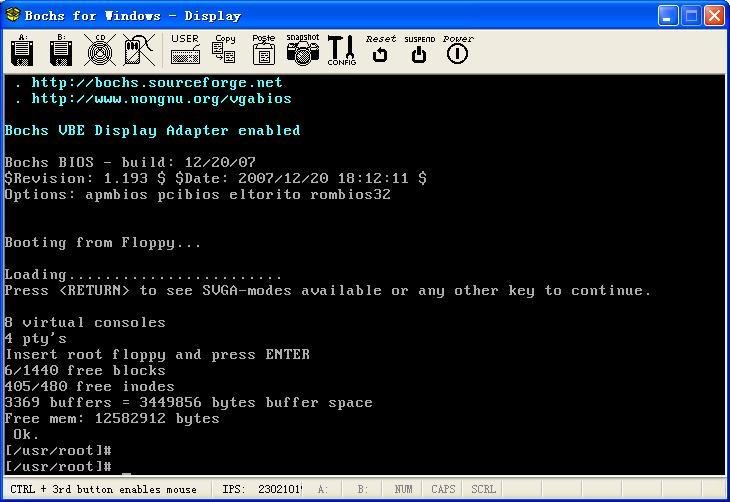


図17-2 フロッピー・ディスクを「交換」し、Enterを押して実行を続ける

2. 2. bochsrc-0.12-hd.bxrcファイルを使って、Linux 0.12システムを実行する。

## この設定ファイルでは、ブートフロッピーディスク（Aディスク）からLinux 0.12カーネルイメージファイルbootimage-0.12-hdを読み込み、ハードディスクイメージファイルrootimage-0.12-hdの第1パーティションのルートファイルシステムを使用します。bootimage-0.12-hdファイルの509バイト目と510バイト目がCドライブの第1パーティションのデバイス番号0x0301（つまり0x01, 0x03）に設定されているため、カーネルの初期化時に自動的に仮想Cドライブの第1パーティションからルートファイルシステムの読み込みが開始されます。この時点で、bochsrc-0.12-hd.bxrcファイル名をダブルクリックして、Linux 0.12システムを直接実行すると、図17-2のような画面が表示されます。

## 17.3 Access Information in a Disk Image File

Bochsは、ディスク・イメージ・ファイルを使って、シミュレーション・システムの外部記憶装置をエミュレートしています。模擬オペレーティングシステムのすべてのファイルは、フロッピーディスクやハードディスク装置の形式でイメージファイルに保存されます。このため、BochsではホストOSと模擬システムの間で情報を交換するという問題が生じる。Bochsシステムは、ホストのフロッピーディスクドライブやCDROMドライブなどの物理的なデバイスを使って直接実行するように構成することができますが、そのような情報交換方法を利用するのは面倒です。したがって、Imageファイルの情報を直接読み書きするのがベストです。模擬OSにファイルを追加したい場合は、Imageファイルに保存し、ファイルを取得したい場合は、Imageファイルから読み出します。しかし、Imageファイルに保存されている情報は、対応するフロッピーディスクやハードディスクのフォーマットに保存されているだけでなく、特定のファイルシステムのフォーマットにも保存されているため、Imageファイルにアクセスするプログラムは、そのファイルシステムのフォーマットに対応していなければなりません。そのため、Imageファイルにアクセスするプログラムは、そのファイルシステムを認識できなければ動作しません。本章の目的のためには、Imageファイルの中のMINIXおよび/またはDOSファイルシステム形式を識別するためのいくつかのツールが必要です。

一般的に、模擬システムと交換するファイルのサイズが小さい場合は、フロッピーのイメージファイルを交換媒体として使用することができます。模擬システムから取得したり、模擬システムに入れたりするファイルのサイズが大きい場合は、既存のLinuxシステムを使ってイメージファイルをマウントすることができます。以下では、この2つの観点から使用できるいくつかの方法について説明します。

 Use the disk image tool to access information (small files or split files) in the floppy image file;  Use the loop device to access the hard disk image file in Linux. (a large amount exchange);  Use iso format file for information exchange (a large amount exchange).

### 17.3.1 Using WinImage Software

フロッピーイメージファイルを使うことで、模擬システムと少量のファイルを交換することができます。前提条件として、模擬システムがDOSフォーマットのフロッピーディスクの読み書きをサポートしている必要があります。例えば、mtoolsソフトウェアを使用します。mtoolsは、UNIXライクなシステムでMSDOSファイルシステムのファイルにアクセスするためのプログラムです。このソフトウェアは、copy、dir、cd、format、del、md、rdといったMSDOSの一般的なコマンドを実装しています。これらのコマンド名にmの文字を加えると、mtoolsの対応するコマンドになります。以下、具体的な操作方法を例を挙げて説明します。

ファイルを読み書きする前に、まず、上述の方法で1.44MBのImageファイル（ファイル名はdiskb.imgとする）を用意し、Linux 0.12の設定ファイルboochs.bxrcを修正する必要があります。floppyaパラメータに以下の行を追加します。

floppyb: 1\_44="diskb.img", status=inserted

つまり、2台目の1.44MBフロッピーディスク装置がシミュレーションされたシステムに追加され、その装置が使用するイメージファイル名はdiskb.imgとなります。

Linux 0.12システムからファイル(hello.c)を取り出したい場合は、設定ファイルのアイコンをダブルクリックして、Linux 0.12システムを起動することができるようになりました。Linux 0.12 システムに入ったら、DOS フロッピーディスク読み書きツール mtools を使って、 hello.c ファイルを 2 枚目のフロッピーイメージに書き込みます。フロッピーイメージがBochsを使って作成されたものであったり、フォーマットされていない場合は、mformat b:コマンドを使って先にフォーマットしておきます。

/ [/usr/root]# mcopy hello.c b: Copying HELLO.C [/usr/root]# mdir b:

ドライブBのボリュームにはラベルがない

B:/のディレクトリ

HELLO C 74 4-30-104 4:47p

1 ファイル（s） 1457152 バイト フリー

[/usr/root]#

### ここでBochsシステムを終了し、WinImageでdiskb.imgファイルを開きます。WinImageのメインウィンドウにhello.cファイルが表示されます。マウスでそのファイルを選択し、デスクトップにドラッグすると、ファイルの取得が完了します。ファイルをシミュレートされたシステムに入れる必要がある場合は、手順が全く逆になります。また、WinImageはDOS形式のディスクファイルにのみアクセスして操作することができ、MINIXファイルシステムのような他の形式のディスクファイルにはアクセスできないことに注意してください。

### 17.3.2 Using an Existing Linux System

既存のLinuxシステム（Redhatなど）では、ループデバイスを使ってイメージファイルに格納されたファイルシステムにアクセスするなど、さまざまなファイルシステムに対応しています。フロッピーのイメージファイルの場合、マウントコマンドを使ってイメージ内のファイルシステムを読み込み、読み書き可能な状態にすることができます。例えば、rootimage-0.12にあるファイルにアクセスする必要がある場合、次のコマンドを実行すればよい。

/ [root@plinux images]# mount -t minix rootimage-0.12 /mnt -o loop

[root@plinux images]# cd /mnt [root@plinux mnt]# ls bin dev etc root tmp usr

[root@plinux mnt]#.

mountコマンドの「-t minix」オプションは、読み込まれるファイルシステムのタイプがMINIXであることを示し、「-o loop」オプションは、ファイルシステムがループデバイスによって読み込まれることを示します。DOSフォーマットのフロッピーイメージファイルにアクセスする必要がある場合は、mountコマンドのファイルタイプオプション「minix」を「msdos」に置き換えるだけです。 ハードディスクのイメージファイルにアクセスする場合は、上記とは操作方法が異なります。フロッピーディスクのイメージファイルは、通常、完全なファイルシステムのイメージを含んでいるので、マウントコマンドを使ってフロッピーディスクのイメージ内のファイルシステムを直接読み込むことができますが、ハードディスクのイメージファイルは、通常、パーティション情報を含んでおり、ファイルシステムは各パーティションに作成されます。そのため、まず必要なパーティションをロードしてから、そのパーティションを完全な "ビッグ "フロッピーディスクとして扱う必要があるのです。

したがって、ハードディスクのイメージファイルのパーティションの情報にアクセスするためには、まず、イメージファイルのパーティション情報を理解して、イメージファイルでアクセスするパーティションの開始セクタ・オフセット位置を決定する必要があります。ハードディスクのイメージファイルのパーティション情報については、fdiskコマンドを使ってシミュレーションシステム上で確認する方法と、ここで説明する方法があります。ここでは、以下のパッケージに含まれるイメージ・ファイル rootimage-0.12-hd.img を例にとり、第1パーティションのファイル・システムにアクセスする方法を説明します。

http://

oldlinux.org/Linux.old/bochs/

linux

-

0.1

2

-

0

8

032

4

.zip

ここでは、ループデバイスの設定や制御コマンドであるlosetupを使用する必要があります。このコマンドは主に、通常のファイルやブロックデバイスとループデバイスの関連付けを行ったり、ループデバイスの解除やループデバイスの状態を問い合わせるために使用します。このコマンドの詳細な説明については、Linux のオンラインマニュアルのページを参照してください。

まず、以下のコマンドを実行して、rootimage-0.12-hdファイルをloop1に関連付け、fdiskコマンドでパーティション情報を表示します。

[root@www linux-0.12-080324]# losetup /dev/loop1 rootimage-0.12-hd

[root@www linux-0.12-080324]# fdisk /dev/loop1

コマンド（mはヘルプ）：x

エキスパートコマンド（m for help）：p

ディスク /dev/loop1: 16 ヘッド, 63 セクタ, 487 シリンダ

Nr AF Hd Sec Cyl Hd Sec Cyl Start Size ID

1. シンプルカーネルのコンパイルと実行
2. 80 1 1 0 15 63 130 1 132047 81
3. 00 0 1 131 15 63 261 132048 132048 81
4. 00 0 1 262 15 63 392 264096 132048 81
5. 00 0 1 393 15 63 474 396144 82656 82

エキスパートコマンド(m for help)：q

[root@www linux-0.12-080324]#.

上記のfdiskによるパーティション情報からわかるように、イメージファイルには3つのMINIXパーティション（ID=81）と1つのスワップパーティション（ID=82）が含まれています。第1パーティションの内容にアクセスする必要がある場合は、そのパーティションの開始セクタ番号（つまり、パーティションテーブルの「開始」列の内容）を書き留めます。別のパーティションのハードディスクイメージにアクセスする必要がある場合は、該当するパーティションの開始セクタ番号を覚えておけばよいのです。

次に、losetupの"-d "オプションを使用して、rootimage-0.12-hdファイルをloop1からアンリンクし、イメージファイルの第1パーティションの先頭に再関連付ける作業を行う。このためには、関連する開始バイトのオフセット位置を示す losetup の「-o」オプションを使用する必要があります。上記のパーティション情報からわかるように、ここでの最初のパーティションの開始オフセット位置は1 \* 512バイトです。したがって、最初のパーティションをloop1に再関連づけした後、マウントコマンドを使ってファイルにアクセスできます。

[root@www linux-0.12-080324]# losetup -d /dev/loop1

[root@www linux-0.12-080324]# losetup -o 512 /dev/loop1 rootimage-0.12-hd

[root@www linux-0.12-080324]# mount -t minix /dev/loop1 /mnt

[root@www linux-0.12-080324]# cd /mnt

[root@www mnt]# ls

bin etc home MCC-0.12 mnt1 root usr dev hdd image mnt README tmp vmlinux

[root@www mnt]#

パーティション内のファイルシステムへのアクセスが終了したら、最後にファイルシステムをアンマウントして、関連付けを解除します。

/ [root@www mnt]# cd

[root@www ~]# umount /dev/loop1

[root@www ~]# losetup -d /dev/loop1

[root@www ~]#

## 17.4 Compiling and running the simple kernel

簡単なマルチタスクカーネルのサンプルプログラムは、前章の80386保護モードと第4章のそのプログラミングで与えられており、これをLinux 0.00システムと呼んでいます。これには特権レベル3で動作する2つのタスクが含まれており、画面上の文字AとBを循環させ、クロックタイミング制御のもとでタスク切り替え動作を行います。Bochsシミュレーション環境で動作するように設定されたパッケージは、書籍のウェブサイトで与えられています。

シンプルカーネルのコンパイルと実行

http://oldlinux.org/Linux.old/bochs/linux-0.00-050613.zip http://oldlinux.org/Linux.old/bochs/linux-0.00-041217.zip

私たちは、上記のファイルのいずれかをダウンロードして実験することができます。最初のパッケージで与えられているプログラムは、ここで説明したものと同じです。2つ目のパッケージのプログラムは，若干の違い（カーネルの先頭コードが0x10000番地に直接実行される）がありますが，原理はまったく同じです。ここでは、第1パッケージのソフトウェアのプログラムを例にして、実験を説明します。第2パッケージのソフトウェアは、実験的な解析は読者にお任せします。linux-0.00-050613.zipパッケージを解凍ソフトで解凍すると、カレントディレクトリにlinux-0.00サブディレクトリが生成されます。このパッケージには、以下のファイルが含まれていることがわかります。

1. linux-0.00.tar.gz - Compressed source file;
2. linux-0.00-rh9.tar.gz - Compressed source file;
3. Image - Kernel boo image file;
4. bochsrc-0.00.bxrc - Bochs configuration file;
5. rawrite.exe - The program to write an Image to a floppy disk under Windows.
6. README - Package documentation;

最初のファイルである linux-0.00.tar.gz は、カーネルサンプルのソースファイルを圧縮したものです。Linux 0.12系でコンパイルすることで、カーネルイメージファイルを生成することができます。2番目のファイルもカーネル例の圧縮されたソースファイルですが、ソースプログラムはRedHat 9 Linuxでコンパイルすることができます。3つ目のファイルImageは、ソースプログラムをコンパイルした実行可能コードの1.44MBフロッピーイメージファイルです。4番目のファイルbochsrc-0.00.bxrcは、Bochs環境で実行するときに使用するBochs設定ファイルです。PCエミュレーションソフトBochsがシステムにインストールされていれば、bochsrc-0.00.bxrcのファイル名をダブルクリックすることで、イメージ内のカーネルコードを実行することができる。5つ目は、イメージファイルをフロッピーディスクに書き込むためのDOSまたはWindows上のユーティリティプログラムです。RAWRITE.EXEプログラムを直接実行して、ここのカーネルイメージファイルを1.44MBのフロッピーディスクに書き込んで実行することができる。

上記のカーネル例のソースコードは、linux-0.00-tar.gzファイルに含まれています。このファイルを解凍すると、boot.sとhead.sのプログラムに加えて、makefileを含むソースファイルを含むサブディレクトリが生成されます。as86/ld86のコンパイルとリンクによって生成された「boot」ファイルの先頭部分には32バイトのMINIX実行ファイルのヘッダ情報が、as/ldのコンパイルとリンクによって生成された「head」ファイルの先頭部分には1024バイトのa.outヘッダデータが含まれているので、カーネルの「Image」ファイルを作成する際には、これらのヘッダ情報を削除する必要があります。2つの「dd」コマンドを使ってヘッダー情報を除去し、それをカーネルイメージのImageファイルにまとめることができます。

このImageファイルは、ソースコードのディレクトリで直接makeコマンドを実行することで生成されます。すでにmakeコマンドを実行している場合は、先に「make clean」を実行してからmakeコマンドを実行してください。

[/usr/root/linux-0.0]# ls -l total 9

-rw------- 1 root root 487 Jun 12 19:25 Makefile

-rw------- 1 root 4096 1557 Jun 12 18:55 boot.s -rw------- 1 root root 5243 Jun 12 19:01 head.s

[/usr/root/linux-0.0]# make

シンプルカーネルのコンパイルと実行

as86 -0 -a -o boot.o boot.s ld86 -0 -s -o boot boot.o gas -o head.o head.s gld -s -x -M head.o -o system > System.map

dd bs=32 if=boot of=Image skip=1

16+0の記録で

16+0 レコードアウト

dd bs=512 if=system of=Image skip=2 seek=1

16+0の記録で

16+0 レコードアウト

[/usr/root/linux-0.0]#.

イメージファイルをAディスクイメージファイルにコピーするには、次のように「make disk」コマンドを実行します。ただし、このコマンドを実行する前に、Linux 0.12系をBochsでコンパイルしている場合は、テスト後にLinux 0.12系を復元できるように、ブートイメージのディスクファイル（例えば、bootimage-0.12-hd）をコピーして保存しておいてください。イメージファイルです。

[/usr/root/linux-0.0]# ls

イメージ System.map boot.o head.o system

Makefile boot boot.s head.s

[/usr/root/linux-0.0]# make disk dd bs=8192 if=Image of=/dev/fd0

1+1レコードイン 1+1レコードアウト sync;sync;sync

[/usr/root/linux-0.0]#.

このシンプルなカーネルの例を実行するには、マウスを使ってBochsウィンドウのRESETアイコンを直接クリックします。その操作は下図のようになります。その後、Linux 0.12システムの実行を再開したい場合は、起動ファイルを先ほど保存したイメージファイルで上書きしてください。

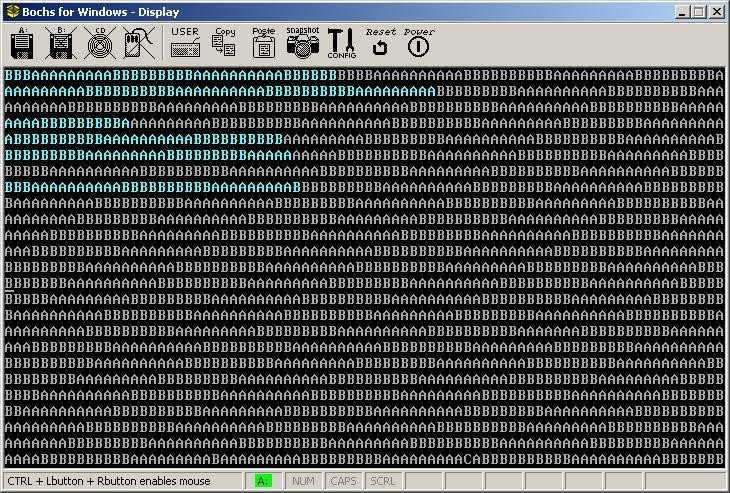


図17-3 シンプルなカーネルランタイムの画面表示

## 17.5 Using Bochs to Debug the Kernel

### Bochsは非常に強力なOSカーネルのデバッグ機能を備えており、これがBochsを実験環境として選択した主な理由の一つです。Bochsのデバッグ機能については、Windows環境でのBochsデバッグ操作の基本的な方法を説明するために、Linux 0.12カーネルを使用している上記のセクション17.2を参照してください。

### 17.5.1 Running the Bochs Debugger

Bochsシステムが「C:\\Bochs-2.3.6\」というディレクトリにインストールされ、Linux 0.12システム用のBochs設定ファイル名が「bochsrc-0.12-hd.bxrc」であるとします。次に、カーネルイメージファイルのあるディレクトリに、次のような簡単なバッチファイルrun.batを作成します。

"C:

\

Program Files

\

Bochs

-

2.

3

.

6

\

bochsdbg"

-

q

-

f bochsrc

-

0.12

-

hd.bxrc

その中で、bochsdbgは、Bochsのデバッグ実行プログラムです。パラメータ「-q」はクイックスタート（設定インターフェースをスキップする）を意味し、パラメータ「-f」は設定ファイル名が続くことを意味します。パラメーターが"-h "の場合、プログラムはすべてのオプションパラメーターのヘルプ情報を表示します。バッチコマンドを実行して、デバッグ環境に入ります。この時点では、Bochsのメイン表示ウィンドウは空白で、コントロールウィンドウには以下のような同様の内容が表示されます。

C:Linux-0.12>"C:Program Files\Bochs-2.3.6\bochsdbg" -q -f bochsrc-0.12-hd.bxrc

00000000000i[APIC?] 初期化中のローカルAPIC

========================================================================

Bochs x86エミュレータ 2.3.6

2007年12月24日のCVSスナップショットからの構築

========================================================================

00000000000i[ ] bochsrc-hd-new.bxrcから設定を読み込む

00000000000i[ ] Bochs GUIとしてのwin32モジュールのインストール

00000000000i[ ] using log file bochsout.txt

次はt=0で

(0) [0xfffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b ; ea5be000f0

<bochs:1>

この時点で、Bochsデバッグシステムの実行開始準備が整い、CPU実行ポインタがROM BIOSのアドレス0x000fffff0の命令に向けられました。ここで、「<bochs:1>」はコマンドライン入力のプロンプトで、数字は現在のコマンドのシリアル番号を示しています。コマンドプロンプトの後に「help」コマンドを入力すると、システムをデバッグするための基本的なコマンドを一覧表示することができます。コマンドの詳細な使い方を知りたい場合は、「help」コマンドの後に、一重引用符で囲まれた特定のコマンドを入力します。" help 'vbreak' " と入力します。以下を参照してください。

<bochs:1>のヘルプ

help - デバッガーのコマンドのリストを表示 help 'command' - 短いコマンドの説明を表示

-\*- デバッガコントロール -\*-

help, q|quit|exit, set, instrument, show, trace-on, trace-off, record, playback, load-symbols, slist

-\*- 実行制御 -\*-

c|cont, s|step|stepi, p|n|next, modebp

-\*- ブレークポイントの管理 -\*-

v|vbreak, lb|lbreak, pb|pbreak|b|break, sb, sba, blist, bpe, bpd, d|del|delete -\*- CPUとメモリの内容 -\*-

x, xp, u|disas|disassemble, r|reg|registers, setpmem, crc, info, dump\_cpu, set\_cpu, ptime, print-stack, watch, unwatch, ?|calc.

<bochs:2> help 'vbreak' help vbreak vbreak seg:off - 仮想アドレス命令のブレークポイントを設定します。

<bochs:3>

よく使われるコマンドの一部を以下に示します。すべてのデバッグコマンドの完全なリストは、Bochs自身のヘルプファイル(internal-debugger.html)に記載されているか、オンラインヘルプ情報("help" コマンド)を参照してください。

1. 制御コマンドを実行する。シングルステップまたはマルチステップの命令実行を制御します。

/ c 連続実行

stepi [count] 「count」命令を実行、デフォルトは1 si [count] lbid. step [count] lbid. s [count] lbid.

p 's'と似ていますが、割り込みや関数の命令をシングルステップで実行するものです。 n（または次）のlbid。

Ctrl-C 実行を停止し、コマンドラインプロンプトに戻る。

Ctrl-D 空のコマンドラインプロンプトでコマンドを入力すると、Bochsを終了します。

quit, q デバッグを終了する。

2. ブレークポイント設定コマンド。ここで、「seg」「off」「addr」には、「0x」で始まる16進数、「0」で始まる10進数、8進数が使用できます。

/ vbreak seg:off 仮想アドレスに命令ブレークポイントを設定します。

vb seg:off lbid.

lbreak addr Set the instruction breakpoint on the linear address.

lb addr lbidです。

pbreak [\*] addr Set the instruction breakpoint on the physical address. Where '\*' is an option for compatibility with GDB. pb [\*] addr lbid. break [\*] addr lbid. b [\*] addr lbid. info break Displays the status of all current breakpoints. delete n Delete a breakpoint. del n lbid.

d n lbid。

1. Memory operation commands

/ x /nuf addr リニアアドレス'addr'のメモリ内容を確認します。addr'が指定されていない場合、デフォルトでは次のアドレスになります。

xp /nuf addr Check the memory contents at the physical address 'addr'.

オプションのパラメーター「n」、「u」、「f」は、次のようになります。

n 表示するメモリーユニットの数です。初期値は1です。

b(Bytes)は1バイト、h(Halfwords)は2バイト、w(Words)は4バイト、g(Giantwords)は8バイトとなります。 注：これらの略語は、主にGDBデバッガの表現との整合性をとるために、インテルの略語とは異なります。

f 表示形式、デフォルトでは「x」が選択されています。

x (hex); d (decimal); u (unsigned); o (octal); t (binary); c (char) charsを表示します。charとして表示できない場合は、コードを直接表示します。

crc addr1 addr2 addr1 から addr2 までの物理メモリの CRC チェックサムを表示します。

info dirty 前回このコマンドを実行した後に変更された物理メモリページを表示します。ページの最初の20バイトのみが表示されます。

1. Information display and CPU register operation commands

/ info program プログラムの実行状況を表示します。

info registers CPUレジスタを表示（浮動小数点レジスタは表示しない） info break 現在のブレークポイント設定状態を表示 set $reg = val CPUレジスタ（セグメントとフラグレジスタを除く）の内容を変更します。 例えば、 set $eax = 0x01234567; set $edx = 25 dump\_cpu CPUのすべてのステータス情報を表示します。

set\_cpu CPUの全ステータス情報を設定。

dump\_cpu "および "set\_cpu "コマンドで表示されるコンテンツのフォーマットは次のとおりです。

"eax:0x%x\n"

"ebx:0x%x\n"

"ecx:0x%x\n"

"edx:0x%x\n"

"ebp:0x%x\n"

"esi:0x%x\n"

"edi:0x%x\n"

"esp:0x%x\n"

"eflags:0x%x\n"

"eip:0x%x\n"

"cs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"

"ss:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"

"ds:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"

"es:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"

"fs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"

"gs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"

"ldtr:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"

"tr:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"

"gdtr:base=0x%x, limit=0x%x\n"

"idtr:base=0x%x, limit=0x%x\n"

"dr0:0x%x\n"

"dr1:0x%x\n"

"dr2:0x%x\n"

"dr3:0x%x\n"

"dr4:0x%x\n"

"dr5:0x%x\n"

"dr6:0x%x\n"

"dr7:0x%x\n"

"tr3:0x%x\n"

"tr4:0x%x\n"

"tr5:0x%x\n"

"tr6:0x%x\n"

"tr7:0x%x\n"

"cr0:0x%x\n"

"cr1:0x%x\n"

"cr2:0x%x\n"

"cr3:0x%x\n"

"cr4:0x%x\n"

"inhibit\_int:%u\n"

"dead\"

among them:  's' means a selector;

* 'dl' is the low 4-byte of the segment descriptor in the selector shadow register;
* 'dh' is the high 4-byte of the segment descriptor in the selector shadow register;
* 'valid' indicates whether a valid shadow descriptor is being stored in the segment register;
* 'inhibit\_int' is an instruction delay interrupt flag. If set, it means that the instruction that has just been executed by the previous one is an instruction that delays the CPU from accepting the interrupt (for example, STI, MOV SS);

また、"set\_cpu "コマンドの実行時には、"Error: ... "というフォーマットでエラーが報告されます。これらのエラーメッセージは、入力行ごとに表示されたり、最後の「done」が表示された後に表示されたりします。set\_cpu "コマンドが正常に実行された場合は、"OK "を表示してコマンドを終了します。

5. 解体コマンド

disassemble start end Disassemble instructions within a given linear address range.

私は

* 以下は、Bochsの新しいコマンドの一部ですが、Windows環境のファイル名を含むコマンドは正常に動作しない場合があります。
* record *filename* -- Write your input command sequence to the file 'filename' during execution. The file will contain lines of the form "%s %d %x". The first parameter is the event type; the second is the timestamp; the third is the data of the related event.
* playback *filename* -- The execution command is played back using the contents of the file 'filename'. You can also type other commands directly in the control window. Each event in the file will be played back, and the playback time will be counted relative to the time the command was executed.
* print-stack [num words] -- Display num 16-bit words at the top of the stack. The default value of num is 16. When the base address of the stack segment is 0, the command can be used normally only in protected mode.
* load-symbols [global] *filename* [offset] -- Load symbol information from the file 'filename'. If the keyword global is given, then all symbols will be visible in the context before the symbol was loaded. The 'offset' (default is 0) is added to each symbol item. The symbol information is loaded into the context of the currently executing code. The format of each line in the symbol file filename is "%x %s". The first value is the address and the second is the symbol name.

BochsがLinuxブートローダの先頭までの実行を直接シミュレートするためには、まずbreakpointコマンドを使って0x7c00にブレークポイントを設定し、システムを0x7c00まで継続して実行させて停止させます。実行されるコマンドの順序は以下の通りです。

<bochs:3> vbreak 0x0000:0x7c00

<bochs:4> c

(0) ブレークポイント1, 0x7c00 (0x0:0x7c00)

次はt=4409138で

(0) [0x00007c00] 0000:7c00 (unk. ctxt): mov ax, 0x7c0 ; b8c007

<bochs:5>

この時点で、CPUはboot.sプログラムの先頭の命令を実行し、Bochsのメインウィンドウには「Boot From floppy...」などの情報が表示されます。あとは、コマンド「s」や「n」を使ってデバッガを追いかけることができます（サブルーチンには追いかけることはできません）。Bochsのブレークポイント設定コマンド、逆アセンブルコマンド、情報表示コマンドなどは、私たちのデバッグ作業を支援するために使用することができます。ここでは、一般的なコマンドの例を紹介します。

/ <bochs:8> u /10 # 10個の命令を分解します。

00007c00: ( ): mov ax, 0x7c0 ; b8c007 00007c03: ( ): mov ds, ax ; 8ed8

00007c05: ( ): mov ax, 0x9000 ; b80090 00007c08: ( ): mov es, ax ; 8ec0

00007c0a: ( ): mov cx, 0x100 ; b90001

00007c0d: ( ): sub si, si ; 29f6

00007c0f: ( ): sub di, di ; 29ff

00007c11: ( ): rep movs word ptr [di], word ptr [si] ; f3a5

00007c13: ( ): jmp 9000:0018 ; ea18000090

00007c18: ( ): mov ax, cs ; 8cc8

<bochs:9> info r # 現在のレジスタの内容を見る eax 0xaa55 43605 ecx 0x110001 1114113 edx 00 0 ebx 00 0 esp 0xfffe 0xfffe ebp 0x0 0 esi 0 0 edi 0xffe4 65508 eip 0x7c00 0x7c00 eflags 0x282 642 cs 0 0 0 ss 0 0 ds 0 0 0 es 0 0 fs 0 0 gs 0 0 0

<bochs:10> print-stack # 現在のスタックを表示します。

0000fffe [0000fffe] 0000

00010000 [00010000] 0000

00010002 [00010002] 0000

00010004 [00010004] 0000

00010006 [00010006] 0000

00010008 [00010008] 0000

0001000a [0001000a] 0000

...

<bochs:11> dump\_cpu # CPUの全レジスタを表示します。

eax:0xaa55 ebx:0x0 ecx:0x110001 edx:0x0 ebp:0x0 esi:0x0 edi:0xffe4 esp:0xfffe eflags:0x282 eip:0x7c00 cs:s=0x0, dl=0xffff, dh=0x9b00, valid=1 # s-selector;dl,dh - descの下位と上位の4バイト。

ss:s=0x0, dl=0xffff, dh=0x9300, valid=7 ds:s=0x0, dl=0xffff, dh=0x9300, valid=1 es:s=0x0, dl=0xffff, dh=0x9300, valid=1 fs: s=0x0, dl=0xffff, dh=0x9300, valid=1 gs:s=0x0, dl=0xffff, dh=0x9300, valid=1 ldtr:s=0x0, dl=0x0, dh=0x0, valid=0 tr: s=0x0, dl=0x0, dh=0x0, valid=0 gdtr:base=0x0, limit=0x0 idtr:base=0x0, limit=0x3ff dr0:0x0 dr1:0x0 dr2:0x0 dr3:0x0 dr6: 0xffff0ff0 dr7:0x400 tr3:0x0 tr4:0x0 tr5:0x0 tr6:0x0 tr7:0x0 cr0:0x60000010 cr1:0x0 cr2:0x0 cr3:0x0 cr4:0x0 inhibit\_mask:0 done

<bochs:12>

Linux 0.1Xカーネルの32ビットコードは絶対物理アドレス0から格納されているので、32ビットコードの先頭（つまりhead.sプログラムの先頭）に直接実行したい場合は、リニアアドレス0x0000にブレークポイントを設定し、「c」コマンドを実行することでその場所まで実行することができます。

また、コマンドプロンプトで直接Enterキーを押すと、前のコマンドが繰り返し実行されます。上矢印を押すと前のコマンドが表示されます。その他のコマンドについては、「help」コマンドを参照してください。

### 17.5.2 Locating Variables or Data Structures in the Kernel

カーネルのコンパイル時に「system.map」ファイルが生成されます。このファイルには、カーネルイメージ（ブートイメージ）ファイル内のグローバル変数と、各モジュール内のローカル変数のオフセットアドレスの位置が記載されています。カーネルがコンパイルされた後、上述のファイルエクスポート方法を使って、「system.map」ファイルをホスト環境（Windows）に展開することができます。「system.map」ファイルの詳しい目的や役割については、第3章を参照してください。サンプルファイル「system.map」の内容の一部を以下に示します。このファイルを使うことで、Bochs debug systemの中で、変数の位置を特定したり、指定した関数コードにジャンプしたりすることができます。

/ ...

グローバルシンボル。

\_dup: 0x16e2c \_nmi: 0x8e08

\_bmapです。0xc364

\_iput: 0xc3b4

\_blk\_dev\_init: 0x10ed0

を開きます。0x16dbc

\_do\_execveです。0xe3d4

\_con\_init: 0x15ccc

\_put\_super: 0xd394

\_sys\_setgid: 0x9b54 \_sys\_umask: 0x9f54

\_con\_write: 0x14f64

\_show\_task: 0x6a54

\_buffer\_init: 0xd1ec

\_sys\_settimeofday: 0x9f4c \_sys\_getgroups: 0x9edc

...

同様に、Linux 0.1Xカーネルの32ビットコードは、絶対物理アドレス0から格納されているので、「system.map」のグローバル変数のオフセット位置は、CPU内のリニアアドレス位置になります。そのため、目的の変数や関数名の位置に直接ブレークポイントを設定し、指定した位置まで連続してプログラムを実行させることができます。例えば、関数buffer\_init()をデバッグしたい場合、「system.map」ファイルから0xd1ecの位置にあることがわかります。この時点で、そこにリニア・アドレス・ブレークポイントを設定し、コマンド「c」を実行すると、以下のように指定した関数の先頭までCPUを実行させることができます。

/ <bochs:12> lb 0xd1ec # リニアアドレスのブレークポイントを設定します。

<bochs:13> c # 連続実行。

(0) ブレークポイント 2, 0xd1ec in ? ()

次はt=16689666で

(0) [0x0000d1ec] 0008:0000d1ec (unk. ctxt): push ebx ; 53

<bochs:14> n # 次の命令です。

次はt=16689667で

(0) [0x0000d1ed] 0008:0000d1ed (unk. ctxt): mov eax, dword ptr ss:[esp+0x8] ; 8b442408

<bochs:15> n # 次の命令です。

次はt=16689668で

(0) [0x0000d1f1] 0008:0000d1f1 (unk. ctxt): mov edx, dword ptr [ds:0x19958] ; 8b1558990100

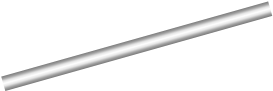
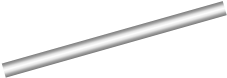
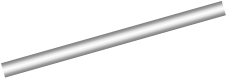
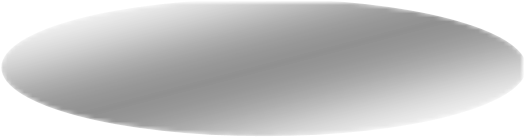
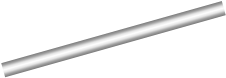
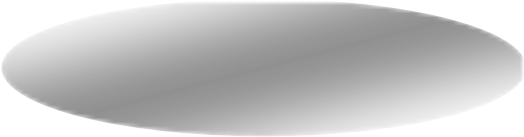
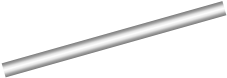
<bochs:16>

プログラムのデバッグは、それを実現するためにはより多くの練習が必要なスキルです。前述の基本的なコマンドのいくつかを組み合わせて、カーネルコードの実行環境全体を柔軟に観察する必要があります。

## 17.6 Creating a Disk Image File

ディスクイメージファイルは、フロッピーディスクやハードディスク上のデータを完全にイメージ化し、ファイルとして保存したものです。ディスクイメージファイルに格納されている情報の形式は、実際のディスクに格納されている情報の形式と全く同じです。空のディスクイメージファイルとは、作成したディスクと同じ容量で、中身がすべて「0」のファイルである。この空のイメージファイルは、購入したばかりの新品のフロッピーディスクやハードディスクのようなもので、使用するためには、パーティションや/を切ってフォーマットする必要があります。

ディスクイメージファイルを作成する前に、まず、作成したイメージファイルの容量を決定する必要があります。フロッピーのイメージファイルの場合、各種仕様の容量（1.2MBや1.44MB）が決まっています。そこで、ここでは、必要なハードディスクのイメージファイルの容量を決める方法を紹介します。従来のハードディスクの構造は、金属製のディスクを積み重ねたものである。各ディスクの上面と下面にはデータが格納されており、表面全体が同心円状の「シリンダー」によってトラックに分割されている。ディスクにデータを読み書きするためには、各ディスクの両面にヘッドが必要である。ディスクが回転すると、ヘッドは半径方向に移動するだけでどのトラックにも移動できるため、ディスク表面のすべての有効な位置にアクセスできる。各トラックはセクタに分割され、セクタサイズは一般的に256〜1024バイトで構成されている。多くのシステムでは、セクタサイズは通常512バイトである。典型的なハードディスクの構造を図17-4に示します。



Top View

1

sector

Track

Head

図17-4 典型的なハードディスクの内部構造

図は、2枚のメタルディスクに4つの物理ヘッドを搭載したハードディスクの構造を示している。円筒の最大収容数は製造時に決定される。ハードディスクの分割・フォーマット時には、ディスク表面の磁気媒体は、各トラック（またはシリンダー）が指定された数のセクタに分割されるように、指定されたフォーマットのデータに初期化される。したがって、このハードディスクの総セクタ数は

総セクタ数 = 物理トラック数 \* 物理ヘッド数 \* トラックあたりのセクタ数

OSで使われるトラックやヘッドなどのパラメータは、論理パラメータと呼ばれるハードディスク内の実際の物理パラメータとは異なります。しかし、これらのパラメータによって算出される総セクタ数は、ハードディスクの物理的パラメータによって算出されるものと間違いなく同じである。PCシステムを設計する際に、ハードウェアデバイスの性能や容量がそれほど急速に開発されていないため、ROM BIOSのハードディスクのパラメータの表現が小さすぎて、実際のハードディスクの物理的なパラメータを満たせないものがあります。そのため、OSやマシンBIOSで一般的に使用されている現在の対処法は、トラック数、ヘッド数、トラックあたりのセクタ数を適切に調整して、互換性とパラメータ表現の制約を確保しつつ、ハードディスクの総セクタ数が一定になるようにしています。ハードディスク・デバイス・パラメーターのBochs設定ファイルの「translation」オプションも、この目的のために設定されています。

Linux 0.1Xシステムのハードディスク・イメージファイルを作成する際、独自のコードが少ないことと、使用するMINIX 1.5ファイルシステムの最大容量が64MBであることを考慮すると、各ハードディスク・パーティションの最大サイズは64MBにしかなりません。また、Linux 0.1Xシステムでは拡張パーティションがまだサポートされていないため、ハードディスクのImageファイルでは、最大4つのパーティションが存在します。したがって、Linux 0.1Xシステムで使用可能なハードディスクイメージファイルの最大容量は、64×4＝256MBとなります。以下の説明では、4つのパーティションを持ち、1つのパーティションにつき60MBのハードディスク・イメージ・ファイルを作成する例を示します。

### USBフラッシュディスクの場合は、ハードディスクとして扱うことができます。フロッピーディスクの場合は、トラック数（シリンダー数）、ヘッド数、トラックあたりのセクタ数が決まっている、分割されていない超小型ハードディスクと考えることができます。例えば、容量1.44MBのフロッピーディスクは、1トラック80本、1ヘッド2セクタ18個、1セクタ512バイトである。総セクタ数は2880で、総容量は80×2×18×512＝1474560バイトとなります。したがって、以下に説明するハードディスクのイメージファイルの作成方法は、すべてUSBディスクやフロッピーのイメージファイルの作成にも利用できます。説明の便宜上，特に断りのない限り，すべてのディスクイメージファイルをイメージファイルと呼びます。

### 17.6.1 Using Bochs' own Image Creation Tool

Bochsシステムには、ディスクイメージ作成ツール「bximage.exe」が付属しており、これを使ってフロッピーディスクやハードディスク用の空のイメージファイルを作成することができます。bximage.exeを実行すると、イメージ作成画面が表示され、作成するイメージの種類（ハードディスクhdまたはフロッピーディスクfd）を選択する画面が表示されます。ハードディスクを作成する場合は、ハードディスクイメージのモードタイプを入力するように促されますが、通常はデフォルト値の「フラット」を選択するだけです。次に、作成するイメージのサイズを入力します。プログラムは、対応するハードディスクのパラメータ値（シリンダー数（トラック数）、ヘッド数、トラックあたりのセクタ数）を表示し、イメージファイルの名前を尋ねます。イメージファイルが生成された後、プログラムはBochs設定ファイルにハードディスクのパラメータを設定するための設定メッセージを表示します。この情報をメモして、設定ファイルに編集することができます。以下は、256MBのハードディスクのImageファイルを作成する手順です。

======================================================================== bximage

Bochs用ディスクイメージ作成ツール

$Id: bximage.c,v 1.19 2006/06/16 07:29:33 vruppert Exp $.

========================================================================

フロッピーディスクのイメージを作りたいのか、ハードディスクのイメージを作りたいのか。

hd または fd と入力してください。[hd]

どのようなイメージを描けばいいのか？

flat、sparse、growthのいずれかを選択してください。[flat] です。

ハードディスクのサイズを1～32255の間でメガバイト単位で入力する

[10] 256

cyl=520 heads=16の「フラット」なハードディスク・イメージを作成します。

トラックあたりのセクタ数=63 合計セクタ数=524160 合計サイズ=255.94メガバイト

画像の名前はどうすればいいですか？[c.img] hdc.img

書き方。[] Done.

(null)に268369920バイト書き込みました。

bochsrcに次のような行が現れるはずです。

ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=520, heads=16, spt=63

いずれかのキーを押して続行

### すでに必要な容量のハードディスクのイメージファイルがある場合は、そのファイルを直接コピーして別のイメージファイルを作成し、必要に応じて加工することもできます。フロッピーディスクのイメージファイルを作成するプロセスは、フロッピーディスクの種類を選択するプロンプトが表示されることを除いて、上記と同様です。同様に、すでに他のフロッピーイメージファイルを持っている場合は、直接コピーする方法を使うことができます。

### 17.6.2 Creating an Image File Using the dd Command on a Linux

ddコマンドは、Linuxシステムのコマンドラインツールで、主にファイルのコピーやファイルのデータ形式の変換に使用します。上記で説明したように、先ほど作成したImageファイルは、すべてのコンテンツが0の空のファイルですが、その容量は必要条件と一致しています。そこで、まず、容量を必要とするImageファイルのセクタ数を算出し、「dd」コマンドを使用して対応するImageファイルを生成します。

例えば、1トラックのシリンダー数が520、ヘッド数が16、セクタ数が63のハードディスクのイメージファイルを作成したいとします。総セクタ数は 520 \* 16 \* 63 = 524160 とすると、コマンドは次のようになります。

dd if=/dev/zero of=hdc.img bs=512 count=524160

パラメータの'if'はコピーされた入力ファイルの名前、'/dev/zero'は0値バイトを生成できるデバイスファイル、'of'は生成された出力ファイル名、'bs'はコピーされたデータブロックのサイズ、'count'はコピーされたデータブロックの数を指定します。1.44MBのフロッピーイメージファイルの場合、セクタ数は2880なので、コマンドは次のようになります。

dd if=/dev/zero of=diska.img bs=512 count=2880

### 17.6.3 Creating a floppy disk image file in DOS format using WinImage

1. WinImage は、DOS フォーマットの Image ファイルへのアクセスおよび作成ツールです。ソフトウェアとの関連付けを行った後、DOSフロッピーのImageファイルのアイコンをダブルクリックすると、そのファイルを閲覧したり、削除したり、追加したりすることができます。さらに、CDROMのisoファイルの内容を閲覧することもできます。WinImageを使ってフロッピーディスクのイメージを作成する場合、DOS形式のイメージファイルを生成することができます。以下のような方法があります。
2. Run WinImage. Select the "Options->Settings" menu and select the Image Settings page. Set
3. 圧縮を "なし "にする（つまり、インジケータを左端まで引く）。
4. Create an Image file. Select the menu "File->New" and a floppy disk selection box will pop up.
5. 1.44MBの容量のフォーマットをお選びください。
6. Select the boot sector property menu item "Image->Boot Sector properties" and click the MS-DOS button in the dialog box;
7. Save the file.

* なお、「保存の種類」ダイアログボックスで「すべてのファイル（\*.\*）」を選択する必要があります。そうしないと、作成されたイメージファイルにWinImageの情報が含まれてしまい、Bochs上でイメージファイルが正常に動作しなくなります。作成されたイメージが要件を満たしているかどうかは、ファイルサイズを見ればわかります。標準的な1.44MBのフロッピーディスクの場合、1474560バイトの容量が必要です。新しいImageのファイルサイズがこの値より大きい場合は、再作成するか、Notepad++などのバイナリエディタ（Hex-Editorプラグインが必要）を使って余分なバイトを削除してください。削除の方法は以下の通りです。
* Open the Image file with Notepad++ and run the plugin Hex-Editor. According to the 511, 512 bytes of the disk image file are 55, AA two hexadecimal numbers, we push back 512 bytes, delete all the previous bytes. At this point, for a disk using MSDOS 5.0 as the boot, the first few bytes of the file should be similar to "EB 3C 90 4D ...".
* Then pull down the right scroll bar and move to the end of the img file. Delete all data after "...F6 F6 F6". Usually it is to delete all data starting from 0x168000. The last line when the operation is completed should be a complete line "F6 F6 F6...". Save and exit to use the Image file.

## 17.7 Making a Root File System

ここでは、ハードディスクにルートファイルシステムを作成することが目的です。フロッピーディスクとハードディスクのルートファイルシステムのImageファイルはoldlinux.orgのウェブサイトからダウンロードできますが、ここでは参考のために作成手順を詳しく説明します。作成の過程では、Linus氏が書いたインストール記事：INSTALL-0.11も参考にしてください。ルートファイルシステムのディスクを作成する前に、まず、rootimage-0.12とbootimage-0.12のイメージファイルをダウンロードします（最新のファイルをダウンロードしてください）。

http://oldlinux.org/Linux.old/images/bootimage-0.12-20040306 http://oldlinux.org/Linux.old/images/rootimage-0.12-20040306

この2つのファイル名を覚えやすいboot-image-0.12とroot-image-0.12に変更し、Linux-0.12というサブディレクトリを作成してください。作成過程では、rootimage-0.12のフロッピーディスクにある実行ファイルをコピーし、bootimage-0.12の起動ディスクを使ってシミュレーションシステムを起動する必要があります。そのため、ルートファイルシステムの作成に取り掛かる前に、まず、これら2つのフロッピーイメージファイルからなる最小のLinuxシステムが実行できることを確認する必要があります。

### 17.7.1 Root File System and Root File Device

Linuxの起動時には、ルートディレクトリを含むデフォルトのファイルシステムがルートファイルシステムとなります。一般的にルートディレクトリには、以下のサブディレクトリやファイルが含まれています。

* etc/ This directory mainly contains some configuration files, such as 'rc' file;
* dev/ Contains device special files for operating the device with files;
* bin/ Store system execution programs. Such as sh, mkfs, fdisk, etc.;
* usr/ Store library functions, manuals, and other files;
* usr/bin/ Store commands commonly used by users;
* var/ Used to store data when the system is running or for information such as logs.

ファイルシステムを保持している装置がファイルシステム装置である。例えば、Windowsの場合、ハードディスクのCドライブがファイルシステムデバイスであり、ハードディスクに一定の規則に従って格納されたファイルがファイルシステムを構成している。WindowsはNTFSやFAT32などの形式のファイルシステムを持つことができるが、Linux 0.1XのカーネルがサポートしているファイルシステムはMINIX 1.0のファイルシステムである。

Linuxの起動ディスクがルートファイルシステムをロードする際には，起動ディスクのブートセクタの509バイト目と510バイト目にあるワード（ROOT\_DEV）のルートファイルシステムのデバイス番号に従って，指定されたデバイスからルートファイルシステムがロードされます。デバイス番号が0の場合、ルートファイルシステムは、起動ディスクがある現在のドライブからロードされる必要があることを意味します。デバイス番号がハードディスク・パーティション・デバイス番号の場合、ルート・ファイル・システムは指定されたハードディスク・パーティションからロードされます。Linux 0.1X カーネルでサポートされているハードディスクのデバイス番号を表17-3に示します。

|  |  |  |
| --- | --- | --- |
| 表17-3 ハードディスクの論理デバイス番号  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 |

デバイス番号がフロッピーデバイス番号の場合、カーネルはそのデバイス番号で指定されたフロッピードライブからルートファイルシステムをロードする。Linux 0.1Xのカーネルで使用されているフロッピーデバイス番号を表17-4に示します。フロッピーディスクドライブのデバイス番号の計算方法については，第9章のfloppy.cプログラム以降の記述を参照してください．

|  |  |  |
| --- | --- | --- |
| 表17-4 フロッピーディスクドライブのロジックデバイス番号  Device Nr | Device file | Description |
| 0x0208 | /dev/at0 | 1.2MB A drive |
| 0x0209 | /dev/at1 | 1.2MB B drive |
| 0x021c | /dev/fd0 | 1.44MB A drive |
| 0x021d | /dev/fd1 | 1.44MB B drive |

### 17.7.2 Creating a File System

1. 上記で作成したハードディスクのイメージファイルは、使用する前にパーティションを切り、ファイルシステムを作成する必要があります。通常は、処理するハードディスクイメージファイルをBochsの下にある既存のシミュレーションシステム（前述のSLS Linuxなど）にアタッチし、シミュレーションシステムのコマンドを使用して新しいイメージファイルを処理することになります。以下では、SLS Linuxエミュレーションシステムがインストールされており、SLS-Linuxというサブディレクトリに格納されていることを前提としています。これを使って、上記で作成した256MBのハードディスクイメージファイルhdc.imgをパーティショニングし、その上にMINIXファイルシステムを作成します。このイメージファイルにパーティションを作成し、MINIXファイルシステムを作成します。実行した手順は以下の通りです。
2. Create a subdirectory named Linux-0.12 in the SLS-Linux directory and move the hdc.img file to it.
3. Go to the SLS-Linux directory and edit the Bochs configuration file 'bochsrc.bxrc' for the SLS Linux system. Add the configuration parameter line of our hard disk Image file under the option 'ata0-master':

ata0-slave:type=disk, path=...linux-0.12hdc.img, cylinders=520, heads=16, spt=63

1. Exit the editor. Double-click the icon for the 'bochsrc.bxrc' file to run the SLS Linux emulation system. Type 'root' at the Login prompt and press Enter. If Bochs does not work properly at this time, generally because the configuration file information is incorrect, please re-edit the configuration file.
2. Use fdisk to create 1 partition in the hdc.img file. Below is the sequence of commands to create the first partition. The process of creating another three partitions is similar. Since the partition type established by SLS Linux by default is 81 type (Linux/MINIX) that supports the MINIX2.0 file system, you need to use the fdisk t command to change the type to 80 (Old MINIX) type. Please note here that we have hooked hdc.img to the second hard drive under the SLS Linux system. According to the Linux 0.1X hard disk naming rules, the overall device name of the hard disk should be /dev/hd5. However, since the Linux kernel version 0.95, the naming rules for the hard disk have been changed to the currently used rules, so the device name of the second hard disk under SLS Linux is /dev/hdb.

[/]# fdisk /dev/hdb

コマンド (m for help): n コマンドアクション e extended p プライマリパーティション (1-4) p

パーティション番号（1-4）：1

第1シリンダー（1～520）：1

最後のシリンダーまたは+サイズまたは+サイズMまたは+サイズK（1-520）。+63M

コマンド（mはヘルプ）：t

パーティション番号（1-4）：1

1. ヘックスコード（Lを入力するとコードが一覧表示されます）。L
2. Empty 8 AIX 75 PC/IX b8 BSDI swap
3. DOS 12-bit FAT 9 AIX bootable 80 Old MINIX c7 Syrinx
4. XENIX root a OPUS 81 Linux/MINIX db CP/M
5. XENIX user 40 Venix 82 Linux swap e1 DOS access
6. DOS 16-bit <32M 51 Novell? 83 Linux extfs e3 DOS R/O
7. Extended 52 Microport 93 Amoeba f2 DOS secondary
8. DOS 16-bit >=32 63 GNU HURD 94 Amoeba BBT ff BBT
9. OS/2 HPFS 64 Novell b7 BSDI fs

16進コード（タイプL～リストコード）：80

コマンド（mはヘルプ）：p

ディスク /dev/hdb: 16 ヘッド、63 セクタ、520 シリンダ

単位：1008×512バイトのシリンダー

デバイス ブート開始 開始 終了 ブロック ID システム

/dev/hdb1 1 1 129 65015+ 80 古い MINIX

コマンド（mはヘルプ）：w

パーティションテーブルが変更されています。

再起動してからご利用ください。

[/]#

1. Remember the number of data blocks in this partition (here is 65015), which is used when creating the file system. When the partition is set up, it is necessary to restart the system as usual, so that the SLS Linux kernel can correctly identify the newly added partition.
2. After entering the SLS Linux emulation system again, we use the mkfs command to create the MINIX file system on the first partition we just created. The commands and information are as follows. This creates a partition with 64,000 data blocks (one block of data is 1 KB).

[/]# mkfs /dev/hdb1 64000

21333 inodes

64000ブロック

ファーストダットゾーン＝680（680）

Zonesize=1024

最大サイズ=268966912

[/]#

この時点で、hdc.imgファイルの第1パーティションにファイルシステムを作成する作業が完了しました。もちろん、ファイルシステムの作成は、Linux 0.12のフロッピーディスクでルートファイルシステムを実行する際にも成立します。今度は、このパーティションをルートファイルシステムに構築します。

### 17.7.3 Bochs Configuration File for Linux-0.12

BochsでLinux 0.12システムを運用する場合、通常は設定ファイルbochsrc.bxrcに以下の設定が必要となります。

/ romimage: file=$BXSHARE/BIOS-bochs-latest vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest megs: 16 floppya: 1\_44="bootimage-0.12", status=inserted ata0-master: type=disk, path="hdc. img", mode=flat, cylinders=520, heads=16, spt=63 boot: a log: bochsout.txt panic: action=ask #error: action=report #info: action=report #debug: action=ignore ips: 1000000 マウス: enabled=0

SLS LinuxのBochs設定ファイルbochsrc.bxrcをLinux-0.12のディレクトリにコピーして、上記と同じ内容に修正すればよい。特に注意すべきは、「floppya」、「ata0-master」、「boot」である。この3つのパラメータは、上記の内容と一致していなければなりません。では、この設定ファイルをマウスでダブルクリックしてみます。まず、Bochsの表示ウィンドウに図17-5のような画面が表示されるはずだ。

/ 図 17-5 Bochs システムの実行ウィンドウ

この時点で、ウィンドウのメニューバーにあるA:フロッピーディスクのアイコンをクリックし、ダイアログボックスでAディスクをrootimage-0.12のファイルとして設定してください。または、メニューバーの「CONFIG」アイコンをクリックして、Bochsの設定ウィンドウを使って設定します（ウィンドウを前面に出すには、マウスをクリックする必要があります）。設定ウィンドウの表示内容を図17-6に示します。

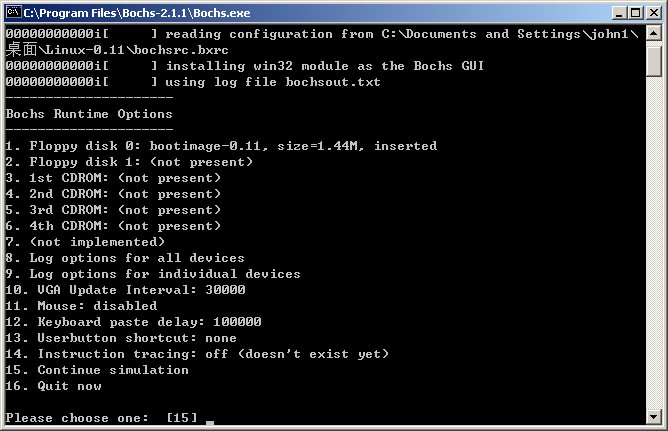


図 17-6 Bochs システム・コンフィギュレーション・ウィンドウ

項目1のフロッピーディスクの設定を変更して、rootimage-0.12のディスクを指すようにします。その後、セットアップウィンドウの最終行に「Continuing simulation」と表示されるまで、Enterキーを押し続けます。この時点でBochs Runウィンドウに切り替え、Enterをクリックすると、図17-7に示すように、Linux 0.12システムに正式に入ります。

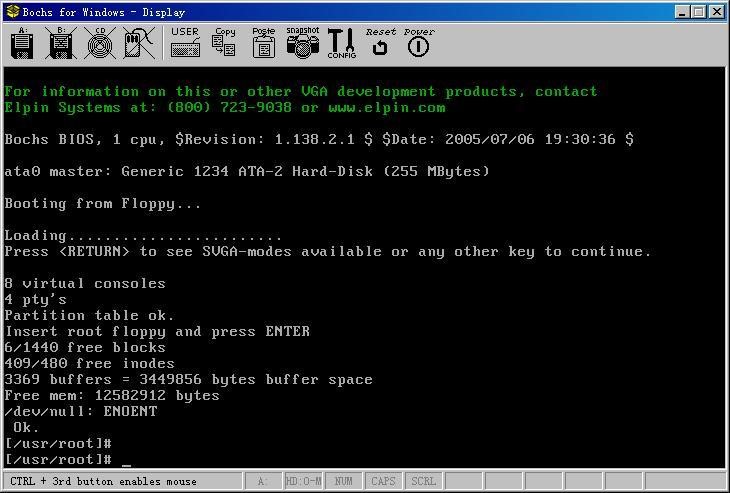


図17-7 Bochsで動作するLinux 0.12システム

### 17.7.4 Establishing a Root File System in hdc.img

フロッピーディスクの容量が小さすぎるため、Linux 0.12系で本当に何かをしたいのであれば、ハードディスクにルートファイルシステム（ここでは、ハードディスクイメージファイル）を作成する必要があります。前節で、256MBのハードディスク・イメージ・ファイルhdc.imgを作成しましたが、すでに実行中のBochs環境に接続されているので、上図のようにハードディスクに関するメッセージが表示されます。

“ata0 master: Generic 1234 ATA

-

2

Hard

-

Disk (255 Mbytes)”

このメッセージが表示されない場合は、Linux 0.12の設定ファイルが正しく設定されていません。bochsrc.bxrcファイルを再編集して、上記と同じ画面が表示されるまでBochsシステムを再実行してください。さて、以前、hdc.imgの第一パーティションにMINIXファイルシステムを作成しました。まだ構築していない場合や、もう一度やってみたい場合は、64MBのファイルシステムを作成するコマンドを入力してください。

[

/usr/root]#

mkfs /dev/hd1 64000

これで、ハードディスクへのファイルシステムのロードを開始できます。以下のコマンドを実行して、新しいファイルシステムを/mntディレクトリにロードします。

/ [/usr/root]# cd /

[/]# mount /dev/hd1 /mnt

[/]#

ハードディスクのパーティションにファイルシステムを読み込んだ後、フロッピーディスクのルートファイルシステムをハードディスクにコピーします。以下のコマンドを実行してください。

[/]# cd /mnt

[/mnt]# for i in bin dev etc usr tmp

> do

> cp +recursive +verbose /$i $i

> done

この時点で、フロッピーのルートファイルシステムにあるすべてのファイルが、ハードディスクのファイルシステムにコピーされます。コピーの過程では、以下のような多くの情報が表示されます。

/usr/bin/mv -> usr/bin/mv

/usr/bin/rm -> usr/bin/rm

/usr/bin/rmdir -> usr/bin/rmdir

/usr/bin/tail -> usr/bin/tail

/usr/bin/more -> usr/bin/more

/usr/local -> usr/local

/usr/root -> usr/root

/usr/root/.bash\_history -> usr/root/.bash\_history

/usr/root/a.out -> usr/root/a.out

/usr/root/hello.c -> usr/root/hello.c

/tmp -> tmp

[/mnt]#

これでハードディスクに基本的なルートファイルシステムが構築できました。新しいファイルシステムのどこでも見ることができます。その後、ハードディスクのファイルシステムをアンマウントし、「logout」または「exit」と入力してLinux 0.12システムを終了してください。以下のようなメッセージが表示されます。

/ [/mnt]# cd /

[/]# umount /dev/hd1

[/]# logout

子供4人がコード0000で死亡

[/usr/root]# \_。

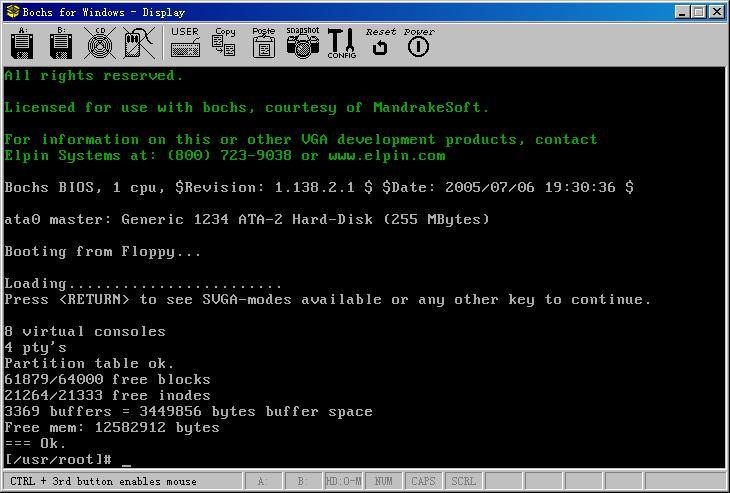
### 17.7.5 Using the Root File System on the Hard Disk Image

1. ハードディスクのイメージファイルにファイルシステムを作成したら、Linux 0.12がそのファイルシステムをルートファイルシステムとして起動することができます。これは、bootimage-0.12ファイルの509、510バイト目(0x1fc, 0x1fd)の内容を変更することで可能になります。以下の手順で作業を行ってください。
2. First copy the two files bootimage-0.12 and bochsrc.bxrc to generate the bootimage-0.12-hd and bochsrc-hd.bxrc files.
3. 2. Edit the bochsrc-hd.bxrc configuration file, change the file name on the 'floppya:' option to 'bootimage-0.12-hd', and save it;
4. 3. Edit the bootimage-0.12-hd binary with Notepad++ or any other binary editor and modify the 509th and 510th bytes (ie 0x1fc, 0x1fd). The original value should be 00, 00, modified to 01, 03, indicating that the root file system device is on the first partition of the hard disk Image, and then save and exit. If you have the file system installed on another partition, you will need to modify the first byte to correspond to your partition.

000001f0h: 00 00 00 00 00 00 00 00 00 01 03 55 aa ; ..............u?

これで、bochsrc-hd.bxrcファイルのアイコンをダブルクリックすることができる。BochsシステムはすぐにLinux 0.12システムに入り、図17-8のようなグラフィックを表示するはずです。

17.8 Linux 0.12 システムでのカーネルのコンパイル



## 図17-8 ハードディスク・イメージ・ファイルのファイル・システムの使用

## 17.8 Compile Kernel on Linux 0.12 System

筆者は、Linux 0.12システムパッケージをgcc 1.40のビルド環境で再構成しました。このシステムは、Bochsシミュレーション・システムの下で動作するように設定されており、対応するbochs設定ファイルが設定されています。このパッケージは、以下のアドレスから入手できます。

http://oldlinux.org/Linux.old/bochs/linux

-

0.1

2

-

0

8

0

324

.zip

このパッケージには、パッケージ内のすべてのファイルの目的と使用方法を説明したREADMEファイルが含まれています。あなたのシステムにbochsシステムがインストールされていれば、設定ファイルbochsrc-0.12-hd.bxrcのアイコンをダブルクリックするだけで、このLinux 0.12を実行することができます。このシステムでは、ハードディスク・イメージ・ファイルをルート・ファイル・システムとして使用します。システムを起動した後、/usr/src/linuxディレクトリでmakeコマンドを入力すると、Linux 0.12のカーネルソースコードがコンパイルされ、ブートイメージファイル「Image」が生成されます。このImageファイルを出力する必要がある場合は、まずbootimage-0.12-hdファイルをバックアップし、次のコマンドでbootimage-0.12-hdを新しいブートファイルに置き換えてください。これでBochsを直接再起動すると、新しいコンパイラで生成されたbootimage-0.12-hdを使ってシステムを起動することができます。

[/usr/src/linux]# make

[/usr/src/linux]# dd bs=8192 if=Image of=/dev/fd0

[/usr/src/linux]#

Redhatシステムでのカーネルのコンパイル

また、mtoolsコマンドを使用して、新たに生成されたImageファイルを2番目のフロッピーイメージファイル「diskb.img」に書き込み、ツールソフト「WinImage」を使用して「diskb.img」内の「Image」ファイルを抽出することもできます。

/ [/usr/src/linux]# mdir a: おそらくMSDOS以外のディスク mdir: A: "を初期化できません

[/usr/src/linux]# mcopy 画像bです。

IMAGEのコピー

[/usr/src/linux]# mcopy System.map b: Copying SYSTEM.MAP [/usr/src/linux]# mdir b:

ドライブBのボリュームはBです。

B:/のディレクトリ

GCCLIB-1 TAZ 934577 3-29-104 7:49p

IMAGE 121344 4-29-104 11:46p

システムマップ 17162 4-29-104 11:47p

README 764 3-29-104 8:03p

4 ファイル（複数） 382976 バイトの空き容量

[/usr/src/linux]#

フロッピーディスク上のルートファイルシステムrootimage-0.12を持つ新しいブートイメージファイルを使用したい場合は、まずコンパイル前にMakefileを編集し、'ROOT\_DEV='の行を'#'でコメントアウトしてください。

## 通常、カーネルをコンパイルする際には非常にスムーズに行われます。考えられる問題は、gcc コンパイラが「-mstring-ins」というオプションを認識しないことです。このオプションは、Linus が gcc 1.40 コンパイラを単体でコンパイルする際に実装した拡張実験パラメータです。このオプションは、文字列命令を生成する際に gcc を最適化するために使用されます。この問題を解決するには、すべてのMakefileでこのオプションを直接削除し、カーネルを再コンパイルします。また、「gar」コマンドが見つからないという問題もあります。この場合、/usr/local/bin/ の下にある 'ar' を直接リンクするか、 'gar' にコピー/リネームしてください。

## 17.9 Compile kernel under Redhat system

オリジナルのLinuxオペレーティングシステムのカーネルは、Minix 1.5.10オペレーティングシステムの拡張版であるMinix-i386上でクロスコンパイルされ開発された。Minix 1.5.10オペレーティングシステムは、A.S.Tanenbaumの「Design and Implementation of Minix」の初版とともにPrentice Hall Publishing Companyからリリースされました。このバージョンのMinixは、80386およびその互換性のあるマイクロコンピュータ上で動作可能ですが、80386の32ビット保護メカニズムを利用していません。このシステムで32ビットのオペレーティングシステムを開発するために、リーナスはブルース・エバンスのパッチを使ってMINIX-386にアップグレードし、GNUシリーズの開発ツールであるgcc、gld、emacs、bashなどをMinix-386に移植しました。このプラットフォームで、リーナスはバージョン0.01、0.03、0.11、0.12のカーネルをクロスコンパイルして開発した。Linuxメーリングリストの記事によると、筆者は当時のリーナス氏と同様の開発プラットフォームを構築し、初期バージョンのLinuxカーネルのコンパイルに成功したという。

しかし、Minix 1.5.10は古く、開発プラットフォームも非常に面倒なので、ここでは、Linux 0.12カーネルのソースコードを、現在のRedHatシステムのコンパイル環境でコンパイルできるように修正し、実行可能なブートイメージファイルbootimageを生成する方法を簡単に紹介します。読者は、通常のPCでも、Bochsなどの仮想マシンソフトウェアでも実行できます。ここでは、主な修正点のみを示します。すべての修正点は、ツールdiffを使って、修正されたコードと修正されていないコードを比較して、次のことがわかります。

Redhatシステムでのカーネルのコンパイル

という違いがあります。修正されていないコードがlinux/ディレクトリにあり、修正されたコードがlinux-mdf/にある場合、以下のコマンドを実行する必要があります。

diff

-

r linux linux

-

mdf > dif.out

ファイル「dif.out」には、ソースコードの修正箇所がすべて記載されています。RedHat 9でコンパイルできるように修正されたLinux 0.1Xカーネルのソースコードは、以下のアドレスからダウンロードできます。

/ http://oldlinux.org/Linux.old/kernel/linux-0.11-040327-rh9.tar.gz http://oldlinux.org/Linux.old/kernel/linux-0.11-040327-rh9.diff.gz http://oldlinux.org/Linux.old/kernel/linux-0.11-060617-gcc4-diff.gz http://oldlinux.org/Linux.old/kernel/linux-0.11-060618-gcc4.tar.gz http://oldlinux.org/Linux.old/kernel/linux-0.12-080328-gcc4-diff.gz http://oldlinux.org/Linux.old/kernel/linux-0.12-080328-gcc4.tar.gz

コンパイルされたブートイメージファイルで起動すると、以下の情報が画面に表示されます。

/ フロッピーからの起動...

ローディングシステム ...

ルートフロッピーを挿入し、ENTERを押す

### なお、「Loding system...」を表示しても何の反応もない場合は、カーネルがコンピュータのハードディスク・コントローラ・サブシステムを認識していないことを意味します。この時点で、VirtualBox、VMware、bochsなどの仮想マシンソフトを使ってテストすることができます。ルートファイルシステムのディスクを入れるように言われたときに、直接Enterキーを押してしまうと、ルートファイルシステムをロードできないという以下の情報が表示され、クラッシュしてしまいます。Linux 0.1XのOSを完全に動作させるには、マッチしたルートファイルシステムが必要で、これはoldlinux.orgのウェブサイトからダウンロードできます。

### 17.9.1 Modifying the Makefile

1. Linux 0.1Xのカーネルソースディレクトリには、ほぼすべてのサブディレクトリにMakefileが含まれており、これを以下のように修正する必要があります。
2. Rename 'gas' to 'as', 'gld' to 'ld'. Because now 'gas' and 'gld' have been directly renamed to 'as' and 'ld'.
3. 'as' (original gas) has no need to use the '-c' option, so the -c compilation option needs to be removed from the Makefile in the kernel home directory Linux.
4. c. Remove gcc's compile flag options: '-fcombine-regs', '-mstring-insns', and these two options in the Makefile in all subdirectories. The '-fcombine-regs' option was not found in the 1994 gcc manual, and '-mstring-insns' is an option that Linus added to gcc modifications, so this optimization option is definitely not included in your gcc.
5. In the gcc compile option, add the '-m386' option. In this way, the kernel image file compiled under RedHat 9 will not contain the instructions of the CPU of 80486 and above, so the kernel can run on the 80386 machine.

### Redhatシステムでのカーネルのコンパイル

### 17.9.2 Modifying Comments in the Assembly Language Programs

### as86コンパイラは、cコメント文を認識しないので、boot/bootsect.sファイルの中で、'!'を使ってCコメントをコメントアウトする必要があります。

### 17.9.3 Modifying the align value of the memory alignment statement

ブートディレクトリにある3つのアセンブリ言語プログラムでは、'align'ステートメントの使い方が変わっています。オリジナルの'align'の後の値は、メモリ位置の累乗値を参照していますが、現在では、整数アドレスの値を直接与える必要があります。そのため、オリジナルのステートメントである

.align

3

(2^3=8)に修正する必要があります。

.align 8

### 17.9.4 Modifying Inline Macro Assembly Language Programs

Asアセンブラの継続的な改良により、現在では自動化が進んでいるため、変数に使用するCPUレジスタを手動で指定する必要はありません。そのため、カーネルコード内のすべての「\_\_asm\_\_("ax")」を削除する必要があります。例えば、fs/bitmap.cファイルの20行目と26行目、fs/namei.cファイルの68行目。

また、インラインアセンブリコードでは、レジスタ（変更されるレジスタ）の内容に対して無効な宣言をすべて削除する必要があります。例えば、include/string.hの84行目。

:

"si","di","ax","cx");

すべてのレジスターを削除し、コロンと右括弧「::);」だけを残す必要があります。

### この修正にはいくつかの問題があるときがあります。gccは上記の記述にしたがってプログラムを最適化することがあるため、場所によっては、修正されるレジスタの内容を削除するとgccの最適化エラーが発生します。そのため、プログラムコードの中には、include/string.hファイルのmemcpy()定義の342行目のように、状況に応じてこれらの宣言の一部を残しておく必要がある場所があります。

### 17.9.5 Reference representation of C variables in assembly statements

Linux 0.1X カーネルの開発に使用されているアセンブラでは、C 変数を参照する際に、変数名にアンダースコア文字「\_」を追加する必要があります。現在の gcc コンパイラは、これらのアセンブリで参照される C 変数を直接認識できるため、アセンブラ内のすべての C 変数の前にアンダースコアを削除します（埋め込みアセンブリステートメントを含む）。例えば、boot/head.sプログラムの15行目にあるステートメントです。

.globl \_idt,\_gdt,\_pg\_dir,\_tmp\_floppy\_area

Redhatシステムでのカーネルのコンパイル

に変更する必要があります。

.globl idt,gdt,pg\_dir,tmp\_floppy\_area

31行目のステートメントの変数名「\_stack\_start」を「stack\_start」に修正する必要があります。

### 17.9.6 Debug Display Function in Protected Mode

プロテクトモードに入る前は、ROM BIOSのint 0x10コールを使って画面に情報を表示することができます。しかし、プロテクトモードに入った後は、これらの割り込みコールは使用できません。プロテクトモード環境でのカーネルの内部データ構造や状態を把握するためには、以下の関数check\_data32()を使ってカーネルデータを表示することができます（以前、oldlinux.orgフォーラムの友人「notrump」が提供してくれました）。カーネルにはprintk()という表示関数がありますが、これはtty\_write()を呼び出す必要があり、カーネルが完全に機能していないときには利用できません。

プロテクトモードに入った後、このcheck\_data32()関数は、興味のあるものを画面に表示することができます。ページ機能を有効にしてもしなくても、使用には影響しない。なぜなら、4Mの仮想メモリは最初のページテーブルディレクトリエントリを使うだけで、ページテーブルディレクトリは物理アドレス0から始まり、さらにカーネルデータセグメントのベースアドレスは0なので、4Mの範囲では、仮想メモリ、リニアメモリ、物理メモリのアドレスは同じになるからだ。リーナス氏はそもそもこのことを考慮して、この設定の方が使い勝手が良いと感じているのだろう。

/\*

* Purpose: Display a 32-bit integer in hexadecimal on the screen.
* Params: value -- the integer to display.
* pos -- The screen position, in units of 16 chars wide, for example, 2, which means
* that the display starts at the width of 32 chars from the upper left corner.
* Return: None.
* If you want to use it in an assembly language program, make sure that the function is compiled \* and linked into the kernel. The usage in gcc assembly is as follows:
* pushl pos // 'pos' should be replaced with your actual data, such as pushl $4 \* pushl value // 'pos' and 'value' can be any legal addressing method. \* call check\_data32

\*/

inline void check\_data32(int value, int pos)

{

\_\_asm\_\_ \_\_volatile\_\_( // %0 -the value to be displayed; EBX -screen position pos. "shl $4, %%ebx\n\t" // Multiply the pos by 16, plus VGA memory start address, "addl $0xb8000, %%ebx\n\t" // get the position from top left of the screen in EBX.

"movl $0xf0000000, %%eax\n\t" // 4 ビットのマスクを設定します。

"movb $28, %%cl\n" // 右シフトの初期ビット値を設定します。

|  |  |
| --- | --- |
| "1:n\\"  "movl | %0, %%edx\n\t" // Put the displayed value to EDX. |
| "andl | %%eax, %%edx\n\t" // Take 4 bits specified by EAX in EDX. |
| "shr | %%cl, %%edx\n\t" // Shift 28 bits to right, EDX is the value of 4 bits taken. |
| "add | $0x30, %%dx\n\t" // Convert this value to ASCII code. |
| "cmp | $0x3a, %%dx\n\t" // If less than 10, jumps forward to label 2. |

"jb2f\\"

"add $0x07, %%dx\n" // Otherwise add 7 and convert the value to A-F.

|  |  |
| --- | --- |
| "2:n\\"  "add | $0x0c00, %%dx\n\t" // Set the display attributes. |
| "movw | %%dx, (%%ebx)\n\t" // Put this value in the display memory. |
| "sub | $0x04, %%cl\n\t" // Prepare the next hex number, the number of shifts minus 4. |
| "shr | $0x04, %%eax\n\t" // The mask is shifted to the right by 4 bits. |
| "add | $0x02, %%ebx\n\t" // Update the display memory position. |
| "cmpl | $0x0, %%eax\n\t" // The mask has moved out of the right (8 hexs displayed)? |

"jnz1b\n" // いいえ、まだ表示すべき数字があるので、テーブル1にジャンプします。 :: "m"(value), "b"(pos))

}

## 17.10 Integrated Boot Disk and Root FS

ここでは、カーネルブートイメージファイルとルートファイルシステムを組み合わせた統合ディスクイメージファイルの作成方法を説明します。その主な目的は、Linux 0.1Xカーネルメモリ仮想ディスクの動作原理を理解し、起動ディスクとルートファイルシステムのディスクの概念をさらに理解し、kernel/blk\_drv/ramdisk.cプログラムの実行方法の理解を深めることにあります。実際、一般的な組み込みシステムでFlashに格納されているブートモジュール、カーネルモジュール、ファイルシステムモジュールのイメージ構造は、ここでの統合ディスクと同様です。

以下では、Linux 0.11カーネルを使った統合ディスクの作成プロセスを例に挙げて説明します。読者は練習として、0.12カーネルを使って同様の統合ディスクを実装する。この統合ディスクを作成する前に、まず以下の実験用ソフトウェアをダウンロードまたは用意する必要がある（後者2つは0.12カーネル統合ディスクの構築に使用する）。

http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-040923.zip http://oldlinux.org/Linux.old/images/rootimage-0.11-for-orig http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip http://oldlinux.org/Linux.old/images/rootimage-0.12-20040306

linux-0.11-devel」は、Bochsで動作する開発環境付きのLinux 0.11システムです。rootimage-0.11」は、1.44MBのフロッピーイメージファイルのLinux 0.11ルートファイルシステムです。接尾辞「for-orig」は、未修正のLinux 0.11カーネルソースコード用にコンパイルされたカーネルブートイメージファイルを指します。もちろん、ここでいう「未修正」とは、カーネルに大きな変更が加えられていないことを意味します。なぜなら、メモリ仮想ディスクを含むカーネルコードをコンパイルするためには、コンパイル済みの設定ファイルMakefileを修正する必要があるからです。

### 17.10.1 Integrated disk building principle

通常、Linux 0.1Xシステムをフロッピーディスク（ここでいうディスクとは、フロッピーディスクに対応するイメージファイルのこと）で起動する際には、カーネルブートディスクとルートファイルシステムディスクの2つのディスクが必要になります。このため，基本的なLinuxシステムを動作させるためには，システムを起動するために2つのディスクが必要となり，実行時にはルートファイルシステムディスクをフロッピーディスクドライブに残しておく必要がある。ここで紹介する統合ディスクは、カーネルブートディスクと基本的なルートファイルシステムディスクの内容を1枚のディスクにまとめたものです。このようにして、Linuxを起動することができます。

0.1Xシステムでは、1枚の統合ディスクを使ってコマンドプロンプトを表示します。この統合ディスクは、実際にはルートファイルシステムを持つカーネルブートディスクです。

#### 統合ディスクシステムを動作させるためには、ディスク上のカーネルコードでメモリ仮想ディスク（RAMDISK）の機能をオンにする必要がある。統合ディスク上のルートファイルシステムをメモリ上の仮想ディスクにロードすることで、システム上の2つのフロッピードライブを他のファイルシステムディスクのマウントなどのために解放することができます。以下、1.44MBのディスクに統合ディスクを作成する原理と手順を詳しく紹介します。

#### 17.10.1.1 Principle of the boot process

Linux 0.1X カーネルは、コンパイル時の Makefile で設定された RAMDISK オプションに応じて、システムの物理メモリに仮想ディスク領域を確保するかどうかを判断します。RAMDISK が設定されていない (サイズが 0) 場合，カーネルは ROOT\_DEV で設定されたルートファイルシステムのデバイス番号に従って，フロッピーディスクまたはハードディスクからルートファイルシステムをロードし，仮想ディスクがない場合の一般的な起動処理を行う．

Linux 0.1Xのカーネルソースコードをコンパイルする際に、そのlinux/makefileにRAMDISKのサイズが定義されている場合、カーネルコードは起動後、RAMDISKのメモリ領域を初期化した後、まず256番目のディスクブロックから起動ディスクを検出しようとします。ルートファイルシステムがあるか？検出方法は、257番目のディスクブロックに有効なファイルシステムのスーパーブロックがあるかどうかを確認します。もしあれば、そのファイルシステムがRAMDISK領域にロードされ、ルートファイルシステムとして使用されます。そのため、ルートファイルシステムを統合した起動ディスクを使用して、システムをシェルコマンドプロンプトに起動することができます。有効なルートファイルシステムが起動ディスクの指定されたディスクブロックの位置（256番目のディスクブロックから始まる）に格納されていない場合、カーネルはルートファイルシステムディスクの挿入を促します。ユーザーがEnterキーを押して確認すると，カーネルは独立したディスク上のルートファイルシステムを読み取り，それをメモリの仮想ディスク領域に読み込む。この検出と読み込みのプロセスを図9-7に示す。

#### 17.10.1.2 Structure of the integrated disk

Linux 0.1Xのカーネルでは、コード＋データセグメントのサイズが非常に小さく、120KB～160KB程度となっています。Linuxシステム開発の初期段階では、カーネルの拡張を考慮しても、リーナス氏はカーネルのサイズが256KBを超えることはないと考えているので、1.44MBの起動ディスクの256番目のディスクブロックの先頭に、基本的なルートファイルシステムを格納することができます。これにより、統合されたディスクとなります。基本的なルートファイルシステムを追加した起動ディスク(すなわち統合ディスク)の模式図を図17-9に示す。ファイルシステムの詳細な構造については、「ファイルシステム」の章の記述を参照してください。

Root FS area

Kernel code

256

257

Boot block

Super

block

図17-9 統合ディスクのコード構造

### 前述したように、統合ディスク上のルートファイルシステムの配置位置とサイズは、主にカーネルの長さと定義されたRAMDISK領域の大きさに関係します。リーナス氏はramdisk.cプログラムの中で、ルートファイルシステムの開始配置位置を256番目のディスクブロックの先頭と定義しています。Linux 0.1Xカーネルの場合、コンパイルされたカーネルイメージファイル（ブートディスクのイメージファイル）は、約120KB～160KBあります。したがって、ルートファイルシステムをディスクの256番目のブロックの先頭に置くことは、確かに問題ないのですが、ディスクスペースが少し無駄になるだけです。ルート・ファイル・システムのために使えるスペースは、まだ合計1440 - 256 = 1184KBあります。もちろん、特定のコンパイル済みカーネルサイズに応じて、ルートファイルシステムの開始ディスクブロックの位置を調整することもできます。例えば、ramdisk.cの75行目の「block」の値を130に変更することで、ルートファイルシステムの開始位置を後方に移動し、ディスク上のルートファイルシステム用のディスクスペースをより多く確保することができます。

### 17.10.2 Integrated disk construction process

#### カーネルプログラムramdisk.cのデフォルトのディスクブロックの位置を変えずに、統合ディスクのルートファイルシステムは1024KB（最大1184KB）必要だと仮定します。統合ディスクの作成の主旨は、まず1.44MBの空のイメージディスクファイルを作成し、RAMDISK機能で新たにコンパイルしたカーネルイメージファイルをディスクの先頭にコピーします。次に、1024KB以下のカスタマイズされたファイルシステムを、ディスクの256番目のディスクブロックの先頭にコピーする。具体的な構築手順は以下の通りです。

#### 17.10.2.1 Recompiling the kernel

RAMDISK領域が2048KBに設定されていると仮定して、RAMDISK定義を持つカーネルImageファイルを再コンパイルします。方法は、linux-0.1XシステムをBochsで実行します。usr/src/linux/Makefileファイルを編集し、以下の設定行を修正します。

/ ramdisk = -dramdisk = 2048

root\_dev = floppy

その後、カーネルのソースコードを再コンパイルして、新しいカーネルイメージファイルを生成します。

make clean; make

#### 17.10.2.2 Making a Temporary Root File System

1. ルートファイルシステムのImageファイルを1024KBのサイズで作成し、そのファイル名を'rootram.img'とする。Bochsシステムは、ハードディスクのImageを持つ設定ファイル(bochsrc-hd.bxrc)を使って、構築過程で実行される。構築方法は以下の通りです。
2. Make an empty Image file of 1024KB in size using the method described earlier in this chapter. The name of the file is specified as 'rootram.img'. You can use the following command to generate under the current Linux system:

dd bs=1024 if=/dev/zero of=rootram.img count=1024

1. Run the linux-0.1X system in Bochs. Then configure the driver disks in the main Bochs window: disk A is rootimage-0.1X (0.11 kernel is rootimage-0.11-orign); disk B is rootram.img.
2. Use the following command to create an empty file system of size 1024KB on the rootram.img disk. Then mount the A and B disks to the /mnt and /mnt1 directories respectively. If the directory /mnt1 does not exist, you can create one.

/ mkfs /dev/fd1 1024 mkdir /mnt1 mount /dev/fd0 /mnt mount /dev/fd1 /mnt1

1. Use the 'cp' command to selectively copy /mnt files from rootimage-0.1X to the /mnt1 directory and create a root filesystem in /mnt1. If you encounter any error message, then the content is usually more than 1024KB. First reduce the files in /mnt/ to meet the capacity requirements of no more than 1024KB. We can remove some files under /bin and /usr/bin to achieve this. Regarding capacity, we can use the 'df' command to view it. For example, the files we can choose to keep are the following:

[/mnt/bin]# ll total 495

-rwx--x--x 1 root root 29700 Apr 29 20:15 mkfs

-rwx--x--x 1 root root 21508 Apr 29 20:15 mknod

-rwx--x--x 1 root root 25564 Apr 29 20:07 mount

-rwxr-xr-x 1 root root 283652 Sep 28 10:11 sh

-rwx--x--x 1 root root 25646 Apr 29 20:08 umount

-rwxr-xr-x 1 root 4096 116479 Mar 3 2004 vi

[/mnt/bin]# cd /mnt/usr/bin [/mnt/usr/bin]# ll total 364

-1 root 29700 Jan 15 1992 cat

-rwxr-xr-x 1 root root 29700 Mar 4 2004 chmod

-rwxr-xr-x 1 root root 33796 Mar 4 2004 chown

-rwxr-x 1 root root 37892 Mar 4 2004 cp

-rwxr-xr-x 1 root root 29700 Mar 4 2004 dd

-rwx--x 1 root 4096 36125 Mar 4 2004 df

-rwx--x 1 root root 46084 Sep 28 10:39 ls

-rwxr-x 1 root 29700 Jan 15 1992 mkdir

-rwxr-xx 1 root root 33796 Jan 15 1992 mv

-rwxr-xr-x 1 root root 29700 Jan 15 1992 rm

-rwxr-xr-x 1 root root 25604 Jan 15 1992 rmdir

[/mnt/usr/bin]#.

1. Then use the following command to copy the file. In addition, you can modify the contents of /mnt/etc/fstab and /mnt/etc/rc as needed. At this point, we have created a file system with a size of 1024KB or less in fd1(/mnt1/).

/ cd /mnt1 for i in bin dev etc usr tmp do cp +recursive +verbose /mnt/$i $i done sync

1. Use the 'umount' command to unmount the filesystems on /dev/fd0 and /dev/fd1, then use the 'dd' command to copy the filesystem from /dev/fd1 to the Linux-0.1X system and create a name called rootram- 0.1X root file system Image file:

dd bs=1024 if=/dev/fd1 of=rootram

-

0.1

X

count=1024

このとき、Bochs氏のLinux-0.1Xシステムでは、新たにコンパイルされたカーネルイメージファイル/usr/src/linux/Imageと、1024KB以下の容量のシンプルなルートファイルシステムイメージファイルrootram-0.1Xが用意されている。

#### 17.10.2.3 Creating an Integrated Disk

ここで、上記2つのイメージファイルを組み合わせて統合ディスクを作成します。BochsのメインウィンドウでAディスクの設定を変更し、先に用意したbootroot-0.1Xという1.44MBのイメージファイルに設定します。そして、以下のコマンドを実行します。

/ dd bs=8192 if=/usr/src/linux/Image of=/dev/fd0 dd bs=1024 if=rootram-0.1X of=/dev/fd0 seek=256 sync;sync;sync。

オプションの「bs=1024」は、定義バッファのサイズが1KBであることを意味し、「seek=256」は、出力ファイルを書き込む際に、最初の256個のディスクブロックをスキップすることを意味する。そしてBochsシステムを終了する。この時点で、ホストのカレントディレクトリに実行中の統合ディスクイメージファイルbootroot-0.1Xを取得します。

### 17.10.3 Running the Integrated Disk System

まず、統合ディスク用の簡単なBochs設定ファイル、bootroot-0.1X.bxrcを作ってみましょう。主な設定内容は以下の通りです。

floppya: 1\_44=bootroot

-

0.1

X

その後、設定ファイルをマウスでダブルクリックして、Bochsシステムを実行します。この時点で、図17-10のような結果になっているはずです。

GDBとBochsによるカーネルコードのデバッグ

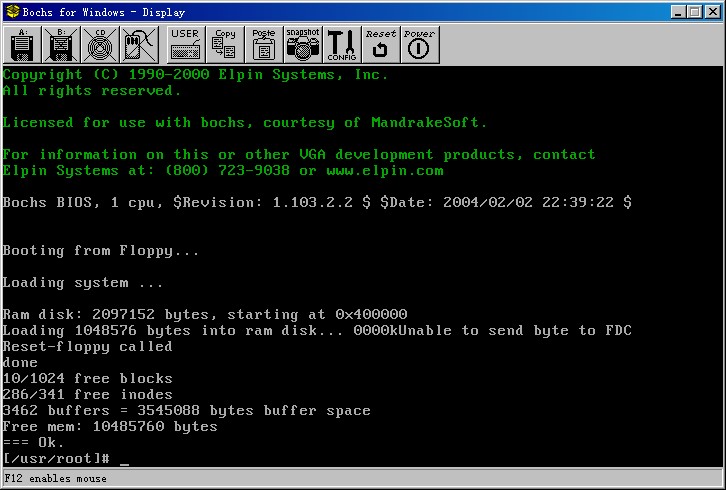


図17-10 統合ディスク実行インターフェース

また、実験を円滑に進めるために、準備が整い、すぐに実行できる0.11カーネルの統合ディスクソフトを以下のサイトからダウンロードすることができます。

http://oldlinux.org/Linux.old/bochs/bootroot

-

0.11

-

040928

.zip

## 17.11 Debugging Kernel Code with GDB and Bochs

ここでは、Bochs エミュレーション環境と gdb ツールを使用して、RedHat や Fedora などの既存の Linux システム上で Linux 0.1X カーネルのソースコードをデバッグする方法について説明します。この方法を使用する前に、既存の Linux システムに X window システムが既にインストールされている必要があります。Bochsのウェブサイトで提供されているRPMインストールパッケージに含まれるBochsの実行ファイルには、gdbデバッガと通信する「gdbstub」モジュールが含まれていないため、Bochsのソースコードをダウンロードして、このモジュールを使って実行プログラムをコンパイルする必要があります。

### gdbstub」モジュールは、Bochsプログラムがローカルの1234ネットワークポート上でgdbからのコマンドを待ち受け、コマンド実行結果をgdbに送信することを可能にします。つまり、gdbを使ってLinux0.1XカーネルのC言語レベルのデバッグを行うことができるのです。もちろん、生成されたカーネルコードにデバッグ情報を持たせるためには、Linux0.1Xカーネルも「-g」オプションを付けて再コンパイルする必要があります。

### 17.11.1 Compiling a Bochs System with gdbstub

Bochsユーザーマニュアルには、自分でBochsシステムをコンパイルする方法が書かれています。ここでは、gdbstubを使ってBochsシステムをコンパイルする方法と手順を紹介します。まず、最新のBochsシステムのソースコードを以下のサイトからダウンロードします（例：bochs-2.6.tar.gz）。

http://sourceforge.net/projects/bochs/

パッケージを'tar'で解凍すると、カレントディレクトリにbochs-2.6のサブディレクトリが生成されます。このサブディレクトリに入った後、設定プログラム「configure」を「--enable-gdb-stub」オプション付きで実行し、次に「make」と「make install」を以下のように実行します。

[root@plinux bochs-2.2]# ./configure --enable-gdb-stub checking build system type...i686-pc-linux-gnu checking host system type...i686-pc-linux-gnu checking target system type...i686-pc-linux-gnu

...

[root@plinux bochs-2.2]# make

[root@plinux bochs-2.2]# make install

./configure」の実行時に問題が発生し、コンパイルに使用するMakefileが生成されない場合、通常、X window開発環境ソフトウェアや関連するライブラリファイルがインストールされていないことが原因となります。この場合、まず必要なソフトウェアをインストールしてから、Bochsを再コンパイルする必要があります。

### 17.11.2 Compiling the Linux 0.1X Kernel with Debug Information

Bochs社のシミュレーション実行環境とシンボリック・デバッグ・ツールgdbをリンクすることで、Linux 0.1X系でコンパイルされたデバッグ情報付きカーネルモジュールを使ってデバッグすることも、RedHat環境でコンパイルされた0.1Xカーネルモジュールを使ってデバッグすることも可能になります。どちらの環境でも、0.1XカーネルのソースディレクトリにあるすべてのMakefileを修正し、コンパイルフラグラインに「-g」オプションを追加し、リンクフラグラインの「-s」オプションを削除する必要があります。

/ LDFLAGS = -M -x // '-s ' フラグを削除します。CFLAGS = -Wall -O -g -fomit-frame-pointer \ // '-g' フラグを追加する。

カーネルのソースディレクトリに入った後、「find」コマンドを使って、修正が必要な以下のMakefileをすべて見つけます。

/ [root@plinux linux-0.1X]# find ./ -name Makefile

./fs/Makefile

./kernel/Makefile

./kernel/chr\_drv/Makefile ./kernel/math/Makefile

./kernel/blk\_drv/Makefile

./lib/Makefile

./Makefile

./mm/Makefile

[root@plinux linux-0.1X]#.

また、この時点でコンパイルされたカーネルコードモジュールにはデバッグ情報が含まれているため、システムモジュールのサイズが、書き込み用カーネルコードイメージファイルのSYSSIZE = 0x3000（boot/bootsect.sファイルの7行目で定義されている）のデフォルトの最大値を超えてしまうことがあります。この時点で、ソースコードのルートディレクトリにあるMakefileで生成されたImageファイルのルールを変更し、カーネルモジュール'system'のシンボル情報を削除してからImageファイルに書き込みます。シンボル情報を持つオリジナルの'system'モジュールは、gdbデバッガが使用するために確保されています。なお、Makefileのターゲットに対する実装コマンドは、タブで始まる必要があります。

Image: boot/bootsect boot/setup tools/system tools/build cp -f tools/system system.tmp strip system.tmp tools/build boot/bootsect boot/setup system.tmp $(ROOT\_DEV) $(SWAP\_DEV) > Image rm -f system.tmp sync

もちろん、boot/bootsect.sやtools/build.cのSYSSIZEの値を0x8000に変更することでも対応可能です。

### 17.11.3 Debugging methods and steps

#### 以下では、最新のLinuxシステム（RedHatやFedoraなど）でコンパイルされたカーネルコードと、Bochsで動作するLinux 0.1Xシステムでコンパイルされたカーネルコードに応じた、デバッグ方法と手順を説明します。Linux 0.11のカーネルコードのデバッグ方法と手順を以下に示します。0.12カーネルのデバッグ方法と手順は全く同じです。

#### 17.11.3.1 Debugging the Linux 0.11 kernel compiled on modern Linux

Linux 0.11のカーネルソースのルートディレクトリがlinux-rh9-gdb/だとすると、まずこのディレクトリにあるすべてのMakefileを上記の方法で修正し、その中にBochs設定ファイルを作成して、カーネルをサポートするルートファイルシステムのイメージファイルをダウンロードします。また、設定されている以下のパッケージをWebサイトから直接ダウンロードして実験を行います。

http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-rh9-050619.tar.gz

このパッケージを「tar zxvf linux-gdb-rh9-050619.tar.gz」コマンドで解凍すると、以下のファイルとディレクトリが含まれていることがわかります。

[root@plinux linux-gdb-rh9]# ls -l total 1600

-1 root root 18055 Jun 18 15:07 bochsrc-fd1-gdb.bxrc drwxr-xr-x 10 root root 4096 Jun 18 22:55 linux

-rw-r--r-- 1 root root 1474560 Jun 18 20:21 rootimage-0.11-for-orig

-rwxr-xr-x 1 root root 35 Jun 18 16:54 run

[root@plinux linux--gdb-rh9]#.

最初のファイル「bochsrc-fd1-gdb.bxrc」は、Bochsの設定ファイルで、ファイルシステムのイメージファイル「rootimage-0.11-for-orig」が、2番目の「フロッピードライブ」に挿入されるように設定されています。このBochs設定ファイルと他のLinux 0.1X設定ファイルとの主な違いは、ファイルの先頭に以下の行が追加されていることで、この設定ファイルでBochsを実行すると、ローカルネットワークのポート1234にあるgdbデバッガからのコマンドを待ち受けることを示しています。

gdbstub: enabled=1, port=1234, text\_base=0, data\_base=0, bss\_base=0

1. linux/の2番目の項目は、Linux 0.11のソースコードディレクトリで、すべてのMakefileに対して修正されたカーネルソースコードファイルが含まれています。3番目のファイル「rootimage-0.11-for-orig」は、このカーネルコードに関連付けられているルートファイルシステムのイメージファイルです。4つ目のファイル'run'は、Bochsの起動コマンドを1行含んだシンプルなスクリプトです。この実験を実行するための基本的な手順は以下の通りである。
2. Open two terminal windows under the X window system;
3. In one of the terminal windows, switch the working directory to the linux-gdb-rh9/ directory and run the program './run'. At this point, a message waiting for gdb to connect is displayed: "Wait for gdb connection On localhost:1234", and the system will create a Bochs main window (no content at this time);
4. In another terminal window, we switch the working directory to the kernel source directory linux-gdb-rh9/linux/ and run the command: "gdb tools/system";
5. Type the command "break main" and "target remote localhost:1234" in the window where gdb is run. At this time, gdb will display the information that has been connected to Bochs.
6. Execute the command "cont" in the gdb environment. After a while, gdb will show that the program stops at the main() function of init/main.c.

その後、gdbコマンドを使ってソースコードを観察し、カーネルをデバッグすることができます。例えば、ソースコードの観測には「list」コマンドを、オンラインヘルプ情報の取得には「help」コマンドを、他のブレークポイントの設定には「break」を、変数値の表示・設定には「print/set」を、シングルステップデバッグの実行には「Next/step」を、gdbの終了には「quit」コマンドを、といった具合です。gdbの具体的な使い方については、gdbのマニュアルを参照してください。以下に、gdbを起動し、その中で実行するコマンドの例を示します。

/ [root@plinux linux]# gdb tools/system // システムモジュールを実行するためにgdbを起動します。GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)

Copyright 2003 Free Software Foundation, Inc.

GDBはGNU General Public Licenseに準拠したフリーソフトウェアであり，一定の条件の下で変更やコピーの配布が可能です。

show copying」と入力すると、条件が表示されます。

GDBには一切の保証がありません。 詳しくは "show warranty "と入力してください。

このGDBは、"i386-redhat-linux-gnu "として設定されていました...。

(gdb) break main // main()関数にブレークポイントを設定します。

ブレークポイント1 at 0x6621: file init/main.c, line 110.

(gdb) target remote localhost:1234 // Bochsに接続しています。

localhost:1234を使ったリモートデバッグ

0x0000fff0 in sys\_mkdir (pathname=0x0, mode=0) at namei.c:481

481 namei.c: そのようなファイルやディレクトリはありません。

in namei.c

(gdb) cont // ブレークポイントまで実行を続ける。

継続中です。

Breakpoint 1, main () at init/main.c:110 // ブレークポイントで実行を停止します。

110 root\_dev = orig\_root\_dev;

(gdb) list // ソースコードを表示します。

1. 105 { /\* スタートアップ・ルーチンは、（まあ、...）これを前提としている \*/ 106/ /\*
2. \* Interrupts are still disabled. Do necessary setups, then
3. \* enable them
4. \*/
5. ROOT\_DEV = ORIG\_ROOT\_DEV;
6. drive\_info = DRIVE\_INFO;
7. memory\_end = (1<<20) + (EXT\_MEM\_K<<10);
8. memory\_end &= 0xfffff000;
9. if (memory\_end > 16\*1024\*1024)
10. (gdb) next // シングルステップ実行。
11. drive\_info = DRIVE\_INFO;
12. (gdb) next // シングルステップ。
13. memory\_end = (1<<20) + (EXT\_MEM\_K<<10);

(gdb) print /x ROOT\_DEV // 変数ROOT\_DEVを表示します。

$3 = 0x21d // 2番目のフロッピーデバイスの番号です。

(gdb) quit // gdbを終了します。

プログラムは動いています。 とにかく終了しますか？(y または n) y

[root@plinux linux]#

gdbでカーネルのソースコードをデバッグしていると、ソースプログラムが見つけられない問題が表示されることがあります。例えば、gdbでは時々「memory.c: No such file or directory "と表示されることがあります。これは、mm/memory.cなどをコンパイルする際に、ldリンカーがmm/以下のファイルモジュールをリンクして、リロケータブルモジュール「mm.o」を生成したことをMakefileが示しており、ソースコードのルートディレクトリlinux / Downでは、再びldの入力モジュールとして使用されているためです。したがって、これらのファイルをlinux/ディレクトリにコピーして、カーネルのデバッグ作業を再実行することができます。

#### 17.11.3.2 Debug the 0.1X kernel compiled on Linux 0.1X system

1. 0.1XシステムでコンパイルされたカーネルをRedHatなどの最新のLinux OSでデバッグするためには、カーネルイメージファイルImageを修正・コンパイルした後、0.1Xカーネルソースディレクトリ全体をRedHatシステムにコピーする必要があります。その後、上記の同様の手順を行います。前述のlinux-0.1X環境を使ってカーネルをコンパイルした後、Imageファイルを含むカーネルソースツリーを圧縮し、mcopyコマンドを使ってBochsの2番目のフロッピーイメージファイルに書き込み、最後にWinImageソフトウェアまたはmountコマンドを使って圧縮ファイルを取り出すことができます。ファイルのコンパイルと抽出のプロセスの基本的な手順を以下に示す。
2. Run Linux-0.1X under Bochs, enter the directory /usr/src/, and create the directory 'linux-gdb ';
3. Use the command to copy the entire 0.1X kernel source tree first: "cp -a linux linux-gdb/". Then enter the linux-gdb/linux/ directory, modify all Makefiles as described above, and compile the kernel;
4. Go back to the /usr/src/ directory and use the 'tar' command to compress the linux-gdb/ directory to get the 'linux-gdb.tgz' file.
5. Copy the compressed file to the second floppy (b drive) image file: "mcopy linux-gdb.tgz b:". If the b disk space is not enough, please use the delete file command "mdel b: file name" to make some space on the b disk.
6. If the host environment is a Windows operating system, then use WinImage to extract the compressed
7. ホスト環境がRedhatやその他の最新のLinuxシステムの場合は、「mount」コマンドを使ってbディスクイメージファイルを読み込み、そこから圧縮カーネルファイルをコピーします。
8. Decompressing the copied compressed file on the modern Linux system will generate a linux-gdb/ directory containing the 0.1X kernel source tree. Go to the linux-gdb/ directory and create the bochs configuration file 'bochsrc-fd1-gdb.bxrc'. You can also take the contents of the 'bochsrc-fdb.bxrc' configuration file from the 'linux-0.11-devel' package and add the 'gdbstub' parameter line yourself. Then download the 'rootimage-0.11' root file system floppy image file from the oldlinux.org website, which is also saved in the linux-gdb/ directory.

その後は、前節の手順に従って、ソースコードのデバッグ実験を続けます。以下に、上記の手順の例を示します。ホスト環境をRedhat系とし、BochsのLinux 0.11系を実行するとします。

/ [/usr/root]# cd /usr/src // ソースコードのディレクトリに入る。

[/usr/src]# mkdir linux-gdb // Ceate ディレクトリ linux-gdb/ です。

[/usr/src]# cp -a linux linux-gdb/ // ソースコードを linux-ddb/ にコピーする。

[/usr/src]# cd linux-gdb/linux

[/usr/src/linux-gdb/linux]# vi Makefile // Makefileを修正します。

...

[/usr/src/linux-gdb/linux]# make clean; make // カーネルをコンプリートする。

...

[/usr/src/linux-gdb/linux]# cd ../../.

[/usr/src]# tar zcvf linux-gdb.tgz linux-gdb // 圧縮ファイルを作成します。

...

[/usr/src]# mdir b: // bディスクの内容を確認します。 BドライブのボリュームはBt

B:/のディレクトリ

LINUX-GD TGZ 827000 6-18-105 10:28p

TPUT TAR 184320 3-09-132 3:16p

LILO TAR 235520 3-09-132 6:00p

SHOELA~1 Z 101767 9-19-104 1:24p

システムマップ 17771 10-05-104 11:22p

5ファイル 90624バイトの空き容量

[/usr/src]# mdel b:linux-gd.tgz // 容量が足りないので、いくつかのファイルを削除します。

[/usr/src]# mcopy linux-gdb.tgz b: // linux-gdb.tgz を b ディスクにコピーします。

LINUX-GD.TGZのコピー

[/usr/src]#

Bochsシステムを終了すると、bディスクイメージファイルの中に「LINUX-GD.TGZ」という圧縮ファイルが得られます。デバッグ実験用ディレクトリは、Redhat Linuxホスト環境で以下のコマンド列を使用して確立できます。

/ [root@plinux 0.11]# mount -t msdos diskb.img /mnt/d4 -o loop,r // bのディスクイメージファイルをマウントする。

[root@plinux 0.11]# ls -l /mnt/d4 // 内容を確認します。

合計 1234

-rwxr-xr-x 1 root root 235520 Mar 9 2032 lilo.tar

-rwxr-xr-x 1 root root 723438 Jun 19 2005 linux-gd.tgz

-rwxr-xx 1 root root 101767 Sep 19 2004 shoela~1.z

17.12 まとめ

-rwxr-xr-x 1 root root 17771 Oct 5 2004 system.map

-rwxr-xr-x 1 root root 184320 Mar 9 2032 tput.tar

[root@plinux 0.11]# cp /mnt/d4/linux-gd.tgz . // ファイルをコピーします。

[root@plinux 0.11]# umount /mnt/d4 // bディスクをアンマウントします。

[root@plinux 0.11]# tar zxvf linux-gd.tgz // ファイルをアンタールする。

...

[root@plinux 0.11]# cd linux-gdb [root@plinux linux-gdb]# ls -l total 4 drwx--x--x 10 15806 root 4096 Jun 19 2005 linux

[root@plinux linux-gdb]#.

その後、Bochs設定ファイル'bochsrc-fd1-gdb.bxrc'をlinux-gdb/ディレクトリに作成し、フロッピーのルートファイルシステムのイメージファイル'rootimage-0.11'をダウンロードする必要もある。また、便宜上、"bochs -q -f bochsrc-fd1-gdb.bxrc "の1行だけを含むスクリプトファイル'run'を作成し、ファイル属性を実行可能に設定することもできる。また、直接デバッグ実験を行うためのパッケージがoldlinux.orgで皆のために作成されており、Redhatで直接コンパイルされたパッケージと同じ内容になっています。

http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-050619.tar.gz

## 17.12 Summary

この本の最後の章です。本章では，Bochs シミュレーション環境を用いた Linux 0.1X の実験運用について説明します．Bochsシステムの基本的な使い方を説明します。シミュレーション・システムとホスト・システムの間でファイルを転送する方法を詳細に説明している。また，Linux 0.1X カーネルのコンパイルとデバッグの具体的な方法と手順を説明している．

本書の内容はここまでですが、読者の皆様には、これを新たな旅の出発点と考えていただき、今日のLinuxシステムのカーネルコードに使われている新技術や新機能について、さらに学び、研究を始めていただきたいと思います。筆者のつたない文章に付き合ってくれる強い意志を持った友人たちに、改めて感謝します。皆さんの新たな旅が楽しいものになりますように。ありがとうございました。

リファレンス

# References

1. Intel Co. INTEL 80386 Programmer's Reference Manual 1986, INTEL CORPORATION,1987.
2. Intel Co. IA-32 Intel Architecture Software Developer's Manual Volume.3:System Programming Guide. http://www.intel.com/, 2005.
3. James L. Turley. Advanced 80386 Programming Technigues. Osborne McGraw-Hill,1988.
4. Brian W. Kernighan, Dennis M. Ritchie. The C programming Language. Prentice-Hall 1988.
5. Leland L. Beck. System Software: An Introduction to Systems Programming,3nd. Addison-Wesley,1997.
6. Richard Stallman, Using and Porting the GNU Compiler Collection,the Free Software Foundation, 1998.
7. The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001, The IEEE and The Open Group.
8. David A Rusling, The Linux Kernel, 1999. http://www.tldp.org/
9. Linux Kernel Source Code，http://www.kernel.org/
10. Digital co.ltd. VT100 User Guide, http://www.vt100.net/
11. Clark L. Coleman. Using Inline Assembly with gcc. http://oldlinux.org/Linux.old/ [12] John H. Crawford, Patrick P. Gelsinger. Programming the 80386. Sybex, 1988.
12. FreeBSD Online Manual, http://www.freebsd.org/cgi/man.cgi
13. Andrew S.Tanenbaum.Operating Systems: Design and Implementation. Prentice-Hall-International Editions. 1990.4
14. Maurice J. Bach. The Design of the UNIX Operating System. Prentice Hall. 1990
15. John Lions. Lions’ Commentary on UNIX 6th Edition with Source Code. Peer-to-Peer Communications, Inc. 1996
16. Andrew S. Tanenbaum, Albert S. Woodhull. Operating Systems:Design and Implementation (Second Edition). Prentice Hall. 1997.
17. Alessandro Rubini, Jonathan. Linux Device Drivers. O’Reilly & Associates. Inc. 2001
18. Daniel P. Bovet, Marco Cesati. Understanding The Linux Kernel. China Electric Power Press. 2001.
19. 张载鸿. 微型机(PC 系列)接口控制教程, 清华大学出版社，1992.
20. 李凤华，周利华，赵丽松. MS-DOS 5.0 内核剖析. 西安电子科技大学出版社，1992.
21. RedHat 9.0 Online manual. http://www.plinux.org/cgi-bin/man.cgi
22. W.Richard Stevens. Advanced Programming in the UNIX Environment. China Machine Press. 2000.2
23. Linux Weekly Edition News. http://lwn.net/
24. P.J. Plauger. The Standard C Library. Prentice Hall, 1992
25. Free Software Foundation. The GNU C Library. http://www.gnu.org/ 2001
26. Chuck Allison. The Standard C Library. C/C++ Users Journal CD-ROM, Release 6. 2003
27. Bochs simulation system. http://bochs.sourceforge.net/
28. Brennan "Bas" Underwood. Brennan's Guide to Inline Assembly. http://www.rt66.com/~brennan/
29. John R. Levine. Linkers & Loaders. http://www.iecc.com/linker/
30. Randal E. Bryant, David R. O'Hallaron. Computer Systems A programmer's Perspective. Publishing House of Electronics Industry. 2004.3
31. Intel. Data Sheet: 8254 Programmable Interval Timer. 1993.9
32. Intel. Data Sheet: 8259A Programmable Interrupt Controller. 1988.12
33. Intel. Data Sheet: 82077A CHMOS Single-chip Floppy Disk Controller. 1994.5
34. Robert Love. Linux Kernel Development. China Machine Press. 2004
35. Adam Chapweske. The PS/2 Keyboard Interface. http://www.computer-engineering.org/
36. リファレンス
37. Dean Elsner, Jay Fenlason & friends. Using as: The GNU Assembler. http://www.gnu.org/ 1998
38. Steve Chamberlain. Using ld: The GNU linker. http://www.gnu.org/ 1998
39. Michael K. Johnson. The Linux Kernel Hackers' Guide. http://www.tldp.org/ 1995
40. Richard F. Ferraro. Programmer’s Guide to the EGA, VGA, and Super VGA Cards. 3rd ed. Addison-Wesley, 1995.

A1 ASCIIコード表

**付録**

A1 ASCII Code Table

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Decimal | Hex | Character | Decimal | Hex | Character | Decimal Hex | Character |
| 0 | 00 | NUL | 43 | 2B | + | 86 56 | V |
| 1 | 01 | SOH | 44 | 2C | , | 87 57 | W |
| 2 | 02 | STX | 45 | 2D | - | 88 58 | X |
| 3 | 03 | ETX | 46 | 2E | . | 89 59 | Y |
| 4 | 04 | EOT | 47 | 2F | / | 90 5A | Z |
| 5 | 05 | ENQ | 48 | 30 | 0 | 91 5B | [ |
| 6 | 06 | ACK | 49 | 31 | 1 | 92 5C | \ |
| 7 | 07 | BEL | 50 | 32 | 2 | 93 5D | ] |
| 8 | 08 | BS | 51 | 33 | 3 | 94 5E | ^ |
| 9 | 09 | TAB | 52 | 34 | 4 | 95 5F | \_ |
| 10 | 0A | LF | 53 | 35 | 5 | 96 60 | ` |
| 11 | 0B | VT | 54 | 36 | 6 | 97 61 | a |
| 12 | 0C | FF | 55 | 37 | 7 | 98 62 | b |
| 13 | 0D | CR | 56 | 38 | 8 | 99 63 | c |
| 14 | 0E | SO | 57 | 39 | 9 | 100 64 | d |
| 15 | 0F | SI | 58 | 3A | ： | 101 65 | e |
| 16 | 10 | DLE | 59 | 3B | ； | 102 66 | f |
| 17 | 11 | DC1 | 60 | 3C | < | 103 67 | g |
| 18 | 12 | DC2 | 61 | 3D | = | 104 68 | h |
| 19 | 13 | DC3 | 62 | 3E | > | 105 69 | i |
| 20 | 14 | DC4 | 63 | 3F | ? | 106 6A | j |
| 21 | 15 | NAK | 64 | 40 | @ | 107 6B | k |
| 22 | 16 | SYN | 65 | 41 | A | 108 6C | l |
| 23 | 17 | ETB | 66 | 42 | B | 109 6D | m |
| 24 | 18 | CAN | 67 | 43 | C | 110 6E | n |
| 25 | 19 | EM | 68 | 44 | D | 111 6F | o |
| 26 | 1A | SUB | 69 | 45 | E | 112 70 | p |
| 27 | 1B | ESC | 70 | 46 | F | 113 71 | q |
| 28 | 1C | FS | 71 | 47 | G | 114 72 | r |
| 29 | 1D | GS | 72 | 48 | H | 115 73 | s |
| 30 | 1E | RS | 73 | 49 | I | 116 74 | t |
| 31 | 1F | US | 74 | 4A | J | 117 75 | u |
| 32 | 20 | (space) | 75 | 4B | K | 118 76 | v |
| 33 | 21 | ! | 76 | 4C | L | 119 77 | w |
| 34 | 22 | " | 77 | 4D | M | 120 78 | x |
| 35 | 23 | # | 78 | 4E | N | 121 79 | y |
| 36 | 24 | $ | 79 | 4F | O | 122 7A | z |
| 37 | 25 | % | 80 | 50 | P | 123 7B | { |
| 38 | 26 | & | 81 | 51 | Q | 124 7C | | |
| 39 | 27 | ' | 82 | 52 | R | 125 7D | } |
| 40 | 28 | ( | 83 | 53 | S | 126 7E | ~ |
| 41 | 29 | ) | 84 | 54 | T | 127 7F | DEL |
| 42 | 2A | \* | 85 | 55 | U |  |  |

A2 コモン C0, C1 制御文字

A2 Common C0, C1 Control Characters

|  |  |  |
| --- | --- | --- |
| 共通C0制御文字表  **Mnemonic** | **Code** | **Actions taken** |
| NUL | 0x00 | Null -- Ignored when received (not saved in the input buffer). |
| ENQ | 0x05 | Enquiry -- Sends a reply message. |
| BEL | 0x07 | Bell -- makes a sound. |
| BS | 0x08 | Backspace -- Moves the cursor one character position to the left. If the cursor is already on the left edge, there is no action. |
| HT | 0x09 | Horizontal Tabulation -- Moves the cursor to the next tab stop. If there is no tab stop on the right side, move to the right edge. |
| LF | 0x0a | Linefeed -- This code causes a carriage return or line feed operation (see linefeed mode). |
| VT | 0x0b | Vertical Tabulation -- acts like LF. |
| FF | 0x0c | Form Feed -- acts like LF. |
| CR | 0x0d | Carriage Return -- Moves the cursor to the left edge of the current line. |
| SO | 0x0e | Shift Out -- Uses the G1 character set selected by the SCS control sequence. G1 can specify one of five character sets. |
| SI | 0x0f | Shift In -- Uses the G0 character set selected by the SCS control sequence. G0 can specify one of five character sets. |
| DC1 | 0x11 | Device Control 1 -- XON. Let the terminal resume transmission. |
| DC3 | 0x13 | Device Control 3 -- XOFF. Stop sending all other codes except sending XOFF and XON. |
| CAN | 0x18 | Cancel -- 制御シーケンス中に送信された場合、シーケンスは実行されず、直ちに終了します。  The error character is also displayed. |
| SUB | 0x1a | Substitute -- works the same as CAN. |
| ESC | 0x1b | Escape -- Generates an Escape Control Sequence. |
| DEL | 0x7f | Delete -- Ignore when typing (not saved in the input buffer). |

|  |  |  |  |
| --- | --- | --- | --- |
| C1制御共通文字表  **Mnemonic** | **Code** | **7B seq.** | **Actions taken** |
| IND | 0x84 | ESC D | Index -- The cursor moves down one row in the same column. If the cursor is already on the bottom line, a scrolling operation is performed. |
| NEL | 0x85 | ESC H | Next Line -- The cursor moves to the first column of the next line. If the cursor is already on the bottom line, a scrolling operation is performed. |
| HTS | 0x88 | ESC E | Horizontal Tab Set -- Sets a horizontal tab stop at the cursor. |
| RI | 0x8d | ESC M | Reverse Index -- The cursor moves one line up the same column. If the cursor is already on the top line, a scrolling operation is performed. |
| SS2 | 0x8e | ESC N | Single Shift G2 -- Temporarily uses the G2 character set in GL for the display of the next character. G2 is specified by the Selective Character Set (SCS) control sequence (see the escape sequence and control sequence table in Appendix 3). |
| SS3 | 0x8f | ESC O | Single Shift G3 -- Temporarily calls the G3 character set in GL for the display of the next character. G3 is specified by the Selective Character Set (SCS) control sequence (see the |
| A2 コモン C0, C1 制御文字 |  |  | escape sequence and control sequence table in Appendix 3). |
| DCS | 0x90 | ESC P | Device Control String -- Used as the starting qualifier of the device control string. |
| CSI | 0x9b | ESC [ | Control Sequence Introducer -- Used as a control sequence leader code. |
| ST | 0x9c | ESC \ | String Terminator -- Used as the ending qualifier of the DCS string. |

A3 Escape and Control Sequences

|  |  |
| --- | --- |
| Sequence and Name | Description |
| ESC ( Ps または  ESC ) Ps  Select Character Set | キャラクタセットの選択（SCS） -- G0とG1のキャラクタセットは、それぞれ5つのキャラクタセットのいずれかを指定できる。ESC (Ps)」は G0 で使用する文字セットを指定し、「ESC) Ps」は G1 で使用する文字セットを指定する。パラメータ Ps: A - UK キャラクタセット、B - US キャラクタセット、0 - グラフィックキャラクタセット、1 - 代替の ROM キャラクタセット、2 - オプションの ROM 特殊キャラクタセット。  A terminal can display up to 254 different characters, however the terminal only stores 127 display characters in its ROM. You must install additional character set ROM for the other 127 display characters. At some point, the terminal is able to select 94 characters (one character set). Therefore, the terminal can use one of five character sets, some of which appear in multiple character sets. At any one time, the terminal can use two active character sets. The computer can use the SCS sequence to specify any two character sets as G0 and G1. You can then switch between these two character sets using a single control character. The Shift In - SI (14) control character is used to select the G0 character set, and the Shift Out - SO (15) control character can be used to select the G1 character set. The specified character set will be used as the current character set until the terminal receives another SCS sequence. |
| ESC [ Pn A  カーソルアップ  (Terminal <--> Host) | Cursor Up (CUU) -- The CUU control sequence moves the cursor up but the column position is unchanged. The number of moving character positions is determined by parameters. If the parameter is 'Pn', the cursor moves up the 'Pn' line. The cursor is moved up to the top row at most. Note that 'Pn' is an ASCII numeric variable. If you do not select a parameter or the parameter value is 0, the terminal will assume a parameter value of 1. |
| ESC [ Pn B または  ESC [ Pn e  カーソルダウン  (Terminal <--> Host) | Cursor Down (CUD) -- The CUD control sequence moves the cursor down but the column position is unchanged. The number of moving character positions is determined by parameters. If the parameter is 1 or 0, the cursor moves down 1 line. If the parameter is 'Pn', the cursor moves down the 'Pn' line. The cursor moves down to the bottom line at most. |
| ESC [ Pn C または  ESC [ Pn a  カーソル前進  (Terminal <--> Host) | Cursor Forward (CUF) -- The CUF control sequence moves the current cursor to the right. The number of moving positions is determined by parameters. If the argument is 1 or 0, move 1 character position. If the parameter value is 'Pn', the cursor moves by 'Pn' character positions. The cursor moves up to the right border at most. |
| ESC [ Pn D  カーソルバックワード  (Terminal <--> Host) | Cursor Backward (CUB) -- The CUB control sequence moves the current cursor to the left. The number of moving positions is determined by parameters. If the argument is 1 or 0, move 1 character position. If the parameter value is 'Pn', the cursor moves by 'Pn' character positions. The cursor moves up to the left border at most. |
| ESC [ Pn E  Cursor moves down | Cursor Next Line (CNL) -- This control sequence moves the cursor to the first character of the 'Pn' line below. |
| ESC [ Pn F  Cursor moves up | Cursor Last Line (CLL) -- This control sequence moves the cursor up to the first character of the 'Pn' line. |
| ESC [ Pn G または  ESC [ Pn `  Cursor moves in line | Cursor Horizon Absolute (CHA) -- This control sequence moves the cursor to the 'Pn' character position of the current line. |
| ESC [ Pn ; Pn H or | Cursor Position (CUP), Horizontal And Vertical Position (HVP) -- The CUP control sequence moves |

|  |  |
| --- | --- |
| ESC [ Pn;Pn f  Cursor positioning | the current cursor to the position specified by the parameter. The two parameters specify the row and column values, respectively. If the value is 0, it is the same as 1, indicating that one position is moved. In the default condition without parameters, it is equivalent to moving the cursor to the home position (ie ESC [ H ). |
| ESC [ Pn d  Set the row position | Vertical Line Position Absolute -- Moves the cursor to the 'Pn' line of the current column. If you try to move below the last line, the cursor will stay on the last line. |
| ESC [ s  Save cursor position | Save Current Cursor Position -- This control sequence has the same effect as DECSC except that the page number displayed on the cursor is not saved. |
| ESC [ u  Restore cursor position | Restore Saved Cursor Position -- This control sequence has the same effect as DECRC except that the cursor is still on the same display page and not moved to the display page where the cursor is saved. |
| ESC D  Index | Index (IND) -- This control sequence moves the cursor down one line, but the column number does not change. If the cursor is on the bottom line, it will cause the screen to scroll up one line. |
| ESC M  Reverse index | Reverse Index (RI) -- This control sequence moves the cursor up one line, but the column number does not change. If the cursor is on the top line, it will cause the screen to scroll down one line. |
| ESC E  Move down one line | Next Line (NEL) -- This control sequence will move the cursor to the beginning of the left side of the next line. If the cursor is on the bottom line, it will cause the screen to scroll up one line. |
| ESC 7  Save cursor | Save Cursor (DECSC) -- This control sequence will cause the cursor position, graphics to be reproduced, and the character set to be saved. |
| ESC 8  Restore cursor | Restore Cursor (DECRC) -- This control sequence will cause the previously saved cursor position, graphics to be reproduced, and the character set to be restored. |
| ESC [ Ps; Ps; ... ; Ps m  Set character attributes | Select Graphic Rendition (SGR) - 文字の再表示と属性は、文字コードを変更することなく、文字の表示に影響を与える特性である。制御シーケンスは、パラメータに応じて文字表示の属性を設定する。今後、端末に送信されるすべての文字は、制御シーケンスが文字の属性を再びリセットするまで、ここで指定された属性を使用する。パラメータ「Ps」。  0 - no attribute (default attribute); 1 - bold and bright; 4 - underline; 5 - flashing; 7 - reverse; 22 - non-bold; 24 - no underline; 25 - no flicker; ;30--38 Set foreground color; 39 - Default foreground color (White); 40--48 - Set background color; 49 - Default background color (Black). 30--37 and 40-47 correspond to colors: Black, Red, Green, Yellow, Blue, Magenta, Cyan, White. |
| ESC [ Pn L  Insert line | Insert Line (IL) -- This control sequence inserts one or more blank lines at the cursor. The cursor position does not change after the operation is completed. When a blank line is inserted, the line in the scroll area below the cursor moves down. The line scrolling out of the display page is lost. |
| ESC [ Pn M  Delete line | Delete Line (DL) -- This control sequence deletes one or more lines from the line where the cursor is located in the scroll area. When the line is deleted, the line below the deleted line in the scroll area moves up, and 1 blank line is added to the bottom line. If 'Pn' is greater than the number of lines remaining on the display page, then this sequence only deletes these remaining lines and does not work outside the scroll area. |
| ESC [ Pn @  Insert character | Insert Character (ICH) -- This control sequence inserts one or more space characters at the current cursor using the normal character attribute. 'Pn' is the number of characters inserted. The default is 1. The cursor will still be at the first inserted space character. The character at the cursor and right border will shift to the right. Characters that exceed the right border will be lost. |
| ESC [ Pn P  Delete character | Delete Character (DCH) -- This control sequence deletes 'Pn' characters from the cursor. When a character is deleted, all characters to the right of the cursor are shifted to the left. This will produce a |
|  | null character at the right border. Its properties are the same as the last left-shift character. |
| ESC [ Ps J  Erase character | Erase In Display (ED) -- この制御シーケンスは、パラメータに応じて、表示されている文字の一部または全部を消去します。消去操作は、他の文字に影響を与えることなく、画面から文字を取り除きます。消去された文字は破棄されます。文字や行を消去しても、カーソル位置は変わりません。文字を消去すると、その文字の属性も破棄されます。この制御シーケンスで行全体を消去した場合、その行は一文字幅モードに戻ります。パラメータ「Ps」。  0 - Erase the cursor to all characters at the bottom of the screen; 1 - Erase the top of the screen to the cursor except all characters; 2 - Erase the entire screen. |
| ESC [ Ps K  Erase in line | Erase In Line (EL) -- パラメータに従って、カーソルのある行の一部または全部の文字を消去します。消去操作は、他の文字に影響を与えることなく、画面から文字を取り除きます。消去された文字は破棄されます。文字や行を消去しても、カーソル位置は変わりません。文字を消去すると、その文字の属性も破棄されます。パラメータ 'Ps':  0 - Erases the cursor to all characters at the end of the line; 1 - Erases the left border to all characters at the cursor; 2 - Erases an entire line. |
| ESC [ Pn ; Pn r  Set top &bottom margins | Set Top and Bottom Margins (DECSTBM) -- This control sequence sets the upper and lower areas of the scroll screen. The scrolling margin is an area on the screen where we can receive new characters by taking away the original characters from the screen. This area is defined by the top and bottom borders of the screen. The first parameter is the first line of the start of the scrolling area, and the second parameter is the last line of the scrolling area. By default it is the entire screen. The smallest scrolling area is 2 lines, ie the top border line must be smaller than the bottom border line. The cursor will be placed in the home position. |
| ESC [ Pn c or  ESC Z  デバイスの属性  (Terminal <--> Host) | |  |  | | --- | --- | | ホストの要求に応じて、端末はレポートメッセージを送信することができます。これらのメッセージは、アイデンティティ（端末の種類）、カーソル位置、および端末の動作状態を提供します。レポートには、デバイスアトリビュートとデバイスステータスレポートの2種類があります。デバイスアトリビュート(DA) -- ホストは、パラメータなしまたはパラメータ0のデバイスアトリビュート(DA)制御シーケンス(ESC Zと同じ)を送信する。 端末は、ホストのシーケンスに応答して、以下のシーケンスの1つを送信する。  Terminal optional attribute | Send sequence | | None, VT101 | ESC [?1;0c | | Processor option (STP) | ESC [?1;1c | | Advance Video (AVO) VT100 | ESC [?1;2c | | AVO and STP | ESC [?1;3c | | Graphic property option (GPO) | ESC [?1;4c | | GPO and STP | ESC [?1;5c | | GPO and AVO, VT102 | ESC [?1;6c | | GPO, STP and AVO | ESC [?1;7c | |
| ESC c  Reset to initial state | Reset To Initial State (RIS) -- Lets the terminal reset to its initial state, that is, just turned on. All characters received during the reset phase will be lost. There are two ways to avoid this: 1. (Auto XON/XOFF) After the transmission, the host assumes that the terminal has sent XOFF. The host stops sending characters until it receives XON. 2. Delay at least 10 seconds and wait for the terminal reset operation to complete. |

A4 キーボードスキャンコードの最初のセット

A4 The First Set of Keyboard Scan Code

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| KEY | MAKE | BREAK | KEY | MAKE | BREAK | KEY | MAKE |  | BREAK |
| A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z  0  1  2 | 1E  30  2E  20  12  21  22  23  17  24  25  26  32  31  18  19  10  13  1F  14  16  2F  11  2D  15  2C  0B  02  03 | 9E  B0  AE  A0  92  A1  A2  A3  97  A4  A5  A6  B2  B1  98  99  90  93  9F  94  96  AF  91  AD  95  AC  8B  82  83 | 9  `  -  =  \  BKSP  スペース  TAB  CAPS  左SHFT  Left CTRL | 0A  29  0C  0D  2B  0E  39  0F  3A  2A  1D | 8A  89  8C  8D  AB  8E  B9  8F  BA  AA  9D | [ | 1A |  | 9A |
| INSERT | E0,52 |  | E0,D2 |
| HOME | E0,47 |  | E0,97 |
| PG UP | E0,49 |  | E0,C9 |
| DELETE | E0,53 |  | E0,D3 |
| END | E0,4F |  | E0,CF |
| PG DN | E0,51 |  | E0,D1 |
| Up Arrow | E0,48 |  | E0,C8 |
| Left Arrow | E0,4B |  | E0,CB |
| Down Arrow | E0,50 |  | E0,D0 |
| Right Arrow | E0,4D |  | E0,CD |
| Left GUI | E0,5B | E0,DB | NUM LOCK | 45 |  | C5 |
| 左ALT  Right SHFT | 38  36 | B8  B6 | KP / | E0,35 |  | E0,B5 |
| KP \* | 37 |  | B7 |
| Right CTRL | E0,1D | E0,9D | KP - | 4A |  | CA |
| Right GUI | E0,5C | E0,DC | KP + | 4E |  | CE |
| Right ALT | E0,38 | E0,B8 | KP ENTER | E0,1C |  | E0,9C |
| APPS | E0,5D | E0,DD | KP . | 53 |  | D3 |
| ENTER  ESC  F1  F2  F3  F4  F5  F6  F7  F8  F9 | 1C  01  3B  3C  3D  3E  3F  40  41  42  43 | 9C  81  BB  BC  BD  BE  BF  C0  C1  C2  C3 | KP 0 | 52 |  | D2 |
| KP 1 | 4F |  | CF |
| KP 2 | 50 |  | D0 |
| KP 3 | 51 |  | D1 |
| KP 4 | 4B |  | CB |
| KP 5 | 4C |  | CC |
| KP 6 | 4D |  | CD |
| KP 7 | 47 |  | C7 |
| KP 8 | 48 |  | C8 |
| KP 9 | 49 |  | C9 |
| ] | 1B |  | 9B |
| 3 | 04 | 84 | F10 | 44 | C4 | ; | 27 |  | A7 |
| 4 | 05 | 85 | F11 | 57 | D7 | ' | 28 |  | A8 |
| 5  6  7 | 06  07  08 | 86  87  88 | F12 | 58 | D8 | , .  / | 33  34  35 |  | B3  B4  B5 |
| PRNT SCRN | E0,2A, E0,37 | E0,B7, E0,AA |
| SCROLL | 46 | C6 |
| 8 | 09 | 89 | PAUSE | E1,1D,45  E1,9D,C5 | 无 |  |  |  |  |

注1：表中の値はすべて16進法です。

Note 2: In the table, KP - KeyPad, represents the key on the numeric keypad. Note 3: The colored parts in the table are all extended keys.