

13 メモリ管理(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

ファイル名	サイズ	最終更新時刻(GMT)	デス
 Makefile	1221バイト	1992-01-12 19:49:22	コ
 memory.c	13464バイト	1992-01-13 22:57:04	
 page.s	508バイト	1991-10-02 14:16:30	
 swap.c	5193バイト	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ライトプロテクション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 主な機能

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の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。について

ここでは、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する内容をさらに説明します。

13.1.1 メモリ・ページング・メカニズム

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 $1024 \times 1024 \times 4096 = 4G$ 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。

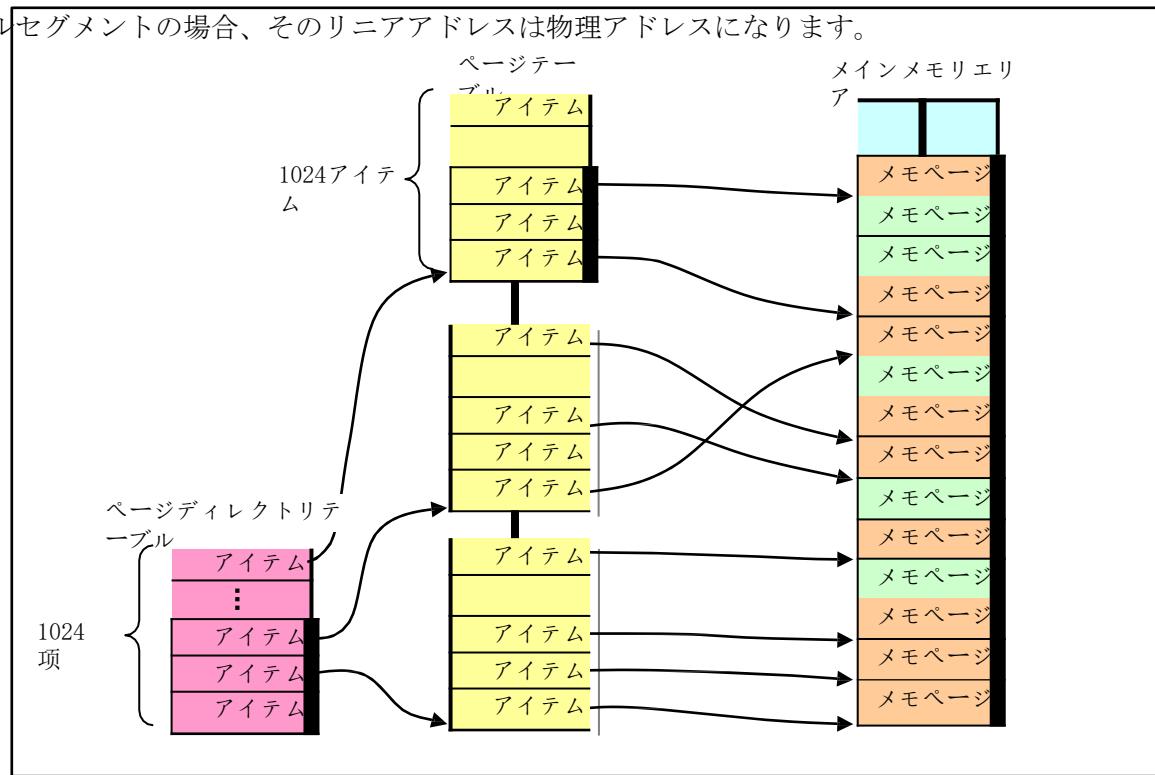


図13-1 ページディレクトリテーブルとページテーブルの構造図

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。ページング機構を利用するため、32ビットのリニアアドレスは3つの部分に分けられ、ページディレクトリエントリ、ページテーブル、そして、物理アドレスを指定するため使われます。

図13-2に示すように、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、エントリ、および対応する物理メモリページ上のオフセットアドレスを設定しています。

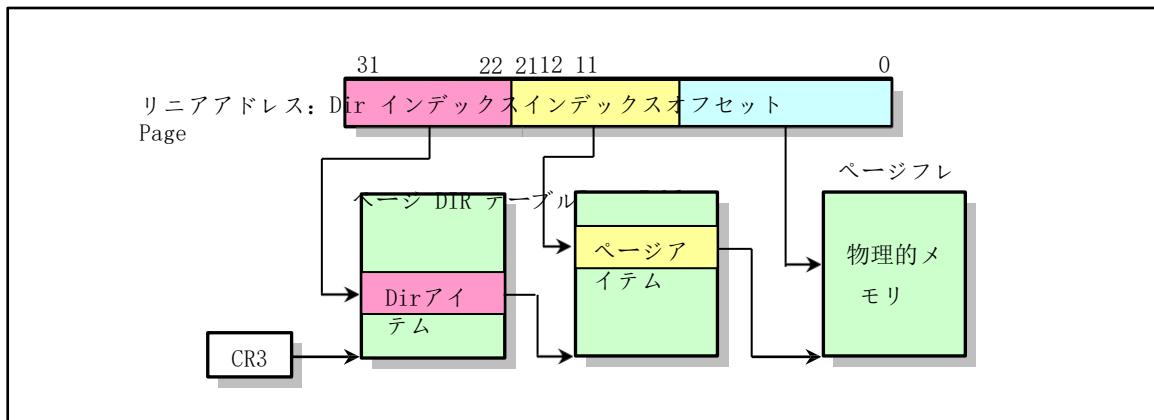


図13-2 リニアアドレス変換の模式図

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。

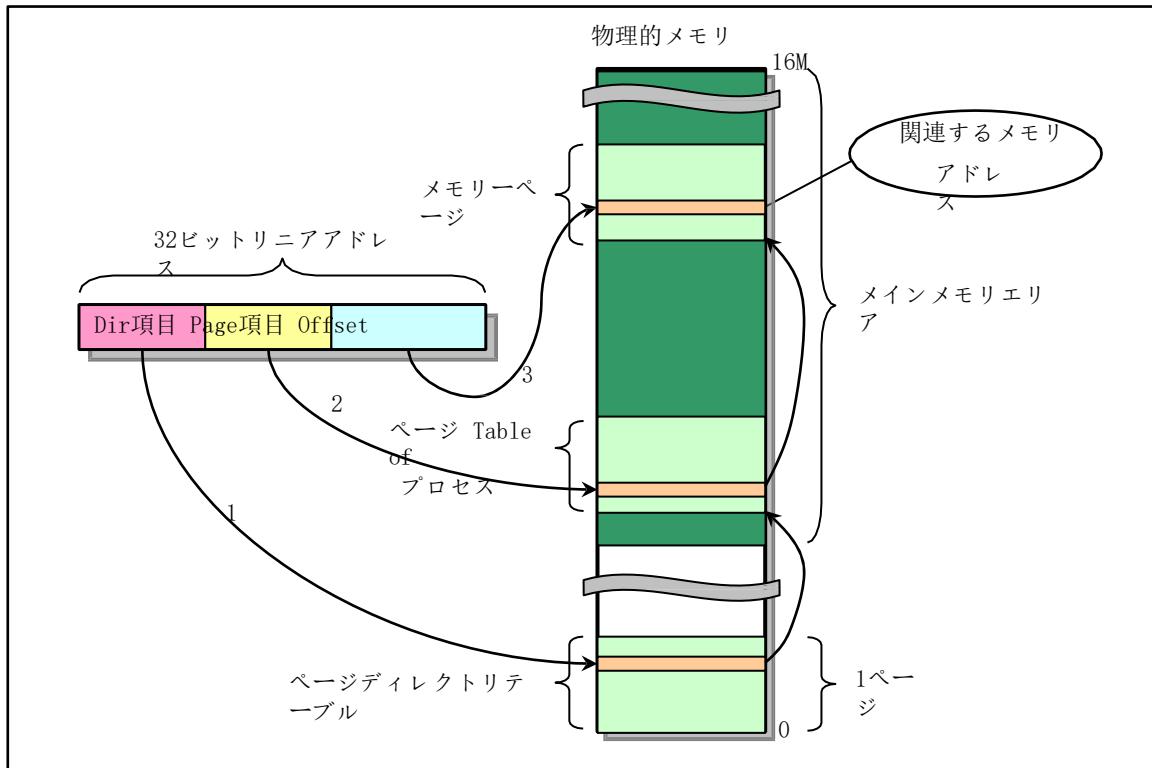


図13-3 リニアアドレスに対応する物理アドレス

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る。

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。

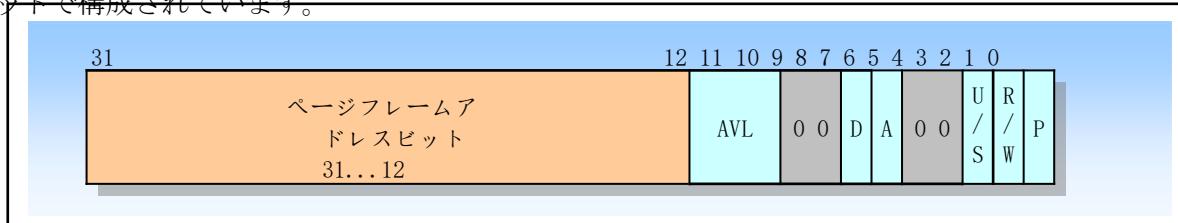


図13-4 ページディレクトリとページテーブルのエントリ構造

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、アドレスにページテーブルエントリを使用できるかどうかを決定します。

の翻訳処を行います。P=1は、そのエントリーが使用可能であることを意味する。ディレクトリ・エントリまたは第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、アドレス変換プロセスで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はアクセスされたビットを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。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修正ビット (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 物理的なメモリの割り当てと管理

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。

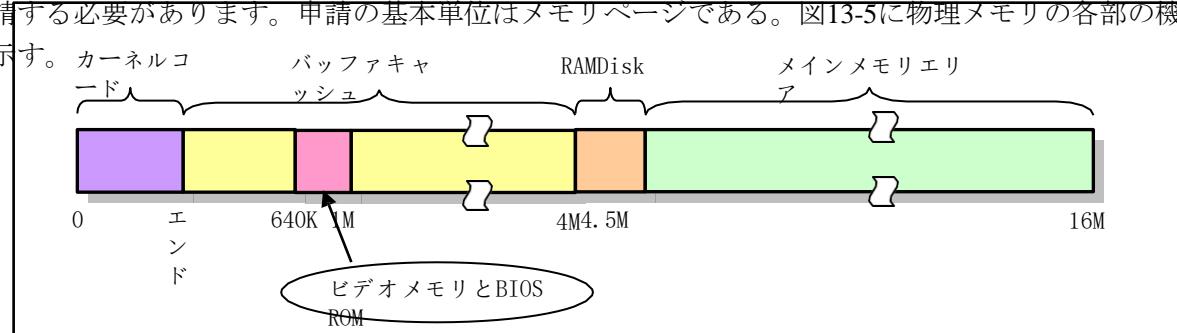


図13-5 メインメモリ領域の模式図

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온라이트の概念である。

`page.s`プログラムは、ページフォルトまたは例外処理 (INT 14) を実装するために使用されます。割り込みの場合

`do_no_page()`関数は、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온라이트)、ページの共有をキャンセルします。

13.1.3 リニアなアドレス空間の割り当て

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理解しておく必要があります。

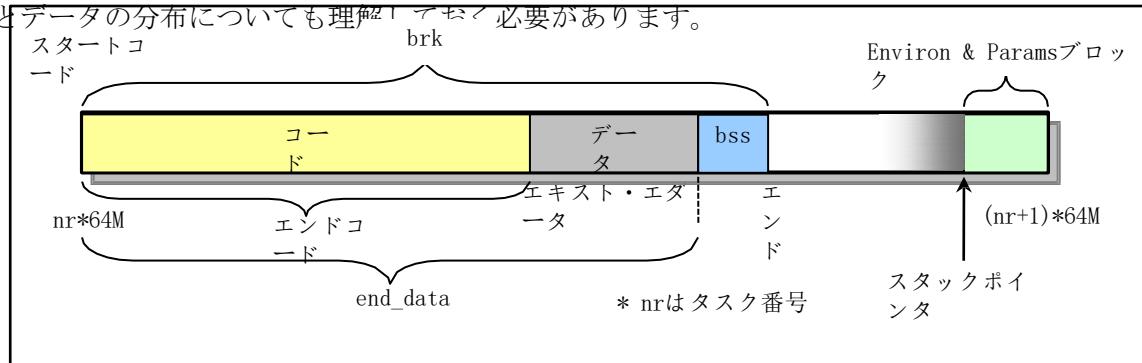


図13-6 論理アドレス空間におけるプロセスの分布

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となります。図中、`bss`はプロセスで初期化されていないデータセグメントです。`bss`セグメントの最初のページは、プロセスが生成されるときにすべて0に初期化されます。

13.1.4 論理アドレス、リニアアドレス、物理アドレス間のトランスマッピング

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得る必要があります。コピー온라이트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作業があります。また、ページを共有しようとする際には

は、プログラムの論理アドレスページをCPUのリニアアドレス空間にマッピングする操作を含みます。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ます。

1. アドレスから対応するページアドレスへの変換

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。

2. 線形アドレスと論理アドレスの分解

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のエントリ番号。PDE_No = (addr >>  
22); ページテーブルエントリナンバー。PTE_No = (addr  
>> 12) & 0x3ff; ページ内のオフセット。Offset = (addr &  
0xffff);
```

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);  
Logic_PTE_No = (Vaddr >> 12) & 0x3ff;  
Logic_Offset = (Vaddr & 0xffff) となり  
ます。
```

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

875

```
PDE_No = (Base >> 22) + Logic_PDE_No;  
= (p->start_code >> 22) + (Vaddr >> 22) です。
```

```

PTE_No = Logic_PTE_No
        = (Vaddr >> 12) & 0x3ff;
Offset = Logic_Offset
        = (Vaddr & 0xffff) となります。

```

3. プログラムページの論理アドレスと物理アドレスの変換

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ると、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、さらに物理ベース・アドレス

ページディレクトリテーブルのPDT_Baseを見れば、ディレクトリエントリアドレス（ポインタ）PDEを得ることができます。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) & 0xffffffff000;
PTE = PT + (PTE_No << 2)
      = PT + (((Vaddr >> 12) & 0x3ff) << 2)。
      = ((*PDE) & 0xffff000) + ((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) & 0xffffffff000です。

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）を得ることができます。

```
PT = (*PDE) & 0xfffff000;
PTE = PT + (PTE_No << 2)
      となります。
      = PT + ((Laddr >> 12) & 0x3ff) << 2) となります。
      = (*PDE) & 0xffff000 + ((Laddr >> 10) & 0xffc) となります。
      = *((Laddr >> 20) & 0xffc) & 0xffff000) + ((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;
Paddr = PPaddr + Laddr & 0xffff
      となります。
      = (*PTE) & 0xfffff000 + Laddr & 0xffff;
      = *((((Laddr >> 10) & 0xffc) + *((Laddr >> 20) & 0xffc) & 0xffff000)) + Laddr & 0xffff;
```

5. ページフォールトの例外処理

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、ページフォールト例外割り込みINT14が発生します。

- アドレス変換処理で使用されるページディレクトリエントリまたはページテーブルエントリのプレゼンスピット (P) が0になると、オペランドを含むページテーブルまたはページが物理メモリに存在しないことを示します。
- 現在の実行コードが、指定されたページにアクセスするための十分な権限を持っていない、またはユーザー モードのコードが読み取り専用のページを書き込んでいる、などの問題があります。

ページの例外処理手順では、中断されたプログラムやプロセスを回復し、再起動することができます。

page-not-presentの状態で、プログラムの実行の継続には影響しません。また、特権侵害後にプログラムやタスクを再起動することができますが、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。

- スタック上のエラーコードです。エラーコードの形式は32ビット長のワードですが、下位3ビットのみが有効です。これらの名前は、ページテーブルエントリの最後の3ビット (U/S, W/R, P) と同じです。それらの意味と役割は
 - ◆ ビット0(P)の場合、存在しないページやアクセス権の侵害による例外が発生しています。P=0は、そのページが存在しないことを示し、P=1は、ページレベルの保護特権が

- を破った。
- ◆ ビット1 (W/R)、メモリの読み出しまだ書き込み操作により例外が発生したことを示す。W/R=0であれば、読み出し操作によるものであることを示し、W/R=1であれば、書き込み操作によるものであることを示す。
 - ◆ ビット2 (U/S)、例外発生時にCPUが実行していたコードレベルを示す。U/S=0であれば、CPUはスーパーバイザモードで実行していたことになり、U/S=1であれば、CPUはユーザモードで実行していましたことになります。
 - コントロールレジスタCR2のリニアアドレスです。CPUは、例外を発生させたリニア・アドレスをCR2に格納します。ページ・フォールト例外ハンドラは、このアドレスを使用して、関連するページ・ディレクトリとページ・テーブル・エントリを見つけることができます。ページ例外ハンドラの実行中に別のページ例外の発生が許される場合、ハンドラはCR2をスタックにプッシュする必要があります。

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メカニズム

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fork()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。これがコピーイ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、上記のパッショブなコピーイ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して、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、まず積極的にメモリページ検証関数 write_verify()で、ページ共有条件があるかどうかを判断します。ある場合は、そのページのコピーインライト動作を行います。

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、コピーイ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なので、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、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確保します。⁸⁷⁸

コピーインライトでは、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、この技術は不必要的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 ロード・オン・デマンド・メカニズム

`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、エンタ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升る。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。

2. メモリ.c

1. 機能

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。各バイトエンタリには、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。

図13-7に示すように、1MB以上のメモリ領域に対応するページ(PAGING_PAGES)を作成し、`mem_map[]`の全項目の値を100(占有)に設定した後、主記憶領域に対応する`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 $(16\text{MB} - 1\text{MB}) / 4\text{KB} = 3840$ 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 $(16\text{MB} - 4.5\text{MB}) / 4\text{KB} = 2944$ 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事はできない。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。

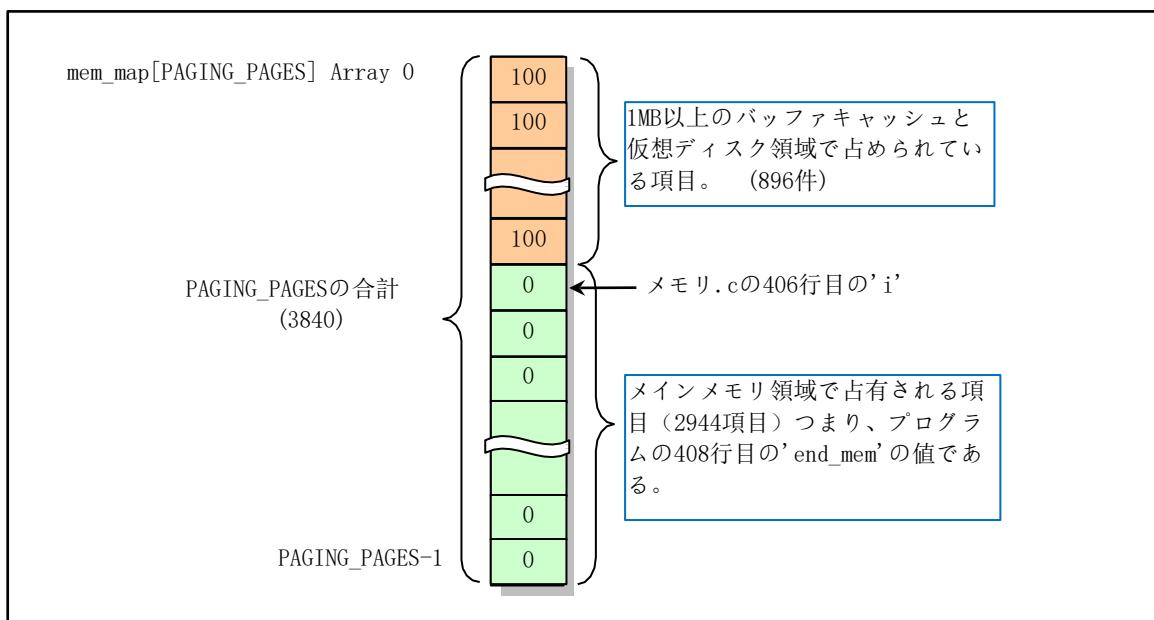


図13-7 16MBのメモリと512KBのvdiskを使用した`Mem_map[]`アレイの初期化

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、特に占領と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フリーネスを確保することができます。`get_free_page()`関数は、メインメモリ領域内の空きメモリのページを要求し、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[]`をスキヤンし、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。まず、指定されたメモリアアドレスが<1M>であるかどうかを判断し、<1M>であれば、1M以内はカーネル固有であるため、リターンします。

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にはエラーメッセージが表示され、指定されたメモリアドレスでページ番号が変換されます。(ページ番号に対応するmem_map[]バイト項目が0であるかどうかを判断し、0でなければ1をデクリメントして返します。

`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タ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境界上に配置し、指定されたアドレス値がカーネルとバッファキャッシュが占める空間よりも上になるようにします。次に、ページディレクトリテーブルで占有されているディレクトリエンタリの数 (すなわち、ページテーブルの数) を計算し、対応する開始ディレクトリエンタリの番号を計算します。そして、対応する開始ディレクトリエンタリから順に、占有されているディレクトリエンタリを解放し、対応するディレクトリエンタ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タ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タ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タリ (`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タリ) の数を算出します。そして、元のディレクトリ・エンタリとページ・テーブル・エンタリを、それぞれ新しい空きディレクトリ・エンタリとページ・テーブル・エンタ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タ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ン操作を行います。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タリを計算します。ディレクトリエンタリが有効であれば、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、空き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タリの属性が設定されます。最後に指定された物理メモリのページアドレスはそのまま返されます。

`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、コピーオンライン操作を行います。

`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。^{88F} 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) の長さがを超える場合は

ページの内容が追加されたときに、プロセスコードの長さにデータをえた分だけ、超過分がクリアされます。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 コードアノテーション

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
7 /*
8 * 01. 12. 91から始まったデマンド・ローディングは、リストの上位にあるようです。
9 * 望んでいたものであり、簡単に実装できるはずです。 - Linus
10 */
11
12 /*
13 * OK、デマンドローディングは簡単でしたが、シェアードページは少し難しいですね。共有ページ
14 * 02. 12. 91に開始したページは、動作しているようです。 - Linus
15 */
16 * 古いカーネルの下で30回ほど/bin/sh:を実行して共有をテストしました。
17 * 私が持っている6M以上の資金が必要でしたが、これでうまくいきました。
18 * 私が見た限りでは
19 */
20 * また、いくつかの「invalidate()」を修正しました - 十分に行っていませんでした。
21 */
22
23 /* 18. 12. 91にリアルVM（ディスクへのページング/ディスクからのページ
   ング）を開始しました。より多くの作業と
24 * このようなことは多かったわけではありません。ああ出ます理由はわかり
25 * ません。 見つけました。これですべてがうまくいくようになりました。
26 * 19. 12. 91 - より
27 * なりました。
28 * 20. 12. 91 - よし、swap-deviceをrootのように変更可能にする。
29 */
30 // <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナル
   の定義
   // 操作関数のプロトタイプ。
   // <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
   // のデの初期タブタゼ割記述系のパラなどが設置を取得に関するいくつかの組み込みアセンブリ関
   // <linux/head.h> 文が含まれていても、ヘッダーファイルでは、タスク構造体task_struct、データ
   // <linux/head.h> ヘッドのヘッダーファイルです。セグメントディスクリプターの簡単な構造が定義
   されています。
   // いくつかのセレクタ定数と一緒に
   // <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
   // カーネルの使用する機能
31 #include <signal.h> (英語)
32

```

```

33 #include <asm/system.h>
34
35 #include <linux/sched.h>
36 #include <linux/head.h>
37 #include <linux/kernel.h> (日本語)
38
// CODE_SPACE(addr) (((addr)+0xffff)&~0xffff) < current->start_code + current->end_code)
// このマクロは、与えられたリニアアドレスがコードセクションの
// 現在のプロセス。"((addr)+4095)&~4095)"は、メモリの終了アドレスを取得するために使用され
// ます。
// リニアアドレス'addr'が存在するページです。265行目を参照してください。
39 #define CODE_SPACE(addr) (((addr)+4095)&~4095) <
 40 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ページ分のメモリをfromからtoにコピーする(4Kバイト)。
44 #define copy_page(from,to) \(^o^)
45 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。で
// 初期化関数mem_init()で、メインメモリとして使用できない位置に
// エリアページはあらかじめUSED(100)に設定され
// ています。 47 unsigned char mem_map [
PAGING_PAGES] = {0,}; 48
49 /*
50 * 物理アドレス'addr'のメモリのページを解放します。使用されるのは
51 * 'free_page_tables()'
52 */
53 void free_page(unsigned long addr)
54 {
    // 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
    // カーネルのaddrログLOW_MEMをreturnする範囲内である1MBを意味し、処理されませんで定義されて
    // がシステム内の物理アドレスのHIGH_MEMORY大きいか同じであれば
    // エラーメッセージが表示され、Freeルが動作しなくなりました。
    // パラメーター'addr'の検証が渡された場合、//からカウントされたメモリーページ番号が表示され
    // この物理アドレスをもとに、メモリのローエンドが計算されます。
    // "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
    // ページがリリースされました。しかし、対応するページバイトがもともと0であれば、それは
    // 物理ページが元々アイドル状態であることを示す//カーネルコードに不具合があることを示すその
    // ため

```

```

// その後、エラーメッセージが表示され、カーネルが停止します。
58     addr -= LOW MEM;
59     addr >= 12;                                //を4096で割ったも
60     if (mem_map[addr]--) return;               のです。
61     panic("try to free page")と表
62 } 示されます。
63
64 /*
65 */
66 * この関数は、必要に応じて連続したページテーブルのブロックを解放します。
67 exit()'によって *. copy_page_tables()と同様に、4MBのブロックのみを扱います。
68 */
69
70 ///////////////////////////////////////////////////////////////////
71 // 4MBの物理メモリ領域を解放します。
72 // パラメータ'from'は4MB境界であるかどうかをチェック
73 // します。
74 // この関数はこの状況しか扱えないからです。from = 0 の場合はエラーとなります。この
75 // カーネルやバッファが占有していた領域を解放しようとしていることを示します。
76 // そして、次の式で与えられる長さに対応するページディレクトリエンタリの数を計算します。
77 // パラメータ「size」（キャリー付き4MBの倍数）、つまり占有されるページテーブルの数です。
78 // 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になります。
79 // コピーしたものを22ビット右にシフトして4MBで割る。0x3fffff(つまり4MB -1)を加えると
80 // キャリー付き整数倍の結果。
81 // 演算を行います。例えば、元のサイズ=4.01Mbであれば、結果のサイズ=2が得られます。
82 // 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タリは
83 // 算出されます。
84 // 対応するディレクトリエンタリ番号は、「from >> 20」と同じです。各エンタリが占めるのは
85 // 4バイトであり、ページディレクトリテーブルは物理アドレス0から格納されるため、実際には
86 // ディレクトリエンタリポインタは「ディレクトリエンタリ番号 << 2」で、「(from >> 20)」とな
87 // ります。"&0ffc"
88 // ディレクトリエンタリポインタが有効範囲内にあることを確認する if
89     (from & 0x3fffff)
90         panic("free_page_tables called with wrong alignment");
91     if (!from)
92         panic("Trying to free up swapper memory space")が発生します。
93     size = (size + 0x3fffff) >> 22;
94
95 // この時点でのunsigned long 解放する必要のある0xffe1; /* pgdir数=0 */ のようになります。
96 // ページのディレクトリエンタリの//,'dir'は開始ディレクトリエンタリポインタです。では、開始

```

```

// 各ページのディレクトリエントリに対してループ処理を行い、各ページのエントリを解放する
// のテーブルを順番に表示します。現在のディレクトリエントリが無効（Pビット=0）の場合、その
// ことを示す
// 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、物理メモリ
// 有効なページテーブルエントリ（Pビット=1）に対応する // ページが解放されるか、無効なページの
// 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
80 ントリが
81 // ページテーブルが処理された後、ページテーブル自体が占めていたメモリページが解放されて
82 // 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
83 // 0を返します。
84     for ( ; size-->0 ; dir++) {
85         if (!(1 & *dir))
86             を続けています。
87         pg_table = (unsigned long *) (0xfffff000 & *dir); // ページテーブルのアド
88         レスを取得する for (nr=0 ; nr<1024 ; nr++)
89         if (*pg_table) {
90             *pg_table++; // 有効なページを下位ビットを解放する。
91             .
92         }
93     } pg_table++; // 次のページを指します。
94     free_page(0xfffff000 & *dir) // ページテーブルのページを解放しま
95     す。
96 } *dir = 0; // ディレクトリエントリをリセットし
97 invalidate(); // CPUページの変換キャッシュを更新し
98 return 0; ます。
99 }

100 /*
101 * さて、ここでは中でも最も複雑な関数の一つを紹介します。それは
102 * ライナー地址の範囲を、ページのみをコピーすることでコピーします。
103 * 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:-)
104 *
105 */
106 * 注意!メモリの塊をそのままコピーするのではなく、アドレスが
107 * は、4Mb (1ページディレクトリエントリ) で割り切れるものでなければなりません。
108 * 機能をより簡単にします。どうせforkでしか使わないんだし。
109 *
110 * NOTE 2!!!from=0の時は、最初にカーネル空間をコピーしています。
111 * fork().この場合、ページディレクトリのエントリ全体をコピーすることはできません。
112 * このような場合、メモリの無駄遣いになります。
113 * 最初の160ページで640kB。これでも必要以上に多いのですが、これは
114 * メモリの使用量を増やすことはありません。
115 * 1 Mbレンジなので、ページはカーネルと共有できます。したがって
116 * nr=xxxxの場合の特別なケース。
117 */
118 //// 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。
119 // この関数は、ページ・ディレクトリ・エントリとページ・テーブル・エントリをコピーするため
// に使用されます。
120 指定されたリニア・アドレスとメモリ・サイズに // 対応する物理メモリ・エリアが

```

それらへの // は、2組のページテーブルマッピングで共有されます。コピーするときには、申請して // 新しいページテーブルを格納するために新しいページを作成すると、元の物理メモリ領域が共有されます。

// その後、2つのプロセス（親プロセスとその子プロセス）がメモリを共有するようになる
カーネルが書き込み操作を行うまでの間、// の領域は、カーネルが新しいメモリページを割り当てる

// (コピーオンライト) の書き込み操作の処理を行います。パラメータの'from' と 'to' は、リニアな
// アドレス、「サイズ」はコピー（共有）する必要のあるメモリの長さをバイト単位で表します。

```

118 int copy_page_tables(unsigned long from, unsigned long to, long size)
119 {
120     from_page_table; 符号化されてい
121     ない長さの * to_page_table; 符号
122     化されていない長さの this_page;
123     このような場合には、以下のようにな
124     ります。
125
126     unsigned long nr;
127
128     // 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。
129     // 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
130     // 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
131     // 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
132     // 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
133     // 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
134     // ートを取得します。
135     // 送信元アドレスと送信先アドレスのディレクトリエントリー pointer (from_dir と to_dir)。そ
136     // の後
137     // が占める (from >> 0x3fffff) の数 (つまり page_table の数) を計算します。
138     // パラメータで指定された copy_page_tables が成功すれば、to_page_table が
139     // 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
140     // 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
141     // 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
142     // 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
143     // ートを取得します。
144     // 送信元の開始ディレクトリエントリ pointer from_dir を取得した後、送信先の開始
145     // ディレクトリエントリ pointer to_dir と、コピーするページテーブルの数を指定すると
146     // 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
147     // 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
148     // テーブルが存在しない場合は
149     // 宛先ディレクトリのエントリがすでに存在する (P=1) 場合、カーネルはクラッシュします。ソ
150     // ースディレクトリの
151     // エントリが舞動、つまり指定された (from_dir) ポイントが存在しない (P=0) 場合は、次のページ
152     // ディレクトリエントリ (from_dir + 1) へと進み続けます。
153     // panic ("copy_page_tables: already exist")
154     // となります。
155     if (!(1 & *from_dir))
156         を続けています。
157
158     // 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
159     // 。
160     // 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。
161     // にするために
162     // 宛先の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には、以下が必要です。
163     // 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します。関数get_free_page() が
164     // が0を返した場合は、空きメモリページが得られず、メモリが不足している可能性があります。
165     // その後、-1の値を返して終了します。
166     from_page_table = (unsigned long *) (0xfffffff000 & *from_dir);
167     if (!(to_page_table = (unsigned long *) get_free_page()))
168         return -1; /* Out of memory, see freeing */.

```

```

// 次に、デスティネーションのディレクトリエントリ情報を設定します。
// にマッピングされたメモリページであることを示しています。
// 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る // コードは、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
定されている場合、そこには
//は読み書きの許可）を設定します。次に、コピーされるページアイテムの数を
// 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
//スペースの場合は、最初の160ページ（nr=160）の対応するページテーブルエントリをコピーするだ
けでよい。
140 // 640KBの物理メモリの先頭に対応しています。
141 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 // エントリです。
    *to_dir = ((unsigned long) to_page_table) | 7;
    nr = (from==0)?0xA0:1024; // 0xA0 = 160

// この時点で、現在のページテーブルに対して、指定された'nr'を周期的にコピーし始めます。
142 // メモリのページテーブルエントリ。まず、ソースページテーブルエントリの内容を取り出します。
143 // 現在のソースページが使用されていない場合（コンテンツが0の場合）、テーブルアイテムはコピ
一されない
144 // そして、次のエントリーが処理されます。
145 for ( ; nr-- > 0 ; from_page_table++, to_page_table++ ) {
    this_page = *from_page_table;
    if (!this_page)
        を続けています。
// 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
146 // (スワップデバイスにあれば!) (1 & this_page) のエントリをコピーして、宛先の
147 // ページテーブルエントリを変更し if 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。
148 // 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、
149 // 次の処理を続けます。
150 // ページエントリになります。それ以外の場合は *to_page_table = this_page;
151 // つまり、ページテーブルエントリに物理メモリページが読み取り専用に設定され、その後
152 // ページテーブルのエントリがコピー先のページテーブルにコピーされます。
153 }
154     this_page &= ~2;
155     *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
// 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ときにのみ実行されます。
// (ページ位置が1MBより大きい) となります。これは、プロセスがexecve()を呼び出したときにのみ
発生し
// 新しいプログラムコードをロードします。
// 157行目のステートメントの意味は、ソース887ページが指すメモリページが
// これは、2つのプロセスがメモリ領域を共有しているためです。もし1つの

```

プロセスの // 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が
 // 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が割り当てられる、つまり、コピー
 // 書き込み操作時に

```

156         if (this_page > LOW MEM) {...  

157             *from_page_table = this_page; // ソースの読み取り専用も設定  

158             // します。  

159             this_page -= LOW MEM;  

160             this_page >>= 12;  

161             mem_map[this_page]++です。  

162         }  

163         Invalidate(); // CPUページの変換キャッシュを更新し  

164         return 0;  

165     }  

166 }  

167 /*  

168 * この関数は、希望するアドレスのメモリにページを置きます。  

169 * ページゲットの物理アドレスを返します。  

170 * メモリ不足(ページテーブルへのアクセス時や  

171 * ページ)  

172 */  

173 */  

174 ///////////////////////////////////////////////////////////////////  

175 // 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。  

176 // 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  

177 // // 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  

178 // 。  

179 // 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  

180 // されるのは  

181 // 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  

182 // については  

183 // 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  

184 // CPUのページ変換バッファ(TLB)をリフレッシュします。  

185 // 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  

186 // 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  

187 // する必要はありません。  

188 // ここでは、Invalidate()関数を呼び出さずに、 // を表示しています。パラメータの 'page' は、次  

189 // のようなポインタです。  

190 // 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)。「address」はリニアアドレス。  

191 static unsigned long put_page(unsigned long page, unsigned long address)  

192 {  

193     unsigned long tmp, *page_table; 177  

194     /* 注意 !!!これは pg_dir=0 であることを利用しています。  

195     */  

196     // 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。  

197     // ページ位置がLOW_MEMよりも低い場合や、システムが実際に  

198     // 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。  

199     // 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  

200     // 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  

201     // は、LOW_MEMから直接始まります。次に、その上へHIGH_MEMORYが適用されたものであるかどうかを確認します  

202     // 。  

203     if (page < LOW MEM || page >= HIGH MEMORY)  

204         printk("Trying to put page %p at %p", page, address);  

205     // メモリページマップ mem_map 内の対応するバイトが設定されているかどうか、ということです。  

206     if (mem_map[(page - LOW MEM) >> 12] != 1)  

207         printk("mem_map disagrees with %p at %p", page, address);  

208     // そうでない場合は、警告が必要です。  

209     // そして、パラメータで与えられたリニアアドレスに応じて、対応するエントリポインタ  

210     // ページディレクトリテーブルの//を計算し、そこからページテーブルアドレスを取得します。
  
```

```

// 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
場合は
// 指定されたページテーブルのアドレスがそこから取得され、変数page_tableに格納されます。それ
以外の場合は
// ページテーブルに空きページを適用し、対応するフラグを設定する (7 - User, U/S, R/W)
184 //を対応するディレクトリエントリに入れ、ページテーブルのアドレスをpage_table
185 // 变数です。
186     page_table = (unsigned long *) ((address>>20) & 0xffc);
187     if ((*page_table)&1)
188         page_table = get_free_page(0xffff000 & *page_table);
189         0を返す。
190         *page_table = tmp | 7;
191         page_table = (unsigned long *) tmp;
192     }
193 // 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
// を同時に表示します。ページテーブルのエントリのインデックス値は、10ビットの値に等しい
// リニアアドレスのビット21～ビット12で構成されています。合計1024個のエントリがあります (0
-- 0x3ff)。
194 /* 無効化の必要なし */。
195     page_table[addr物理的なアドレス-0x3ff]ドロップします。
196 }
197
198 /*
199 * 前述の機能は、マークを付けたい場合にはあまりうまく機能しません。
200 * the page dirty: exec.cは、以前にページを変更したので、これを望んでいます。
201 * また、dirty-statusは (VMにとって) 正しい値であることが望れます。したがって、同じ
202 * しかし、今回はダーティなマークもつけています。
203 */
204 ///////////////////////////////////////////////////////////////////
205 // 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。
206 // 線形アドレス空間。この関数は、先ほどの関数とほぼ同じです
207 // put_page(). 223行目でページテーブルのエントリ内容を設定していることに加え、この関数は
208 // また、ページ修正フラグ (ビット6、PAGE_DIRTY) を設定します。
209 unsigned long put_dirty_page(unsigned long page, unsigned long address)
210 {
211     unsigned long tmp, *page_table;
212
213 /* 注意 !!!これは pg_dir=0 であることを利用しています。
214
215     if (page < LOW_MEM || page >= HIGH_MEMORY)
216         printk("Trying to put page %p at %p\n", page, address);
217     if (mem_map[(page-LOW_MEM)>>12] != 1)
218         printk("mem_map disagrees with %p at %p\n", page, address);
219     page_table = (unsigned long *) ((address>>20) & 0xffc);
220     if ((*page_table)&1)
221         page_table = (unsigned long *) (0xffff000 & *page_table);
222     else {
223         if (! (tmp=get_free_page()))
224             return 0;
225         *page_table = tmp|7;
226         page_table = (unsigned long *) tmp;
227     }

```

```

223 page_table[(address>>12) & 0x3ff] = page | (PAGE_DIRTY | 7); 224 /*  

無効にする必要はありません */.  

225     を返します。  

226 }  

227
    //// 書き込みページ保護を解除します。書き込み保護例外の処理に使用  

    // ページの例外（コピーオンライン）時には  

    // 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。共有されていない場合は、そのペー  

    // ジを書き込み可能な状態にしてから  

    // exit; ページが共有状態の場合は、新しいページを再適用し、書き込まれたものをコピーする必要  

    // がある  

    書き方のための//ページの内容を別にして使用している。そのため、このシェアはキャンセルされま  

    した。  

    // 全てのページは必ず物理位置での位置（アドレス）を取得します。  

228 void un_wp_page(unsigned long *table_entry)  

    // 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  

229 い場合  

230 unsigned long old_page,new_page; 231  

    // メモリLOW_MEM（メインメモリ内）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  

    // キャッシュがリフレッシュされてから返されます。つまり、メモリページが1つのプロセスでしか  

232 使用されていない場合は  

233 // 現時点ではold_pageが内LOW_MEM表記ではなくMAP_NR属性を直接変更して  

234 // 書き込み可能であれば新しいページを再適用する必要がありませを設  

235     old_page invalidate(); // 定義せず。一内の物理ページアドレスを取得。  

236         return;  

237 }
    // そうではない場合は、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  

    // 指定されたページテーブルエントリーの // を新しいページアドレスに変更し、読み取りフラグと書  

    // き込みフラグを設定します。  

238 // (u/s, r/w, p)。ページ変換キャッシュ（TLB）をリフレッシュした後、元のページ内容の  

239 // が最終的に新しいと使います。 // Out of Memory.  

240     if (!new_page==get_lowfmem_page())  

241         mem_map[MAP_NR(old_page)]--;  

242     copy_page(old_page,new_page).  

243     *table_entry = new_page | 7;  

244     invalidate();  

245 }
246
247 /*
248 * このルーチンは、ユーザーが現在のページに  

249 * を共有ページにコピーします。これは、ページを新しいアドレスにコピ  

    // ーすることで行われます。  

250 * また、古いページの共有ページカウンタをデクリメントします。

```



```

// は、より直感的に次のように表されます。
// "(0xfffffff000 & *((unsigned long *) (((アドレス>>22) & 0x3ff)<<2))))".
// (3) ページテーブルエントリのポインタ(物理アドレス)は、オフセットアドレス
// (1)のページテーブルの//に加えて、コンテンツ内の対応するページテーブルのアドレスを
// (2)のディレクトリエントリの//。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。
268     un_wp_page((unsigned long *)(
269         (((アドレス>>10) & 0xffc) + (0xfffffff000 &
270         *((unsigned long *) ((アドレス>>20)
271         &0xffc))))));
271 }
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、ページをコピーします。この関数は、メモリを検証した上で
// fork.cの34行目にあるジェネリック関数verify_area()です。
// 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。
274 void write_verify(unsigned long address)
275 {
276     unsigned long page; 277
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は共有やコピー온ライトを行わず、プログラムが書き込みを行った場合は
// 存在しないページに対して操作を行うと、システムが do_no_page() を実行するため、ページ
// fault例外が発生し、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
// ページテーブルの指定されたページのエントリーオフセット値を取得し、ページテーブルのエント
278     リー
279 // そのアドレスに対応するポインタ。与えられたリニアに対応する物理ページ
280 // のアドレスを入力してください。
281     if (!(page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
282         return;
283     page &= 0xfffffff000;
284     page += ((address>>10) & 0xffc) となります。
285 // そして、(page = *((unsigned long *) ((address>>20) & 0xffc)) )&1) とビット書き込みを行なうと、存在する
// し、ページ un\_wp\_page\(\(unsigned long \*\) page\) が書き込み可能ではなく(R/W = 0)、かつ存在している場合、シェアチェックとページコピーの
// 操作は を返すことができます。
286 // (コピー온ライト)が実行された場合は、何もせずにそのまま終了します。
287
//// 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。
// 関数 get_free_page() は、物理メモリのページを
// メインメモリ領域を取得します。この関数は、物理メモリのページを取得するだけでなく
// 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。
// 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。
288 void get_empty_page(unsigned long address)
289 {
290     unsigned long tmp; 290
291
// 空いているページが得られない場合や、撮影したページを指定した場所に配置できない場合
// のアドレスを入力すると、メモリ不足を示すメッセージが表示されます。の意味は以下の通りです
// 。
292 // 292行目のコメント: free_page()関数のパラメータ'tmp'は0にすることができます。

```

```

111 if (!(tmp=get_free_page()) || ! || ! put_page(tmp, address)) { .
291     free_page(tmp); /* 0 is ok - ignored */.
292     oom()を使用しています。
293
294 }
295 }
296
297 /*
298 * try_to_share()は、タスク "p" のアドレス "address" のページをチェックします。
299 * が存在するかどうか、そしてそれがクリーンであるかどうかを確認します。存在する場合は、
300 * 現在の
301 * タスクです。
302 * 注意! ここでは、p != current を確認した上で、彼らが
303 * 同じ実行ファイルやライブラリを共有しています。
304 */
305 static int try_to_share(unsigned long address, struct task_struct * p)
306 {
307     符号なし グ です。
308     符号なし ロン しています。
309     符号なし ロン from_page;           // プロセス'p' のページディレクトリエントリ
310     符号なし ロン to_page;           // 現在のプロセスのページディレクトリエントリ
311     // 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
312     // 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
313     // 「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
314     // (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
315     // CPUの4Gリニアアドレス空間におけるプロセスpとカレントプロセスの開始アドレスは
316     // 対応する実際のページ(=start_addrの20 & 0xffff)のアドレス'from_page'を取得します
317     // それをpageドリфт(current->start_from_page20 & 0xffff)。
318     // そこ(=address>>20) & 0xffc);           // カレントのページdirエントリ
319     // 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
320     // 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
321     // 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
322     // 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
323     // す。このメソッドは
324     // は、まずディレクトリエントリの内容を取得し、次に物理的なアドレスを取得します。
325     // に対応するページテーブルエントリポインタを計算して、その中のページテーブルの
326     // 論理アドレス「アドレス」で、ページテーブルエントリの内容を抽出し、一時的に

```

```

// physic_addrに保存されます。
316 /* is there a page-directory at from?*/
317     from = *(unsigned long *) from_page;           // ページの内容を取得する dir
318     if (!(from & 1))                                entry.
319         return 0;                                     // ページテーブルが存在しない?
320     from &= 0xfffffff000;                            // ページテーブルのアドレス
321     from_page = from + ((address>>10) & 0xffc);    .
322     phys_addr = *(unsigned long *) from_page;       // ページテーブルエントリポ
                                                               インタ。

// 次に、ページテーブルエントリがマッピングされている物理ページが存在する内容リーンであるかどうかを調べます。
// 値「0x41」は、ページテーブルエントリのD（ダーティ）フラグとP（プレゼント）フラグに対応しています。
// 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
// エントリーを作成し、それを phys_addr に保存します。最後に、この物理アドレスの有効性をチェックする必要があります。
324 // ページアドレス(phys_addr & 0x40)が0x40の最大物理アドレス値を超えてはならず、また
325 // メモリの下限値(1MB)よりも小さいこと。
326 /* ページがきれいに存在しているか? */           // 物理的なページアドレス
327     if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)
328         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
// エントリー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が要求されます。そして、更新
329 // 元々レジストリードアーズ(unsigned long*)の内容を、新ドアーズの江を指すよの内容更します。
// います

330    もし !(to & 1))
331         if (to = get_free_page())
332             *(unsigned long *) to_page = to | 7;
333 // 次に、ディレクトリエントリの使用ビテ位をアドレスを'to'に取り、さらにオフセットアドレス
334 // 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
// 対応する物理ページがすでに存在していることを確認した場合 (P=1) 、 // ページテーブルエントリ
335 .      to &= 0xfffffff000となります。           // ページテーブルのアドレス
// これは、対応する物理ページをプロセスpで共有したいという意味ですが、現在は
336 // それを吐き出す=カーネル(物理間違)と0x40とを     // ページテーブルエントリポ
// ります。                                                インタ。
337     if (1 & *(unsigned long *) to_page)
338 // アドレス'address'が既存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
// 現在のプロセスは、pプロセスのエントリをコピーします。この時点で、現在のプロセスのページ
// アドレス'address'は、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。
// pプロセスの//。
339 /* 共有する: 書き込み禁止 */ (注)

```

```

340      *(unsigned long *) from_page &= ~2;
341      *(unsigned long *) to_page = *(unsigned long *) from_page;
// その後、ページ変換キャッシュを更新し、物理ページのページ番号を計算します
// を操作し、マップバイト配列のエントリの参照を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。
342      invalidate()です。
343      phys_addr -= LOW_MEM;
344      phys_addr >>= 12;                                // メインメモリ領域のページ番号を取得しま
345     す。
346      mem_map[phys_addr]++;
347  }
348
349 /* share_page()は、ページを共有できるプロセスを探します。
350 * 現在のものです。Addressは、相対的に欲しいページのアドレス
351 * 現在のデータスペースに*。
352 */
353 *
354 * まず、実行可能かどうかを executable->i_count で確認します。
355 * 他のタスクがこのinodeを共有している場合は、>1でなければなりません。
356 */
// // 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、他の
// // 同じ実行ファイルを実行しているプロセス。この関数は、まず、他の
// // システム内で現在のプロセスと同じ実行ファイルを実行しているプロセス。
// // 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
// // で指定されたアドレスのページを共有することができます。システム内で他のタスクが実行されて
// // いない場合
// // 現在のプロセスと同じ実行ファイルで、共有ページの前提となる
// // 操作は存在しないので、この関数はすぐに終了します。
// // 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可能フィールドを比較して、その中に
// 同じ実行ファイルを実行するプロセス。
// パラメータの' inode'は、共有したいプロセスの実行ファイルのi-nodeです。
// 'address'は、現在のプロセスが望んでいるページの論理アドレスです。
// あるプロセスと共有します。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。
357 static int share_page(struct m_inode *inode, unsigned long address)
358 // 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
359 // 参照カウント値が1になっている (executable->i_count = 1) か、i-nodeポインタが空になってい
360 // る。
// 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ことを意味するか
361 // 提供されたi-nodeは無効です。そのため、共有は一切行わず、そのまま関数を終了します。 if
362     (inode->i_count < 2 || !inode)
        0を返す。
// 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。プロセスの論理アドレス'address'の方が小さい場合
// は
// 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 // 大きい値であることを示
// しています。
// 共有ページは、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ります。

```

```

// 次に、与えられたi-nodeがプロセスの実行ファイルのi-nodeと同じであるかどうかをチェックします。
// (つまり、プロセスの実行可能なフィールド)を表示します。同じでない場合は、検索を続けます。
//
// 論理アドレス'address'が開始アドレス以上の場合 LIBRARY_OFFSET
// プロセス・ライブラリ・ファイルの // をクリックすると、共有されるべきページがライブラリ・ファイルにあることを示します。
// 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ます。
363 // 同じでない場合は、検索を続けてください。
364 // プロセス'p'の'ifexecutable'または'library'の場合は検索を続いたものと同じであれば
365 // 'inode'では、try_to_share()という関数が呼ばれ、ページ共有を試みます。この関数は
366 // for (p = & LAST_TASK; current_p >= & *p; --p) { // それが現在のものであれば、検索を続
367 // を続けています。
368 // を続けています。
369 if (address < LIBRARY_OFFSET) {
370     if (inode != (*p)->executable) // 実行ファイルのi-nodeです。
371         continue;
372     } else {
373         if (inode != (*p)->library) // ライブラリファイルのi-node
374             continue;
375         if (try_to_share(address, *p))
376             return 1;
377     }
378     0を返す。
379 }
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
// のプログラムです。パラメータの'error_code'と'address'は、CPUによって自動的に生成されます
//
// page-not-present faultによりプロセスがページにアクセスしたとき。'error_code'は
// エラーの種類。'address'は例外を発生させたページのリニアアドレス。
// 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
// プロセスが動的に要求するからといって、単に物理メモリのページをマッピングするだけではありません。
383 // メモリ不足があります。共有操作が失敗した場合、不足しているデータページは次のようにしか
384 // できません。
385 // 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ます。
386 void do_no_page(unsigned long error_code, unsigned long address)
387 {
388     struct m_inode * inode;
389
// 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.
// コードの開始アドレス)が1つのプロセスのサイズ(64MB)よりも大きいことから、リニア
// アドレスが例外を発生させたプロセスの空間内にない場合は、次のコマンドを発行して終了します
389 .
// エラーメッセージが表示されます。
    if (address < TASK_SIZE)

```

```

390         printk ("");
391     if (address - current->start_code > TASK_SIZE) {...  

392         printk ("Bad things happen: nonexistent page error in do_no_page\n");
393         do_exit(SIGSEGV) です。
394     }
395
// そして、指定されたリニアアドレス'address'に従って、対応する2次ページの
// テーブルエントリのポインタが取得され、エントリの内容に応じて決定される
// 'アドレス'のページがスワップデバイスにあるかどうか。もしそうなら、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。
// 页内にありイズを使用してでれません。トリの内容を取得します。エントリの内容が
395 // がpage &= 0xffff0000; であるればlong型のaddress(20bit)で指すされた物理ページの内容 dir エントリ
// がスワップデバイスにあることを確認します。この関数は、指定されたページが
396     if (page & 1) {
397         page &= 0xffff0000;                                // ページテーブルのアドレス。
398         page += (address >> 10) & 0xffc;                  // ページテーブルエントリポイ
// ンタ。
399         tmp = *(unsigned long *) page;                   // ページエントリの内容。
400         if (tmp && !(1 & tmp)) {
401             swap_in((unsigned long *) page);           // スワップデバイスから読み込
// まれる。
402 // それ以外の場合は、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
403 // 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
404 // 対応する論理アドレスです。このようにして、特定の開始ブロック番号を計算することができます
// 。
405 実行ファ address &= 0xffff0000; とならないアドレスの中にある欠落したページのアドレス
406         tmp = address - current->start_code;          // ページの論理アドレス。
407
// 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があるので
// 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、ページが存在しないのはプロセスが原因です。
// 動的に要求されたメモリページデータにアクセスしているので、対応するi-nodeはありません。
// とデータブロック番号（どちらもNULLに設定されています）。
// 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
407 ます。なぜなら、各
408 // データブロックの長さがBLOCK_SIZE(4KB)である場合、1ページのレコードは4ブロック分のデータが
409 // 格納できます。 block = 1 + (tmp-LIBRARY_OFFSET) / BLOCK_SIZE;
410 // データです。プロセスの論理アドレス「tmp」をデータブロックサイズ+1で割ると
// 実行イメージファイルの大損ページの開始ブロック番号'block' if (tmp >=
// LIBRARY_OFFSET) {

```

```

411         inode = current->executable;           // 実行ファイルのinodeとblock no.
412         ブロック = 1 + tmp / BLOCK_SIZE;
413     } else {
414         inode = NULLです。                  // 動的に適用されるページの場合
415         ブロック = 0;
416     }
417 // 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
418 // が発生した場合
419 // 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。
420 // 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
421 // ジがスコープ内の
422 // 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
423 // ます。
424 // 成功した場合は、物理メモリのページを申請する必要があります。
425 // その後、実行するnodeページのaddress論理アドレスtmpへ。
426 // プロセススペースの論理アドレスtmpへ。
427 // デバイスから読み込んで配置（マッピング）する
428 // 無料ページに応募する。
429 // 1ブロックがヘッダーに使用されることを覚えておいてください。
430 // このブロック番号と実行ファイルのi-nodeに基づいて、デバイスを見つけることができます。
431 // 対応するブロックデバイスの論理ブロック番号(nr[]配列に格納)をマッピングしたもの
432 // ビットマップを作成し、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みます。
433     for (i=0 ; i<4 ; block++, i++)
434         nr[i] = bmap(inode, block);
435     bread_page(page, inode->i_dev, nr)で
436 // す。
437 // 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
438 // 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
439 // 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。
440 // 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、読まれません
441 // 実行ファイルイメージ内のファイルから//が読み込まれるのではなく、ライブラリファイルから//が読
442 // み込まれるので
443 // クリア操作の実行の必要はcurrentでend_data;          // 余分なバイトの長さを
444 // if (i>4095)                                         // 最後から1ページ以上の
445 //     i = 0;
446 //     tmp = page + 4096;                                // ページの最後を指します。
447 //     while (i-- > 0)                                    // iバイトは、ページの終わりから始まり
448 //     {...                                                 // ます。
449 //         tmp--;
450 //         *(char *)tmp = 0;
451 //     }
452 // 最後は、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
453 // る
454 // のアドレスになります。操作が成功した場合は戻り、そうでない場合は、メモリページが
455 // メモリが不足してpageをaddresspageをaddressがリリースされました。
456 // を返すことができます。
457 // free_page(page)です。
458 // oom()を使用しています。
459 }
460 /////
461 // 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つずつデクリメントされます。バイト項目が0の場合は、対応するページがあることを示し
// ます。
// バイト値が1以上の場合は、ページが占有されていることを示します。
// または複数のプロセスで共有されます。
// 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]となります
// 。
// となり、合計3072の物理ページが割り当て可能となります。範囲は0~1MBのメモリ
// カーネルのための領域です。
// パラメータ' 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。
// ページの割り当てを行います (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。
443 void mem_init(long start_mem, long end_mem)
444 {
445     int i; 446
        // この関数は、まず、メモリマップされたバイト配列の項目を、すべてのページに対応する
447     // 1MB~16MBの範囲を占有状態つまり全バイトをUSED(100)にします (16MB) を設定
        // PAGING_PAGESは、(PAGING_MEMORY>>12)として定義されしますればすべての物理メモリの数である。
448     // 1MB以下の領域 (15MB/4KB=3840) ; i++) // paging_pages = 3840.
449     // 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
        // 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 (表示は
        // idle)になります。16MBの物理メモリを持つシステムでは、4MB~16MBのメインメモリに対応する
        // バイトが表示されます。
450     // mem_map[]内のNR領域がクリアされます。 // MMAの冒頭にあるページ番号。
451         end_mem -= start_mem;
452         end_mem >>= 12; // MMAの総ページ数。
453         while (end_mem-->0)
454             mem_map[i++]=0 // MMAページのバイトがリセットさ
455         } れる。
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
        // 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 186 行目で呼び出されま
        // す。
        // chr_drv/keyboard.S プログラムは、「Shift + Scroll」でシステムメモリの統計情報を表示しま
        // す。
        // Lock "キーを押します。
457 void show_mem(void)
458 {

```

```

459     int i, j, k, free=0, total=0;
460     int shared=0;
461     unsigned long * pg_tbl;
462
// まず、バイト配列mem_map[]に従って、メインのページの合計数を数えます。
// メモリ領域'total' と空きページ数'free' と共有ページ数
// printf("Mem-info:$n$r")を表示します。
463     for(i=0 ; i< PAGING_PAGES ; i++) {
464
        if (mem_map[i] == USED)           // 割り当てのないページ
465            continue;
466
        total++;                         // total++です。
467
        if (! mem_map[i])
            free++;                      // メインメモリの空きページ。
468
       その他
469            shared += mem_map[i]-1;    // 共有ページ（バイト値>1）。
470
471    }
472
473     printf("%d free pages of %d$%r", free, total);
474     printf("%d pages shared$%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
// 対応する2次ページテーブルが存在する場合、2次ページテーブルが占有するメモリページは次のようになります。
475 // ページテーブル自体がまずカウントされ (484行用) その後に物理メモリページの統計値をカウント
476 // されます for(i=4 ; i<1024 ; ) {
477 // ページテーブルの各ページ(pg_dir)に対する // がカウントされます。
478 // ((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
479 // 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。
480 // 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
481 // エントリーの処理を繰り返す) pg_dir[i]> HIGH_MEMORY) { // 内容が異常です。
482
483 // 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
484 // 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。
485 // 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。
486 // 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
487 // ページテーブルのすべての項目の内容を表示しています。表示される物理ページが
488 // 現在のページテーブルエントリが存在し、物理ページの "アドレス" がLOW_MEMよりも大きい場合
489 // の場合は、ページテーブル if((pg_dir[i]> LOW_MEM) が統計値に含まれます。
490 // (pg_dir[i]> LOW_MEM) で占有されているページ
491 // pg_tbl=(unsigned long *) (0xfffff000 & pg_dir[i]) となります。
492 // for(j=0 ; j<1024 ; j++)
493 // if ((pg_tbl[j]> LOW_MEM) && pg_tbl[j]> HIGH_MEMORY)
494 // (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
495 // 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。
496 // 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
497 // ージ

```

ページテーブル項目の//は統計値に含まれます)。

```

488             if (pg_tbl[j] > HIGH_MEMORY)
489                 printk ("page_dir[%d][%d]: %08X\n", i, j, pg_tbl[j]) となります。
490
491             その他
492             k++, free++; // ページテーブル項目のページ。
493         }
494
495         // 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
496         // します。
497         // 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、プロセスが占有する
498         // ページテーブルが
499         // タスク構造がカウントされます。この時点でk=0であれば、タスク構造に対応するプロセスが
500         // 賽奪⑩個の次のアロセスが占有するスペースを確保しなける危機を切りぬけました(終了して
501         // ます)。
502         // そして、対応するプロセス番号と物理ページ統計値kの後に
503         if (!(i&15) && k) { // k != 0 はプロセスが存在することを示す。
504             k++, free++; /* task_struct のための1ページ/1プロセス */。
505             printk ("Process %d: %d pages\n", (i>>4)-1, k);
506             k = 0;
507         }
508
509     }
510
511     // 最後に、システムで使用されているメモリページと、メインのページの合計数
512     // のメモリ領域が表示されます。
513     printk ("Memory found: %d (%d)\n", free-shared, total);
514 }
```

3. page.s

1. 機能

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。

2. コードアノテーション

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 */
7 */
```

```

8 * page.s には、低レベルの page-exception コードが含まれ  

9 // 実際の作業は mm.c で行われます。 // memory.c
10 */
11
12 // 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。
13 .globl _page_fault      グローバル変数として # 宣言され
14                               ています。
15 _page_fault:
16     xchgl %eax, (%esp)    # エラーコードをEAXに取り込み
17     pushl %ecx            ます。
18     pushl %edx
19     push %ds
20     push %es
21     movl $0x10, %edx      # カーネルデータのセグメントセレクタを
22     mov %dx, %ds          設定します。
23     mov %dx, %es
24     mov %dx, %fs
25     movl %cr2, %edx      # ページ例外の原因となったリニアアドレスを取得します。
26     pushl %edx            # 呼び出される関数の引数として、線形アドレスとエラーコードが
27     pushl %eax            # スタックにpushされます。
28 // "ページが存在する"というフラグP(ビット0)をテストし、例外であればdo_no_page()関数を呼び出す
29 // ページfaultにより例外が起きた場合、do_no_page()が呼び出されます。
30     jne 1f
31     call _DO_NO_PAGE    # mm/memory.c, line 381.
32 1:   jmp 2f
33 2:   call _do_wp_page  # mm/memory.c, line 254.
34     addl $8, %esp
35     popl %fs
36     popl %es
37     popl %ds
38     popl %edx
39     popl %ecx
40     popl %eax
41     iret

```

4. swap.c

1. 機能

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、具体的な保存場所やマップの位置を

スワップされたメモリページ。

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、include/linux/mm.hという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));
#define write_swap_page(nr, buffer) ll_rw_page(WRITE, SWAP_DEV, (nr), (buffer));
```

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SWAP_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すると、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）となる。swap_size - SWAP_BITSの範囲のビットマップのビットを0, 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。スワップビットマップは、ラミオズ上に対応するスワップページが付与されず、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っているはずです（斜線部分はスワップに利用できません）。

13.4.2 コードアノテーション

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 /*
7 */
8 * このファイルには、ディスクとのスワップを行うほとんどのものが含まれています。
9 * 18.12.91開始
10 */
11
// <string.h> 文字列のヘッダーファイルです。文字列操作に関するいくつかの組み込み関数を定義しています。
// <linux/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。
// <linux/kernel.h> カーネル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います。
// いくつかのセレクタ定数と一緒に
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネルの使用する機能
12 #include <string.h>
13
14 #include <linux/mm.h>
15 #include <linux/sched.h>
16 #include <linux/head.h>
17 #include <linux/kernel.h> (日本語)
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
// のビットマップは32,768ページまで管理でき、1ページ分のメモリに対応しています。
// から128MBのメモリ容量を持つ。このビットがセットされると、対応するスワップ
// ページはアイドルです。
19 #define SWAP_BITS (4096<<3) // 1ページ内のスワップ合計ビット数を定義 (32768). 20
// bitop()は、ビット操作用のマクロです。異なる「op」を与えることで、3つの「op」を定義することができます。
// 指定したビットをテスト、設定、クリアする操作。
// 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、'nr'番目のビットの値を
// 与えられたアドレス'addr'をキャリーフラグに入力し、ビットをセットまたはリセットしてキャリー
// 一を返す
// フラグの値（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 - Bit test and set, and the original bit is sets to
// carry.
// "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する。
// 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ーの初期値 (0) です。
// インライン・アセンブリ・コードは、ベース904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
// キャリービット付きで、キャリービットCFに応じてオペランド(%0)を設定します。CF = 1の場合、
// リターン

```

```

// レジスタの値=1、そうでなければリターンレジスタの値=0となります。
21 #define bitop(name, op) ♪ ♪ ♪
22 static inline int name(char * addr, unsigned int nr)
"^\w^、)
24 int res; "^\w^"
25 asm volatile ("bt" op "%1,%2; adc1 $0,%0" ) 26
: "=g" (res)
27 : "r" (nr), "m" (*(addr)), "0" (0)); □。
28 return res; "^\w^"
29 }
30

// 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。
31 bitop(bit, "") // 定義 bit(char * addr, unsigned int nr)
32 bitop(setbit,
"") // 定義 setbit(char * addr, unsigned int nr)
33 bitop(clrbit,
35 static char * swap_bitmap = NULL;
36 int SWAP_DEV = 0; // カーネルの初期化時に設定されたスワップデバイスの番号です。
37
38 /*
39 * task[0] - カーネルメモリ内のページをページングすることはありません。
40 * 私たちは、他のすべてのページをページングします。
41 */
42 // 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。
43 #define FIRST_VM_PAGE (TASK_SIZE)>>12) // = 64mb/4kb = 16384
44 #define LAST_VM_PAGE (1024*1024) // = 4gb/4kb = 1048576
45 #define VM_PAGES (LAST_VM_PAGE - FIRST_VM_PAGE) // = 1032192 (0からのカウント)。

//// 応募して、スワップのページ番号をゲット。
// スワッピングビットマップ全体(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、リセットします。
// 最初に見つかったビットの位置(現在のフリースワップページ番号)を返します。戻り値
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。
46 static int get_swap_page(void)
47 {
48     int nr;
49
50     if (!swap_bitmap)
51         0を返す。
52     for (nr = 1; nr < 32768 ; nr++)
53         if (clrbit(swap_bitmap, nr))
54             nrを返す。 // 現在のフリースワップのページ番号を返します。
55
56 }
57 // スワップデバイスで指定されたswap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。
// 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。そのため
// オリジナルビットが1の場合は、スワップデバイスのオリジナルページが
// が占有されていないか、ビットマップにエラーが発生しています。そして、エラーメッセージが表示され、返されます。
58 void swap_free(int swap_nr)
59 {
    を返す
    ことが
    できま
    す。

```

```

62     if (swap_bitmap && swap_nr < SWAP_BITS)
63         if (!setbit(swap_bitmap, swap_nr))
64             を返すことができます。
65         printk ("Swap-space bad (swap_free())……");
66         return;
67     }
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
// 要求されたメモリページを同時に、スワップビットマップの対応するビットを変更する
// (セット)し、ページテーブルのエントリの内容を変更し、メモリページを指すようになります。
// そして、対応するフラグを設定します。
69 void swap_in(unsigned long *table_ptr)
70 {
71     int swap_nr;
72     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します。もし、スワップ
// ビットマップが
// 存在しないか、指定されたページテーブルエントリーに対応するページがすでに存在する場合
// メモリ内に // があるか、スワップページ番号が 0 である場合、警告メッセージを表示して終了しま
// す。
// スワップデバイスに配置されたメモリページに対して、対応するページテーブルの
74 // エントリ(swap bit map)ページ番号*2、つまり (swap_nr << 1)でなければなりません。111行目の説
// 明を参照してください。
75 // 以下の関数 try_swap_in("trying to swap in without swap bit-map")
    .
76         を返すことができます。
77     }
78     も (1 & *table_ptr) {
79     }
80     swap_nr = *table_ptr >> 1;
81     if (!swap_to_use)
82         .
83         .
84         printk ("No swap page in swap_in\n");
85         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
物理ページへの // エントリーポイントを設定し、ページの修正、ユーザーの読み取り可能性、書き
87     .
88     .
89     .
90     .
91     .
92     .
93 }     *table_ptr = page | (PAGE_DIRTY | 7);
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は、スワップデバイスに保存する必要はありません。
// 対応するページは、対応する画像ファイルからも直接読み取ることができます。

```

```

// そのため、対応する物理ページを直接リリース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
// を対応するページテーブルエントリに残しておく必要があります。また、ページテーブルエントリ
// パラメータの table_ptr は、ページテーブルエントリへのポインタです。の場合は1を返します。
// ページが正常にスワップまたはリリースされた場合は 0 を返します。
95 int try_to_swap_out(unsigned long * table_ptr)
96 {。 符号付きロングページ、
97     符号付きロング swap_nr
98
99     .
100    // 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
101    // スワップアウトされるべき // が存在しない（または無効である）場合、コードは終了します；物理
ページアドレス
102    // ページテーブルのエントリで指定された値が、メモリ管理されたハイエンドのPAGING_MEMORYより
も大きい場合。
103    // (15MB)でも終了しています。
104    page = *table_ptr;
105    if (!(PAGE_PRESENT & page))
106        return 0;
107    if (page - LOW_MEM > PAGING_MEMORY)
108        0を返す。
109    // 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
110    // このようなページは &0xfffffff000; されるべきではないので、この関数は物理的なページアドレスを取
111    // 0で存在する。それ以外の場合 NR(page)アボート番号を取得します。それをページテーブルのエント
リに保存します。
112    // ページをスワップアバートし、対応する物理メモリページを解放する。スワップページ番号を取
113    // 得します。
114    // 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
115    // 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ピット (P、ビッ
ト0) を解放するためです。
116    // 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
117    // P = 0 (無効なページ)、その他のビット (ビット31~1) は自由に使用できます。ページ書き込み
機能
118    // write_swap_page(nr, buffer) は、CPUの変換記憶装置 (WRITER_SWAPDEV, ます nr), (buffer) として定義され
119    // ています。
120    // linux/mm.h ファイルの12行目です。
121    write_swap_page(swap_nr, (char *) page);
122    // grab_page(pag swap_nr << 1;
123    // を返します。
124    // それ以外の場合は、ページが変更されていないことを意味するので、交換する必要はありません
。
125    // となっていますが、直接リリースすることも可能です。
126    *table_ptr = 0;
127    invalidate();
128    free_page(page);
129
130    return 1;
131
132 /*
133 * OK、これはかなり複雑なロジックを持っています - アイデアは、良好な
134 * そして、高速なマシンコード。それを気にしなければ、物事は

```

126 *はもっと簡単です。

127 */

```
// スwap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。
// 2つの静的なローカル変数は、一時的な格納のために使用されます。
// 次の検索を開始するための現在の検索ポイント。この関数は、次のように呼び出されます。
// get_free_page() (172行目で定義)の中で。
```

128 int swap_out(void)

```
129 {    static int dir_entry = FIRST_VM_PAGE>>10; // 16, タスク1の最初のディレクトリエント
130     リ static int page_entry = -1;
131     int counter = VM_PAGES; // 1032192, 44行目を参照。 int pg_table;
132
133
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
// 続けて次のディレクトリエントリを確認します。適切な（既存の）ページディレクトリが
135 // while (counter>0) {...
```

```
136         pg_table = pg_dir[dir_entry];           // ディレクトリエントリの内容。
137         if (pg_table & 1)                      // テーブルが存在する場合はループを終
138             break;                         了します。
139         counter -= 1024;                     // 1ページのテーブルには1024個の
140         dir_entry++;                      // アイテムがあります。
141         if (dir_entry >= 1024)                 // 次のディレクトリ・エントリ。
142             dir_entry = FIRST_VM_PAGE>>10;
143     }
```

// 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

// try_to_swap_out()は、ページテーブルにある1024ページ全てに対して1つずつ呼び出され、スワップを試みます。

// 並んでおむす。ページがリスエップ引の処理で失敗した場合は、警告された場合を戻しも返します

```
144 // すべてのページアドレス0xfffff000;           // ページテーブルポインタ（アド
145     while (counter-- > 0)                   // レス）。
146     {.                                         // ページテーブルのエントリ（初期値
// 現在のページアドレスまで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
// スワップアウト可能なページ、つまりページテーブルアイテムのインデックス番号の発見に成功する
// が1024以上の場合は、前の135～143行と同じように使用します。
// 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する
147 // 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す
148     if (page_entry >= 1024) {
149         page_entry = 0;
150         を繰り返しています。
151         dir_entry++です。
152         if (dir_entry >= 1024)
153             dir_entry = FIRST_VM_PAGE>>10;
154             pg_table = pg_dir[dir_entry]; // ページディレクトリエントリの内容
155             if (!pg_ifake&1) counter -= 1024) > 0)
156                 goto リピート。
157             その他
158             ブレークします。
```

```

159             pg_table &= 0xfffffff000; // ページテーブルの取得 ponter.
160         }
161         if (try\_to\_swap\_out(page_entry + (unsigned long *) pg_table))
162             return 1;
163     }
164     printk ("Out of swap-memory\n");
165     return 0;
166 }
167 */
168 /*
169 * 最初の（実際には最後の）フリーページの物理的なアドレスを取得し、それをマークします。
170 * を使用しています。フリーページが残っていない場合は0を返します。
171 */
172 /**
173 ///////////////////////////////////////////////////////////////////
174 // メインメモリ領域の空き物理ページを取得します。
175 // 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ってから
176 // 申し込みをして、再度ページを取得しようとします。
177 // 入力: %1(ax = 0) ; %2(LOW_MEM) バイトビットマップで管理されているメモリの開始位置。
178 // %3(cx = PAGING_PAGES); %4(edi = mem_map + PAGING_PAGES - 1)。
179 // 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。
180 // 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[].Thisの最後のバイトを指して
181 // います。
182 // この関数は、すべてのページフラグをビットマップの最後から逆方向にスキャンします（ページの
183 // 合計数は
184 // がPAGING_PAGESの場合）、ページが空いていればページアドレスを返します（ビットマップバイト
185 // は0）。
186 // 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
187 // 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
188 // 下のようになっています。
189 // 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されます。
190 // もちろん、このプログラムで
191 // カーネル用のこの関数では、マッピングに put_page() を使用する必要がありません。
192 // 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ます
193 // 。
194 // 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eaxレジスタに保存されます。
195 // を使用することで、効率的なアクセスと操作が可能になります。この変数の定義方法は、主にイ
196 // ンラインの
197 // アセンブリファイルです。
198 unsigned long get\_free\_page\(void\)
199 {
200     register unsigned long res asm("ax");
201
202     asm ("std ; repne ; scasb\n\t") // アル(0)と各ページ(デイ)の内容を比較してみまし
203     // ます、メモリバイドのビットマップで値が0のバイトを調べて、(デイ)アじよす。
204     // 対応する物理メモリページの結果として得られるページアドレスが、実際の
205     // movb $1, (%edi)In\n\t // 結果として得られるページアドレスがラベルにジャンプします。
206     // 物理メモリの容量をもう一度探してみてください。一度探してみてください。一度探してみてください。
207     // movl $12, %ecx\n\t // バイト[1+edi]をページの上に設定します。
208     // sopl %2, %ecx\n\t // ページ最初の4K = ページの相対アドレスのアドレスを返し
209     // の操作をしてから、もう一度調べてみます。 // LOW_MEMを加えて絶対的なページアドレスを得る。
210     // addl %2, %ecx\n\t // モリページをゼロにします。
211     // movl %%ecx, %%edx\n\t // edxレジスタにページ開始アドレスをロードしま
212     // 回繰り返す。 // メモリバイドのビットマップを使って、空きページを探すmem_map[]。
213     // movl $1024, %%ecx\n\t // 1024のカウント数でecxをロードします。
214     // leal 4092(%edx), %%edi\n\t // 1024のカウント数でediをロードします。
215     // rep ; stosl\n\t // ediレジスタ内の各ワードをトリセッ(4092ケル振)をロー
216     // dします。
217     // movl %%edx, %%eax\n\t // ページ開始アドレスをeaxにロードします
218     // "1:\n\t : =a" (res)

```

```

189      : "0"(0)、"i"(LOW MEM)、"c"(PAGING PAGES)
190      となります。
191      "D"(mem_map+PAGING_PAGES-1)
192      if ("d"res >> HIGHMEMORY)    ページがメインページの領域外にある場合は、// 再検索
193          goto repeat;           します。
194      if (! res && swap_out())   // 無料のページが見つからない場合は、スワップを行い、
195          goto repeat;           再度検索します。
196      return res;              // 自由なページのアドレスを返す。
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。
// 配列（ブロックの配列）を作成し、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
// スワップページのビットマップ配列swap_bitmap[]を格納し、スワップ管理ページを読み出すメモリの
デバイスのスワップパーティションから // （最初のページ）をスワップビットマップアレイに入れ
ます。その後、慎重に
// 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チェックして、最後にリターンします。
199 void init_swapping(void)
200 {
    // Blk_size[]は、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す。
    // メジャーとマイナーの各項目はblk_size[MAJOR(SWAP_DEV)]とblk_size[MINOR(SWAP_DEV)]で、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します
    // システムにスワップデバイスの番号が定義されていない場合、コードは戻ります。
    // ブロックの配列を設定しない場合は、警告メッセージを表示して返します。
    if (! blk_size[MAJOR(SWAP_DEV)]) {...}
    printk("Unable to get size of swap device\n");
    // 表示されます。
    // を返すことができます。
206
    // 次に、swapのswapパーティションにあるブロックの合計数' swap_size' を取得して確認します。
    // デバイスになります。0であればリターンします。ブロックの総数が100未満であれば、警告の
    // のメッセージが表示され、終了します。
210     swap_size = blk_size[MAJOR(SWAP_DEV)][MINOR(SWAP_DEV)]となります。
211     if (!swap_size)
        // を返すことができます。
213     if (swap_size < 100) {...}
        // Swap Device Too Small (%d Blocks) ..., swap_size)。
215     // を返すことができます。
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。
// ページ（ブロック/4）です。この値は、SWAP_BITSが使用できるページ数よりも大きくすることはで
きません。
// (32768)を表す。次に、スワップビットマップ配列' swap_bitmap' を格納するための空きメモリペー
217 ジを取得します。
218 // 各ビットが1つのスワップページを表します
219
    if (swap_size > SWAP_BITS)
        swap_size = SWAP_BITS;
    swap_bitmap = (char *) get_free_page();
    if (! swap_bitmap) {...}
        printk("Unable to start swapping: out of memory :-)\n");
        (^); return;

```

```

224      }
// 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」。機能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す。そこで、警告メッセージを表示し、取得したばかりのメモリページを解放
// して終了します。
// 関数を実行します。それ以外の場合は、メモリページ内の機能文字列（10バイト）がクリアされま
225 す。
226 // Read_swap_page(0, swap_bitmap)は、swap_bitmap+4086,10)linux/mm.hの11行目で定義されている
// セクションです。
227     printk ("Unable to find swap-space signature\n");
// と表示されます。
228     free_page((long) swap_bitmap);
229 }
230     swap_bitmap = NULL;
231 // を返すことができます。
232     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チェックします。もし、そのビット
// が
// ビットマップが0の場合は、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）であることを示していま
// 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0（占有）に初期化されるべきです。
233 // 対応するスワップページ以下、ビットマップをチェックする場合、2つのステップでチエッ
// クします。
234     if (i == 1)
// 利用できない部分と利用できる部分に基づいて
235     // 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。
236     if (!bit(swap_bitmap, i))
237     // 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ります。エラーメッ
// セージは
238     printk ("Bad swap-space bit-map\n");
239 // と表示され、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。
240     // を返すことができます。
241 }
242 // 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
// 統計によると、空きのあるスワップページがない場合、問題があるということです
// スワップ関数なので、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。
// そうでなければ、スワップデバイスは正常に動作し、スワップページ数とスワップの総バイト数は
243 // スペースが表示されま
244    す。 j = 0;
245     for (i = 1 ; i < swap_size ; i++)
246         if (bit(swap_bitmap, i))
247             j++;
248     if (!j) {
249         free_page((long) swap_bitmap);
250         swap_bitmap = NULL;
251         // を返すことができます。
252     }
253     printk ("Swap Device ok: %d pages (%d bytes) swap-space\n", j, j*4096) となります。

```

[254](#)

13.5 まとめ

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。

14 ヘッダーファイル (include)

プログラムは、関数を使用する前に、まず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 <ファイル名>
```

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",;,"^w^")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、ヘッダー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、該当するヘッダー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ます。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、リンクがプログラムのすべてのオブジェクト・モジュール（ライブラリ・ファイル内のモジュールを含む）を結合します。

使って、実行可能なプログラムを形成します。

標準的なCライブラリには、約15個の基本的なヘッダー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の詳細については

は、Plauger氏の著書「The Standard C Library」に掲載されています。

本書で紹介するカーネルのソースコードにおいて、関係するヘッダーファイルは、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ー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ーファイルは、いくつかの特殊なヘッダーファイル（ロックデバイスのヘッダーファイル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ー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。⁹¹³

カーネルコードです。

カーネルバージョン0.95からは、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 ディレクトリには、カーネルをスムーズにコンパイルするためのヘッダーファイルが用意されています。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 include/ ディレクトリのファイル

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ーファイルと共に存しています。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ーファイルを置き換えるために、いくつかのヘッダーファイルを書き換えようとしたしました。そのため、これらのカーネルソースに含まれるヘッダーファイルは、開発環境のヘッダー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ーファイルは、通常、標準Cライブラリのヘッダー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ー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ー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、使用するコンピュータ・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ため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、インテル社のCPUポートIOアセンブリマクロ

`linux/`ディレクトリには、Linuxカーネルプログラムで使用されるヘッダーファイルがあります。`linux/` ディレクトリには、スケジューラが使用するヘッダーファイル`sched.h`、メモリ管理ヘッダーファイル`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ー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ー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ー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ーファイルは、linux/include/ディレクトリ内に移動しています。

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個名ヘッダーファイルが存在します。次節からは、まず`include/`ディレクトリにある13個のヘッダーファイルを説明し、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。

	A. OUT. H	6047 バイト	1991-09-17 15:10:49
	const. h	321 バイト	1991-09-17 15:12:39
	ctype. h	1049 バイト	1991-11-07 17:30:47

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

2. A.OUT.H

1. 機能

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 (Assembly 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ため、ファイルは3つの部分に分けられます。

- 1-108行目では、オブジェクトのヘッダー構造と関連するマクロ定義を示し、説明しています。
- 109~185行目は、シンボルエントリの定義と構造の説明です。
- 186-217行目では、再配置テーブルエントリの構造を定義し、説明しています。

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、その他の部分から構成されています。実行ファイルのヘッダ部には、主に exec 構造体があり、その中には以下の情報が含まれています。

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、コードの実行開始位置などの情報です。カーネルはこの情報をを使って実行ファイルをメモリにロードして実行し、リンク (ld) はこれらのパラメータを使ってモジュールファイルの一部を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 のヘッダーファイル 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、そのフィールドはさらに以下のように分かれています。

の3つのサブドメインに分かれています。フラグ」、「マシンタイプ」、「マジックナンバー」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対応するマクロN_MACHTYPEとN_FLAGSが定義されています。

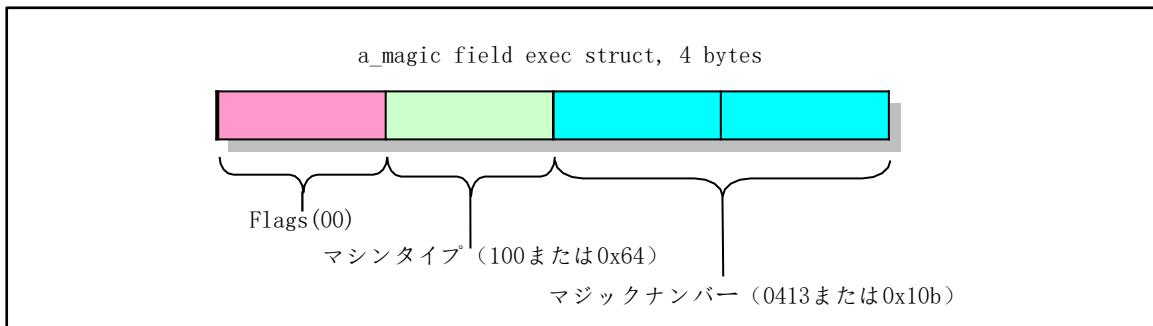


図14-1 exec構造体の最初のフィールド `a_magic(a_info)`

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

0x0b, 0x01, 0x64, 0x00

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。

0x0b, 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ルート・ファイル・システムの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。

2. コードアノテーション

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、主に実行構造を定義した文書の最初の部分です
// オブジェクトファイルと関連する操作のマクロの//。
7 //unsigned long magicファイルのマジック構造を定義するマクロ。N_MAGIC等を用いています。
8 struct _exec {a_text;
/* テキストの長さをバイト単位で表示します。

```

```

9  符号なし a_dataです。          /* データの長さをバイト単位で表示します。
10 符号なし a_bssです。          /* ファイルの初期化されていないデータ領域の長さをバイト単
11 符号なし a_syms;           単位で示しています。
12 符号なし a_entryです。        ファイル内のシンボル・テーブル・データの長さ（バイト単位）
13 }, 符号なし a_trsize;       /* */。
14 , 符号なし a_drsizesです。   開始アドレス */ /* スタート
15 // 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。
16 // 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。
17 #ifndef N_MAGIC
18#define N_MAGIC(exec) ((exec).a_magic)
19#endif
20
21 #ifndef OMAGIC
22 /* オブジェクトファイルや不純な実行ファイルを示すコードです。*/
23 // 歴史的には、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であった
24 // (0x107). 歴史的には、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
25 // 実行ファイルのヘッダー構造の先頭にあった // 0407番 (0x107)。
26 // 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。
27 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
28 とができます。
29 実行ファイルをメモリに入れた後の命令の//。するプログラムはありません。
30 // はこの方法を採用していますが、この8進数は識別のためのフラグ（マジックナンバー）として残
31 // されています。
32 // ファイルの種類です。OMAGIC』は「オールドマジック」と考えられます。
33 #define OMAGIC_0407
34 /* 純粋な実行ファイルを示すコード*/
35 // 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。
36 #define NMAGIC_0410 // 0410 == 0x108
37 /* デマンドページングされた実行ファイルを示すコード*/
38 // 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。
39 #define ZMAGIC_0413 // 0413 == 0x10b 28#endif /* not
OMAGIC */.
40 // QMAGICもあり、これはディスク容量を節約し、ヘッダー構造を保存するために使用されます
41 // ディスク上のファイルの//やコードをコンパクトにまとめてくれます。
42 // 以下のマクロ (Bad Magic) でマジックナンバーの正しさを判定します。
43 #define N_BADMAG(x) 表示されます。マジックナンバーが認識できない場合は、trueを返します。
44 #if !N_MAGIC(x) || !BADMAG(x) && !ZMAGIC(x) != 0
45 #endif
46
47 #define N_BADMAG(x) 表示されます。マジックナンバーが認識できない場合は、trueを返します。
48 #if !N_MAGIC(x) || !OMAGIC(x) && !N_MAGIC(x) != 0
49 #endif
50
51 // ヘッダー構造の終わりまでの残りの長さを与えるマクロ定義です。
52 // と1KBの位置関係になっています。
53 #define N_HDROFF(x) (SEGMENT_SIZE - sizeof (struct exec))
54
55 // 以下のマクロは、.oを含むオブジェクトファイルの内容を操作するために使用されます。
56 // モジュールファイルと実行ファイル。
57
58 // コード部分の開始オフセットです。

```

```

// ファイルのタイプがZMAGIC、つまり実行ファイルの場合、コード部分は1024バイトの
// 実行ファイルからのオフセット。それ以外の場合は、コード部分は実行の終わりから始まる。
// モジュールファイル（OMAGICタイプ）であることを示す、 // ヘッダー構造 (32バイト) .
42 #ifndef N_TXTOFF
43 #define N_TXTOFF(x) [ ]。
44(N_MAGIC x) == ZMAGIC ? N_HDROFF((x))+ sizeof (struct exec) : sizeof (struct exec)) 45
#endif
46
    // 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。
47 #ifndef N_DATOFF
48#define N_DATOFF(x) (N_TXTOFF(x) + (x).a_text)
49#endif
50
    // 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。
51 #ifndef N_TRELOFF
52#define N_TRELOFF(x) (N_DATOFF(x) + (x).a_data)
53#endif
54
    // 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。
55 #ifndef N_DRELOFF
56#define N_DRELOFF(x) (N_TRELOFF(x) + (x).a_trsize)
57#endif
58
    // 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
    .
59 #ifndef N_SYMOFF
60#define N_SYMOFF(x) (N_DRELOFF(x) + (x).a_drsize)
61#endif
62
    // シンボルテーブルの後の、文字列情報のオフセットです。
63 #ifndef N_STROFF
64#define N_STROFF(x) (N_SYMOFF(x) + (x).a_syms)
65#endif
66
    // 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。
67 /* ロードされた後のメモリ上のテキストセグメントのアドレス。
68 #ifdef NTTEXTADDR 0                                // コードセグメントは、アドレス0から始ま
70#endif                                              ります。
71
72 /* ロードされた後のメモリ上のデータセグメントのアドレス
73     なお、ここに記載されていないマシンでは、
74     SEGMENT_SIZEの定義は任意となります。*/
75 #if defined(vax) || defined(hp300) || defined(pyr)
76#define SEGMENT_SIZE PAGE_SIZE
77#endif
78 #ifdef hp300
79 #define PAGE_SIZE      4096
80#endif
81 #ifdef sony
82 #define SEGMENT_SIZE 0x2000
83#endif
84 /* Sony. */ #ifdef
85 #define SEGMENT_SIZE 0x20000

```

```

86 #endif
87 #if defined(m68k) && defined(PORTAR)
88 #define PAGE_SIZE_0x400
89 #define SEGMENT_SIZE PAGE_SIZE
90 #endif
91
// ここでは、カーネルがメモリページを4KB、セグメントサイズを1KBと定義しています。
// なので、上記の定義は使われていません。
92 #define PAGE_SIZE_4096
93 #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。
// ファイルがOMAGICタイプの場合、データセグメントはコードセグメントの直後にあります。
// それ以外の場合、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。
ZMAGICタイプのファイルの場合は、// (1KB境界のアラインメント)になります。
99 #ifndef _N_DATADDR
100 #define(_N_DATADDR(x))OMAGIC?(_N_TXTENDADDR(x))¥
102     : (_N_SEGMENT_ROUND (_N_TXTENDADDR(x))))
103 #endif
104
105 /* ロードされた後のメモリ上のbssセグメントのアドレス。*/
// 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。
106 #ifndef N_BSSADDR
107 #define _N_BSSADDR(x) (_N_DATADDR(x) + (x).a_data)
108 #endif
109
// 110~185行目がパート2です。オブジェクトファイル内のシンボルテーブルのエントリを記述しており、以下を定義しています。
// 関連する操作マクロです。プログラムリストの後にある詳細な説明をご覧ください。
// a.outオブジェクトファイル内のシンボルテーブルエンタリー（レコード）構造です。
110 #ifndef N_NLIST_DECLARED
111 struct nlist {
113     char *n_name;
114     struct nlist *n_next;
115     long n_strx;
116 } n_un;
117     unsigned char n_type;           // バイトは3つのフィールドに分かれています
118     char n_other;                 // 146~154行目は、各フィールドのマスクコードです。
119     short n_desc;
120     符号付き長さ n_value
121 };
122 #endif
123
// 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。
124 #ifndef _N_UNDF
125 #define _N_UNDF 0

```

```

126 #endif
127 #ifndef N_ABS
128 #define N_ABS_2
129 #endif
130 #ifndef N_TEXT
131 #define N_TEXT_4
132 #endif
133 #ifndef N_DATA
134 #define N_DATA_6
135 #endif
136 #ifndef N_BSS
137 #define N_BSS_8
138 #endif
139 #ifndef N_COMM
140 #define N_COMM_18
141 #endif
142 #ifndef N_FN
143 #define N_FN_15
144 #endif

// 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。
146 #ifndef N_EXT
147 #define N_EXT_1           // 0x01 (0b0000,0001) シンボルは外部（グローバル）？
148 #endif
149 #ifndef N_TYPE
150 #define N_TYPE_036        // 0x1e (0b0001,1110) シンボルのタイプビット。
151 #endif
152 #ifndef N_STAB
153 #define N_STAB_0340      // STAB -- シンボルテーブルのタイプ。
154 #endif

156 /* 以下のタイプは、シンボルの定義を、以下のように示しています。
157   は、他のシンボルへの間接的な参照です。他のシンボル
158   は、このシンボルの直後に未定義の参照として表示されます。
159
160 インダイレクションは非対称です。他のシンボルの値は、間接的なシンボ
161 ルの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。
162 他のシンボルに定義がない場合は、ライブラリを検索して定義を見つ
163 けます。
164 #define N_INDR_0xa
165
166 /* 以下の記号は、セット要素を表しています。
167 同じ名前のN_SET[ATDB]シンボルは、すべて1つのセットを形成します。
168 テキストセクションにセット用のスペースが割り当てられ、各セット要
169 素の値はスペースの1ワードに格納されます。
170 スペースの最初の単語は、セットの長さ（要素の数）です。
171
172 セットのアドレスは、セットの名前と同じ名前のN_SETVシ
173 ンボルにされます。
174 このシンボルは、N_DATAグローバルシンボルと同様に機能しま
175 す。
176 未定義の外部参照を満たすことができるという点で*/
177 /* これらはLDの入力として、.oファイルに表示されま
す。

```

```

178 #define N_SETA 0x14      /* 絶対値設定要素記号 */
179 #define N_SETT 0x16      /* テキストセット要素のシンボル
180 #define N_SETD 0x18      */
181 #define N_SETB 0x1A      データセット要素のシンボル */ /*
182                         データセット要素のシンボル
183 /* LDからの出力です。 */      /* Bss セット要素記号 */
184#define N_SETV_0x1C /* データ領域のセットベクタへのポインタ。 */ 185
186#ifndef N_RELOCATION_INFO_DECLARED
187
188 /* この構造体は、実行される1つの再配置を記述する。
189
190     ファイルのtext-relocationセクションは、これらの構造体のベクトルであり、
191     すべての構造体がtextセクションに適用されます。
192
193     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。 */ (注)
194
195 // 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。
196 struct relocation_info
197 {
198     /* 再配置されるアドレス(セグメント内)です。*/
199     int r_address;
200
201     /* r_symbolnum の意味は r_extern に依存する。*/
202     符号付き整数 r_symbolnum:24;
203
204     /* 0ではない場合は、pc-relative offsetであることを意味します。
205        と、自分のアドレスの変更のために再配置する必要があります。
206        また、指定されたシンボルやセクションの変更にも対応しています。*/
207     符号付き整数 r_pcrel:1;
208
209     /* 再配置されるフィールドの長さ(2の指数として)。
210        つまり、2という値は、1<<2バイトを表します。
211     符号付き整数 r_length:2;
212
213     /* 1 => シンボルの値で再配置。
214        r_symbolnum はシンボルのインデックスです。
215        ファイルのシンボルテーブルで
216        0 => セグメントのアドレスで再配置。
217        r_symbolnum は、N_TEXT、N_DATA、N_BSS または N_ABS です。
218        (N_EXTビットもセットされているかもしれません、何の意味もありません)。*/
219     符号付き整数 r_extern:1;
220
221     /* 使われない4つのビットだが、オブジェクトファイルを書くときには
222        をクリアすることが望ましいです。*/
223     unsigned int r_pad:4;
224 };
225
226#endif /* no N_RELOCATION_INFO_DECLARED.*/
227
228
229#endif /* A_OUT_GNU_H */
230
231

```

3. インフォメーション

1. a.out 実行可能ファイル形式

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。

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

- a) **execヘッダー** -- このセクションには、カーネルが実行ファイルをメモリにロードして実行するため使用するいくつかのパラメータ（exec構造）が含まれており、リンカー（ld）はこれらのパラメータを使用して、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。これは、オブジェクトファイルの唯一の必要なコンポーネントです。
- b) **テキストセグメント** -- プログラムの実行時にメモリに読み込まれる命令コード（テキスト）と関連データが含まれる。読み取り専用で読み込まれることもある。
- c) **データセグメント** -- このセクションには、すでに初期化されたデータが含まれており、常に読み書き可能なメモリにロードされています。
- d) **text relocations** -- このセクションには、リンカーが使用するレコードデータが含まれています。オブジェクト・モジュール・ファイルを結合する際に、コード・セグメント内のポインタやアドレスの位置を特定し、更新するために使用されます。
- e) **データ再配置** -- コード再配置セクションの役割に似ているが、データセグメント内のポインタの再配置のためのものである。
- f) **シンボルテーブル** -- このセクションには、リンカーがバイナリ・オブジェクト・モジュール・ファイル間で名前の付いた変数や関数（シンボル）を相互に参照するためのレコードデータも含まれています。

構造体 **exec_string table** -- シンボル名に対応する文字列を格納する部分。

```
符号付き長さ a_magic           // N_MAGICのようなアクセス用のマクロを使用します。
符号付き長さ a_text            符号付き長さ a_text // バイナリไฟアルは、最初あらかじめ実行データ構造(exec構造)で始ま
符号付き長さ a_data           // データの長さをバイト単位で表します。
unsigned a_bss                 // ファイルの初期化されていないデータ領域の長さをバイト単位
unsigned a_syms                // 表します。
unsigned a_entry               // ファイル内のシンボルテーブルデータの長さをバイト単位で表
unsigned a_trsize              // します。
unsigned a_drsiz               // 実行開始アドレス。
};                           // テキストの再配置情報の長さをバイトで表します。
                                // データの再配置情報の長さをバイト単位で指定します。
```

各フィールドの機能は以下の通りです。

- **a_magic** -- このフィールドには、図14-1に示すように、フラグフィールド、マシンタイプIDフィールド、マジックナンバーフィールドの3つのサブフィールドがあります。しかし、Linux 0.12システムでは、そのオブジェクトファイルにはマクロN_MAGIC()を使ってアクセスし、バイナリ実行ファイルと他のロードされたファイルとの違いを一意に決定します。このサブフィールドには、以下の値のいずれかを含める必要があります。
 - ◆ **OMAGIC**・・・テキストセグメントとデータセグメントが実行ヘッダの直後にあり、連続して格納されていることを示す。カーネルはテキストセグメントとデータセグメントの両方を読み取り可能なメモリと書き込み可能なメモリにロードします。コンパイラがコンパイルしたオブジェクトファイルのマジックナンバーはOMAGIC（8進数0407）です。
 - ◆ **NMAGIC** -- OMAGICと同様に、テキストとデータのセグメントは実行ヘッダに追従し、継続的に保存されます。しかし、カーネルはテキストをリードオンリーメモリにロードし、データをリードオンリーメモリにロードします。

のセグメントを、テキストの次のページ境界で書き込み可能なメモリに保存します。

- ◆ ZMAGIC -- カーネルは、必要に応じてバイナリ実行ファイルから別のページをロードします。実行ヘッダ、テキストセグメント、データセグメントはすべてリンクによって複数のページサイズのブロックに処理されます。カーネルがロードするテキストページは読み取り専用で、データセグメントのページは書き込み可能です。リンクによって生成された実行ファイルのマジックナンバーは「ZMAGIC」(0413, ie 0x10b)です。
- a_text -- このフィールドには、テキストセグメントのサイズ (バイト数) が入ります。
- a_data -- このフィールドには、データセグメントのサイズ (バイト数) が入ります。
- a_bss -- データセグメントの後にカーネルが最初のブレーク(brk)を設定するために使用する「bssセグメント」の長さが格納されている。カーネルがプログラムをロードしているとき、この書き込み可能なメモリはデータセグメントの後ろにあるように見え、最初はすべて0である。
- a_syms -- シンボルテーブルセクションのサイズをバイト単位で表しています。
- a_entry -- カーネルが実行ファイルをメモリにロードした後の、プログラム実行開始点のメモリアドレス。
- a_trsize -- このフィールドには、テキスト再配置テーブルのサイズが、バイト単位で格納されています。
- a_drsiz -- このフィールドには、データ再配置テーブルのサイズがバイト単位で入ります。

Non-hugepageファイルにはa_magicが定義されません。0以外の値を返す場合はexec構造を利用せず整合性をテストせずに、実行マゼイム内の構造を直接見つけたりします。

N_BADMAG(exec)	データセグメントの開始バイトオフセットです。
N_PXTOFF(exec)	データ再配置テーブルの開始バイトオフセットです。
N_TREOFF(exec)	トです。 テキスト再配置テーブルの開始バ
N_SYMOFF(exec)	トオフセットです。 シンボルテーブルの開始
N_STROFF(exec)	バイトオフセットです。

文字列テーブルの開始バイトオフセットです。

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。

構造体relocation_info

```
{
    int r_address;           // セグメント内で再配置される必要のあるアドレスです。
    符号付き整数 r_symbolnum:24; // その意味は、r_externに関連しています。
                                // 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。
    符号付き整数 r_pcrel:1,   // pc関連のフラグです。
    符号付き整数 r_length:2,  // 再配置されるフィールドの長さ（2の指数として）。
    符号付き整数 r_extern:1,  // 1: シンボルの値で再配置、0: セグメントのアドレスで再配置。
    符号付き整数 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 -- このフィールドには、リンクが処理（編集）する必要のあるポインタのバイトオフセットが含まれています。 テキスト再配置のオフセットは、テキストセグメントの先頭から数え、データ再配置のオフセットは、データセグメントの先頭から計算されます。リンクはこの値に

オフセットで既に保存されている値と、リロケーションレコードを使って計算された新しい値を比較します。

- r_symbolnum -- このフィールドは、シンボルテーブル内のシンボル構造体の序数（バイトオフセットではない）を表す。リンクは、シンボルの絶対アドレスを計算した後、再配置されるポインタにそのアドレスを追加します。（r_externビットが0の場合は、状況が異なります。後述します）。
 - r_pcrel -- このビットがセットされていると、リンクは、pc関連のアドレッシングモードを使用し、マシンコード命令の一部であるポインタが更新されていると見なします。実行中のプログラムがこの再配置されたポインターを使用するとき、ポインターのアドレスは暗黙のうちに追加されます。
 - r_length -- このフィールドは、ポインタの長さの2の累乗を含みます。0は1バイト長、1は2バイト長、2は4バイト長を意味します。
 - r_extern -- このビットがセットされていると、再配置に外部参照が必要であることを示し、リンクはシンボルアドレスを使用してポインタを更新する必要があります。このビットが0の場合、再配置は "ローカル" です。リンクは、様々なセグメントのロードアドレスの変更を反映させるためにポインタを更新します。
- シンボルの値の変更。この場合、r_symbolnumフィールドの内容はn_typeの値であり、再配置されたポインタがどのセグメントを指しているかをリンクに伝えます。
- r_pad -- これらの4ビットは、Linuxシステムでは使用されません。オブジェクトファイルを書くときは、すべて0にしておくとよいでしょう。

~~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）。リンク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。~~

```
long n_strx;
} n_un;
unsigned char n_type;           // は3つのフィールドに分かれています、146～154行目はそのマスク
チャ          n_other;          です。
シ          n_desc;
符号付き長さ n_value
};
```

各フィールドの意味は

- n_un.n_strx -- 文字列テーブル内のシンボル名のバイトオフセットが格納されています。プログラムがnlist()関数を使ってシンボルテーブルにアクセスすると、このフィールドはn_un.n_nameフィールドに置き換えられます。
- n_type -- リンクがシンボルの値をどのように更新するかを決定するために使用します。8ビット幅のn_typeフィールドは、図14-2に示すように、146～154行目から始まるbitmasksコードを使って、3つのサブフィールドに分割することができます。N_EXTタイプのロケーション・ビットのシンボルについては、リンクは「外部」シンボルとして扱い、他のバイナリ・オブジェクト・ファイルが参照できるようにします。N_TYPEマスクは、リンクが関心を持つビットを選択するために使用されます。
 - ◆ N_UNDF -- 未定義のシンボルです。リンクは、シンボルの絶対的なデータ値を決定するために、他のバイナリ・オブジェクト・ファイル内の同名の外部シンボルを見つける必要があります。特殊なケースとして、n_typeフィールドが0以外で、このシンボルを定義しているバイナリファイルがない場合⁸²⁴、リンクはシンボルをBSSセグメントのアドレスに解決し、n_valueに等しい長さのバイトを予約する。もし

シンボルが複数のバイナリ・オブジェクト・ファイルで定義されておらず、バイナリ・オブジェクト・ファイルの長さの値が一致しない場合、リンカーはすべてのバイナリ・オブジェクト・ファイルで見つかった最も長い値を選択します。

- ◆ N_ABS -- 絶対記号です。リンカは絶対記号を更新しません。
- ◆ N_TEXT -- テキスト（コード）シンボルです。このシンボルの値は、テキストアドレスであり、リンカーは、バイナリオブジェクトファイルをマージする際に、その値を更新します。
- ◆ N_DATA -- データシンボル。N_TEXTと似ているが、データアドレスのためのものである。対応するテキストおよびデータシンボルの値は、ファイルのオフセットではなく、アドレスである。ファイルのオフセットを求めるためには、該当するセクションが読み込みを開始するアドレスを決定し、それを差し引いた上で、セクションのオフセットを加える必要がある。
- ◆ N_BSS -- BSSシンボル。テキストやデータのシンボルに似ているが、バイナリオブジェクトファイルに対応するオフセットがない。
- ◆ N_FN -- ファイル名のシンボルです。バイナリ・オブジェクト・ファイルをマージするとき、リンカはシンボルを、バイナリファイル内のシンボルの前に配置します。シンボルの名前はリンカーに与えられたファイル名で、その値はバイナリファイルの最初のテキストセグメントのアドレスです。ファイル名シンボルは、リンクやロードには必要ありませんが、プログラムのデバッグには非常に便利です。
- ◆ N_STAB -- シンボリックデバッガ(gdbなど)が注目するビットを選択するためのマスクコードで、その値はstab()で指定します。
- n_other -- このフィールドは、n_typeで決定されたセグメントにしたがって、シンボル再配置操作に関するシンボル独立情報を提供する。現在、n_otherフィールドの最下位4ビットには2つの値のうち1つが格納されています。AUX_FUNC "と "AUX_OBJECT "です。AUX_FUNCはシンボルと呼び出し可能な関数を関連付け、AUX_OBJECT はシンボルがテキスト・セグメントかデータ・セグメントかに関わらず、データを関連付けます。このフィールドは、主にリンカldが動的実行プログラムを作成する際に使用します。
- n_desc -- デバッガが使用するために予約されており、リンカはこのフィールドを処理しません。リンカでは処理されません。デバッガによって、このフィールドの使用目的は異なります。
- n_value -- シンボルの値を格納する。テキスト、データ、BSSシンボルの場合はアドレス、他のシンボル（デバッガシンボルなど）の場合は、任意の値を指定できます。

文字列テーブルは、^{6 5}_{4 3 2 1 0} これに続くnullで構成されています。
長さはテーブル全体のノードのオフセット）は常に4です。

N_STAB	N_TYPE	N_EXT
--------	--------	-------

図 14-2 シンボルタイプフィールド
n_type

3. const.h

1. 機能

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。

2. コードアノテーション

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
6 // i-node構造体のi_modeフィールドの各フラグビット。
7 #define I_TYPE          0170000    // i-nodeのタイプを示す（タイプマスク）。
8 #define I_DIRECTORY     0040000    // ディレクトリファイルです。
9 #define I_REGULAR       0100000    // dirや特殊ファイルではなく、通常のファイルです。
10 #define I_BLOCK_SPECIAL 0060000    // ブロックデバイスのスペシャルファイルです。
11 #define I_CHAR_SPECIAL  0020000    // キャラクター・デバイス・スペシャル・ファイルです。
12 #define I_NAMED_PIPE    0010000    // 名前付きパイプノードです。
13 #define I_SET_UID_BIT   0004000    // 実行時に効率的なユーザーIDタイプを設定します。
14                                         // 実行時に効率的なグループIDタイプを設定します。
15 #endif
16

```

4. ctype.h

1. 機能

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、アンダースコア2つ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、abcや_SPのように、通常はヘッダーライター用に予約されています。

2. コードアノテーション

プログラム 14-3 linux/include/ctype.h

```

1 #ifndef _CTYPE_H
2 #define _CTYPE_H
3
4 #define _U     0x01    /* 上          // 大文字の[A-Z]に使用されます。
5 #define _L     0x02    /* 下げる      // 小文字の[a-z]に使用されます。
6 #define _D     0x04    /* 数字で表    // 0-9」のデジタル表示に使用されます。
7 #define _C     0x08    示していま    // 制御文字に使用されます。
8 #define _P     0x10    す。          // 句読点に使用されます。
9 #define _S     0x20    ホワイトスペース(space/lf/tab) // 16進法の数字に使用されます。
10 #define _X    0x40    16進数で表示されます // スペース文字 (0x20) に使用されます。
11 #define _SP   0x80    */
12           /* ハードスペース (0x20)
13           // 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。
14           // 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。
15 extern unsigned char _ctype[];
16 extern char _ctmp;
17
18 // 以下に、文字の種類を決めるマクロを紹介します。
19 #define isalnum(c) ((ctype+1)[c]&(_U|_L|_D)) // は、文字またはデジタルです
20 #define isalpha(c) ((ctype+1)[c]&(_U|_L)) 18
21 #define iscntrl(c) ((ctype+1)[c]&(_C)) // は文字です。
22 #define isdigit(c) ((ctype+1)[c]&(_D)) // は制御文字です。
23 #define isgraph(c) ((ctype+1)[c]&(_P|_U|_L|_D)) // はデジタルです。
24 #define islower(c) ((ctype+1)[c]&(_L)) // はグラフィック文字です。
25 #define isprint(c) ((ctype+1)[c]&(_P|_U|_L|_D|_SP)) // は小数点可能な文字であること。
26 #define ispunct(c) ((ctype+1)[c]&(_P)) // は句読点です。
27 #define isspace(c) ((ctype+1)[c]&(_S)) // is a space, " " " " " "
28 #define isupper(c) ((ctype+1)[c]&(_U)) // は大文字の文字です。
29 #define isxdigit(c) ((ctype+1)[c]&(_D|_X)) // は16進数である。
30
31 // 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'
32 // 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
33 // ため、'c' は
34 // 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。(符号なし
35 // )
36 #define _tolower(c) (isupper(c)? _ctmp=c, isupper(_ctmp)? _ctmp-('A'-'A'): _ctmp) // を小文字のcharに変換します。
37 #define _toupper(c) (islower(c)? _ctmp=c, islower(_ctmp)? _ctmp-('a'-'A'): _ctmp) // 文字を大文字に変換します。
38
39 #endif
40

```

5. errno.h

1. 機能

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ついて、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だからです。もし、各関数に真/偽の指示値を返させての結果の値が別々に返ってくるので、システムコールの結果を簡単に得ることができません。1つの解決策は、この2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ー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。

2. コードアノテーション

プログラム 14-4 linux/include/errno.h

```

1 #ifndef _ERRNO_H
2 #define _ERRNO_H
3
4 /*
5 * の他の情報源を得ていなかったので、OKしました。
6 * エラーの可能性のある数字を使用せざるを得ませんでした。
7 * minixとして。
8 * 願わくば、これらが posix か何かであることを。私にはわかりませんが（しかも posix
9* は私には教えてくれません。彼らは自分たちの***ingスタンダードに$$
10* を求めています）。 10 *
11 * minixの_SIGNのようなごちゃごちゃしたものは使ってないので、カーネルのリターンは
12* 自分でサインを見る。 13 *
13 * 注意！このファイルを変更した場合は、strerror()を変更すること
14* を忘れないでください。 15 */
15 */
16
// システムコールや多くのライブラリ関数は、操作を示す特別な値を返します。
// 失敗またはエラーになります。この値は、通常は '-1' か、その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。型を知る必要がある場合は
// エラーの場合、システムのエラー番号を表す変数'errno'に注目する必要があります。
// 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。

```

```

// 実行します。
17 extern int errno;
18

// 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'に格納します。
// (負の値)で、-1を返します。したがって、プログラムが特定のエラーを知る必要がある場合は
// の数が多い場合は、'errno'の値を確認する必要があります。
19 #define ERROR 99 // 一般的なエラーです。
20 #define EPERM 1 // その操作は許可されていません。
21 #define ENOENT 2 // ファイルやディレクトリが存在しません。
22 #define ESRCH 3 // 指定されたプロセスは存在しません。
23 #define EINTR 4 // システムコールを中断しました。
24 #define EIO 5 // 入力/出力エラー
25 #define ENXIO 6 // 指定されたデバイスまたはアドレスが存在しません。
26 #define E2BIG 7 // パラメータリストが長すぎます。
27 #define ENOEXEC 8 // 実行ファイルの形式が正しくありません。
28 #define EBADF 9 // ファイルハンドル（記述子）が正しくありません
29 #define ECHILD 10
30 #define EAGAIN 11
31 #define ENOMEM 12 // 子プロセスは存在しません。
32 #define EACCES 13 // そのリソースは一時的に利用できません。
33 #defineEFAULT 14 // 十分なメモリがありません。
34 #define ENOTBLK 15 // アクセス許可がありません。
35 #define EBUSY 16 // アドレスが間違っています。
36 #define EEXIST 17 // ブロックデバイスファイルではありません。
37 #define EXDEV 18 // リソースはビジー状態です。
38 #define ENODEV 19 // ファイルはすでに存在しています。
39 #define ENOTDIR 20 // デバイスへの接続が不正です。
40 #define EISDIR 21 // そのデバイスは存在しません。
41 #define EINVAL 22 // ディレクトリファイルではありません。
42 #define ENFILE 23 // ディレクトリファイルです。
43 #define EMFILE 24 // 無効な引数です。
44 #define ENOTTY 25 // システムが開いているファイルが多すぎます。
45 #define ETXTBSY 26 // 開いているファイルが多すぎます。
46 #define EFBIG 27 // 不適切なI/O (ttyターミナルがない)。
47 #define ENOSPC 28 // (もう使わない)。
48 #define ESPIPE 29 // ファイルサイズが大きすぎます。
49 #define EROFS 30 // デバイスが満杯です（デバイスに空きがありません）。
50 #define EMLINK 31 // 無効なファイルポインタの再配置。
51 #define EPIPE 32 // ファイルシステムは読み取り専用です。
52 #define EDOM 33 // リンクが多すぎます。
53 #define ERANGE 34 // パイプが間違っています。
54 #define EDEADLK 35 // ドメインエラーです。
55 #define ENAMETOOLONG 36 // 結果の範囲のエラーです。
56 #define ENOLCK 37 // リソースのデッドロック
57 #define ENOSYS 38 // ファイル名が長すぎます。
58 #define ENOTEMPTY 39 // 利用できるロックはありません。
59
60 /* ユーザープログラムからは絶対に見られない// 未実装です。
由 来#define ERESTARTSYS 512 // デスクトップシステム再起動ません。
62 #define ERESTARTNOINTR 513 // 割り込みなしでシステムコールを再実行する
63
64#endif
65

```

6. fcntl.h

1. 機能

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。

2. コードアノテーション

<pre> 1 #ifndef _FCNTL_H 2 #define _FCNTL_H 3 4 #include <sys/types.h> 5 6 /* open/fcntl - NOCTTY, NDELAYはまだ実装されていません */. 7 #define O_ACCMODE 00003 // ファイルアクセスモードマスク。 8 // 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。 9 #define O_RDONLY 00 // ファイルを開くでリードオンリーモード。 10 #define O_WRONLY 01 // ファイルを開くでライトオンリーモード。 11 // 以下はopen()のファイル作成および操作フラグです。上記の「[bit or]」を「bit or」モードにする。 12 #define O_CREAT 00100 /* not fcntl */ // ファイルが存在しない場合に作成します 13 #define O_EXCL 00200 /* not fcntl */ . 14 #define O_NOCTTY 00400 /* not fcntl */ // ファイルの排他的使用 15 #define O_TRUNC 01000 /* not fcntl */ // 制御端子はありません。 16 #define O_APPEND 02000 // 書き込まれた場合は、0に切り捨てられます。 17 #define O_NONBLOCK 04000 /* not fcntl */ // アペンドモードでファイルを開く。 18 // ノンブロッキングでファイルを開く。 19 /* fcntl-commandの定義です。なお、現在 20 * ロッキングがサポートされていないなど、実際には 21 * テスト済みです。 22 */ 22 23 #define インターフェースクリップの一の操作関数fcntl()で使用するコマンド(cmd)です。を複製します。 24 #define F_GETFD 1 /* ゲ f_flags */ // ファイルハンドルフラグ(FD_CLOEXEC)を取得します。 25 #define F_SETFD 2 /* セ f_flags */ // ファイルハンドルフラグを設定します。 26 #define F_GETFL 3 /* その他のフラグ(cloexec) */ 27 #define F_SETFL 4 // ファイルの状態を表すフラグやアクセスモードの取得・設定。 28 // 以下はファイルロックコマンド。Th fcntl()の'lock'はflock構造体を指しています。 29 #define F_GETLK 5/* not implemented */ // ロックをブロックするロックを取得します。 30 // ロックの設定(F_RDLCKまたはF_WRLCK)またはクリア(F_UNLCK)を行います。 </pre>	<p>プログラム 14-5 linux/include/fcntl.h</p> <p>// 型のヘッダーファイルです。基本的なシステムデータ型が定義されています。</p> <p>/* open/fcntl - NOCTTY, NDELAYはまだ実装されていません */.</p> <p>#define O_ACCMODE 00003 // ファイルアクセスモードマスク。</p> <p>// 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。</p> <p>#define O_RDONLY 00 // ファイルを開くでリードオンリーモード。</p> <p>#define O_WRONLY 01 // ファイルを開くでライトオンリーモード。</p> <p>// 以下はopen()のファイル作成および操作フラグです。上記の「[bit or]」を「bit or」モードにする。</p> <p>#define O_CREAT 00100 /* not fcntl */ // ファイルが存在しない場合に作成します</p> <p>#define O_EXCL 00200 /* not fcntl */ .</p> <p>#define O_NOCTTY 00400 /* not fcntl */ // ファイルの排他的使用</p> <p>#define O_TRUNC 01000 /* not fcntl */ // 制御端子はありません。</p> <p>#define O_APPEND 02000 // 書き込まれた場合は、0に切り捨てられます。</p> <p>#define O_NONBLOCK 04000 /* not fcntl */ // アペンドモードでファイルを開く。</p> <p>// ノンブロッキングでファイルを開く。</p> <p>/* fcntl-commandの定義です。なお、現在</p> <p>* ロッキングがサポートされていないなど、実際には</p> <p>テスト済みです。 22 */</p> <p>#define インターフェースクリップの一の操作関数fcntl()で使用するコマンド(cmd)です。を複製します。</p> <p>#define F_GETFD 1 /* ゲ f_flags */ // ファイルハンドルフラグ(FD_CLOEXEC)を取得します。</p> <p>#define F_SETFD 2 /* セ f_flags */ // ファイルハンドルフラグを設定します。</p> <p>#define F_GETFL 3 /* その他のフラグ(cloexec) */</p> <p>#define F_SETFL 4 // ファイルの状態を表すフラグやアクセスモードの取得・設定。</p> <p>// 以下はファイルロックコマンド。Th fcntl()の'lock'はflock構造体を指しています。</p> <p>#define F_GETLK 5/* not implemented */ // ロックをブロックするロックを取得します。</p> <p>// ロックの設定(F_RDLCKまたはF_WRLCK)またはクリア(F_UNLCK)を行います。</p>
--	---

```

31
32 /* F_[GET|SET]FLのためのものです。
   // 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。
33#define FD_CLOEXEC 1 /* 実際にロービットが設定されているものは何でも行く */
34
35 /* OK、これらはロック機能であり、どのような場合でも実装されていません。
36* レベルである。POSIXはそれ
   を望んでいる。 37 */
37

38 #define F_RDLCK      0      // ファイルのロックを共有または
39 #define F_WRLCK      1      読み取ります。
40 #define F_UNLCK      2      // 排他的または書き込みのファイルロック。
41
42 /* もう一度言いますが、実装はされないかもしれません。*/
   // 以下は、ファイルロックのデータ構造で、影響を受けるタイプ(l_type)を記述しています。
   // ファイルセグメント、開始オフセット(l_whence)、相対オフセット(l_start)、ロック長
   // (l_len)、そしてロックを実装するpidです。
43 struct flock {
44     short l_type;           // ロックタイプ (F_RDLCK, F_WRLCK, F_UNLCK)。
45     short l_whence;         // 開始オフセット (SEEK_SET, SEEK_CUR, SEEK_END)。
46     off_t l_start;          // ロックの始まりです。相対的なオフセット (バイト単位) です。
47     off_t l_len;            // ロックのサイズ。0の場合はファイルの終わり。
48     pid_t l_pid;            // ロックのプロセスIDです。
49 };
50

51 // 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。
52 // 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'filename'はファイル名
   // 作成されるファイルの//、'mode'はプロパティ (include/sys/stat.h参照)。
53 extern int creat(const char * filename, mode_t mode);
54 // ファイルハンドル操作関数です。これらは、ファイルオープン操作に影響を与えます。パラメータ
   // 'fildes' は
   // 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);
   // int fcntl(int fildes, int cmd, struct flock *lock);
55 extern int fcntl(int fildes, int cmd, ...);
56

57 #endif
58

```

7. signal.h

1. 機能性

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。シグナルは

最も重要な機能は、シグナルの処理方法を変える関数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で要求される20個のシグナルをすべて実装している。つまり、Linuxは最初からPOSIX.1との互換性を考慮して設計されていると言えるのです。具体的な機能の実装は、kernel/signal.cというプログラムにあります。

14.7.2 コードアノテーション

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です。基本的なシステムデータ型が定義さ
5                                         れています。
6 typedef int sig_atomic_t;        // シグナルのアトミックな操作タイプを定義しま
7 typedef unsigned int sigset_t;   // す。
8                                         /* 32ビット */ // シグナルセットタイプの定義
9 #define _NSIG          32         // 信号の数
10 #define NSIG          _NSIG      // nsig = _nsig
11
12 // 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。20個のシグナルがすべて含ま
13                                         れています
14 #ifndef _POSIX_SIGNAL // 要求されている。          // ハングアップ -- コントロールターミナルやプロセスをハング
15                                         // アップする。
16 #define SIGINT         2          // インタラップ -- キーボードからのインターラップト
17                                         // ト
18 #define SIGQUIT        3          // 辞める -- キーボードからの終了コマンド
19 #define SIGILL         4          // イルイーグ -- 違法な命令です。
19                                         // ル
20 #define SIGTRAP        5          // トランプ -- ブレイクポイントの追跡
21 #define SIGABRT        6          // アボート
22 #define SIGIOT          6          // IOトランプ
23 #define SIGUNUSED       7          // 未使用
24 #define SIGFPE          8          // FPE -- コプロセッサのエラーです。
25 #define SIGSEGV        11         // キャンセル違反 -- 無効なメモリ参照
26                                         // プロセスを強制的に終了させます。
27 #define SIGUSER1        10         // User2 -- ユーザー1 == ユニバーサル信号1.
28 #define SIGPIPE         13         // パイプ -- パイプ書き込みエラー、リーダ
29                                         // ム
30 #define SIGALRM        14         // アラーム -- なし。
31 #define SIGTERM        15         // ターミネ -- リアルタイムのタイマーアラー
32 #define SIGSTKFLT       16         // Stack Fault -- スタックエラー（コプロセッ
33 #define SIGCHLD         17         // ム）子供 -- 子プロセスが終了または終了する。
34 #define SIGCONT        18         // 続ける -- プロセスの実行を再開します。
35 #define SIGSTOP         19         // ストップ -- プロセスの実行を停止します。
36 #define SIGTSTP         20         // TTYスト -- プロセスが送信するStop sigは無視しても構
37                                         // ット
38 #define SIGTTIN         21         // TTYイン -- バックグラウンド処理では、入力を求められ
39                                         // ます。
39
40 /* OK, シグナルは実装していないませんが、ヘッダをPOSIXを維持するための要求する
41 // 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。
42 // 以下は、「sa_flags」フラグフィールドから取得できるシンボリック定数です。
43 // Σ(1..111)
44 // 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。
45 // SA_INTERRUPT - シグナルで中断された後、システムコールは再開されません。
46 // SA_NOMASK - この信号は、その信号ハンドラで受信がブロックされていません。

```

```

// SA_ONESHOT - シグナルハンドラーが呼び出されると、デフォルトのものに戻されます。
37 #define SA_NOCLDSTOP      1
38 #define SA_INTERRUPT     0x20000000
39 #define SA_NOMASK        0x40000000
40 #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
42 #define SIG_BLOCK        1 /* シグナルをブロックするために
43 #define SIG_SETSIG        2 /* シグナルの動作を変更するために使用します。
44 #define SIG_SETMASK       2 シグナルのブロック解除のための
45 /* */
// 以下の3つの定数記号は // の値を持ち、INTの整数パラメータとして使用することができます。
// 3つの定数記号は // の値を持ち、INTの整数パラメータとして使用することができます。
// 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。
// 以下のsignal()関数で、カーネルにシグナルを処理させ、無視するように指示します。
// 信号の処理を行うとエラーになります。その使い方については
// kernel/signal.c の 156-158 行目をご覧ください。
46 #define SIG_DFL          ((void (*)(int))0)    /* デフォルトのシグナル処理 */
47 #define SIG_IGN          ((void (*)(int))1)    /* シグナルを無視する */
48 #define SIG_ERR          ((void (*)(int))-1)   /* シグナルからのエラーリター
49 */
// 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。
50 #ifdef notdef
51 #define sigemptyset(mask) ((*(mask) = 0), 1) // マスクをクリアします。
52 #define sigfillset(mask) ((*(mask) = ~0), 1) // マスクの全ビットが設定されます。
53 #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。このシグナルを無視するには
// 上記のSIG_DFLやSIG_IGNのほか、シグナルを処理する関数へのポインタを指定することができます
// 'sa_mask' はシグナルのマスキングコードを与え、これらの処理をブロックします。
// シグナルプログラムが実行されると、 // シグナルが表示されます。
// '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、その定義は
// 37-40行目のビットフラグによって
// '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、以下のように使用されます。
// でユーザースタックをクリーンアップします。signal.cを参照してください。
// さらに、トリガー信号処理の原因となる信号も、以下の場合を除き、ブロックされます。
56 // SA_NOMASK(既定のマスク);
57 struct sigaction sa_mask;
58     int sa_flags;
59     void (*sa_restorer)(void)
60 }; // です。
61
// 以下の signal() 関数は、シグナル _sig の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、シグナル _sig を
// キャプチャされたもので、もう一つは、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、int型の引数を持つ関数ポインターです（最後の
// (int)）となり、戻り値はなく、シグナルの元のハンドルとなります。
62 void (*signal(int _sig, void (*_func)(int))(int);

```

```

// 以下の2つの関数はシグナルを送るために使用されます。kill()はシグナルを
// raise()は、現在のプロセス自身にシグナルを送るために使用されます。
// これは kill(getpid(), sig)と同じです。kernel/exit.cの205行目を参照してください。
63 int raise(int sig);
64 int kill(pid_t pid, int sig);
// プロセスのタスク構造では、以下を示す32ビットのシグナルフィールド'signal'に加えて
// 現在のプロセスの処理対象となる信号には、32ビットのブロッキング・シグナルが設定されています。
// 信号をマスクするためのフィールド「blocked」で、各ビットが対応するブロックされた信号を表します。
// マスクシグナルセットを変更することで、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、信号セットをマスクするためにプロセスを操作するために使用されます。しかし、それは非常に
// 実装は簡単ですが、このバージョンのカーネルに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
// すべてのシグナルをシグナルセットに入れる、つまりすべてのシグナルをマスクする。もちろん、
// SIGINT と SIGSTOP は
// がブロックされます。sigismember()関数は、指定されたシグナルが、シグナル
// セット(1 - はい、0 - いいえ、-1 - エラー)。
65 int sigaddset(sigset_t *mask, int signo);
66 int sigdelset(sigset_t *mask, int signo);
67 int sigemptyset(sigset_t *mask);
68 int sigfillset(sigset_t *mask);
69 int sigismember(sigset_t *mask, int signo); /* 1 - is, 0 - not, -1 error */.
// '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、現在の
// プロセス中にブロックされたシグナルセット。
70 int sigpending(sigset_t *set);
// 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。もし 'set' ポイントが
// がNULLでない場合は、プロセスマスクシグナルセットを「どのように」(42-44行目)に従って変更します。
71 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。
// して、シグナルを受け取るまでプロセスを一時停止します。シグナルを捕捉して返すと
// シグナルハンドラーから // を受け取ると、この関数は戻り、シグナルマスクは
//呼び出しの前に
72 int sigsuspend(sigset_t *sigmask);
8. stdarg.h
// sigaction 関数は、プロセスが次のようなものを受け取ったときに取る行動を変更するために使用されます。
// シグナルの処理ハンドラを変更することです。の説明を参照してください。
1. 機能
73 int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
74 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。
75 #endif /* _SIGNAL_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。

2. コードアノテーション

プログラム 14-7 linux/include/stdarg.h

```

1 #ifndef STDARG_H
2 #define STDARG_H
3
4 typedef char *va_list; // va_listの定義は、文字ポインタ型です。 5
5 /* TYPE型のargのために引数リストに必要なスペースの量。
6 TYPEには、タイプが使用される式を指定することもできます。*/ 8
7
8 // 次の文は、丸みを帯びたTYPE型のバイト長を定義しており、その倍数は
9 // intの長さ(4)の
10 #define va_rounded_size(TYPE) \(^o^)
11 (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * 11
12 // 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。
13 // 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
14 // は最初に va_start マクロを呼び出す必要があります。パラメーターLASTARGは最右翼の
15 // 関数定義のパラメータ、すなわち、「...」の左側の識別子です。APは
16 // 可変パラメータテーブルのパラメータポインタ、LASTARGは最後に指定されたパラメータ。
17 // &(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。
18 // LASTARG の幅の値を追加した後、AP は、最初のパラメータへのポインタである
19 // 変数のパラメータリストです。このマクロには戻り値はありません。
20 // 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で定義されています。
21 // で、レジスタを保持するために使用されます。gccの“Implementing the Varargs Macros”の項を
22 // 参照してください。
23 // マニュアル「Target Description Macros」で説明しています。
24 #ifndef sparc
25
26 #define va_start(AP, LASTARG) ¥
27 (AP = ((char *) &(LASTARG) + va_rounded_size (LASTARG)))
28
29 #else
30 #define va_start(AP, LASTARG) ¥
31 ( builtin_saveregs ()です。 ¥
32 AP = ((char *) &(LASTARG) va_rounded_size (LASTARG)))
33
34 #endif
35
36 // 以下のマクロは、呼び出された関数が正常なリターンを完了するために使用されます。va_end
37 // は、以下のように変更できます。
38 // va_start が再び呼び出されるまで使用されない AP。va_end は va_arg の後に呼び出されなければ
39 // なりません(va_list); /* gnu/libで定義されています */
40
41 // 実際にva_end(AP)マクロを読みます */
42
43 // 次のマクロは、式を拡張して、同じタイプと値を持つ
44 // 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

```

```

// 文字、および浮動小数点型です。va_arg を最初に使用したとき、最初の
// テーブルの // 引数を返し、それ以降の呼び出しでは、テーブルの次の引数を返します。この
// は、まずAPにアクセスし、その値を増やして次の項目を指し示すことで行われます。
// va_arg は TYPE を使ってアクセスを完了し、次のアイテムを探します。va_argが呼ばれるたびに
// テーブルの次のパラメータを示すようにAPを変更します。
24 #define va_arg(AP, TYPE)           \
25   (AP += va_rounded_size (TYPE),    \
26    *((TYPE *) (AP - va_rounded_size (TYPE)))) 27 \
28 #endif /* _STDARG_H */ \
29

```

9. stddef.h

1. 機能性

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サポートしていないスタンダードアロン環境でも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る人もいる。スタンダードアロンの場合

環境では、C標準では、C言語のすべての属性を提供することが求められます。標準Cライブラリでは、float.h、limit.h、stdarg.h、stddef.hの4つのヘッダーファイルに含まれる関数をサポートするだけでよいとされています。浮動小数点表現の特徴を説明する内容を明確にしたもので、他の3つのヘッダーファイルは基礎的で、より具体的な部分に使用されます。

■ stdarg.h 変数のパラメータリストにアクセスするためのマクロ定義

スタンダードアロン環境で使われる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つのヘッダー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、様々なコンパイラー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 コードアノテーション

プログラム 14-8 linux/include/stddef.h

```
1 ifndef _STDDEF_H
```

```

2 #define _STDDEF_H
3
4 #ifndef PTRDIFF_T
5 #define PTRDIFF_T
6 typedef long ptrdiff_t;           // 2つのポインタを引き算した結果の型。
7#endif
8
9 #ifndef SIZE_T
10#define SIZE_T
11typedef unsigned long size_t;      // sizeof()によって返される結果のタイプ。
12#endif
13
14#undef NULL
15#define NULL ((void *)0)           // null ポインタ
16.
17 // 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。
18 // 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
19 // 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
20 // 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、整数の
21 // タイプ0をデータ・オブジェクト・ポインタ・タイプにキャストして、その操作を
22 // 結果。
23#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
24
25#endif
26

```

10. string.h

1. 機能性

string.hヘッダーファイルでは、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ファイルについては、Linusは各関数の実装を与えていますが、各関数には「extern」と「inline」というキーワードの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、あるプログラムに対して

このヘッダーファイルを含んでいると、何らかの理由で使用するインライン関数を呼び出し側のコードに埋め込むことができない場合、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の関数が使用されます（lib/string.cプログラム参照）。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。

2. コードアノテーション

プログラム14-9 linux/include/string.h

```

1 #ifndef STRING_H
2 #define STRING_H
3
4 #ifndef NULL
5 #define NULL ((void *) 0)
6 #endif
7
8 #ifndef SIZE_T
9 #define SIZE_T
10 typedef unsigned int size_t;
11#endif
12
13 extern char * strerror(int errno);
14
15 /*
16 * このstring-includeは、すべての文字列関数をインラインで定義します。
17 * 機能があります。gccを使用してください。また、ds=es=データスペースを想定しています
18 * が、これは
19 * 正常です。ほとんどの文字列関数は、かなり手作業で最適化されています。
20 * 特にstrtok, strstr, str[c]spnを参照してください。これらは動作するはずですが
21 * 非常にわかりやすいですね。すべてはレジスター内で完結します。
22 * セットされているので、機能が早くてきれいで。文字列指示は
23 * 全体的に使用されているため、「少し」不明瞭なコードになって
24 * います:-) 23 */
25 */ (C)
26 1991 Linus Torvalds
27
28 ///////////////////////////////////////////////////////////////////
29 // 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。
30 // Params: dest - コピー先の str ポインタ, src - コピー元の str ポインタ。
31 // 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。
32 extern inline char * strcpy(char * dest, const char *src)
33 {
34     // DS:[ESI]から1バイトをALにロードし、ESIを更新します。
35     // ALのバイトをES:[EDI]に格納し、EDIを更新します。
36     // 今格納したバイトは0?
37     // そうでない場合は、ラベル1に逆方向にジャンプし、他
38     // は終了。
39     // 変換の結果、dest:(src), dest:(dest): "si", "di", "ax" を表示します。
40     // DS:[ESI]から1バイトをALにロードし、ESIを更新します。
41     // ECX--(カウント--)を登録します。
42     // count<0であれば、ラベル2にジャンプして終了。

```

```

43     /*lodsb$*/
44     (stosb$)
45     "testb %%al, %%al$t"
46     "jne 1b$t" "rep$"
47     "stosb$\n"
48     "2:"           // そうでなければ、ラベル1にジャンプしてコピーを続け
49                           // る。
50     :: "S"(src), "D"(dest), "c"(count): "si", "di", "ax",
51     return dest;           // destの文字列ポインタを返します。
52 }
53
54 ///////////////////////////////////////////////////////////////////
55 // 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。
56 // 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。
57 // アセンブリコードでは、%0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1)
58 // .
59 extern inline char * strcat(char * dest, const char * src)
60 { ...
61   asm ("cld$"
62       "scasb$" "dec1"
63       "%1$"
64       "1:tlodsb$"
65       "stosb$"
66       "testb %%al, %%al$t"
67       "jne 1b"
68     :: "S"(src), "D"(dest), "a"(0), %0(%1) // %0がFFの場合siをベリプして継続し、そ
69     return dest;           // のでない場合は終了しを返します
70   }
71
72 ///////////////////////////////////////////////////////////////////
73 // source strのcountバイトをdest strの最後にコピーし、最後にnull charを追加します。
74 // dest - コピー先の文字列, src - コピー元の文字列, count - コピーするバイト数。
75 // アセンブリコードでは、%0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1), %4 -
76 // (count)となります。
77 extern inline char * strncat(char * dest, const char * src, int count)
78 { ...
79   asm (cld$"
80       "scasb$"
81       "dec1 %1$"
82       "movl %4, %3n"
83       "1:df%3" "js 2f"
84       "lodsb" "stosb"
85       "testb %%al, %%al$\n$t"
86       "jne 1b$\n"
87       "2:txorl %2, %2%n$\t"
88       "stosb"
89     :: "S"(src), "D"(dest), "a"(0), %0(%1) // %0を%4に代入する。"g"(count)の
90     順で表示されます。           // ES:[EDI]に保存します。
91   return dest;           // destの文字列ポインタを返します。
92 }
93
94 ///////////////////////////////////////////////////////////////////
95 // 2つの文字列を比較し
96 // パラメータ: cs-文字列1、ct-文字列2。

```

```

// %0 - eax( res ) の戻り値、 %1 - edi(cs) の str1 ポインタ、 %2 - esi(ct) の str2 ポイ
ンタ。
// 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。
88 extern inline int strcmp(const char * cs, const char * ct)
89 {
90 register int res asm ("ax"); // res はレジスタ変数 (eax) です。 91_asm
92 "cld\n" "1:tlodsb\n\t" // str2バイトのDS:[ESI]をALに取得し、 ESI++とする。
93 "scasb\n\t" "jne 2f\n\t" // ALをstr1バイトES:[EDI]、 EDI++と比較する。
94 "2f\n\t" // 等しくない場合は、 ラベル2にジャンプします。
95 "testb %%al, %%al\n\t" // これはバイト0(文字列の終わり)ですか?
96 "jne 1b\n\t" // いいえ、 ラベル1に戻り、 比較を続けます。
97 "xorl %%eax, %%eax\n\t" // はい、 EAXがクリアされ、 ラベル3, endにジャンプします
98 "jmp 3f\n\t" 。 
99 "2:00" "1:00" "3:00" // EAXは1に設定されています。
100 "4:00" // str2 < str1の場合、 正の値を返し、 終了。
101 "negl %%eax\n\t" // Else EAX = -EAXとし、 負の値を返して終了。
102 "3:" 
103 : "=a" ( res ): "D" ( cs ), "S" ( ct ): "si", // 比較結果を返します。
104 return res; // 比較結果を返します。
105 }
106
107 ///////////////////////////////////////////////////////////////////
107 // 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。
107 // パラメータ: cs - str1, ct - str2, count - 比較する文字数。
107 // %0 - eax( res ), %1 - edi(cs) str1 ポインタ, %2 - esi(ct) str2 ポインタ, %3 - ecx(count).
107 // 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。
107 extern inline int strncmp(const char * cs, const char * ct, int count)
108 {
109 register int res asm ("ax"); // res はレジスタ変数(eax) です。 110_asm
("cld\n")
110 "1:tdec1 %3n\n\t" // ECX--(カウント--)を登録します。
111 "js 2f\n\t" // count<0ならば、 ラベル2にジャンプして進む。
112 "1:lodsb\n\t" // str2のchar DS:[ESI]をALにロードして、 ESI++。
113 "scasb\n\t" // ALとstr1の比較 char ES:[EDI], EDI++。
114 "jne 3f\n\t" // 等しくない場合は、 ラベル3にジャンプします。
115 "3f\n\t" // それはNULL charですか?
116 "testb %%al, %%al\n\t" // いいえ、 ラベル1に戻って比較を続けます。
117 "jne 1b\n\t" // はい、 その場合、 EAXはクリアされます(戻り値)。
118 "2:txorl %%eax, %%eax\n\t" // ラベル4にジャンプして終了
119 "jmp 4f\n\t" // EAXを1にする。
120 "3:to be $1, %%eax\n\t" // 結果がstr2 < str1の場合、 1が返されます。
121 "j1 4f\n\t" // Else EAX = -EAX, 負の値を返して終了。
122 "negl %%eax\n\t" 
123 "4:" 
124 : "=a" ( res ): "D" ( cs ), (ct) , "c" (count) : "si", "di", "cx" ) なっています
125 return res; // 比較結果を返します。
126 }
127
128 ///////////////////////////////////////////////////////////////////
128 // 文字列の中で最初に一致する文字を探します。
128 // パラメータ: s - 文字列、 c - 検出される文字。
128 // アセンブリコードでは、 %0 - eax( res ), %1 - esi (str ポインタ s), %2 - eax (char c) で
す。
128 // 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。
128 extern inline char * strchr(const char * s, char c)
129 {

```

```

130 register char * res asm ("ax"); 131
asm ("cld\n\t")
132     "movb %%al, %%ah\n\t"           // AHと比較するcharを移動させる。
133     "1:t1odb\n\t" "cmpb          // 文字列を取得する char DS:[ESI] to AL, and ESI++.
134     %%ah, %%al\n\t" "je          // char ALは指定されたchar AHと比較されます。
135     2f\n\t"                      // 同じであれば、ラベル2に向かってジャンプします。
136     "testb %%al, %%al\n\t"         // ALの文字はNULL文字ですか？(文字列の終わり？)
137     "jne 1b\n\t"                  // そうでなければ、ラベル1に戻って比較を続ける。
138     "movl $1, %1\n\t"             // はいの場合、マッチするチャーが見つからず、ESIを1に
139     "2:\n\t%1, %0\n\t" "decl      設定します。
140     %0"                          // match charの次のバイトのポインタをEAXに入れる。
141     :"=a" ( res): "S" (s), "0" (c) / ポインタが一致するcharを指すように調整します。
142 return res\";
143 }                                     // ポインタを返します。
144

//// 文字列の中で指定された文字が最後に現れる場所を探す。(逆引き検索文字列)
// 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(string pointer s), %2 - eax(char c) です。
// 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。
145 extern inline char * strchr(const char * s, char c)
146 {
147 register char * res asm ("dx");
148     asm ("cld\n\t%%al, %%ah\n\t"           // AHと比較するcharを移動させる。
149     "1:t1odb\n\t" "cmpb          // 文字列を取得する char DS:[ESI] to AL, and ESI++.
150     %%ah, %%al\n\t" "jne          // char ALは指定されたchar AHと比較されます。
151     2f\n\t"                      // 同じでなければ、ラベル2に向かってジャンプします。
152     "movl %%esi, %0\n\t"             // EDXにcharのポインタを格納する。
153     "dec1 %0\n\t"                  // ポインタが一致するcharを指すように調整します。
154     "2:\n\t" "jne 1b"                // ALの文字はNULL文字ですか？(文字列の終わり？)
155     // そうでなければ、ラベル1に戻って比較を続ける。
156     :"=d" ( res): "0" (0), "S" (s), "a" (c):
157 return res\";
158 }                                     // ポインタを返します。
159 }

// 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)。
// str2に含まれるstr1の最初の文字列の長さを返します。
160
161 extern inline int strspn(const char * cs, const char * ct)
162 {
163 register char * res asm ("si");
164     asm ("cld\n\t")
165     "movl %4, %%edi\n\t"             // str2のlenを計算し、そのポインタをEDIに格納する。
166     "repne_n\n\t"                  // AL(0)とstr2のchar(ES:[EDI])、EDI++を比較する。
167     "scasb\n\t"                   // 等しくない場合は、比較を続ける(ECX--)。
168     "notl %%ecx\n\t"              // ECXの各ビットは反転しています。
169     "dec1 %%ecx\n\t"              // ECX--, str2の長さを取得します。
170     "movl %%ecx, %%edx\n\t"        // str2の長さをEDXに一時的に入れる。
171     "1:t1ods\n\t"                 // str1のchar DS:[ESI]をALに入れて、ESI++。
172     "testb %%al, %%al\n\t"         // charは0(str1の終わり)と等しいか？
173     "je 2f\n\t"                  // Yesの場合、ラベル2にジャンプして終了します。

```

```

174      "movl %4,%%edi$\n"
175      "movl %%edx,%%ecx$\n"
176      "repne$\n" "scasb$\n"
177      "je 1b$\n"
178      "2:$^w^%0"
179      :=""=5" ( res ): "a" (0), "c" (0xffffffff), "0" (cs), "g"
180      (ct)
182 リターンレバックス "$cx", "dx", "di"); // 一連の文字列の長さを返します。
183 }
184 //// 文字列1の中で、文字列2の中のどの文字も含まない最初の文字列を検索します。
185 // パラメータ: cs - 文字列1のポインタ、ct - 文字列2のポインタ。
186 // %0 - esi( res ), %1 - eax(0), %2 - ecx(-1), %3 - esi (str1 ptr cs), %4 - (str2 ptr ct)。
187 // str1の最初の文字列で、str2のどの文字も含まないものの長さを返します。
188 extern inline int strcspn(const char * cs, const char * ct)
189 {
190     register char * res asm ("si");
191     asm ("cld\n\t")
192     "movl %4,%%edi$\n"
193     "repne$\n"
194     "scasb$\n"
195     "notl %%ecx$\n" "decl
196     %%ecx$\n" "movl
197     %%ecx,%%edx$\n"
198     "1:tlodsb$\n" "testb
199     %%al,%%al$\n" "je 2f$\n"
200     "2:$^w^%0"
201     "movl %4,%%edi$\n"
202     "movl %%edx,%%ecx$\n"
203     "repne$\n" "scasb$\n"
204     "jne 1b$\n"
205     "2:$^w^%0"
206     :=""=5" ( res ): "a" (0), "c" (0xffffffff), "0" (cs), "g"
207     (ct)
208 リターンレバックス "$cx", "dx", "di"); // 一連の文字列の長さを返します。
209 }
210 ...
211 register char * res asm ("si");
212 asm ("cld\n\t")
213     "movl %4,%%edin$\n"
214     "repne_n$\n"
215     "scasb$\n"
216     "notl %%ecx$\n"
217     "dec1 %%ecx$\n"
218     "movl %%ecx,%%edx$\n"

```

```

219      "1:lodsb\$n\$t"           // str1のchar DS:[ESI]をALにして、EDI++にする。
220      "testb %%al, %%al\$t"    // charは0(str1の終わり)と等しいか？
221      "je 2f\$"                // Yesの場合、ラベル2にジャンプします。
222      "movl %4, %%eden\$t"    // str2のポインタをEDIに取り込む。
223      "movl %%edx, %%ecx\$n\$t" // str2の長さをECXに入れる。
224      "repne_n\$"              // ALとstr2を比較する char ES:[EDI], and EDI++.
225      "scasb\$"                // 等しくない場合は、比較を続けます。
226      "jne 1b\$"                // 等しくない場合は、ラベル1に戻ってジャンプします。
227      "dec1 %%on\$"              // ESI--は、str2に含まれるstr1のcharを指します。
228      "jmp 3f\$"                // ラベル3にジャンプして終了
229      "2:txorl %%0, %%0\$n"    // 一致するものが無い場合は、NULLを返します。
230      "3:"                     // 
231      :"S" ( res ): "a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
232      :"ax", "cx", "dx", "di");
233 return res;                                // str1のポインタを返す。
234 }
235
236 extern inline char * strstr(const char * cs, const char * ct)
237 {
238 register char * res asm ("ax"); 239
asm ("cld\$n\$t" ^w^..)
239      "movl %4, %%edi\$t"       // str2のlenを計算し、そのポインタをEDIに格納する。
240      "repne\$"                 // AL(0)とstr2のchar(ES:[EDI])、EDI++を比較する。
241      "scasb\$"                  // 等しくない場合は、比較を続ける(ECX--)。
242      "notl %%ecx\$t"          // ECXの全ビットとdec 1を反転させてstr2のlenを算出する
243      "decl %%ecx\$t" /* 注意! 。searchstring='' の場合はZも設定されます。
244      "movl %%ecx, %%edx\$n\$/" // str2の長さをタemporarily格納する。
245      "1:tmovl %4, %%edi\$t"   // str2の長さをEAXにコピーします。
246      "movl %%esi, %%eax%"     // str1のポインタをEAXにコピーします。
247      "movl %%edx, %%ecx%"     // str2の長さをECXに入れる。
248      "repe\$n\$t"               // Comp str1, str2 char DS:[ESI], ES:[EDI], ESI++, EDI++.
249      "cmpsb\$n\$t"              // 文字が等しい場合は、比較を続けます。
250
251      "je 2f\$" /* 空の文字列でも動作しますが、上記を参照してください
252      "xchgl %%eax, %%esi\$t" // str1_ptr => ESI, 比較結果 str1_ptr => EAX.
253      "clock %%esi%%n\$t"     // str1ポインターは次のcharを指す。
254      "cmpb $0,-"             // str1のptrが指すバイトは(EAX-1) 0か？
255      "1(%eax)in\$t" "jne    // そうでなければ、label1に進み、str1の2番目の文字から
256      1b\$t"                   // コンパイルを続ける。
257      "xorl %%eax, %%eax\$n\$t" // EAXをクリアして、マッチしなかったことを示す。
258      :"Z=a" ( res ): "0" (0), "c" (0xffffffff), "S" (cs), "g"
259      (ct)
260 return res;"cx", "dx", "di", "si"); // 結果を返します。
261 }
262
263 ///////////////////////////////////////////////////////////////////
264 // 文字列の長さを計算する。
265 // パラメータ: s - 文字列。
266 // %0 - ecx( res ), %1 - edi (string pointer s), %2 - eax(0), %3 - ecx(0xffffffff)
267 // です。
268 // 文字列の長さを返します。

```

```

263 extern inline int strlen(const char * s)
264 {
265 register int res asm ("cx"); 266 // resはレジスタ変数(ECX)です。
asm ("cld\n\t") // 演出フラグをクリアします。
267 "repne\#\#"
268 "scasb\#\#"
269 "notl %0\#"
270 "dec1 %0\#"
271 : "=c"( res): "D"(s), "a"(0), "0'(0xffffffff): "di"\")とな
272 return res; // 文字列の長さを返します。
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; // 文字列トークン。 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、ヌル文字は
    // は、トークンが返されたときにセパレータに書き込まれます。その後の strtok() の呼び出しでは
    // 最初の引数にnullを指定すると、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。
    // %0=ebx( res), %1=esi( strtok); %2=ebx( strtok), %3=esi(str1 ptr s), %4=(str2 ptr ct)。
    // 文字列 s のトークンを返します。トークンが見つからない場合は null ポインタを返します。後続の呼び出し
    // null 文字列 s を strtok() に渡すと、元の文字列 s の中から次のトークンを探します。
277 extern inline char * strtok(char * s, const char * ct)
278 {...}
279 register char *res asm ("si"); // そうでない場合は、最初の呼び出しを意味し、ラベル1に
280 asm ("testl %1,%1\#\#t" // 最初に ESI(ストリップ) が NULL かどうかをテストします。
281     "testl %0,%0n\#\#"
282     "je 8f\#\#"
283     "movl %0,%1\#"
284     "1:txorl %0,%0n\#\#t"
285     "movl $-1,%ecx\#\#t"
286     "xorl %%eax, %%eax\#\#t"
287     "cld_n\#"
288     "movl %4,%eden\#\#t"
289     "repne_n\#\#"
290     "scasb\#\#"
291
292     "notl %%ecx\#\#"
293     "dec1 %%ecx\#\#"
294     "je 7f\#\#"
295     "movl %%ecx, %%edx\#\#n"
296     "2:t1odsb\#\#t"
297     "testb %%al, %%al\#\#t"
298     "je 7f\#\#"
299     "movl %4,%eden\#\#t"
300     "movl %%edx, %%ecx\#\#t"
301     "repne_n\#\#"
302     "scasb\#\#"
303     "je 2b,\^w,\^"
    // そうでない場合は、最初の呼び出しを意味し、ラベル1に
    // NULLの場合は、後続の呼び出しを意味し、EBX( strtok) をテストします。
    // EBXがNULLの場合は、処理できないので最後までジャンプします。
    // EBXポインタ(現在のstr1 ptr)をESIにコピーします。
    // EBXをクリアします。
    // ECX = 0xffffffff を設定します。
    // EAXをクリアします。
    // 演出フラグのクリア
    // str2の長さを求めよう。EDIはstr2を指します。
    // AL(0)とES:[EDI], EDI++のcharを比較します。
    // str2の終わり(null char)まで、またはカウント ECX = 0.
    // ECXを逆にして1,を引くと、str2の長さになります。
    // str2 lenが0の場合、ラベル7に進む。
    /* 空のデリケートな文字列 */
    // str2の長さをEDXに一時的に格納する。
    // str1のchar DS:[ESI]をALにして、ESI++にする。
    // charは0(str1の終わり)ですか?
    // Yesの場合、ラベル7にジャンプします。
    // EDIは再びstr2の最初のcharを指します。
    // str2の長さをカウンタにコピー ECX。
    // str1のchar ALとstr2の全charsを比較する。
    // その文字がセパレータであるかどうかをチェックします。
    // str2で見つかった場合は、ラベル2に逆方向にジャンプします。

```



```

349 asm ("cld\n\t"
350     "rep\n\t"
351     "movsb"
352     : "c"(n), "S"(src), "D"(dest)。
353     の順で表示されます。
354 他    "cx", "si", "di")。
355 asm ("std\n\t")
356     "rep\n\t"
357     "movsb"
358     : "c"(n),      (src+n-1), "D" (dest+n-1)
359 return dest;
360     "S"
361     "cx", "si", "di")。
362
363     // 方向を設定し、端からコピーを開始する。
364     // DS:[ESI++]からES:[EDI++]へ。
365     // ECXバイトを繰り返しコピーしま
366     //す。
367     :: "c" (n),      (src+n-1), "D" (dest+n-1)
368     // デスクトップアドレスを返
369     //します。
370
371     // 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ない。
372     // 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
373     // バイト数。
374     // %0 - eax(res), %1 - eax(0), %2 - edi(mem block1), %3 - esi(mem block2), %4 -
375     // ecx(count)。
376     // block1 > block2の場合は1を、block1 < block2の場合は-1を、block1 == block2の場合は0を返し
377     // ます。
378
379     extern inline int memcmp(const void * cs, const void * ct, int count)
380     {...    "repeat\n\t" // DS:[ESI++]とES:[EDI++]を比較する。
381     register int res asm ("res\n\t" 366 // resは、比較を続けます。
382     ("cld\n\t"
383     "je 1f\n\t" // すべて同じなら、ラベル1にジャンプし、0(EAX)を返す。
384     "movl $1,%eax\n\t" // else set EAX to 1.
385     "jl 1f\n\t" // mem block2 の値が <block1> の場合、ラベル 1 にジャンプします。
386     "negl %%eax\n\t" // else invert EAX value.
387     "1:""
388     : "=a" (res): "0" (0), "D" (cs), "S" (ct), "c" (count)の順で表示されます。
389     "si", "di", "cx")。
390
391     return res;           // 比較結果(EAX単位)を返します。
392
393     // 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。
394     // Params: cs - メモリブロックのアドレス、c - 指定された文字、count - 比較サイズ。
395     // %0 - edi(res), %1 - eax(char c), %2 - edi(mem block address cs), %3 - ecx(num of
396     // bytes).
397     // 最初にマッチした場所へのポインタを返し、見つからない場合はNULLを返します。
398
399     extern inline void * memchr(const void * cs, char c, int count)
400     {
401     register void * res asm ("di");
402     if (!coNHLを返す)モリブロックサイズが0の場合は、NULLが返されます。
403     asm ("cld\n\t"
404     "repne_n\n\t"           // ALのcharをES:[EDI++]と比較します。
405     "scasb\n\t"            // 等しくなければ繰り返す(最大ECX倍まで)。
406     "je 1f\n\t"             // 等しい場合は、ラベル1にジャンプします。
407     "movl $1,%0\n\t"        // elseはEDIを1に設定します。
408     "1:tdec1 %0"           // EDIに検索されたchar(またはNULL)を指定させます。
409     : "=D" (res): "a" (c), "D" (cs), "c" (count)
410     : "cx")。
411
412     return res; // char のポインタを返します。
413 }

```

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。
//%0 - eax (char c), %1 - edi (memory address s), %2 - ecx (number of bytes count).
```

395 extern inline void * memset(void * s, char c, int count)

396 {

397 asm ("cId~~YY~~"
398 "rep~~YY~~" // ECXを繰り返します。
399 "stosb" // ALのcharをES:[EDI++]に埋めます。
400 :: "a"(c)、"D"(s)、"c"(count)
401 : "cx"、"di") .
402 リターンズ // メモリブロックのポインタを返します。
403 }

404

405 #endif

406

11. termios.h

1. 機能性

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の読み取りや設定などに使用されます。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、現在は使用されていません。

2. コードアノテーション

1 #ifndef TERMIOS_H

2 #define TERMIOS_H プログラム 14-10 linux/include/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ークにするためのマジックナンバー ('T'
) */ 9

// 以下は、ttyのioctlが使用するコマンドセットで、ローバイトでエンコードしています。

```

// 端末のtermios構造体の情報を取得します (tcgetattr()参照)。
10 #define TCGETS 0x5401 // Terminal Command GET Settings.
// 端末のtermios構造体に情報を設定する (TCSANOWのtcsetattr()参照)。
11 #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。
// 出力に影響を与える(tcsetattr()のTCSADRAINオプション参照). (TCSETSW - TCSETS
Wait) 12 #define TCSETSW 0x5403
// termios情報を設定する前に、出力キューにあるすべてのデータを待つ必要がある
// 処理するために、入力キューをフラッシュ（クリア）します (tcsetattr()のTCSAFLUSHオプション
参照)。
13 #define TCSETSF 0x5404
// termio構造体の属性情報を取る (tcgetattr()参照)。
14 #define TCGETA 0x5405
// termio構造体の属性情報を設定する (tcsetattr()、TCSANOWオプション参照)。
15 #define TCSETA 0x5406
// termioの属性情報を設定する前に、すべてのデータが
// 出力キューを処理する（ランアウトする）。を変更する際には、このタイプのコマンドが必要です
。
// 出力に影響を与えるパラメータ (tcsetattr()のTCSADRAINオプション参照)。
16 #define TCSETAW 0x5407
// termioの属性を設定する前に、出力キュー内のすべてのデータが
// 処理され、入力キューをフラッシュ（使い切る）します (tcsetattr()のTCSAFLUSHオプション参照
)。
17 #define TCSETAF 0x5408
// 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、ブレークを送る
// (tcsendbreak()、tcdrain()参照)。
18 #define TCSBRK 0x5409
// 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、それが2の場合は入力が中断され、それが3の場合は保留中の
// 入力は再び開かれます (tcflow()参照)。
19 #define TCXONC 0x540A
// 書き込まれたが送信されていない、または受信されたがまだ送信されていないデータをフラッシュ
します。
// 読み込みます。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、出力の
// キューがフラッシュされます。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)。
20 #define TCFLSH 0x540B
// 端末のシリアルラインの排他モードを設定します。
21 #define TIOCEXCL 0x540C // Terminal I/O Control Exclude.
// 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
// 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)。
23 #define TIOCSCTTY 0x540E
// 指定した端末機器プロセスのグループIDを取得する (tcgetpgrp()参照)。
24 #define TIOCGPGRP 0x540F // Terminal IO Control Get PGRP.
// 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)。
25 #define TIOCSPGRP 0x5410
// 送信されていない出力キューの文字数を返します。
26 #define TIOCOUTQ 0x5411
// 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
// 端末上で文字が入力されたように見せかける。ユーザはスーパーユーザ権限を持っている必要があります
// または制御端末の読み取り許可
27 #define TIOCSTI 0x5412          948
// 端末デバイスのウィンドウサイズ情報を取得する (winsize構造体を参照)。
28 #define TIOCGWINSZ 0x5413
// 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)。
29 #define TIOCSWINSZ 0x5414

```

```

// MODEMのステータスコントロールピンに設定されている現在のステータスピットフラグを返します
(185-196行目参照)。
30 #define TIOCGET_ 0x5415
    // 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。
31 #define TIOCMBIS_ 0x5416
    // 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。
32 #define TIOCMBIC_ 0x5417
    // MODEMの状態制御ラインの状態を設定します。ビットが設定されている場合、対応するステータス
    ラインの
    MODEMへの//が有効になるように設定されます。
33 #define TIOCSET_ 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
    // 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っており、端末ではオフになっています
    。
    // またはMODEM回線を使用する機器。これらの2つのioctlコールを使用できるようにするために、tty
    // 行は O_NDELAY モードで開く必要があり、open() はキャリアを待ちません。
34 #define TIOCGSOFTCAR_ 0x5419
35 // フローティングキャリア検出フラグを設定します (1 - オン、0 - オフ)。
36 #define TIOCINQ_ 0x541A
37 // メモリ領域中のオフセット構造体。返却しない文字の数を返しますに使用することができます。
38 #define TIOCGWINSZ_ 0x541B
    // ワインド文字列の構造体。符号なしショードタイプ。
39 struct winsize {
    short ws_row;           // ウィンドウ高さ。
    short ws_col;           // ウィンドウ文字列の列。
    short ws_xpixel;        // ウィンド幅をピクセル単位
    short ws_ypixel;        // ウィンド高さをピクセル
40 // AT&Tシステムのタ下ミオ構造 V.
41 // 符号なしショード型で表示します。
42 // struct winsize {
43 // 符号なしショード型で表示します。
44 // #define NCC_ 8 // termioの制御文字配列のサイズです。 47 struct termio {
45
46
47 struct termio {
48     short c_iflag;          /* 入力モードフラグ */
49     short c_oflag;          /* 出力モードフラグ */
50     short c_flag;           /* 制御モードフラグ */
51     short c_lflag;          /* ローカルモードフラグ */
52     char c_line;            /* ラインの規律 */
53 // POSIX仕様で定義されているtermios構造です * 制御文字 */
54 #define NCCS_ 17 // 制御文字配列のサイズをtermiosで指定します。 57 struct termios {
55
56     tcflag_t c_iflag;       /* 入力モードフラグ */
57     tcflag_t c_oflag;       /* 入力モードフラグ */
58     tcflag_t c_flag;        /* 出力モードフラグ */
59     tcflag_t c_lflag;        /* 制御モードフラグ */
60     ec_t c_line;           /* 制御モードフラグ */
61     cc_t c_cc[NCCS];        /* 制御文字モードフラグ */
62
63 } ;
64
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
// この配列の値は、include/linux/tty.hで定義されていますが、プログラムは

```

```

// この配列。POSIX_VDISABLE(0)が定義されている場合は、配列の値が
// _POSIX_VDISABLE の場合は、配列中の特殊文字の使用が禁止されていることを意味します。
66 /* c_cc キャラクタ */
-
67 #define VINTR 0          // c_cc[VINTR].    = INTR   (^C), ¥003.
68 #define VQUIT 1          // c_cc[VQUIT].    = QUIT   (^¥), ¥034.
69 #define VERASE 2         // c_cc[VERASE].   = ERASE  (^H), ¥177.
70 #define VKILL 3          // c_cc[VKILL].    = KILL   (^U), ¥025.
71 #define VEOF 4           // c_cc[VEOF].     = EOF    (^D), ¥004.
72 #define VTIME 5          // c_cc[VTIME].    = タイム (¥0), ¥0. タイマーの値（後述）。
73 #define VMIN 6           // c_cc[VMIN].     = MIN    (¥1), ¥1. タイマーの値です。
74 #define VSWTC 7          // c_cc[VSWTC].   = SWTC   (¥0), ¥0. スイッチチャ一。
75 #define VSTART 8         // c_cc[VSTART].  = START  (^Q), ¥021.
76 #define VSTOP 9          // c_cc[VSTOP].   = STOP   (^S), ¥023.
77 #define VSUSP 10         // c_cc[VSUSP].   = SUSP   (^Z), ¥032. サスペンド・チャ一
78 #define VEOL 11          // c_cc[VEOL].     = EOL    (¥0); ¥0.
79 #define VREPRINT 12       // c_cc[VREPRINT] = REPRINT (¥R); ¥0.
80 #define VDISCARD 13       // c_cc[VDISCARD] = DISCARD (^O), „^w“.
81 #define VWERASE 14        // c_cc[VWERASE]. = WERASE  (^W), „^w“ 어플을 사용하고
82 #define VLNEXT 15        // c_cc[VLNEXT].  = LNEXT   있습니다. (^V), ¥026.
83 #define VEOL2 16          // c_cc[VEOL2].   = EOL2   (¥0), ¥0.
84                         // c_cc[VEOL2].
-
// 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。
85 /* c_iflag ビッ */
ト          0000001      // 入力時のBREAK条件を無視します。
86 #define IGNBRK 0000002      // BREAKでシギント信号を発生させる。
87 #define BRKINT 0000004      // パリティエラーの文字を無視します。
88 #define IGNPAR 0000010      // マークパリティエラー。
89 #define PARMRK 0000020      // パリティチェックのための入力を許可する。
90 #define INPCK 0000040       // キャラクターの8ビット目をマスクします。
91 #define ISTRIP 0000100      // ラインフィードNLをキャリッジリターンCRにマッピングする。
92 #define INLCR 0000200       // CRは無視してください。
93 #define IGNCR 0000400       // CRをNLにマッピングする。
94 #define ICRNL 0001000      // 大文字の文字を小文字に変換します。
95 #define IUCLC 0002000      // スタート/ストップ(XON/XOFF)の出力制御が可能
-
96 #define IXON 0004000      // 任意の文字で出力を再開できるようになる。
97 #define IXANY 0010000      // スタート/ストップ(XON/XOFF)の入力制御が可能。
98 // 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。
99 // 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。
100 #define I_SETSIG 0020000      // 入力キューがいっぱいになったときに鳴る。
101 #define I_SETSIG 0000001      // 出力処理を行う。
102 #define OPOST 0000002      // 小文字を大文字に変換します。
103 #define OLCUC 0000004      // NLをCR-NLにマッピングします。
104 #define ONLCR 0000010      // CRをNLにマッピングする。
105 #define OCRNL 0000020      // CRは0列目には出力されません。
106 #define ONOCR 0000040      // NLはキャリッジリターンの機能を果たします。
107 #define ONLRET 0000100      // 遅延時には、時間遅延の代わりにフィル・チャールを使用
-
108 #define OFILL 0000200      // fill charはDELです。設定されていない場合、デフォルト
109 #define OFILL 0000001      // はNULLです。
110 #define NLDLY 0000400      // ラインフィードディレイを選択します。
111 #define NL0 0000000      // NLディレイタイプ0。
112 #define NL1 0000400      // NLディレイタイプ1。
113 #define CRDLY 0003000      // キャリッジリターンの遅延を選択します。
114 #define CRO 0000000      // CR遅延タイプ0。

```

```

115 #define CR1 0001000 // CR 遅延 タイプ1です。
116 #define CR2 0002000 // CR 遅延 タイプ2です。
117 #define CR3 0003000 // CR 遅延 タイプ2です。
118 #define TABDLY 0014000 // 水平方向のTABディレイを選択します。
119 #define TAB0 0000000 // TABディレイタイプ0
120 #define TAB1 0004000 // TABディレイタイプ 1.
121 #define TAB2 0010000 // TABディレイタイプ 2.
122 #define TAB3 0014000 // TABディレイタイプ 3.
123 #define XTABS 0014000 // TABをスペースに置き換えると、数字になりま のスペ
                           //ースを
                           //利用し
                           //ていま
                           //す。
124 #define BSDLY 0020000 // バックスペース (BS) のディレイを選択しま
                           //す。
125 #define BS0 0000000 // BSディレイタイプ0。
126 #define BS1 0020000 // BS遅延タイプ1。
127 #define VTDLY 0040000 // 垂直方向のタブレーション遅延を選択しま
                           //す。
128 #define VTO 0000000 // VT遅延タイプ0。
129 //この ディレクトリもこの フォルダ c_flag で使用されるフラグを termios (8進数) で
130 // 表したもの
131 #define FFDLY 0040000 // フォームフィードディレイを選択します。
132 #define FF0 0000000 // 送信ポートのビットマスク
133 #define FF1 0000000 // FF遅延タイプ1。
134 #define B50 0000001 // フォームフィード
135 #define B75 0000002 // フォームフィード
136 #define B110 0000003 // フォームフィード
137 #define B134 0000004 // フォームフィード
138 #define B150 0000005 // フォームフィード
139 #define B200 0000006 // フォームフィード
140 #define B300 0000007 // フォームフィード
141 #define B600 0000010 // フォームフィード
142 #define B1200 0000011 // フォームフィード
143 #define B1800 0000012 // フォームフィード
144 #define B2400 0000013 // フォームフィード
145 #define B4800 0000014 // フォームフィード
146 #define B9600 0000015 // フォームフィード
147 #define B19200 0000016 // フォームフィード
148 #define B38400 0000017 // フォームフィード
149 #define EXTA_B19200 // 拡張ポート A.
150 #define EXTB_B38400 // 拡張ポート B.
151 #define EXTB_B19200 // 拡張ポート B.
152 #define EXTB_B38400 // 拡張ポート B.
153 #define CSIZE 0000060 // 文字のビット幅のマスク。
154 #define CS5 0000000 // 1文字あたり5ビット。
155 #define CS6 0000020 // 6ビットです。
156 #define CS7 0000040 // 7ビットです。
157 #define CS8 0000060 // 8ビットです。
158 #define CSTOPB 0000100 // ストップビットを1つではなく2つ設定する。
159 #define CREAD 0000200 // 受信を許可する。
160 #define PARENB 0000400 // パリティ演算を有効にする。
161 #define PARODD 0001000 // 入力チェックは奇数チェック。
162 #define HUPCL 0002000 // 最後のプロセスが終了したら電話を切る。
163 #define CLOCAL 0004000 // MODEMの制御線を無視する。
164 #define CBAUD 03600000 // 入力ボーレート (使用しない) */
165 #define CIBAUD

```

```

166 #define CRTSCTS 020000000000 /* フローコントロール */ (英語)
167 // 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
。
168 #define I_SETSIG 0000001 // INTR、QUIT、SUSP、DSUSPのいずれかを選択したときに発生する信号です。
170 #define ICANON 0000002 // カノニカルモード（クックドモード）を有効にします。
171 #define XCASE 0000004 // ICANONが設定されている場合、ターミナルは大文字を表示します。
172 #define ECHO 0000010 // ICANONが設定されている場合、ターミナルは大文字を表示します。
173 #define ECHOE 0000020 // 入力された文字をエコーする。
174 #define ECHOK 0000040 // ICANONの場合、ERASE/WERASEは前の文字/単語を消去します。
175 #define ECHONL 0000100 // SIGINT および SIGQUIT シグナルが発生する場合、入力は現在の行を消されません。
// また、SIGSUSP信号が発生すると、ICANONの場合、ECHOが有効になっていなくてもNLがエコーされます。
176 #define NOFLSH 0000200 // 書き込みを行お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シグナルを送信します。
// 独自の制御端子を備えています。
177 #define TOSTOP 0000400 // ECHOが設定されている場合、TAB、NL、START、STOP以外のASCII制御文字がエコーされます。
// 「^X」パターンに戻り、X値は制御値+0x40となります。
178 #define ECHOCTL 0002000 // ICANON & IECHOの場合、消去時に文字が表示されます。
180 #define ECHOKE 0004000 // ICANON、KILLの場合は、その行の各文字を消してエコーをかけます。
181 #define FLUSHO 0010000 // 出力がフラッシュされます。これはDISCARD charで切り替えることができます。
182 #define PENDIN 0040000 // 次の文字が読み込まれたときに、待ち行列の中のすべての文字が再印刷されます。
183 #define IEXTEN 0100000

185 /* モデムライン */
/*
186 #define TIOCM_LE 0x001 // 実装で対応しない。入力処理を有効にする。
187 #define TIOCM_DTR 0x002 // データターミナルレディ。
188 #define TIOCM_RTS 0x004 // 送信のリクエスト。
189 #define TIOCM_ST 0x008 //シリアル転送。
190 #define TIOCM_SR 0x010 //シリアル受信。
191 #define TIOCM_CTS 0x020 // Clear To Send.
192 #define TIOCM_CAR 0x040 // キャリアディテクト。
193 #define TIOCM RNG 0x080 // リングを示す。
194 #define TIOCM_DSR TIOCM_CAR // データセットレディ。
195 #define TIOCM_CD TIOCM RNG
196 #define TIOCM RI

198 /* tcflow() と TCXONC はこれらを使
用しません */
200 #define TCOFF 0 // 出力を一時停止する（「端子制御出力OFF」）。
201 #define TCOON 1 // サスPENDされた出力を再起動する。
202 #define TCIOFF 2 // STOPを送信して、システムにデータを送信しているデバイスを停止します。
203 #define TCION 3 // デバイスがシステムへの送信を開始するためにSTARTを送信

204 /* tcflush() と TCFLSH はこれらを使
用しません */
205 #define TCIFLUSH 0 // 受信したデータはクリアするが、読まない。
206 #define TCOFLUSH 1 // 出力データはクリアするが、送信はしない。
207 #define TCIOFLUSH 2 // データの出し入れはクリアするが、読み書きはしない。

209 /* tcsetattr はこれらを使
用しません */
210 #define TCSANOW 0 // 今すぐ設定を有効にする。
211 #define TCSADRAIN 1 // すべての出力が送信された後に変更されます。

```

```

212 #define TCSAFLUSH      2          // 入力/出力がすべてフラッシュされた後に変更されます
213                                     .
// 以下の関数は、ビルト中の関数ライブラリ libc.a に実装されています。
// カーネルではなく、// 環境で使用されます。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()を呼び出すことにより // 行う。ioctl()については、fs/ioctl.c プ
// ログラムを参照すること。

// termios_p で参照される termios 構造体の受信ボーレートを返す。
214 extern speed_t cfgetispeed(struct termios *termios_p);
// termios構造体の送信ボーレートを返す。 215 extern
speed_t cfgetospeed(struct termios *termios_p);
// termios構造体の受信ボーレートを'speed'に設定する。 216 extern
int cfsetispeed(struct termios *termios_p, speed_t speed);
// termios構造体に送信ボーレートを設定する。
217 extern int cfsetospeed(struct termios *termios_p, speed_t speed);
// 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。
218 extern int tcdrain(int fildes);
// 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。
219 extern int tcflow(int fildes, int action);
// 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データをすべて破棄す
// るとともに
受信したが、まだ読まれていないすべてのデータとして、//。
220 extern int tcflush(int fildes, int queue_selector);
// 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。
221 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
// 一定時間連続して送信される。
222 extern int tcsendbreak(int fildes, int duration);
223 // 端末に関連するtermiosメソッドを設定するために、termios構造のデータを使用します。
224 extern int tcsetattr(int fildes, int optional_actions,
225 #endif
227

```

3. インフォメーション

1. 制御文字 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。MIN(c_cc[VMIN])とTIMEの設定について

(termios構造体のc_cc[VTIME]) 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を読み取るか、そしていつユーザー プログラムに戻るかを決定するために使用されます。

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

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。

◆ $min > 0$ 、 $time = 0$ です。

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、待機中の読み出し動作が満たされることになる（待機中の動作はMIN文字を受信するまでスリープする）。これをを利用してレコードベースのターミナルIOを読むプログラムはブロックされる

（読み込み時に、（任意に）無期限に

◆ $min = 0$ 、 $time > 0$ となります。

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すぐに復帰します。

12. time.h

1. 機能性

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る。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。

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）、4つの型、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 コードアノテーション

プログラム 14-11 linux/include/time.h

```

1 ifndef _TIME_H
2 define _TIME_H
3
4 ifndef _TIME_T
5 define _TIME_T
6 つのタイ long time_t; // 1970年1月1日午前0時のGMTからの時間です。
    プデフ
7 endif
8
9 ifndef _SIZE_T
10 define _SIZE_T
11 typedef 符号付き整数 size_t
12 endif
13
14 ifndef NULL
15 define NULL ((void *) 0)
16 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通過したティック。
21
22 構造体 tm {
23     int tm_sec; // 秒数 [0, 59].
24     int tm_min; // 分 [0, 59]。
25     int tm_hour; // 時間 [0, 59].
26     int tm_mday; // 月の日数 [0, 31].
27     int tm_mon; // 1年のうちの月数 [0, 11].
28     int tm_year; // 1900年からの年数です。
29     int tm_wday; // 曜日 [0, 6] (日曜日 =0)
30     int tm_yday; // 1年のうちの1日 [0, 365] - 使用中、 = 0 - 不使用、 < 0 -
31     tm_isdst; // 無効。
32 }
33
34 // うるう年かどうかをチェックするマクロです。
#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。
// 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。
37 clock_t clock(void);
// 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。
38 time_t time(time_t * tp);
// 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。
39 double difftime(time_t time2, time_t time1);
// tm構造体で表される時間をカレンダー時間に変換する。
40 time_t mktime(struct tm * tp);
41
// 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。
42 char * asctime(const struct tm * tp);
// カレンダーの時間を "Wed Jun 30 21:49:08:1993" のように文字列に変換します。
43 char * ctime(const time_t * tp);
// 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。
44 struct tm * gmtime(const time_t *tp);
// 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。
45 struct tm * localtime(const time_t * tp);
// tm構造体が表す時間を、最大長の文字列'smax'に変換します。
// フォーマット文字列'fmt'を使用して、結果を's'に格納します。
46 size_t strftime(char * s, size_t smax, const char * fmt, const struct tm * tp);
// 時間の変換情報を初期化し、変数znameの初期化を
// この関数は、時間変換機能で自動的に呼び出されます。
// タイムゾーンに関連付けられています。
47 void tzset(void);
48
49 #endif
50

```

13. unistd.h

1. 機能性

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。このファイルには、さまざまなシンボル定数や型が定義されており、関数の宣言もあります。如しシンボル
プログラムの中でLIBRARYが定義されていると、カーネルのシステムコール番号とインラインアセンブリマクロの`_syscall0()`、`_syscall1()`なども含まれます。

2. コードアノテーション

プログラム 14-12 `linux/include/unistd.h`

```

1 ifndef UNISTD_H
2 define UNISTD_H
3
4 /* OK, これはジョークかもしれないが、私はそれに取り組んでいる*/
// 以下のシンボル定数は、IEEE Standard 1003.1の実装バージョンを示しています。
カーネルが従うことを示す // 整数値である。

```

```

5 #define _POSIX_VERSION 198808L
6
7 #define _POSIX_CHOWN_RESTRICTED /* rootのみがchownを行える（と思う） */.
// 8 #define _POSIX_NO_TRUNC /* パス名の切り詰めは行わない（ただし、カーネルを参照） */
#define _POSIX_NO_TRUNC 8#define _POSIX_NO_TRUNC /* パス名の切り詰めは行わない（ただし、カーネルでは参照） */
// _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します。
// 端末のtermios構造体のc_cc[]配列の文字コード値が等しい場合
// を_POSIX_VDISABLEの値に設定すると、対応する特殊文字の機能が
// は禁止です。
9 #define _POSIX_VDISABLE '\0' /* ^Cなどを無効にする文字 */。
// 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
11 #ifdef _POSIX_SAVED_IDS
12 // 各プロセスが保存されたset-user実装が保存された何かの役割を持つかもしれません。
13 #define STDIN_FILENO 0 // 標準入力ファイルのハンドル（記述子）番号。
14 #define STDOUT_FILENO 1 // 標準出力ファイルのハンドル番号。
15 #define STDERR_FILENO 2 // 標準エラー出力ファイルのハンドル番号
16
17 #ifndef NULL
18 #define NULL ((void *)0) // null ポインタの値を定義する。
19 #endif
20
// access()関数では、以下のシンボル定数が使用されます。
21 /* アクセス */
22 #define F_OK 0 // チェック ファイルが存在する場合は
23 #define X_OK 1 // チェック 実行可能（検索可能）であれば
24 #define W_OK 2 // チェック が書き込み可能であれば
25 #define R_OK 4 // チェック 読むことができれば
26
// 以下のシンボル定数は、lseek() および fcntl() 関数で使用されます。
27 /* lseek */
28 #define SEEK_SET 0 // ファイルの読み取り/書き込みポインタをオフセットに設定します。
29 #define SEEK_CUR 1 // 現在の値にオフセットを加えた値に設定します
30 #define SEEK_END 2
31 .
// 以下のシンボル定数は、sysconf()関数で使用され、長さにオフセットを加えた値を設定
32 /* _SC は System Configuration の略ですあまり使用しません */
33 #define SC_ARG_MAX 1 // 最大数 の の議論をしていま
34 #define SC_CHILD_MAX 2 // 最大数 の 子プロセス。
35 #define sc_clocks_per_sec 3 // 1秒あたりのクリック数
36 #define sc_ngroups_max 4 // 最大数 の のグループがあります。
37 #define SC_OPEN_MAX 5 // 最大数 の のファイルを開いた。
38 #define sc_job_control 6 // ジョブコントロール
39 #define SC_NNFS 7 // pathconf()関数では、以下のシンボル定数を使用した識別子。
40 #define SC_NNFS 7 // 多くの（おそらく）設定可能なものの今度はパス名 */
41 #define PC_LINK_MAX 1 // 最大数の のリンクをクリックします。
42 #define PC_MAX_CANON 2 // 最大数の レギュラーファイルです。
43 #define PC_MAX_INPUT 3 // 最大の入力長です。
44 #define PC_NAME_MAX 4 // 最大の長さは という名前になります。

```

```

47 #define _PC_PATH_MAX      5    // パスの最大長です。
48 #define _PC_PIPE_BUF       6    // パイプのバッファサイズ
49 #define _PC_NO_TRUNC        7    // ファイル名が切り捨てられていない。
50 #define _PC_VDISABLE        8    // 指定されたコントロールチャーは無効にな
51 #define _PC_CHOWN_RESTRICTED 9    っています。
52                                         // オーナーの変更が制限されます。
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造
// 体を含む stat{}。
// そして、定数です。
// <sys/time.h> timeval構造体と itimerval構造体が定義されています。
// <sys/times.h> ランニングタイム構造体tmsと、関数プロトタイプtimes()を定義しています。
// 過程です。
// <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。システムの限界と利用に関する情報が含まれています。
// プロセスが使用するリソース。
// <utime.h> ユーザータイムのヘッダーファイルです。アクセス時間と修正時間の構造体とutime()
// プロトタイプが定義されています。
53 #include <sys/stat.h>
54 #include <sys/time.h>
55 #include <sys/times.h>
56 #include <sys/utsname.h>.
57 #include <sys/resource.h>.
58 #include <utime.h>
59
60 #ifdef LIBRARY
61
// 以下は、カーネルが実装するシステムコールシンボルの定数で、インデックスとして使用されます
62 #define _NR_setup          0    /* システムを起動するためにinitでのみ使用さ
63 #define _NR_exit           1    関数表（include/re直接/sys.h参照）にある
64 #define _NR_fork           2
65 #define _NR_read            3
66 #define _NR_write           4
67 #define _NR_open             5
68 #define _NR_close            6
69 #define _NR_waitpid         7
70 #define _NR_creat            8
71 #define _NR_link             9
72 #define _NR_unlink           10
73 #define _NR_execve           11
74 #define _NR_chdir            12
75 #define _NR_time             13
76 #define _NR_mknod            14
77 #define _NR_chmod            15
78 #define _NR_chown            16
79 #define _NR_break            17
80 #define _NR_stat             18
81 #define _NR_lseek            19
82 #define _NR_getpid           20
83 #define _NR_mount            21
84 #define _NR_umount           22
85 #define _NR_setuid            23
86 #define _NR_getuid           24
87 #define _NR_stime            25

```

88 #define	<u>NR_ptrace</u>	26
89 #define	<u>NR_alarm</u>	27
90 #define	<u>NR_fstat</u>	28
91 #define	<u>NR_pause</u>	29
92 #define	<u>NR_utime</u>	30
93 #define	<u>NR_stty</u>	31
94 #define	<u>NR_gtty</u>	32
95 #define	<u>NR_access</u>	33
96 #define	<u>NR_nice</u>	34
97 #define	<u>NR_ftime</u>	35
98 #define	<u>NR_sync</u>	36
99 #define	<u>NR_kill</u>	37
100 #define	<u>NR_rename</u>	38
101 #define	<u>NR_mkdir</u>	39
102 #define	<u>NR_rmdir</u>	40
103 #define	<u>NR_dup</u>	41
104 #define	<u>NR_pipe</u>	42
105 #define	<u>NR_times</u>	43
106 #define	<u>NR_prof</u>	44
107 #define	<u>NR_brk</u>	45
108 #define	<u>NR_setgid</u>	46
109 #define	<u>NR_getgid</u>	47
110 #define	<u>NR_signal</u>	48
111 #define	<u>NR_geteuid</u>	49
112 #define	<u>NR_getegid</u>	50
113 #define	<u>NR_acct</u>	51
114 #define	<u>NR_phys</u>	52
115 #define	<u>NR_lock</u>	53
116 #define	<u>NR_ioctl</u>	54
117 #define	<u>NR_fcntl</u>	55
118 #define	<u>NR_mpx</u>	56
119 #define	<u>NR_setpgid</u>	57
120 #define	<u>NR_ulimit</u>	58
121 #define	<u>NR_uname</u>	59
122 #define	<u>NR_umask</u>	60
123 #define	<u>NR_chroot</u>	61
124 #define	<u>NR_ustat</u>	62
125 #define	<u>NR_dup2</u>	63
126 #define	<u>NR_getppid</u>	64
127 #define	<u>NR_getpgrp</u>	65
128 #define	<u>NR_setsid</u>	66
129 #define	<u>NR_sigaction</u>	67
130 #define	<u>NR_getmask</u>	68
131 #define	<u>NR_ssetmask</u>	69
132 #define	<u>NR_setreuid</u>	70
133 #define	<u>NR_setregid</u>	71
134 #define	<u>NR_sigsuspend</u>	72
135 #define	<u>NR_sigpending</u>	73
136 #define	<u>NR_sethostname</u>	74
137 #define	<u>NR_setrlimit</u>	75
138 #define	<u>NR_getrlimit</u>	76
139 #define	<u>NR_getusage</u>	77
140 #define	<u>NR_gettimeofday</u>	78

```

141 #define __NR_settimeofday 79
142 #define __NR_getgroups 80
143 #define __NR_setgroups 81
144 #define __NR_select 82
145 #define __NR_symlink 83
146 #define __NR_lstat 84
147 #define __NR_readlink 85
148 #define __NR_uselib 86

// 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、結合された
// をNR_に置き換えて、上記のシステムコールシンボル定数を形成し、これをアドレスとして
// システムコールテーブルの関数ポインタ。マクロ定義では、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、一緒に接続されて新しいシンボルを形成します。例えば、『
NR_##name』。
// 以下の156行目では、パラメータの「名前」（例えば「フォーク」）を置き換えた後、最後に
// プログラムの中で出現するのは、シンボル「NR_fork」です。
// 戻り値: 戻り値が0以上の場合はその値を返し、そうでない場合は
// エラー番号 errno が設定され、-1 が返されます。
150 #define syscall0(type, name)
 151 type name(void)  。
152 { \
153     asm volatile("int $0x80") \
 // システム割り込み 0x80.
154     : "=a" ( res ) \
 // 戻り値はeax( res )に入れられます。
155     : "%0" ( NR_##name ); \
 // 入力はシステム割込み番号NR_nameです。
156     if ( res >= 0 ) \
 // value >=0 の場合は、値を直接返します。
157     return (type) res; eldest \
158     return (type) res; eldest \
159     errno = - res; \
 // それ以外の場合は、エラー番号を設定し、-1
160     return -1; \
 // を返します。
161 }
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)
// %0 - eax( res ), %1 - eax( NR_name ), %2 - ebx(a).
163 #define syscall1(type, name, atype, a)
 164 type name(atype a)  164
165 { \
166     long res; eldest
167     asm volatile(("int $0x80") \
 // 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, btype b)
168     : "%0" ( NR_##name ), "b" ((long)(a)); \
 // %0 - eax( res ), %1 - eax( NR_name ), %2 - ebx(a), %3 - ecx(b) です。
169     if ( res >= 0 ) \
170     return (type) res; \
171     errno = - res; \
172     return -1; \
173 }
174
// 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) です。
176 #define syscall2(type, name, atype, a, btype, b)
 177 type name(atype a, btype b)
178 { \

```

```

179 long res; eldest
180   asm volatile ("int $0x80")¥
181       : "=a" ( res )
182       : "0" ( NR_##name ), "b" ((long)(a)), "c" ((long)(b)) ;
183 if ( res>=0 )  

184 return (type) res; ① 185
185 errno = - res; ② 。
186 return -1; ③④⑤⑥
187 }
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) です。
// 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
// データをバッファに入れ、バッファポインタをパラメータとしてカーネルに渡します。
189 #define _syscall3(type, name, atype, a, btype, b, ctype, c) ↳
190 type name(atype a, btype b, ctype c) ↳ 190 type
name(atype a, btype b, ctype c) ↳ 195
191 { ¥
192 long res; eldestres )
193   asm volatile NR##name##"0x80" ((long)(a)), "c" ((long)(b)), "d" ((long)(c)) ;
194 if ( res>=0 )  

195 return (type) res; ⑨ errno=-
res; ⑨ Erno=- res
196 return -1; 。
197 }
198
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。
// 詳細は include/linux/sys.h を参照してください。
206 int access(const char * filename, mode_t mode);
207 int acct(const char * filename);
208 int alarm(int sec);
209 int brk(void * end_data_segment);
210 void * sbrk(ptrdiff_t increment);
211 int chdir(const char * filename);
212 int chmod(const char * filename, mode_t mode);
213 int chown(const char * filename, uid_t owner, gid_t group);
214 int chroot(const char * filename);
215 int close(int fildes);
216 int creat(const char * filename, mode_t mode);
217 int dup(int fildes);
218 int execve(const char * filename, char ** argv, char ** envp);
219 int execv(const char * pathname, char ** argv);
220 int execvp(const char * file, char ** argv);
221 int execl(const char * pathname, char * arg0, ...);
222 int execlp(const char * file, char * arg0, ...); 223
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

```

```

// このキーワードは、特定の（初期化されていない）変数に対する誤った警告の発生を回避します。
この
// は、gccの関数属性の記述と同じです。
// void do_exit(int error_code) attribute ((noreturn)); 224
volatile void exit(int status);
225 volatile void exit(int status); 226
int fcntl(int fildes, int cmd, ...); 227
int fork(void);
228 int getpid(void);
229 int getuid(void);
230 int geteuid(void);
231 int getgid(void);
232 int getegid(void);
233 int ioctl(int fildes, int cmd, ...);
234 int kill(pid\_t pid, int signal);
235 int link(const char * filename1, const char * filename2);
236 int lseek(int fildes, off\_t offset, int origin);
237 int mknod(const char * filename, mode\_t mode, dev\_t dev);
238 int mount(const char * specialfile, const char * dir, int rwflag);
239 int nice(int val);
240 int open(const char * filename, int flag, ...);
241 int pause(void);
242 int pipe(int * fildes);
243 int read(int fildes, char * buf, off\_t count);
244 int setpgid(void);
245 int setpgid(pid\_t pid, pid\_t pgid);
246 int setuid(uid\_t uid);
247 int setgid(gid\_t gid);
248 void (*signal(int sig, void (*fn)(int))(int));
249 int stat(const char * filename, struct stat * stat_buf);
250 int fstat(int fildes, struct stat * stat_buf);
251 int stime(time\_t * tptr);
252 int sync(void);
253 time\_t time(time\_t * tloc);
254 time\_t times(struct tms * tbuf);
255 int ulimit(int cmd, long limit);
256 mode\_t umask(mode\_t mask);
257 int umount(const char * specialfile);
258 int uname(struct utsname * name); 259
int unlink(const char * filename);
260 int ustat(dev\_t dev, struct ustat * ubuf);
261 int utime(const char * filename, struct utimbuf * times);
262 pid\_t waitpid(pid\_t pid, int * wait_stat, int options);
263 pid\_t wait(int * wait_stat);
264 int write(int fildes, const char * buf, off\_t count);
265 int dup2(int oldfd, int newfd);
266 int getppid(void);
267 pid\_t getpgroup(void);
268 pid\_t setsid(void);
269 int sethostname(char *name, int len);
270 int setrlimit(int resource, struct rlimit *rlp);
271 int getrlimit(int resource, struct rlimit *rlp);
272 int getusage(int who, struct rusage *rusage);
273 int  gettimeofday(struct timeval *tv, struct timezone *tz);

```

```

274 int settimeofday(struct timeval *tv, struct timezone *tz);
275 int getgroups(int gidsetlen, gid_t *gidset);
276 int setgroups(int gidsetlen, gid_t *gidset);
277 int select(int width, fd_set * readfds, fd_set *
278 writefdsfd_set * exceptfds, struct timeval * timeout)
279 .
280 #endif
281

```

14. utime.h

1. 機能性

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。

2. コードアナリシス

プログラム 14-13 linux/include/utime.h

```

1 ifndef UTIME_H
2 define UTIME_H
3
4 // <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
5 メータの
6 // の構造が定義されています。
7 #include <sys/types.h> /* 分かってますよ、こんなことしちゃいけない
8 って。*/
9 struct utimbuf {  
10     time_t actime;           // ファイルのアクセス時間。1970.1.1:0:0:0からの秒数
11     time_t modtime;         .
12 };  
13 // 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。
14 extern int utime(const char *filename, struct utimbuf *times);

```

14.15 include/asm/ ディレクトリのファイル

リスト 14-2 linux/include/asm/ 目录下的

ファイル名	サイズ	最終更新時刻 (GMT)	説明
io.h	477バイト	1991-08-07 10:17:51	
メモリ.h	507バイト	1991-06-15 20:54:44	
セグメント.h	1366バイト	1991-11-25 18:48:24	
system.h	1707バイト	1992-01-13 13:02:10	

16. io.h

1. 機能性

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。

2. コードアナリシス

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。
// パラメータ: value - 出力のバイト、port - ポート。 ①
#define outb(value, port)
② asm ("outb %%al, %%dx": "a" (値), "d" (ポート)) ③
④
//// ハードウェアポートのバイト入力機能。
// 入力バイトを返します。
⑤ #define inb(port) ({ __u8 _v;
})
⑥ unsigned char _v; No.
⑦ asm volatile ("inb %%dx, %%al": "=a" (_v): "d" (port));
⑧
⑨ })
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。
// パラメータ: value - エクスポートされるバイト、port - ポート。
⑪ #define outb_p(value, port) \
⑫ asm ("outb %%al, %%dx\n\t" ⑬ // ラベル1(次の文)にジャンプします。
⑭ "1: jmp 1f\n\t" ⑮ // ラベル1に向かってジャンプします。
⑯ "1:":: "a"(値), "d"(ポート))
⑰

```

```

//// 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。
// 入力バイトを返します。
17 #define inb_p(port) ({ 18
unsigned char _v; ^w^ )
19 asm volatile("inb %%dx, %%al\n\t" // ラベル1(次の文)にジャンプします。
21     "1:jmp 1f\n\t" // ラベル1に向かってジャンプします。
22     "1:";"=a" (_v): "d" (port));
23 _v; )
24 })
25

```

17. メモリ.h

1. 機能性

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。

2. コードアナリシジョン

プログラム 14-15 linux/include/asm/memory.h

```

1 /*
2 * 注意！ ! memcpy(dest, src, n) は ds=es= 通常のデータセグメントを想定しています。これは
3 * すべてのカーネル関数に適用されます (ds=es= カーネル空間、 fs= ローカルデータ ) 。
4 * gs=null と、すべての行儀の良いユーザープログラム (ds=es=
5 * ユーザーデータスペース ) 。これはバグではありません。なぜなら、ユーザープログラムが
6 * es は気をつけないと死んでしまいます。 7
*/
7
8 ///////////////////////////////////////////////////////////////////
9 // 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。
10 // %0 - edi (address dest), %1 - esi (address src), %2 - ecx (number of bytes n).
11 #define memcpy(dest, src, n) ({ ^w^ )
12 void *_res = dest;
13 Asm ("cld;rep;movsb") DS:[ESI++] から ES:[EDI++] へ、 ECX(n) バイト分コピーします。
14     : "D" ((long)_res), "S" ((long)(src)), "c"
15     ((long)(n));
16 _res; di/「si/「cx/」)。
17 eldest
18 })
19

```

18. セグメント.h

1. 機能性

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。

2. コードアノテーション

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)。
// メモリFS:[addr]のバイトを返します。
1 extern inline unsigned char get_fs_byte(const char * addr)
2 {
3     unsigned register char _v; // 効率的なアクセスのためのレジスタ変数です。
4
5     asm ("movb %%fs:%1,%0": "=r" (_v): "m" (*addr));
6     return _v;
7 }
8
9 ///////////////////////////////////////////////////////////////////
10 // 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。
11 // %0 - (返されたワード _v); %1 - (メモリアドレス addr)。
12 // メモリFS:[addr]のワードを返します。
13 extern inline unsigned short get_fs_word(const unsigned short *addr)
14 {
15     符号なしの短編_V;
16
17     asm ("movw %%fs:%1,%0": "=r" (_v): "m" (*addr));
18     return _v;
19 }
20
21 ///////////////////////////////////////////////////////////////////
22 // 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。
23 // %0 - (返されたロングワード _v); %1 - (メモリアドレス addr)。
24 // メモリFS:[addr]のロングワードを返します。
25 extern inline unsigned long get_fs_long(const unsigned long *addr)
26 {
27     符号付きロング_V;
28
29     asm ("movl %%fs:%1,%0": "=r" (_v): "m" (*addr));
30 }

```

```

22         return _v;
23 }
24
25         //// 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。
26         // パラメータ: val - バイト値、addr - メモリアドレス。
27         // %0 - レジスタ (バイト値 val); %1 - (メモリアドレス addr).
28
29 extern inline void put_fs_byte(char val, char *addr)
30 {
31     asm ("movb %0,%fs:%1:: "r" (val), "m" (*addr));
32
33         //// 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。
34         // パラメータ: val - ワード値、addr - メモリアドレス。
35         // %0 - レジスタ (ワード値 val); %1 - (メモリアドレス addr).
36
37 extern inline void put_fs_word(short val, short * addr)
38 {
39     asm ("movw %0,%fs:%1:: "r" (val), "m" (*addr));
40
41         //// 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。
42         // パラメータ: val - ロングワードの値、addr - メモリアドレス。
43         // %0 - レジスタ (ロングワード値 val); %1 - (メモリアドレス addr).
44
45 extern inline void put_fs_long(unsigned long val, unsigned long * addr)
46 {
47     //// FSセグメントレジスタの値（セレクタ）を取
48     // 得します。 47 extern インライン unsigned long
49     get_fs()
50     {
51         符号なしの短編_V;
52         asm ("mov %%fs, %%ax": "=a" (_v):);
53         return _v;
54
55         //// DSセグメントレジスタの値を取得
56         // します。 54 extern inline unsigned
57         long get_ds() 55 {
58             符号なしの短編_V;
59             asm ("mov %%ds, %%ax": "=a" (_v):);
60             return _v;
61
62         //// FSセグメントレジスタの設定
63         // 61 extern inline void set_fs(unsigned long val)
64         {

```

```

63     asm ("mov %0, %%fs": "a" ((unsigned short) val))
64 }
65 .
66

```

19. system.h

1. 機能性

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ものです。

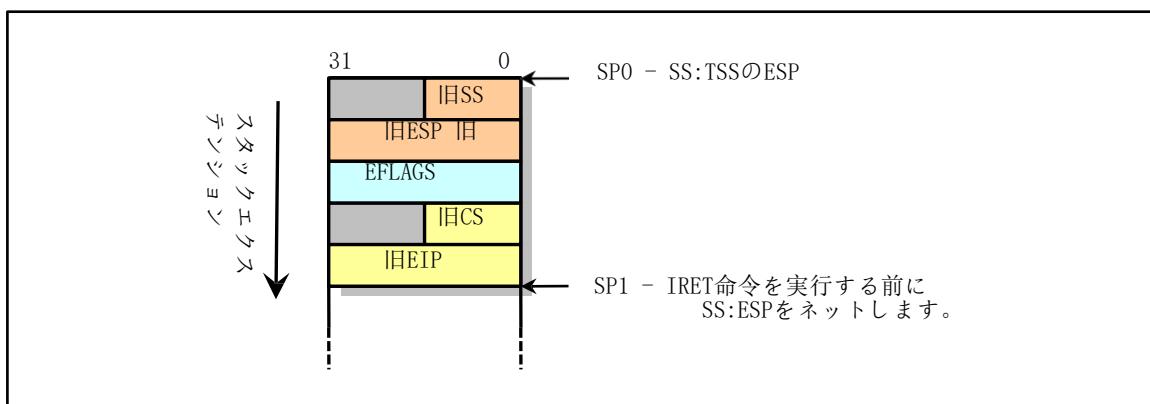


図14-3 IRET命令による特権レベル変更の実行

特権レベルの変更や制御の転送にこの方法が使われる原因是、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、コンテンツが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時に、スタックを使用します。そして、IRET命令が実行されると、システムはタスク0の実行に移ることになります。IRET命令が実行されると、スタックの内容は図14-3のようになります。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がタスク切り替え動作を行うことはありません。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カーネルのコードとデータに直接マッピングされる特別なプロセスです。

物理アドレス0から始まる640Kのメモリ空間であり、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、図中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。

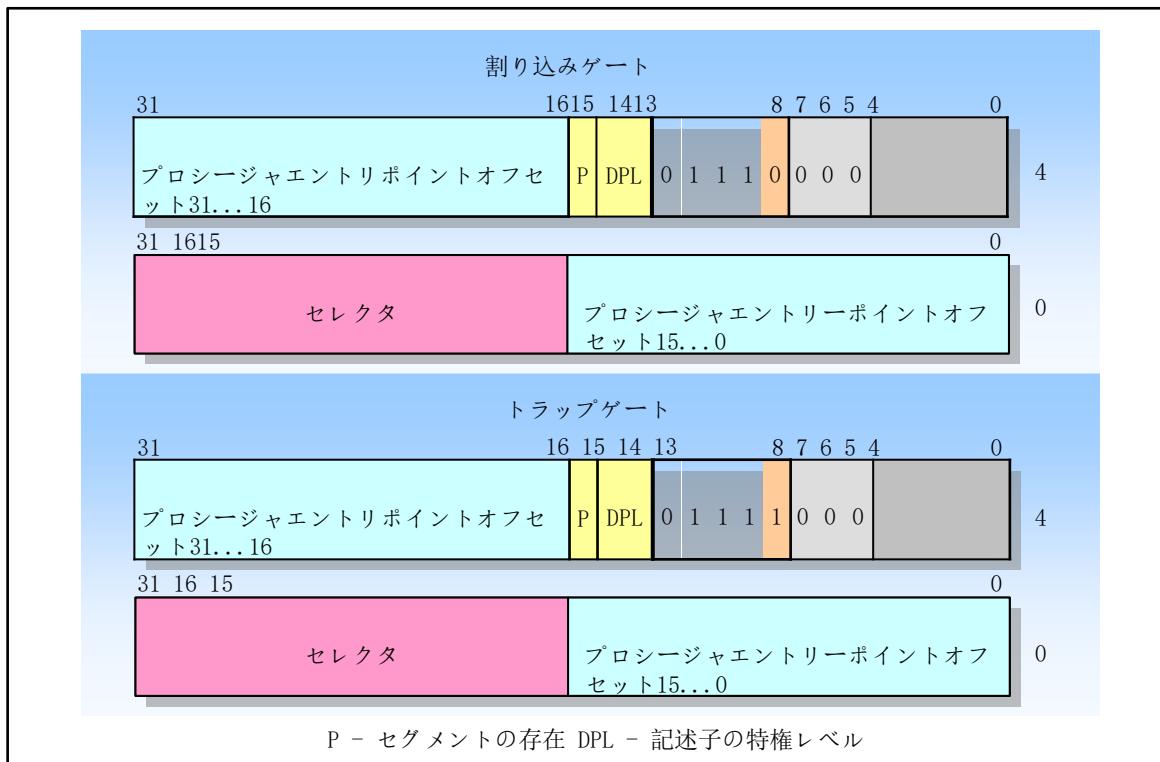


図14-4 テーブルIDTにおけるインタラプトゲートおよびトラップゲート記述子のフォーマット

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の種類を示し、図14-4のディスクリプタフォーマットの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では、一般的な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します。の意味は以下の通りです。

これらのマクロのパラメータは、上記と同様です。

14.19.2 コードアノテーション

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 #define move_to_user_mode() „^w^“ )

2 _asm ("movl %%esp, %%eax\t" ... // スタックポインタESPをEAXレジスタに保存する。

3     "pushl $0x17n\t" \$      // 最初にユーザーをプッシュ スタックセグメントSS。
4     "pushl %%eax%n\t" \$      // その後、スタックを押す ポインタESPをスタックに格納
5     "/pushflowered"          // そして、EFLAGSを押します も登録します。
6     "pushl $0x0f n\t" \$      // そして、タスク0のコードセグメント上CSをpushします。
7     "tmovl $0x17, %%eax,n\t" // そして、カーネルはタスク0の実行を開始します。
8     "pushl $1f\t"             // そして、ラベル1のオフセット(EIP)を押します。
9     "%retto n\t"             // %RETを実行します。各のボトルネックをローベルに初期化。
10    "%ax, %es\t"             // このローカルディスクリプターテーブルのセグメン
11    "、 「movw %%ax, %%fs\t"   // ト。
12    "」を意味します。
13
14    "... "ax")
15
16 #define sti() asm ("sti":)
17 #define cli() asm ("cli":)
18 #define nop() asm ("nop":)
19
20 #define iret() asm ("iret":) // 割り込み返し。
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、ゲートディスクリプターのタイプ type、特権
// パラメータのレベル情報dpl。(注)以下の「オフセット」は、カーネルに対する相対的な
// コードまたはデータセグメントになります。
// 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 - (記述子の上位4バイトのアドレス); %3 - EDX (プログラムオフセットアドレスの addr);
// %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)。
22 #define _set_gate(gate_addr, type, dpl, addr) [●]。
// オフセットアドレスの下位ワードとセレクタは、ディスクリプタ下位4バイト (EAX) にまとめられ
// ます。
// タイプフラグワードとオフセットハイワードを組み合わせて、ディスクリプタハイ4バイト (EDX)
24 ) にします_movw %0, %dx%n\t。
25 // 最後に_movl %%eax, %1\t。ゲートの下位4バイトと上位4バイトを別々に設定します。
26 Asm ("movl %0, %1\t")なります。
27     :\$
28     :"i" ((short) (0x8000+(dpl<<13)+(type<<8))) \$ // %0
29     "o" (*((char *) (gate_addr))), 。 // %1
30     "o" (*4+(char *) (gate_addr)), 。 // %2
31     "d" ((char *) (addr)), "a" (0x00080000)) // %3, %4
32

```



```

55      "rorl $16, %%eax\n\t"      // ベースのハイワードをAXに回転させる（ローワードからハ
56      "movb %%al, %3n\t"        イワード）。
57      "movb $" type ",%4n\t" ..^ w\// // flagタイプのドロードの中を1byteを4byteに移動。
58      \ movb $0x00, %5n\t" \     // ディスクリプタの6バイト目が0になる。
59      \ "movb %%ah, %6n\t" \     // ベースハイワードのハイバイトを7バイト目に移動。
60      \ "rorl $16, %%eax" \     // 16ビット右にループし、EAXは元に戻す。
61      \: "a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)),
62      "m" (n+4), "m" (n+4)
63      ) \ m (*(n+5)), m (*(n+6)), m (*(n+7))\

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。
// n - ディスクリプタへのポインタ; addr - ディスクリプタ内のセグメントのベースアドレス
// のエントリーです。TSS記述子のタイプは0x89である。
65 #define set_tss_desc(n,addr) set_tssldt_desc(((char *) (n)),addr, "0x89")
/// グローバルテーブルにローカルテーブル(LDT)の記述子を設定します。
// n - ディスクリプタへのポインタ; addr - ディスクリプタ内のセグメントのベースアドレス
// のエントリーです。ローカルテーブルセグメント記述子のタイプは0x82です。
66#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

リスト14-3 linux/include/linux/ディレクトリ内のファイル

		説明
	config.h	名前 最終更新時間 (GMT) 00:13:18
	fdreg.h	2466 バイト 1991-11-02 10:48:44
	fs.h	5754 バイト 1992-01-12 07:00:20
	hdreg.h	1968 バイト 1991-10-13 15:32:15
	head.h	304 バイト 1991-06-19 19:24:13
	kernel.h	1036 バイト 1992-01-12 02:17:34
	math_emu.h	4924 バイト 1992-01-01 17:33:04
	mm.h	1101 バイト 1992-01-13 15:46:41
	sched.h	7351 バイト 1992-01-13 22:24:42
	sys.h	3402 バイト 1992-01-13 21:42:37
	tty.h	2801 バイト 1992-01-08 22:51:56

21. config.h

1. 機能性

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。

2. コードアノテーション

プログラム 14-18 linux/include/linux/config.h

```
1 #ifndef _CONFIG_H
2 #define _CONFIG_H
3
4 /*
5 * uname() が何を返すべきかを定義する 6 */
6
7 #define UTS_NODENAME "(none)" /* sethostname() で設定され
8 #define UTS_SYSNAME "Linux"   ます。
9 #define UTS_RELEASE " "
10#define UTS_VERSION "0.12"    パッチレベル */ /* パッチ
11#define UTS_MACHINE "i386"   ベースードウェアタイ
12                                プ
13/* 本当に分かっている人以外は、これらに触れないでください。 */
```

```

// 以下のシンボリック定数は、システム起動時にメモリの位置を示すために使用されます。
// とカーネルのロード、および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指定します
14 #define DEF_INITSEG    0x9000      // ブートセクタを移動させるセグメントです。
15 #define DEF_SYSSEG     0x1000      // システムモジュールがロードされたセグメントです。
16 #define DEF_SETUPSEG   0x9020      // セットアッププログラムが配置されているセグメントで
17 #define DEF_SYSSIZE    0x3000      す。
18                                         // システムモジュールの最大サイズ（16個単位）です。
19 /*
20 * root-deviceがハードコードされなくなりました。デフォルトの
21 * boot/bootsect.s の ROOT_DEV = XXX という行を変更することで、ルートデ
22 * バイスを変更します。
23 */
24 /*
25 * キーボードは kernel/chr_dev/keyboard.S で定義されています
26 *
27 */
28 /*
29 * 通常、Linuxでは、BIOSからドライブのパラメータを
30 * 起動しても、何か不可解な理由で失敗した場合は
31 * が取り残されてしまいます。このような場合には、HD_TYPEを定義すること
32 * で、以下のようになります。
33 * ハードディスクに必要な情報がすべて含まれています。 33 *
34 * HD_TYPEマクロは以下のようにになります。
35 */
36 * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
37 */
38 * 2台のハードディスクを使用する場合は、情報を
39 * カンマ。
40 */
41 * #define HD_TYPE { h, s, c, wpcom, lz, ctl }, { h, s, c, wpcom, lz, ctl }.
42 */
43 /*
44 这は一例で、2台のドライブ、1台目はタイプ2、2台目はタイプ3です:
45
46 #define HD_TYPE { 4, 17, 615, 300, 615, 8 }, { 6, 17, 615, 300, 615, 0 }.
47
48 注: ctlは、ヘッド<=8のすべてのドライブでは0、ドライブではctl=8です。
49 8つ以上のヘッドを持つ 50
50 BIOSにドライブの種類を知らせたい場合は
51 HD_TYPEを未定義のままにします。これが通常のやり方です。
52 */
53
54
55 #endif
56

```

22. fdreg.h

1. 機能性

fdreg.hというヘッダーファイルを使って、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ポート	レジスタ名	機能
0x3f2	Read/Write	レジスタ出力 (制御) レジス
0x3f4	Read only	タ FDCメインステータスレジ
0x3f5	Read/Write	ジスタ
0x3f7	Read/Write	FDCデータレジス
0x3f7	Read only	タ デジタル入力レジス
	Write only	ジスタ

デジタル出力ポート（デジタルコントロールポート）はデジタル出力ポートで、駆動モータオン、駆動選択、FDCのスタート／リセット、DMA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

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。

2. コードアナリシジョン

プログラム14-19 linux/include/linux/fdreg.h

```

1 /*
2 * このファイルには、フロッピーディスクコントローラの定義が含まれています。
3 * 様々なソース。ほとんどが「IBMマイクロコンピュータ。A Programmers
4 * Handbook」, Sanches and Canton.
5 #ifndef FDREG_H // この定義は、コード内の重複するヘッダーファイルを除外するために使用
6 #define FDREG_H されます。
7
8 // 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です
9 extern int ticks_to_floppy_on(unsigned int nr);
10 extern void floppy_on(unsigned int nr);
11 extern void floppy_off(unsigned int nr);
12 extern void floppy_select(unsigned int nr);
13 extern void floppy_deselect(unsigned int nr);
14
15 // 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
16 #ifndef FD_STATUS // FD_STATUSのレグSGG3f約340ページ/*メインのステータスレジスタポート。
17 #define FD_DATA 0x3f5 // データポートです。
18 #define fd_dor 0x3f2 /* デジタル出力レジスタ */ デジタ
19 #define fd_dir 0x3f7 /* ル入力レジスタ(読み出し) */
20 #define fd_dcr 0x3f7 /* ディスクケット・コントロール・レジスタ(書き込
み) */
21
22 /* ビット 主要なステータ レジスタ */
23 #define status_bmask 0x0F /* ドライブビジーマスク */ // (ドライバ毎に1ビ
24 #define STATUS_BUSY 0x10 ット)
25 #define status_dma 0x20 /* FDCビジー */
26 #define status_dir 0x40 /* 0-DMAモード
27 #define status_ready 0x80 /* */
28
29 /* ビット of FD_ST0 */
30 #define ST0_DS 0x03 /* ドライブ・セレクト・マスク */
31 #define ST0_HA 0x04 /* ヘッド(アドレス) */
32 #define ST0_nr 0x08 // 割り込みコードビット(割り込み理由)、00 - Not Ready // ノットリードが正常に終了、01 - コマンドが終了
33 #define ST0_ece 0x10 // 異常: 10 - コマンドが無効; 11 - FDDの準備状態が変化 // ミキエラー
34 #define ST0_INTR 0x20 // 割り込みコードのマスク // seek end or altrate end。
35
36
37 /* ビット of FD_ST1 */
38 #define ST1_MAM 0x01 /* 欠落したアドレスマーク */
39 #define ST1_WP 0x02 /* ライトプロテクト*1
40 #define ST1_ND 0x04 /* No Data - unreadable */ // セクターが見つかり
                               ません。

```

```

41 #define ST1_OR          0x10      /* OverRun */ // データ転送のタイムアウト
42 #define ST1_CRC          0x20      /* データまたはaddrのCRCエラー */
43 #define ST1_EOC          0x80      /* シリンダの終了 */

44
45 /* ビット of FD_ST2 */
46   ト ST2_MAM          0x01      /* アドレスマークの欠落(再) */
47 #define ST2_BC           0x02      /* バッドシリンダー */
48 #define ST2_SNS          0x04      /* Scan Not Satisfied */
49 #define ST2_SEH          0x08      /* スキャンイコールヒット */
50 #define ST2_WC            0x10      /* 誤ったシリンダー */
51 #define ST2_CRC          0x20      /* データフィールドのCRCエラー */
52 #define ST2_CM            0x40      /* コントロールマーク = 削除 */

53
54 /* ビット of FD_ST3 */
55   ト ST3_HA            0x04      /* ヘッド(アドレス) */
56 #define ST3_TZ            0x10      /* トラックゼロ信号(1=トラック0)
57   */ /* (注) */
58 #define FD_WP             0x40      /* テクノロジ移動 */
59 /* FD_COMMAND の値 */ /* (注) */
60 #define FD_RECALIBRATE    0x07      /* 再校正する。 */
61 #define FD_SEEK            0x0F      /* シーク・トラック */
62 #define FD_READ            0xE6      /* MT, MFM, SKipが削除された状態で読み込まれる。
63 #define FD_WRITE           0xC5      /* MT, MFMでの書き込み */ */
64 #define FD_SENSEI          0x08      /* ドライブのパラメータ(ステップレート、セクタソーフト・ステータス */
65 #define FD_SPECIFY         0x03      /* HUTなどを指定する */
66
67 /* DMAコマンド */
68 #define DMA_READ           0x46      /* DMAリードディスク(ポート12, 11へ)のモードワードです。 */
69 #define DMA_WRITE          0x4A      /* DMAライトディスクのモードワードです。 */
70
71#endif
72

```

23. fs.h

1. 機能性

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inode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。.

2. コードアノテーション

プログラム 14-20 linux/include/linux/fs.h

```

1 /*
2 * このファイルには、いくつかの重要なファイルテーブルの定義があります。
3 * 構造体など 4 */

```

```

5 #ifndef _FS_H
6 #define _FS_H
7
8
9 #include <sys/types.h>           // 型のヘッダーファイルです。基本的なシステムデータタイプが定
10                                義されています。
11 /* デバイスは以下の通りです。(minixと同じなので、minixの
12 * ファイルシステムです。これらはメジャーな
13 * 数字です) 13 */
14 * 0 - 未使用(nodev)
15 * 1 - /dev/mem                  // メモリデバイス。
16 * 2 - /dev/fd
17 * 3 - /dev/hd
18 * 4 - /dev/ttys                // tty シリアルターミナルデバ
                                イス
19 * 5 - /dev/tty
20 * 6 - /dev/lp                  // プリンタデバイス。
21 * 7 - 無名のパイプ
22 */
23 #define IS_SEEKABLE(x) ((x)≥1 && (x)≤3)    // デバイスがロケーションを見つけられるかどうかを判断します。
24
25
26 #define READ_0
27 #define WRITE_1
28 #define READA_2      /* read-ahead - don't pause */ (英語)
29 #define WRITEA_3     /* "write-ahead" - 馬鹿げているが、多少は役に立
                                つ */。
30
31 void buffer_init(long buffer_end);        // バッファキャッシュの初期化
32                                関数です。
33 #define MAJOR(a) (((unsigned)(a))>>8)    // デバイスのメジャー番号を
34 #define MINOR(a) ((a)&0xff)                 取得します。
35
36 #define NAME_LEN 14                      // デバイスのマイナー番号を
37 #define ROOT_INO 1                        取得する前の長さは14です
38
39 #define I_MAP_SLOTS 8                    .
40 #define Z_MAP_SLOTS 8                    // ルートのi-node番号
41 #define SUPER_MAGIC 0x137F               // i-nodeのビットマップスロット(ブロック)の
42                                数です。
43 #define NR_OPEN_20                     // 論理ブロックのビットマップスロットの数です
44 #define NR_INODE 32                   .
45 #define NR_FILE 64                   // // ブロックが開いてあるアドレス数。一です。
46 #define NR_SUPER 8                   // // システムが使用するi-nodeの最大数です。
47 #define NR_HASH_307                  // // システム内のファイルの最大数です。
48#define NR_BUFFERS nr_buffers          // // システム内のハッシュテーブルの配列アイテムをバッファリング
49 #define BLOCK_SIZE 1024              // // アイテムをします。
50 #define BLOCK_SIZE_BITS 10           // // システム内のデータブロックのサイズ(バイト)。
51 #ifndef NULL                         // // ブロックサイズで使用されるビット数です。
52 #define NULL ((void *) 0)
53 #endif
54
55 // 各ブロックが格納できるi-nodeの数(1024/32 = 32)。 55#define
INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct d_inode)))
56 // 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。

```

```

56 #define DIR_ENTRIES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct dir_entry)))
57
// 58 #define PIPE_READ_WAIT(inode)
((inode.i_wait)) 58 #define PIPE_READ_WAIT(inode)
((inode).i_wait) 59 #define PIPE_WRITE_WAIT(inode)
((inode).i_wait2) 60 #define PIPE_HEAD(inode)
((inode).i_zone[0])
61 #define PIPE_TAIL(inode) ((inode).i_zone[1])
62#define PIPE_SIZE(inode) ((PIPE_HEAD(inode)-PIPE_TAIL(inode))&(PAGE_SIZE-1))
63#define PIPE_EMPTY(inode) (PIPE_HEAD(inode)==PIPE_TAIL(inode))
64#define PIPE_FULL(inode) (PIPE_SIZE(inode)==(PAGE_SIZE-1))
65#define NIL_FILP ((struct file *)0) // nullファイル構造体のポインタ
66#define SEL_IN 1
67#define SEL_OUT 2
68#define SEL_EX 4
69
70
71 typedef char buffer_block[BLOCK_SIZE]; // バッファブロックの配列（1024個）。 72
// バッファブロックヘッダのデータ構造。（とても重要!!)
// bhはコード上、buffer_headの略語を表すことが多いです。
72 struct buffer_head {
73
74     char * b_data;           /* データブロック(1024バイト)へのポインター */
75    符号付きロング b_blocknr、/* ブロック番号 */
76    符号付きショート b_dev、/* デバイス(0 = フリー) */
77    符号付きチャード b_uptodate、/* 0-clean, 1-dirty */ // 修正フラグ。
78    符号付きチャード b_dirt、/* このブロックを使うユーザー */
79    符号付きチャード b_count、/* 0 - OK, 1 - ロックされています */。
80    符号付きチャード b_lock。/* タスクの待ち行列。 */
81     struct task_struct * b_wait; // ハッシュキューの前のブロック。
82     struct buffer_head * b_prev; // ブロックリストの次のブロックを表示
83     struct buffer_head * b_next; // します。
84     struct buffer_head * b_prev_free; // クリシタ東キの前の次回ブロックを表示
85     struct buffer_head * b_next_free; // します。
86 };
87
// ディスク上のI-nodeデータ構造（32バイト）
88 struct d_符号なし short i_mode; // ファイルタイプと属性(rwxビット)があります。
89
90     符号なし 短い i_uid。 // ユーザーid(ファイル所有者の識別子)
91     符号化されていない長さの // 誤正された時間(1970年1月1日0から、単位は秒)。
92     符号化されていない // ドループID(ファイルオーナーが所属するグループ)。
93     i_time、符号化され // リンク(i-nodeに向けられたエントリー)の数。
94     ない長さの i_gid、 // 論理ブロック番号の配列。
95     符号化されていない // ダイレクト(0-6)、インダイレクト(7)、セカンダリーア
96     長さの i_nlinks、 // サイド(8)のいずれかです。
97     符号化されていない // 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。
98 struct m_inode {
99     符号化されていない短い // 符号化されていない短い
100    i_mode、符号化されてい // i_mode、符号化されてい
101    ない短い i_uid、符号化 // ない短い i_size
102    されていない長い i_size // 、符号化されていない長
103    い i_mtime、符号化され // い i_size
104    ていないchar i_gid。 // 、符号化されていない長
105

```

```

104      符号化されていない char
105          i_nlinks; 符号化されてい
106 /* これ は フィールドの生にあらわします */
107     struct task_struct * i_wait; // i-nodeを待つためのタスク待ち行列。
108     struct task_struct * i_wait2; /* パイプのために */。
109     符号なし ロン i_atimeです // i-nodeのアクセスタイム。
110         グ。
111     符号なし ロン i_ctimeです // i-nodeの変更時間。
112         グ。
113     符号なし ショ i_devです。 // i-nodeが配置されているデバイスです。
114         ート
115     符号なし ショ i_num; // i-nodeの番号。
116         ート
117     符号なし ショ i_countです // i-node used count、0はアイドル（フリー）であるこ
118         ート。
119     符号なし チャ i_lockです。 // ロックフラグです。
120         ー
121     符号なし チャ i_dirt; // 修正フラグ
122         ー
123 // ファイル構造(チャイムペイでされるとi-nodeとの関係を確立せし用めに使われる) struct
124 file {
125     符号なし チャ i_fmodeです。 // mount_flag (RWビット)。
126         ー
127     符号なし チャ i_fsegflagsです // フォライ旗を開設セグメントに使用されます。
128         ー
129     符号なし チャ i_fupdateです // フォルマ参照数更新しました。
130         ー
131 } ; struct m_inode * f_inode; // ファイルのi-nodeになります。
132 off_t f_pos; // ファイル内の読み書きの位置。
133 // インメモリーディスクのスーパーブロック
134 // 構造。
135 struct superblock {
136     s_ninodes; // ファイルシステム内のi-nodeの数。
137     unsigned short s_nzones; // ゾーンの数 (ロジカルブロック)。
138     unsigned short s_imap_blocks; // i-nodeマップが占有するデータブロックの数。
139     unsigned short s_zmap_blocks; // 論理ブロックマップで占有されるブロック数。
140     unsigned short s_firstdatazone; // データゾーンの最初のブロック番号。 unsigned
141     short s_log_zone_size; // Log2(データブロック数/論理ブロック)。
142     符号なし long s_max_size; // 最大ファイルサイズ。
143     符号なし 短い s_magic。 // ファイルシステムのマジックナンバーです。
144 /* これらはメモリ上にのみ存在します */
145     struct buffer_head * s_imap[8]; // i-nodeのビットマップバッファブロックの配列です。
146     struct buffer_head * s_zmap[8]; // 論理ブロックのビットマップバッファブロックの配列
147         です。
148     符号なしの短い s_dev; // スーパーブロックのデバイス番号です。
149     struct m_inode * s_isup; // マウントされたファイルシステムのルートi-node。
150     struct m_inode * s_imount; // ファイルシステムがインストールされるi-nodeです。
151     unsigned long s_time; // 修正された時間です。
152     struct task_struct * s_wait; // それを待っているプロセスの待ち行列。
153     符号なしのchar s_lock; // ロックフラグです。
154     unsigned char s_rd_only; // 読み取り専用フラグ。
155     符号なしのchar s_dirt; // サーティフラッグ
156 }
157 // 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。
158 struct d_super_block {
159     unsigned short s_ninodes;

```

```

154     unsigned short s_nzones,
155     unsigned short s_imap_blocks,
156     unsigned short s_zmap_blocks,
157     unsigned short s_firstdatazone,
158     unsigned short s_log_zone_size,
159     unsigned long s_max_size,
160     unsigned short s_magic;
161 };
162
163 // ファイルディレクトリのエントリ構造(16バイト)。
164 struct unsigned short inode;           // i-nodeの番号。
165     char name[NAME_LEN];             // ファイル名, NAME_LEN =
166 };                                     14.
167
168 extern struct m_inode_inode_table[NR_INODE]; // i-nodeテーブル(32エントリ
169 extern struct file_file_table[NR_FILE];    // )。
170 extern struct super_block_super_block[NR_SUPER]; // ブートローダの配列項目8項
171 extern struct buffer_head * start_buffer; // バッファキャッシュの開始位置。
172 extern int nr_buffers;                // バッファの数です。
173
174 ///////////////////////////////////////////////////////////////////
175 // ディスク操作機能の試作品を紹介します。
176 // 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。
177 extern void check_disk_change(int dev);
178 // 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。
179 extern int floppy_change(unsigned int nr);
180 // ドライブを開始するまでの待ち時間を設定します(待ち時間の設定)。
181 extern int ticks_to_floppy_on(unsigned int dev);
182 // 指定したドライブを起動する。
183 extern void floppy_on(unsigned int dev);
184 // 指定されたフロッピードライブの電源を切る。
185 extern void floppy_off(unsigned int dev);

186 ///////////////////////////////////////////////////////////////////
187 // 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。
188 // 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。
189 extern void truncate(struct m_inode * inode);
190 // i-nodeの情報を更新(同期)します。
191 extern void sync_inodes(void);
192 // 指定されたi-nodeを待ちます。
193 extern void wait_on(struct m_inode * inode);
194 // 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。
195
196 extern int bmap(struct m_inode * inode, int block);
197 // ブロック'block'に対応するデバイス上の論理ブロックを作成し、その論理ブロックを返します。
198 // デバイスのブロック番号。
199 extern int create_block(struct m_inode * inode, int block);
200 // 指定されたパス名のi-node番号を取得します。 184
201 extern struct m_inode * namei(const char * pathname);
202 // 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。
203 extern struct m_inode * lnamei(const char * pathname);
204 // struct m_inode * lnamei(const char * pathname)を開く準備をします。
205 extern node_t open_namei(const char * pathname, flag, int mode),
206 // node_tをopen_namei( pathname, flag, mode )でpathameを書き込む。
207

```

```

188 extern void input(struct m_inode * inode);
    // デバイスからi-nodeを読み込みます。
189 extern struct m_inode * iget(int dev, int nr);
    // inodeテーブルからidle i-nodeエントリを取得します。
190 extern struct m_inode * get_empty_inode(void);
    // 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)。
191 extern struct m_inode * get_pipe_inode(void);
    // 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。
192 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);
    // 低レベルの読み書き可能なデータページ、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);
    // 指定されたバッファブロックを解放します。
196 extern void brelse(struct buffer_head * buf);
    // 指定されたデータブロックを読み込む。
197 extern struct buffer_head * bread(int dev, int block);
    // 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。
198 extern void bread_page(unsigned long addr, int dev, int b[4]);
    // 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。
199 extern struct buffer_head * breada(int dev, int block,...);
    // 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 200
extern int new_block(int dev);
    // デバイステーラ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。
201 extern void free_block(int dev, int block);
    // 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。
202 extern struct m_inode * new_inode(int dev);
    // i-nodeを解放(フリー)する(ファイルを削除する場合)。
203 extern void free_inode(struct m_inode * inode);
    // 指定されたデバイスバッファを更新します。
204 extern int sync_dev(int dev);
    // 指定されたデバイスのスーパーべロックを取得します。
205 extern struct super_block * get_super(int dev);
206 extern int ROOT_DEV; // ルートデバイス番号。 207
    // ルートファイルシステムをマウントします。 208
extern void mount_root(void); 209
210#endif
211

```

24. hdreg.h

1. 機能性

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ミング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、コントローラのポート、ハードディスク・ステータス・レジスタの各ビットの状態、コントローラのコマンド、および

エラー状態の定数記号を示しています。また、ハードディスクの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。

2. コードアナリシジョン

プログラム 14-21 linux/include/linux/hdreg.h

```

1 /*
2 * このファイルには、AT-hd-コントローラの定義が含まれています。
3 * 様々なソース。いくつかの定義をチェックしてみましょう（コメントを参照
4 * a ques）。
5 */
6 #ifndef HDREG_H
7 #define HDREG_H
8
9 #define HD_DATA ハードディスクデータレジスター // AT-COPYTL書き込み時 */
10#define HD_ERROR 0x1f1 // err-bitsを見る */
11#define HD_NSECTOR 0x1f2 // 読み書きされるセクタ数 /* nr of sectors */
12#define HD_SECTOR 0x1f3 // (Read/Write)
13#define HD_LCYL 0x1f4 // /* セクターを開始します。
14#define HD_HCYL 0x1f5 // 始動シリンドー */ /* 始動シリンドー
15#define HD_CURRENT 0x1f6 // スタートシリンドーのハイバイト */。
16#define HD_STATUS 0x1f7 // /* 101dhhh, d=ドライブ, hhh=ヘッド */。
17// /* ステータスピットを見る */
18#define HD_PRECOMP HD_ERROR // 同一の io アドレス、リード=エラー、ライト=プレコンプ */
19#define HD_COMMAND HD_STATUS // /* 同じドライブを read=status, write=cmd */。
20
21#define HD_CMD 0x3f6 // 一ト。
22
23/* HD_STATUSのビット数
24#define ERR_STAT 0x01 // コマンド実行エラー。
25#define INDEX_STAT 0x02 // インデックスを受け取りました。
26#define ECC_STAT 0x04 // /* Corrected error */ // ECCチェックサムエラー。
27#define DRQ_STAT 0x08 // リクエストサービス
28#define SEEK_STAT 0x10 // シーク終了
29#define WRERR_STAT 0x20 // シーク終了
30#define READY_STAT 0x40 // ドライブエラー
31#define BUSY_STAT 0x80 // ドライブレディ
32// // Controller busy.
33/* HD_COMMANDの値 */ (注)
34#define WIN_RESTORE 0x10 // ドライブのリセット（再校正）
35// // を行います。
36#define WIN_READ 0x20 // セクターを読む。
37#define WIN_WRITE 0x30 // セクターを書き込む。
38#define WIN_VERIFY 0x40 // セクターの検証
39#define WIN_FORMAT 0x50 // フォーマットトラック。
40#define WIN_INIT 0x60 // コントローラの初期化
41#define WIN_SEEK 0x70 // シークトラック
42#define WIN_DIAGNOSE 0x80 // コントローラの診断を行います
43// // 。
44// // 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、以下のようにになります。
ト // =====
// // 診断コマンド その他のコマンド
// // =====

```

```

// 0x01 エラーなし コン データマークが
// 0x02 トローラエラー 消えた トランク0エラー
// 0x03 セクターバッファエラ ー ECC部エラー 制御ブロセスエラー コマンドアボート
// 0x04 // 0x10 IDが見つかりません。 ECCエラーです。
// 0x40 // 0x80 バッドセクタ
// -----
45 #define MARK_ERR 0x01 /* 悪いアドレスマーク */
46 #define TRK0_ERR 0x02 /* トランク0が見つからない */
47 #define ABRT_ERR 0x04 /* couldn't find track 0 */
48 #define ID_ERR 0x10 /* */
49 #define ECC_ERR 0x40 /* */
50 #define BBD_ERR 0x80 /* */
51 /* */

// 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。
52 struct punsigned char boot_ind; /* 0x80 - アクティブ(未使用) */ (注)
53     unsigned char head; /* */
54     unsigned char sector; /* */
55     unsigned char cyl; /* */
56     unsigned char sys_ind; /* */
57     unsigned char end_head; /* */
58     unsigned char end_sector; /* */
59     unsigned char end_cyl; /* */
60     unsigned int start_sect; /* セクターを0から数え始めます */
61     unsigned int nr_sects /* パーティション内のセクタ数 */ /* nr
62     of sectors in partition */
63 };

64
65 #endif
66

```

3. インフォメーション

1. ハードディスクのパーティションテーブル

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つのエントリーで構成されており、各エントリーは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、シリンド番号、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、0x1BE--0x1FDの位置に格納されています。

ハードディスクの0シリンドー0ヘッドのセクターです。

オフセット	名義	データタイプ	説明・パーティション・テーブルのエントリ構造
0x00	boot_ind	1バイト	ブートインデックス。4つのパーティションのうち、一度に起動できるのは1つのパーティションのみです。 0x00 - このパーティションから起動しない、0x80 - このパーティションから起動する。
0x01	ヘッド	1バイト	パーティション開始時のヘッド番号です。ヘッドナンバーの範囲は0~255です。
0x02	セクター	1バイト	現在のシリンドー内のセクター番号（ビット0~5）と上位2ビット（ビット6-7）のシリンドー番号をパーティションの最初に表示します。

0x03	円筒	1バイト	パーティションの先頭にあるシリンドー番号の下位8ビット。
0x04	sys_ind	1バイト	パーティションタイプ。0x0b - DOS、0x80 - オールドミニックス、0x83 - Linux ...
0x05	エンド・ヘッド	1バイト	パーティションの最後にあるヘッド番号。0~255の範囲で設定できます。
0x06	end_sector	1バイト	現在のシリンドー内のセクター番号（ビット0~5）と上位2ビット（ビット パーティションの最後にシリンドー番号の6-7）を表示します。
0x07	end_cyl	1バイト	パーティション終了時のシリンドー番号の下位8ビット。
0x08-0x0b	スタート_セクト	4バイト	パーティションの先頭にある物理セクター番号。0から数えます。 を、ハードディスク全体のセクタ番号順に表示します。
0x0c-0x0f	nr_sects	4バイト	パーティションが占有しているセクタの数。

25. head.h

1. 機能性

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。

2. コードアノテーション

プログラム 14-22 linux/include/linux/head.h

```

1 #ifndef HEAD_H
2 #define HEAD_H
3
4 // セグメントディスクリプターのデータ構造は以下のように定義されています。この構造では、以下の
5 // ことだけが書かれています。
6 // 各ディスクリプターは8バイトで構成され、各ディスクリプターテーブルは256エントリであるこ
7 // とを示しています。 4typedef struct desc_struct {
8     unsigned long a, b; 6
9     desc_table[256];
10
11    // 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
12    // 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
13    // る。 9extern desc_table* __adt(long pg_dir)[1024]; // L1 プロトディスクリプターテーブル、グローバルディ
14
15    // GDTの0番目の項目で、使用されていません。
16    // GDT CODE 0 // 最初の項目は、カーネルコードセグメント記述子です。
17    // GDT DATA 2 // 2番目の項目は、カーネルデータセグメントの記述子です。
18    // GDT TMP 3 // 3番目の項目は、システムセグメントの記述子で、使用され
19    // ていません。
20    // LDTの0番目の項目で、使用されていません。
21    // LDT CODE 1 // 最初の項目は、ユーザーコードのセグメント記述子です。
22    // LDT DATA 2 // 2番目の項目は、ユーザーデータセグメントの記述子です
23
24 #endif
25

```

26. kernel.h

1. 機能性

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。

2. コードアナリティクス

プログラム 14-23 linux/include/linux/kernel.h

```

1 /*  

2 * 'kernel.h' には、よく使われる関数のプロトタイプなどが含まれ  

3 ています。  

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。  

5 (kernel/fork.c, 24). 4 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).  

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). 6  

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  

11 参照してください。 7 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。 8 int  

13 printk(const char * fmt, ...);  

14 // コンソールの表示機能です。(kernel/chr_drv/console.c, 995)を参照してく  

15 ださい。 9 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 。  

18 int tty_write(unsigned ch, char * buf, int count);  

19 // 汎用的なカーネルのメモリ割り当て関数です。(lib/malloc.c, 117).  

20 void * malloc(unsigned int size);  

21 // 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  

22 ください。  

23 void free_s(void * obj, int size);  

24 // 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。  

25 extern void hd_times_out(void);  

26 // ピープ音を止めます。(kernel/chr_drv/console.c, 944).  

27 extern void sysbeepstop(void);  

28 // ブラックスクリーン処理。(kernel/chr_drv/console.c, 981)となっています。  

29 extern void blank_screen(void);  

30 // 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。  

31 extern void unblankscreen(void); // ピープ音の時間刻みのカウント (chr_drv/console.c,  

32 950).  

33 extern int hd_timeout; // ハードディスクのタイムアウト秒数  

34 extern int blankinterval; (kernel/blk_drv/blk.h).  

35 extern int blankcount; 22  

36 #define free(x) free_s((x), 0) // 画面が黒くなる interval(chr_drv/console.c, 138).  

37 // 黒い画面の時間数 (chr_drv/console.c, 139).  

38  

39 25 /*  

40 * これはマクロとして定義されていますが、ある時点で、これはある種の  

41 * 真を返した場合にフラグを設定する本物のサブルーチン(以下を行うた  

42 ために)  

43 * プロセスがrootを使用している場合にフラグを立てるBSDスタイルのアカ  

44 ンティングです。  

45 * privs)となっています。この意味するところは、通常の

```

30 * パーミッションのチェックを最初に行い、suser()のチェックを最後に行います。 31 */
32 #define suser() (current->euid == 0) // スーパーユーザーであるかどうかをチェックします。 33

27. math_emu.h

1. 機能性

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するデータ構造の一部も含まれています。

2. コードアナリティクス

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、タスク構造、初期タスク0、およびいくつかの組み込みの
9 #include <linux/sched.h.p.m. // ディスクリプタのパラメータ設定と取得に関するアセンブリ関数
マクロの記述。 9 #include <linux/sched.h>.
10
// CPUが例外INT7（デバイスが存在しない）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 {
12     long math_ret;           // math_emulate()の呼び出し元のリターンアドレス（INT 7）で
                           // す。
13     long orig_eip;          // オリジナルのEIPを一時的に保存する場所です。
14     long edi;               // 例外ハンドラがプッシュしたレジスタです。
15     long esi;
16     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。
// 以下（18～30行目）は、システムコールが呼び出されたときのスタック内の構造と同じです。 long
17     sys_call_ret;
18     long    eax;
19     long    ebx;
20     long    ecx;
21     long    edx;
22     long    orig_eax;        // sys-callではなく他の割り込みの場合は-1です。
23     long    fs;
24     long    es;
25     long    ds;

```

```

26     long eip;           // 26行目～30行目は、CPUが自動的にプッシュします。
27     長いcs。
28     long eflags;
29     long esp;
30     long ss
31 } ;
32

// 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。
33 #define EAX (info-> eax)
34 #define EBX (info-> ebx)
35 #define ECX (info-> ecx)
36 #define EDX (info-> edx)
37 #define ESI (info->
38 eti)esi) 38 #define EDI
39 (info-> edi) 39 #define EBP
40 (info-> ebp) 40 #define ESP
41 (info-> esp) 41 #define EIP
42 (info-> eip)

42 #define ORIG_EIP (info-> orig_eip)
43 #define EFLAGS (info-> eflags)
44 #define DS (*(unsigned short *) &(info-> ds))
45 #define ES (*(unsigned short *) &(info-> es))
46 #define FS (*(unsigned short *) &(info-> fs))
47 #define CS (*(unsigned short *) &(info-> cs))
48 #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 を再定義することです。
// を返さない関数として（つまり、前に volatile を追加する）。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。
// を使用して、math_abort関数の定義を再確認します。これに続いて、対応する
// パラメーターです。関数名の前にキーワード volatile を置くことで、関数を装飾することができます。
// は、関数が返されないことを gcc コンパイラに伝えるためのもので、gcc は次のように生成します
//
// より良いコード
50 void math_abort(struct info *, unsigned int);
51
52 #define math_abort(x,y) ●。
53((volatile void (*)(struct info *,unsigned int)) math_abort)((x),(y)))
54
55 /*
56 * Gccはこの馬鹿げたアライメント問題を強制します：2つのロングだけを使用したい。
57 * 仮の実数である64ビットの仮数に対して、gccでは
58 * 構造体を12バイトにすると、math_emulate.cが壊れてしまします。I
59 * "no-align" プラグマのようなものが必要です。
60 */
61
// 一時的な本物の構造
63 // 合計64ビットのショルダを持っています。ここで、「a」は下位32ビット、「b」は上位32ビットを表
64 す
64     exponent;
65 // (1固定ビットを含む)で、「exponent」は指数値です。
66 typedef struct {

```

```

// 上記のオリジナルノートで述べたアライメント問題を解決するために設計された構造体
// そして、上記の temp_real 構造体のように動作します。
67 typedef struct {
68     short m0, m1, m2, m3;
69     short exponent;
70 } temp_real_unaligned;
71
// 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))。
72 #define real_to_real(a, b) ※※※(a), ((b)->exponent = (a)->exponent)
73((long long *) (b) = *(long long *) (a)), ((b)->exponent = (a)->exponent))
74
// 長い実数(倍精度)の構造体です。
75 typedef struct {
76     long a, b; // 'a' は下位32ビット、'b' は上位32ビットを表します。
77 } long_real;
78
79 typedef long short_real; // 短い実数型を定義します。
80
// 一時的な整数構造。
81 typedef struct {
82     long a, b; // 'a' は下位32ビット、'b' は上位32ビット。 short sign;
83
84 } 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
// のコプロセッサです(図11-6参照)。
86 struct swd {
87     int ie:1;          // 無効な操作の例外。
88     int de:1;          // 非正規化された例外。
89     int ze:1;          // ゼロによる除算の例外。
90     int oe:1;          // オーバーフローの例外。
91     int ue:1;          // アンダーフローの例外。
92     int pe:1;          // 精密な例外。
93     int sf:1;          // アキュムレータのオーバーフローが原因で発生するスタックエラ
94     int ir:1;          // ーフラグ。
95     int c0:1;          // Ir, b: 上記6つのマスクされていない例外が発生した場合に設定
96     int c1:1;          // します。
97     int c2:1;          // c0--c3: コンディションコードビット。
98     int top:3;          // 現在、スタックの先頭にある80ビットのレジスタを示す。
99     int c3:1;
100    int b:1 と
101 }; なります。
102
// 80387内部レジスタ制御モード定数。
103 #define I387 (current->tss.i387)           // 80387 プロセスのステータス情報です。
104#define SWD (*(struct swd *)&I387.swd)       // 80387のステータスコントロールワード。
105#define ROUNDING ((I387.cwd >> 10) & 3)        // コントロールワードで丸めモードを取得します
106#define PRECISION ((I387.cwd >> 8) & 3)        //
107
// 精度の有効桁数を定義する定数。
108#define BITS24      0      // 精度有効ビット数: 24ビット
109#define BITS53      2      // 53ビット
110#define BITS64      3      // 64ビット

```

```

111 // 丸めモードの定数を定義します。
112 #define ROUND_NEAREST 0           // 最も近いか偶数に丸める。
113 #define ROUND_DOWN 1            // 負の無限大へのトレンド
114 #define ROUND_UP 2             // 正の無限大へのトレンド
115 #define ROUND_0 3              // ゼロに切り替わるトレンド

// 定数の定義。
117 #define CONSTZ   (temp_real_unaligned) {0x0000, 0x0000, 0x0000, 0x0000, 0x0000} // 0
118 #define CONST1   (temp_real_unaligned)                                // 1.0
119 #define CONSTPI  {0x0000, 0x0000, 0x0000, 0x8000, 0x3FFF}.                // パイ
120     #define CONSTLN {0xC235, 0x2168, 0x1A2, 0xC90F, 0x14000} // Loge(2)
{0x79AC, 0xD1CF, 0x17F7, 0xB172, 0x3FFE}.          121     #define CONSTLG2 // Log10(2)
(temp_real_unaligned) {0xF799, 0xFBDF, 0x9A84, 0x9A20, 0x3FFD}.      122     #define // Log2(e)
CONSTL2E (temp_real_unaligned) {0xFOBC, 0x5C17, 0x3B29, 0xB8AA, 0x3FFF} 123 #define // Log2(10)
CONSTL2T (temp_real_unaligned) {0x8AFE, 0xCD1B, 0x784B, 0xD49A, 0x4000} 124

// 80387の状態を設定します。
125 #define set_IE() (I387.swd |= 1)
126 #define set_DE() (I387.swd |= 2)
127 #define set_ZE() (I387.swd |= 4)
128 #define set_OE() (I387.swd |= 8)
129 #define set_UE() (I387.swd |= 16)
130 #define set_PE() (I387.swd |= 32)

// 80387の制御条件を設定
132 #define set_C0() (I387.swd |= 0x0100)
133 #define set_C1() (I387.swd |= 0x0200)
134 #define set_C2() (I387.swd |= 0x0400)
135 #define set_C3() (I387.swd |= 0x4000)
136

137 /* ea.c */
138 // エミュレーション命令でオペランドが使用する実効アドレスを計算する、つまり
// 命令内のアドレッシングモードバイトに応じて、実効アドレスを計算します。
// Params: info - 割り込み時のスタックの内容; code - 命令コード。
// 有効なアドレスを返します。
139 char * ea(struct info * info, unsigned short code); 140

141 /* convert.c */
142 // convert.cファイルで実装されている様々なデータ型変換関数。 143 void
short_to_temp(const short_real * a, temp_real * b);
144 void long_to_temp(const long_real * a, temp_real * b);
145 void temp_to_short(const temp_real * a, short_real * b);
146 void temp_to_long(const temp_real * a, long_real * b); 147
void real_to_int(const temp_real * a, temp_int * b); 148 void
int_to_real(const temp_int * a, temp_real * b); 149

150 /* get_put.c */
151 // 様々なタイプの機能にアクセスできます。
152 void get_short_real(temp_real *, struct info *, unsigned short);
153 void get_long_real(temp_real *, struct info *, unsigned short);
154 void get_temp_real(temp_real *, struct info *, unsigned short);
155 void get_short_int(temp_real *, struct info *, unsigned short);
156 void get_long_int(temp_real *, struct info *, unsigned short) と
なります。

```

```

157 void get_longlong_int(temp_real *, struct info *, unsigned short);
158 void get_BCD(temp_real *, struct info *, unsigned short);
159 void put_short_real(const temp_real *, struct info *, unsigned short);
160 void put_long_real(const temp_real *, struct info *, unsigned short);
161 void put_temp_real(const temp_real *, struct info *, unsigned short);
162 void put_short_int(const temp_real *, struct info *, unsigned short);
163 void put_long_int(const temp_real *, struct info *, unsigned short) と
なります。
164 void put_longlong_int(const temp_real *, struct info *, unsigned short);
165 void put_BCD(const temp_real *, struct info *, unsigned short);
166
167 /* add.c */
168 // 浮動小数点の加算命令をシミュレートする関数です。 169 void
fadd(const temp_real *, const temp_real *, temp_real *);
170
171 /* mul.c */
172 // 浮動小数点の乗算命令をシミュレートします。
173 void fmul(const temp_real *, const temp_real *, temp_real *);
174
175 /* div.c */
176 // 浮動小数点の分割命令をシミュレートします。
177 void fdiv(const temp_real *, const temp_real *, temp_real *);
178
179 /* compare.c */
180 // i浮動小数点の比較命令をシミュレートします。 // FCOM、2つの数値を比較する。
181 void fucom(const temp_real *, const temp_real *); // FUCOM、順序比較なし。
182 void ftst(const temp_real *); // FTST、トップスタックのアキュムレータが0と
183 // 比較される。
184
185 #endif
186

```

28. mm.h

1. 機能性

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。

2. コードアナリティクス

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。一般的に使用されているいくつかの製品のプロトタイプ定義が含まれています。
// カーネルの機能を利用しています。

```

```

// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナル
の定義
6 #include <signal.h>
(日本語)
7 #include <signal.h>
8 // メモリページのスワップデバイス番号 (mm/memory.c, line
9 extern int SWAP_DEV; 36).
10 // 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。
11 _rw_page() 関数は
// パラメータ'nr'は、ファイルblk_drv/ll_rw_block.cで定義されているページ番号です。
// buffer'はリード/ライトバッファ。
11#define read_swap_page(nr, buffer) _ll_rw_page(READ, SWAP_DEV, (nr), (buffer));
12#define write_swap_page(nr, buffer) _ll_rw_page(WRITE, SWAP_DEV, (nr), (buffer));
13
// 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
。
14 extern unsigned long get_free_page(void)です。
// 内容が変更された物理的なメモリページを、指定された
// 線形アドレス空間。put_page()とほぼ同じです。
15 extern unsigned long put_dirty_page(unsigned long page, unsigned long address)です。
// アドレス「addr」で始まる物理メモリのページを解放する。
16 extern void free_page(unsigned long addr);
// 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)。
17 void swap_free(int page_nr);
// 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).
18void swap_in(unsigned long *table_ptr);
19
//// アウトオブメモリー (oom) 処理機能。
20 extern inline volatile void oom(void)
21 {。
22 // do_exit()では必ず終了する必要があります。シグナル値と同じ値のエラーコード
23 // SIGSEGV(11)はSIGSEGV(11)は暂时的に unavailable "であり、これは同義である。
24 }
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
// トランステーション・ロックサイド・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は、バッファー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、EAX
= 0であることを示しています。
// ページディレクトリのベースアドレス。
26#define invalidate() 。。。
27asm ("movl %eax,%cr3": : "a" (0))
28
29 #define LOW_MEMORY 0x00000000L // 0x00000000Lを変更せずに変更することは物理的なメモリの下限 (1MB) です。
31extern unsigned long long HIGH_MEMORY; // 32サポートしていない物理最大メモリの容量は 16MB です。これらの
#define 定義を既存メモリ 15MB 10MB のメモリを搭載することができますが、これはメモリのサイズ (15MB) です
33#define PAGING_PAGES (PAGING_MEMORY>>12) 。
34#define MAP_NR(addr) (((addr)-LOW_MEMORY)>>12) // ペーシング (3840) 後のページ数です。
35#define USED 100 // メモリアドレスをページ番号にマッピングしま
36 // す。
// Page used flag, see memory.c, line 449.

```

```

// 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
// ページが現在参照されている（使用されている）回数を示すために使用されます。これは、最大で
// 15Mbのメモリ空間。memory.cプログラムの関数mem_init()では、その位置は
    あらかじめメインメモリ領域のページがUSED(100)に設定されているため、//は使用できません。
37 extern unsigned char mem_map [ PAGING_PAGES ];
38
    // 以下に定義されたシンボル定数は、ページディレクトリのフラグビットのいくつかに対応しています。
39 #define PAGE_DIRTY (セカンドページテーブルのページが修正されている)
    // ビット5、ページにアクセスした。
40 #define PAGE_ACCESSED 0x20
41 #define PAGE_USER     0x04 // ビット2、このページは以下に属します。1-user; 0-superuser.
42 #define PAGE_RW       0x02 // ビット1、リード/ライトの権利。1 - 書き込み、0 - 読み取り。
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

```

29. sched.h

1. 機能性

スケジューラのヘッダファイルであるSched.hには、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と取得のための組み込みアセンブリ関数マクロ、タスクコンテキストスイッチマクロswitch_to()が定義されています。

タスク切り替えマクロswitch_to(n)(222行目から)では、まず構造体「struct {long a,b;} tmp」を宣言し、カーネルのステートstackoverflow上に8バイトのスペースを確保しています。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ありません、そのまま終了します。そうでなければ、新しいタスクのセレクタTSS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内のデータには

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、間接的にtmpにジャンプする命令LJMP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 コードアノテーション

プログラム 14-26 linux/include/linux/sched.h

```

1 #ifndef _SCHED_H
2 #define _SCHED_H
3
4 #define HZ 100           // システムクロックのティック周波数の定義 (10ms/tick)
5
6 #define NR_TASKS       64           // システム内のタスクの最大数です。
7 #define TASK_SIZE      0x04000000  // 各タスクのサイズ (64MB) です。
8 #define LIBRARY_SIZE   0x00400000  // 読み込まれたライブラリのサイズ (4MB)
9
10 #if (TASK_SIZE & 0xffff)
11 #error "TASK_SIZE must be multiple of 4M"
12#endif
13
14 #if (LIBRARY_SIZE & 0xffff)
15#error "LIBRARY_SIZE must be a multiple of 4M"
16#endif
17
18 #if (LIBRARY_SIZE >= (TASK_SIZE/2))
19 #error "LIBRARY_SIZE too damn big!"
20#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
// 次のマクロ CT_TO_SECS および CT_TO_USECS は、現在のシステムのティックを変換するために使
用されます。
// を秒とマイクロ秒に変換します。
28#define CT_TO_SECS(x) ((x) / HZ)
29#define CT_TO_USECS(x) (((x) % HZ) * 1000000/HZ)
30#define FIRST_TASK task[0].          // タスク0は特別なので、そのためのシンボルを定義し
31#define LAST_TASK task[NR_TASKS-1].  // タスク配列の最後のタスクです。
32
33// <linux/head.h> ヘッドのヘッダーファイルです。セグメントディスクリプターの簡単な構造が定義
されています。
// いくつかのセレクタ定数と一緒に
// <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。システムの限界と利用に関する情報が含まれています。
// プロセスが使用するリソース。
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナル
// の定義
// 操作関数のプロトタイプ。
34 #include <linux/head.h>
35 #include <linux/fs.h>
36 #include <linux/mm.h>
37 #include <sys/param.h>
38 #include <sys/time.h>
39 #include <sys/resource.h>
40 #include <signal.h>
41
42 #if (NR_OPEN > 32)
43#error "Currently clos-on-exec-flags and select masks are in one long, max 32 files/proc" 44
#endif
45 // このを定義しています。
46 #define TASK_RUNNING
47 #define task_interruptible
48 #define task_uninterruptible
49 #define TASK_ZOMBIE
50 #define TASK_STOPPED
51
52 #ifndef NULL
53 #define NULL ((void *) 0)
54 #endif // 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)
55 // 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。 56 extern
int copy_page_tables(unsigned long from, unsigned long to, long size);
    // 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) 57
extern int free_page_tables(unsigned long from, unsigned long size);
58
    // スケジューラの初期化関数です。(kernel/sched.c, 417) 59 extern void
sched_init(void);
    // プロセスのスケジューリング機能。(kernel/sched.c, 119)
60 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た後、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に書き込む。(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
    // 以下は、数学コプロセッサが使用する構造で、主に保存するために使用されます。
    // プロセス切り替え時のi387の実行状況情報。
67 struct i387_struct {
68     ロング cwdです。      // コントロール
// ワード。
69     ロング swdです。      // Status Wordで
// す。
70     ロング twd;          // タグ・ワード

```

```

71     ロング    fipです。      // コプロセッサコード ip ポインタ。
72     ロング    fcs;        // コプロセッサのコードセグメントレジスタ。
73     ロング    フーである。   // メモリオペランドのオフセット。
74     ロング    fos;        // メモリオペランドのセグメントです。
75     ロング    st_space[20]で /* 各FP-regの8*10バイト=80バイト */。
76 };
77 // TSS (Task Status Segment) データ構造。
78 struct tss_struct {
79     ロング    back_link    /* 16 ハイ ピッ ゼロ */ト
80     ロング    esp0;
81     ロング    ss0です。    /* 16 ハイ ピッ ゼロ */ト
82     ロング    esp1;
83     ロング    ss1です。    /* 16 ハイ ピッ ゼロ */ト
84     ロング    esp2;
85     ロング    ss2です。    /* 16 ハイ ピッ ゼロ */ト
86     ロング    cr3です。
87     ロング    eip;
88     ロング    eflagsです。
89     ロング    eax,ecx,edx,ebx
90     ロング    。。
91     ロング    esp;
92     ロング    ebpです。
93     ロング    esi;
94     ロング    edi;
95     ロング    es;          /* 16 ハイ ピッ ゼロ */ト
96     ロング    csです。    /* 16 ハイ ピッ ゼロ */
97     long trace_bitmap; /* bits: trace 0, bitmap 16-31 */。
98     ロング struct i387_struct i387; /* 16 ハイ ピッ ゼロ */ト
99 };
100    ロング    ds;          /* 16 ハイ ピッ ゼロ */ト
// 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 gsです。
// 長い状態。
101    ロング    fs;          /* 16 ハイ ピッ ゼロ */ト
// -1 実行不可能、0 実行可能（準備完了）、> 0 停止。
// #16クのアビメイゼロ */トク（デクリメント）、ランタイム
// スライス。ト
// 優先度。タスクの実行を開始すると、counter=priorityとな
// struct sigaction sigaction[32]; // ますシグナル属性構造体。シグナルの操作とフラグです。
// long blocked; // 信頼のやばくドマスクさねだ信頼のビタ木やレディを処理します。1)です。
// int exit_code; // タスクが停止した後にコードを終了すると、親がそれを
// 符号化されていない長い // 取得します。
// start_code; 符号化されて // リニアアドレス空間でのコード開始位置。
// いない長いend_code; 符号 // マークの長さまたはオフセット(バイト)。
// 価値がない長い長い; // データの長さ(バイト)。
// 矢印dataの長さのstart_stack // データの底の位置ターゲット(バイト)。
// 。
// long pid;           // プロセス識別子。
// long pgrp;         // プロセスグループ番号
// ロングセッション。 // セッション番号を処理します。

```

```

// ロングリーダーです。
// int groups[NGROUPS];
// task_struct *p_pptr;
// task_struct *p_cptr;
// task_struct *p_ysptr;
// task_struct *p_osptr;
// unsigned short uid;
// unsigned short euid;
// unsigned short suid;
// unsigned short gid;
// unsigned short egid;
// unsigned short sgid;
// 長いタイムアウト。
// ロングアラーム
// long utime;
// long stime;
// struct rlimit rlim[RLIM_NLIMITS];
// long cutime;
// このような場合には、次の
// ようにします。
// unsigned short umask;
// struct m_inode *start_inode;
// struct m_inode *root;
// struct m_inode *executable;
// library; // ロードされたライブラリのi-node構造体へのポインタ。
// 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
// struct file *filp[NR_OPEN]; // ファイル構造体のポインタテーブル、最大32項目まで。
// 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TSS。
//};

105 struct task_struct {
106 /* これらはハードコードされています。
107     long state; /* -1 unrunnable, 0 runnable, >0 stopped */
108     long counter;
109     long priority;
110     long signal;
111     struct sigaction sigaction[32];
112     long blocked; /* マスクされた信号のビットマッ
113 /* 様々なプロパティ */
114     int exit_code;
115     unsigned long start_code, end_code, end_data, brk, start_stack;
116     long pid, pgrp, session, leader;
117     int groups[NGROUPS];
118     /*
119         * 親プロセス、末っ子、弟妹へのポインタ。
120     * それぞれ年上の兄弟です。(p->father は以下のように置き換えることが
121     * できます。
122         * p->p_pptr->pid
123         */
124     struct task_struct *p_pptr, *p_cptr, *p_ysptr, *p_osptr;

```

```

124     符号化されていない short
125     uid, euid, suid; 符号化されてい
126     ない short gid, egid, sgid; 符号
127     化されていない long
128     timeout, alarm
129     long utime, stime, cutime, cstime, start_time;
130     struct rlimit rlim[RLIM_NLIMITS];
131 /* ファンシーフラグの flags; /* プロセスごとのフラグ、以下に定義 */
132 */
133     unsigned short used_math/* -1 は tty がない場合、署名が必要です
134     unsigned short umask; 。
135     struct m_inode * pwd;
136     struct m_inode * root;
137     struct m_inode * executable;
138     struct m_inode * library;
139     unsigned long close_on_exec;
140     struct file * filp[NR_OPEN]
141 /* このタスクの init 0 - ゼロ 1 - cs 2 - ds&ss */
142 */
143 struct desc_struct ldt[3]; 142 /*
144 のタスクのための tss*/struct tss;
145 */
146 /*
147 * プロセスごとのフラ
148 */
149 #define PF_ALIGNWARN 0x00000001      /* アライメントの警告メッセージを表示
150                                         します。
151                                         /* まだ実装されていない、486*/のみ
152 */
153 /* INIT_TASK は、最初のタスクテーブルをセットアップするために使用され、次のようにタッ
   チします。
154 * your own risk!Base=0, limit=0x9ffff (=640kB) 155
*/
// 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。
156 /* state etc */ASK 0x15015,          // ステート、カウンター、プライオリティ
..^.. ) 158 /* signals */           // シグナル、sigaction[32], blocked
0, {}, {}, 0, ..^.. )               // exit_code, start_code, end_code, end_data, brk, start_stack
159 /* ec, brk... */ 0, 0, 0, 0, 0, 0, ¥ // pid, pgrp, セッション, リーダー
160 /* pid etc... */ 0, 0, 0, 0, 161    // groups[].
161 /* supports links */ init_task. task, 0, 0, 0, ..^.. // p_pptr, p_cptr, p_ysptr,
162 /* uid など */ 0, 0, 0, 0, 0, 0, // uid, euid, suid, gid, egid, sgid
163 /* timeout */ 0, 0, 0, 0, 0, 0, // alarm, utime, stime, cutime, cstime, start_time, used_math
164 /* rlimits */ {0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff
165             {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}, \(^o^)
166             {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}}, .
167 /* flags */ 0, ¥                  // 旗
168 /* math */ 0, ¥                 // used_math, tty, umask, pwd, root, executable,
169 /* fs情報 */ -1, 0022, null, null, lse1, 00, である
170 /* filp */ {NULL}, // \(^o^)
171 /* ldt */ {0, 0}, ¥              // ldt[3].
172 {¥
173 {0x9f, 0xc0fa00}, // コードサイズ 640K, ベース 0x0, G=1, D=1, DPL=3, P=1
174 /* ldt */ . TYPE=0xa
175 {0x9f, 0xc0f200}, // データサイズ 640K, ベース 0x0, G=1, D=1, DPL=3, P=1
176 . TYPE=0x2

```

```

176 }, ¶
177 /*tss*/ {0, PAGE_SIZE+(long)& init_task, 0x10, 0, 0, 0, (long)& pg_dir, 0 // tss
178     0, 0, 0, 0, 0, 0, 0, ¶
179     0, 0, 0x17, 0x17, 0x17, 0x17, 0x17, 0x17, 0x17, ¶
180     LDT(0), 0x80000000, „^w^“
181     {}¶
182 }, ¶
183 }
184
185 extern struct task_struct *task[NR_TASKS]; // タスクのポインタの配列です
186 extern struct task_struct *last_task_used_math; .
187 extern struct task_struct *current; // 現在のプロセスポインタ。
188 extern unsigned long volatile jiffies; // 起動開始からの秒数 (10ms/tick)。
189 extern unsigned long startup_time; // 起動時間。1970:0:0:0からの秒数。
190 extern int jiffies_offset; // 調整が必要な目盛りの数。 191
192 #define CURRENT_TIME (startup_time+(jiffies+jiffies_offset)/HZ) // 現在の時刻(秒)です。
193
194 // 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)
195 extern void add_timer(long jiffies, void (*fn)(void));
196 // 途切れることのないスリープ待ち。(kernel/sched.c)
197 extern void sleep_on(struct task_struct ** p);
198 // スリープ待ちで中断。(kernel/sched.c)
199
200 // スリープの処理を明確に起こす。(kernel/sched.c) 197
201 // 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。
202 extern int in_group_p(gid_t grp);
203 /*
204 // 最初のTSSを探すためのgdtへの入力。0-nul, 1-cs, 2-ds, 3-syscall 202 *
205 // 4-TSS0, 5-LDT0, 6-TSS1など ...
206 // 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
207 // 2-データセグメント ds、3-システムセグメント syscall、4-タスクステートセグメント TSS0
208 // 、5-ローカルテーブル
209 // LTD0、6タスクのステートセグメントTSS1などです。
210 // 元のコメントから推測できるように、リーナス氏はシステムのコードを
211 // GDTテーブルの4つ目の独立したセグメントで呼び出します。しかし、その後、それをしなかったので、彼は
212 // GDTテーブルの4番目のディスクリプターアイテム（syscallアイテム）をアイドル状態にしました。
213 // 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。
214 #define FIRST_TSS_ENTRY 4
215 #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
// 合計16バイトを占めるディスクリプターのうち、n<<4>>は、対応する
// TSSのスタート位置。このマクロで得られる値は、セレクタのインデックス値でもある
// TSSの//です。後者のマクロは、GDT内のLDT記述子のセレクタ（オフセット）を定義します。
216#define TSS(n) (((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3>>)
217#define LDT(n) (((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3>>)

// 埋め込みアセンブリマクロは、n番目のタスクのTSSセグメントセレクタを

```

```

// タスクレジスターTRです。後者のマクロは、n番目のLDTセグメントセレクタをロードするために使用されます。
// タスクをローカルディスクリプター一テーブルレジスタLDTRに入れます。
208 #define ltr(n) asm ("ltr %%ax": "a" ( TSS(n)) ) 209
#define lldt(n) asm ("lldt %%ax": "a" ( LDT(n)) )

// 現在実行中のタスクのタスク番号を取得する（これはタスク配列のインデックスであり
// は、プログラムkernel/traps.cの78行目で使用されているプロセス番号pid）とは異なります。
// 返却: n - 現在のタスク番号。
210 #define str(n) ♪♪♪♪
211     asm ("stib%ax[%eax]" : "a" (FIRST_TSS_ENTRY) ) // タスクレジスターTSSセレクタをEAXに保存します。
212     /* $4,%eax" ♪♪♪♪ // (EAX / 16) => EAX = 現在のタスク番号。
213         : "a" (0), "+i"
214     /* (FIRST_TSS_ENTRY)<<3)
215
216 * switch_to(n) はタスクをタスク nr n に最初に切り替えます。
217 * nが現在のタスクでないことをチェックし、その場合は何もしません。
218 * また、切り替え先のタスクでTSフラグをクリアします。
219 * tha math co-processor latest.
220 */
221 */

// TSSセレクタで構成された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セレクタ用のtmp.bを指す； DX - TSSセグメント
// 新しいタスク n のセレクタ； ECX - 新しいタスク n のタスク構造体ポインタ task[n].
// 一時的なデータ構造であるtmpは、ファージャンプ命令のオペランドを構成するために使用される
222 行目 // です。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
構成されています。
// 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。「a」の値は
// は、タスク切り替えが発生するような長いジャンプには使えません。行目の間接ジャンプ命令は
// 223では、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。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
// 最後に使用されたコプロセッサのタスクポインタが last_task_used_math 変数に保存された状態
で
// 元のタスクが前回実行されたかどうかを判断します。の説明を参照してください。
// kernel/sched.cのmath_state_restore()関数。
222 #define switch_to(n) { 223
224     struct {long a,b;} Ijmp; // そうであれば、何もせずに戻ります。
225     asm ("cmpl %ecx,%dx(%cx,%bx,%b1)" : "+a" (Ijmp.a) : "c" (task[n]), "b" (tmp.b) : "cc") // task[n]はカurrentがtmpに格納される
226     if(Ijmp.a) { // task[n]? Current = task[n]; ECX = タスクが切り替わった。
227         xchgl %ecx, _current // *tmpにロングジャンプし、タスク切り替えを行う
228         ljmp %0n%t" ♪♪♪♪ // 元のタスクがスイッチバックされるまで継続しません。最初のチェックは
// 元のタスクが前回コプロセッサを使用していました。その場合、CR0のTSフラグをクリアする
229         . cmpl %ecx, _last_task_used_math // 元のタスクが最後に数学を使った
230         jne Ijmp // そうでない場合は、前方にジャンプして終
231         clts // 了します。
232         "1:" // そうであれば、CR0のTSフラグをクリアし
233         :: "m" (*& tmp.a), "m" (*& tmp.b) // ください。
234         "w" ) // exit.
235     } [ d( TSS(n)), "c" ((long) task[n]); // ページの配置（カーネルのどこにもこれを参照する場所はありません）
236     // ページの配置（カーネルのどこにもこれを参照する場所はありません）

```

```

237 #define PAGE_ALIGN(n) (((n)+0xffff)&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。
// %0 - addr offset 2; %1 - addr offset 4; %2 - addr offset 7; EDX - base address
。
239 #define _set_base(addr,base) ♪ ♪ ♪
240     asm ("movl %%dl,%0\ln\t" ) // base[EDX[addr+2]] の下位16ビット(15-0)(DL)を表示します
241
242         "movb %%dl,%1\ln\t"      // 上位16ビットのうち下位8ビット(23-16) => [addr+4].
243         "movb %%dh,%2"           // 上位16ビットのうち、上位8ビット(31~24)
244             ります。                  => [addr+7].
245             ;*((addr)+2)の ¥
246             "m" ように)。
247             "m" *((addr)+4)の ¥
248             ように)。
249             "m" *((addr)+7)の ¥
250             // 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
251             定します。 "d" (base) urchin
252             // %0 - アドレスの addr; %1 - アドレスの EDX の値が変更されたことを gcc に伝える。
253             // トの長さ(dx), %0\ln\t" ) // セグメントリミットの下位16ビット(15-0)=>[addr].
254             #define _set_lmt(lmt,addr) \(^o^)/ EDX のリミットの上位4ビット(19-16)=>DL.
255             "movb %1,%dh\ln\t"       // オリジナル [addr+6] => DH、上位4ビットはフラグ。
256             "andb $0xf0,%dh\ln\t"   // DH の下位4ビットをクリアします(19-16に格納されます)
257
258             ;*((addr)), \(^o^)
259             "m" *((addr)+6), "d" (limit) ♪ ♪
260             // 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
261             します。
262             // 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。
263 #define _set_base(ldt,base) _set_base( ((char *)&(ldt)) , base )
264 #define _set_limit(ldt,limit) _set_limit( ((char *)&(ldt)) , (limit-1)>>12 )
265
// 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。
// EDX - store base ( base ); %1 - addr offset 2; %2 - addr offset 4; %3 - addr offset 7
。
266 #define _move_base(addr) (^o^) // 上位16ビットのうち、上位8ビット(31-24) [addr+7]
267     unsigned long b268; } ) =>DH.
268
269     "shll $16,%edx\ln\t"      // 上位16ビットのうちの上位8ビットは移動(0)ま [addr+4]
270     /movw %1,%dx "となりま    // EDX[addr+2] => DX でベースの下位16(15-0)ビットになります
271     す。                      .
272     ;"=d" ( ベース )        // このように、EDXには32ビットのセグメントベースアドレ
273     : "m" *((addr)+2)),      斯が含まれています。
274     ^"m" *((addr)+4)),      .
275     ^"m" (base); } )        // LDTの ldt が指定された段メンバディスクリプターのベースアドレスを取る。
276 #define _get_base(ldt) _get_base( ((char *)&(ldt)) )
277
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の略です。これは、散乱した制限を

```

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、完全なセグメントの限界値を入力します。

// の値を指定されたレジスタに入力します。結果として得られるセグメント・リミット値は、実際の数

// 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。

// %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。

279 #define get_limit(segment) ({
280 unsigned long limit; ".^w^.")
281 asm ("lsl1 %1,%0n#tinc1 %0": "=r" (limit): "r" (segment)); 282
 limit;})
 283
284 #endif
 285

30. sys.h

1. 機能性

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。

2. コードアノテーション

プログラム 14-27 linux/include/linux/sys.h

1 /*
2 * なぜこれは.cファイルではなく 探究心....
 いのか? 3 */

4
5 extern int sys_setup(); // 0 - システムの初期化。 (kernel/blk_drv/hd.c, 74)
6 extern int sys_exit(); // 1 - プログラムの終了。 (kernel/exit.c, 365)
7 extern int sys_fork(); // 2 - 新しいプロセスを作成します。 (kernel/sys_call.s, 222)
8 extern int sys_read(); // 3 - ファイルを読む。 (fs/read_write.c, 55)
9 extern int sys_write(); // 4 - ファイルの書き込み。 (fs/read_write.c, 83)
10 extern int sys_open(); // 5 - ファイルを開く。 (fs/open.c, 171)
11 extern int sys_close(); 12 // 6 - プロセスを閉じるのを待ちます。
 extern int sys_waitpid(); 13 (kernel/exi~~t~~と作成 (fs/open.c, 214)
 extern int sys_creat(); 14
 extern int sys_link(); 15
 extern int sys_unlink(); 16
 extern int sys_execve(); 17
 extern int sys_chdir(); 18
 extern int sys_time(); 19
 extern int sys_mknod(); 20
 エクスターん int
21 エクスターん int sys_chmod(); 22 エクスターん int sys_chown(); 23 エクスターん int sys_break(); 24 エクスターん int sys_lseek(); 25 エクスターん int sys_getpid(); 26 エクスターん int sys_mount(); 27
 エクスターん int sys_umount();

// 10 = プロセス名を実行する (kernel/sys_call.s, 214)
// 11 = ファイル名を新規作成 (fs/open.c, 76)
// 12 = ディレクトリを変更する。 (fs/open.c, 134)
// 13 - 現在の時刻を取得 (kernel/sys.c, 134)
// 14 - ブロック/キャラクタの特殊ファイルを作成します。
// 15 - ファイルモードの変更
// 16 = ファイルモードを変更 (fs/open.c, 106)
// 17 = ファイルモードを変更 (fs/open.c, 33)*.
// 18 = ファイルモードを変更 (fs/open.c, 36)
// 19 = ファイルモードを変更 (fs/open.c, 122)
// 20 = ファイルモードを変更 (fs/open.c, 199)
// 21 - ファイルシステムをマウントします。
// 22 - ファイルシステムをアンマウントします。

```

28 extern int sys_setuid();           // 23 - プロセスユーザー idです (kernel/sys.c, 196)
29 extern int sys_getuid();           // 24 - プロセスユーザー idです (kernel/sched.c, 390)
30 extern int sys_stime();            // 25 - システム時刻の設定 (kernel/sys.c, 207)
33 extern int sys_fstat();           // 26 - 分岐点を用いてファイルの状態を取得 (kernel/sys.c, 38)*
31 extern int sys_ptrace();           // (fs/stat.c, 58)
34 extern int sys_pause();            // 27 - 実行中の時刻を一時停止 (kernel/sched.c, 370)
35 extern int sys_utime();            // (kernel/sched.c, 164)
36 extern int sys_stty();             // 28 - ファイルのアクセス時間と更新時間の設定 (fs/open.c, 25)
37 extern int sys_gtty();              // 29 - ターミナルの設定 (fs/open.c, 43)*
38 extern int sys_access();           // fsopen実行権限を設定します (kernel/sched.c, 370)
39 extern int sys_nice();              // 30 - ターミナルの設定を取得 (kernel/sys.c, 48)*
40 エクスター int
sysftime(); 41 エクスター
int sys_sync(); 42 エク
スター int sys_kill(); 43
エクスター int
sys_rename(); 44 エクスター
int sys_mkdir(); 45 エク
スター int sys_rmdir(); 46 エクスター int
sys_dup(); 47 エクスター int
sys_pipe(); 48 エクスター int
sys_times(); 49 エクスター int
sys_prof(); 50 エクスター int
sys_brk(); 51 エクスター int
sys_setgid(); 52 エクスター int
sys_getgid(); 53 エクスター int
sys_signal(); 54 エクスター int
sys_geteuid(); 55 エクスター int
sys_getegid(); 56 エクスター int
sys_acct(); 57
extern int sys_phys(); 58
extern int sys_lock(); 59
extern int sys_ioctl(); 60
extern int sys_fcntl(); 61
extern int sys_mpx(); 62
extern int sys_setpgid(); 63
extern int sys_ulimit(); 64
extern int sys_uname(); 65
extern int sys_umask(); 66
extern int sys_chroot(); 67
extern int sys_ustat(); 68
extern int sys_dup2(); 69
extern int sys_getppid(); 70
extern int sys_getpgrp(); 71
extern int sys_setsid(); 72
extern int sys_sigaction(); 73
extern int sys_sgetmask(); 74
extern int sys_ssetmask(); 75
extern int sys_setreuid(); 76
extern int sys_setregid(); 77
extern int sys_sigpending(); 78
extern int sys_sigsuspend(); 79
extern int sys_sethostname(); 80
extern int sys_setrlimit(); 81

```

// 23 - プロセスユーザー idです (kernel/sys.c, 196)
 // 24 - プロセスユーザー idです (kernel/sched.c, 390)
 // 25 - システム時刻の設定 (kernel/sys.c, 207)
 // 26 - 分岐点を用いてファイルの状態を取得 (kernel/sys.c, 38)*
 // (fs/stat.c, 58)
 // 27 - 実行中の時刻を一時停止 (kernel/sched.c, 370)
 // (kernel/sched.c, 164)
 // 28 - ファイルのアクセス時間と更新時間の設定 (fs/open.c, 25)
 // 29 - ターミナルの設定 (fs/open.c, 43)*
 // fsopen実行権限を設定します (kernel/sched.c, 370)
 // 30 - ターミナルの設定を取得 (kernel/sys.c, 48)*
 // 31 - 分岐点を用いてファイルの状態を取得 (fs/open.c, 28)*
 // 32 - プロセスを終了させる。 (kernel/exit.c, 205)
 // 33 - ファイル名の変更。 (kernel/sys.c, 53)*
 // 34 - ディレクトリを作る。 (fs/namei.c, 515)
 // 35 - ディレクトリを削除します。 (fs/namei.c, 635)
 // (fs/fcntl.c, 42)
 // 36 - ファイルハンドルを複製します。 (fs/pipe.c, 76)
 // (kernel/sys.c, 216)
 // 37 - パイプを作る。 (kernel/sys.c, 58)*
 // 38 - ランニングタイムの取得 (kernel/sys.c, 228)
 // 39 - 実行タイムゾーン。 (kernel/sys.c, 98)
 // 40 - データ・セグメント・レンを変更します。 (kernel/sched.c, 400)
 // (kernel/signal.c, 85)
 // 41 - プロセスグループIDを設定します。 (kernel/sched.c, 395)
 // (kernel/sys.c, 405)
 // 42 - プロセスグループIDの設定 (kernel/sys.c, 109)*
 // 43 - 信号処理をプロセス空間にマッピングします。 (kernel/sys.c, 119)*
 // (kernel/sched.c, 55) レクリトリの変更。 (fs/iot.c, 31)
 // 44 - デバイスのi/o制御。 (fs/fcntl.c, 47)
 // 45 - 効率的な操作の制御を取得します。 (kernel/sys.c, 124)*
 // 46 - リソース使用状況の統計 (kernel/sys.c, 245)
 // 47 - リソース使用状況の統計 (kernel/sys.c, 129)
 // 48 - システム情報を表示 (kernel/sys.c, 343)
 // 49 - デフォルトのファイル作成モードを取得します。 (fs/open.c, 91)
 // (fs/open.c, 20)
 // 50 - ファイルシステムの状態を取得します。 (fs/fcntl.c, 36)
 // (kernel/sched.c, 385)
 // 51 - ファイルハンドルが重複しています。 (kernel/sys.c, 271)
 // (kernel/sys.c, 276)
 // 52 - パラネットのプロセスIDを取得します。 (kernel/signal.c, 100)
 // (kernel/signal.c, 14)
 // 53 - pidの取得 (getpid(0)) (kernel/signal.c, 19)
 // (kernel/signal.c, 159)
 // 54 - 新しいセッションIDを設定します。 (kernel/sys.c, 74)
 // (kernel/sys.c, 27)
 // 55 - セット信号の操作。 (kernel/signal.c, 48)
 // (kernel/signal.c, 357)
 // 56 - シグナルマスクコードを取得します。 (kernel/sys.c, 357)
 // 57 - シグナルマスクコードの設定 (kernel/sys.c, 357)
 // 58 - シグナルマスクコードの設定 (kernel/sys.c, 357)
 // 59 - シグナルマスクコードの設定 (kernel/sys.c, 357)
 // 60 - リアル/エフィシェンシーのuidを設定します。 (kernel/sys.c, 387)
 // 61 - リアル/エフィシェンシーのpidを設定します。 (kernel/sys.c, 387)
 // 62 - 保留中の信号をチェックします。 (kernel/sys.c, 387)

```

81 extern int sys_getrlimit(); // 76 - 使用するリソースの制限を取得します。
(kernel/sys.c, 375) 82 extern int sys_getrusage(); // 77 - リソースの使用量を取得します。 83
83 extern int sys_gettimeofday(); // 78 - 一日の時間を取得します。 (kernel/sys.c, 440)
84 extern int sys_settimeofday(); // 79 - 一日の時間を設定します。 (kernel/sys.c, 466)
85 extern int sys_getgroups(); // 80 - プロセスの全グループIDを取得
86 extern int sys_setgroups(); (kernel/sys.c, 289)
87 extern int sys_select(); 88 // 81 - プロセスグループ配列の設定
extern int sys_symlink(); 89 // 84 = エクターン引数が状態構成待合 (fs/stat.c, 47)
90 エクスター int
sys_readlink(); 91 エクスター // 85 = エクターン引数の内容を読み (fs/stat.c, 69)
シ int sys_uselib(); 92 む (fs/namei.c, 767)
    // システムコールの割り込みで使用される中断手順を選択します。
    // ハンドラ(int 0x80)をジャンプテーブルとして使用します。
93 fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
94 sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
95 sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
96 sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
97 sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
98 sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
99 sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
100 sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
101 sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
102 sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
103 sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
104 sys_getpgrp, sys_setsid, sys_sigaction, sys_getmask, sys_ssetmask,
105 sys_setreuid, sys_setregid, sys_sigsuspend, sys_sigpending, 106
sys_sethostname, 106 sys_setrlimit, sys_getrlimit, sys_getrusage,
sys_gettimeofday,
107 sys_settimeofday, sys_getgroups, sys_setgroups, sys_select, sys_symlink,
108 sys_lstat, sys_readlink, sys_uselib };
109
110 /* これでもう手動で更新する必要はなくなりました....*/ 111
int NR_syscalls = sizeof(sys_call_table)/sizeof(fn_ptr); 112

```

31. tty.h

1. 機能性

tty.hファイルでは、端末のデータ構造やいくつかの定数、ttyのキューバッファ操作で使用されるマクロなどが定義されています。

2. コードアノテーション

プログラム 14-28 linux/include/linux/tty.h

```

1 /*
2 * 'tty.h' は tty_io.c が使用するいくつかの構造体といくつかの定義をしています。 3 *
4 * 注意! rs_io.s やその他の部分に何もないことを確認せずにこれに触れないでください。
5 * con_io.s が壊れます。いくつかの定数はシステムに組み込まれています (主に

```

```

6 * offsets into 'tty_queue',
7 */
8
9 #ifndef _TTY_H
10 #define _TTY_H
11
12 #define MAX_CONSOLES 8 // パーチャルコンソールの最大数です。
13 #define NR SERIALS 2 //シリアル端子の数です。
14 #define NR_PTYS 4 // psesudo端末の数です。
15
16 extern int NR_CONSOLES; // パ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。主に端末の入出力機能を定義しています。
// 非同期通信ポートを制御するインターフェースです。
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 キャラクタバッファキューデータ構造。読み取り、書き込み、および補助（カノニカル）に
// 使用されます。
// tty_struct構造のバッファキューリ。
// 最初のフィールド「data」には、文字列の値（文字数ではありません）を
// キューバッファです。シリアルターミナルの場合は、シリアルポートのアドレスが格納されます。
22 struct tty_queue {
23     符号化されていない // バッファやシリアルポートにあるチャーライン。
24     ロングデータ、符号 // バッファ内のデータヘッダ。
25     化されていないロン // バッファ内のデータテール
26     グヘッド、符号化さ // このキューバッファを待っているプロセスリスト
27     れていないロングテ // 。
28 }; // ル。 // バッファーです。
29     struct task_struct * proc_list;
// これがタスク[TTY_BUF_SIZE]レタイプをチェックする
30 #define 使用されたCONSOLE(min) (((min) & 0xCO) == 0x00) // コンソールです。
31 #define IS_A_SERIAL(min) (((min) & 0xCO) == 0x40) //シリアル端子。
32 #define IS_A_PTY(min) ((min) & 0x80) // psesudo terminal.
33 #define IS_A_PTY_MASTER(min) (((min) & 0xCO) == 0x80) // master pty.
34 #define IS_A_PTY_SLAVE(min) (((min) & 0xCO) == 0xC0) // スレートPTY
35 #define PTY_OTHER(min) ((min) ^ 0x40) // 他のタイプの端末。
36
// ttyキュー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。(テールは前に、ヘッドは
// tty_io.cの図を参照してください)。
// バッファポインタ'a'は1バイトずつ前後にシフトされ、もしそれが超過していれば
// バッファの右/左で、ポインタは周期的に移動します。
37 #define INC(a) ((a) = ((a)+1) & (TTY_BUF_SIZE-1))
38 #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。
// 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。
39 #define EMPTY(a) ((a)->head == (a)->tail)
40 #define LEFT(a) (((a)->tail-(a)->head-1)&(TTY_BUF_SIZE-1))
41 #define LAST(a) ((a)-> buf[(TTY_BUF_SIZE-1)]&((a)-> head-
1)]) 42 #define FULL(a) (! LEFT(a))
43 #define CHARS(a) (((a)->ヘッド-(a)->テール)&(TTY_BUF_SIZE-1))
// 'queue' バッファの 'tail' から文字を取得し、tail++とします。
// 'queue' キューバッファの 'head' に文字を配置し、head++。

```

```

44 #define GETCH(queue, c) ↳ ↳ ↳
45 (void) ({c=(queue)->buf[(queue)->tail]; INC((queue)-
>tail);}) 46 #define PUTCH(c, queue) ↳ ↳ ↳ > ↳ *。
47 (void) ({(queue)->buf[(queue)->head]=(c); INC((queue)->
head);})
48
49 #define 端末のNTR-CHAR((tty)入力されたいふ文字の種類を確認します。 // シグナルSIGINTを送信
50 #define QUIT_CHAR(tty) ((tty)->termios.c_cc[VQUIT]) します。
51 #define ERASE_CHAR(tty) ((tty)->termios.c_cc[VERASE]) 52 #define KILL_CHAR(tty) ((tty)->termios.c_cc[VKILL]) 53 #define EOF_CHAR(tty) ((tty)->termios.c_cc[VEOF])
54 #define START_CHAR(tty) ((tty)->termios.c_cc[VSTART]) 55 #define STOP_CHAR(tty) ((tty)->termios.c_cc[VSTOP])
56 #define SUSPEND_CHAR(tty) ((tty)->termios.c_cc[VSUSP])
57
58 // 端末のポート構造ios termios;
59 struct ttypgrpt {
60     int session; // 端末のioモードと制御構造
61     int stopped; // その端末が属するpグループ。
62     void (*write)(struct tty_struct * tty); // セッションです。
63     struct tty_queue *read_q; // 端末停止フラグ。
64     struct tty_queue *write_q; // tty write function pointer.
65     struct tty_queue *secondary; // tty read queue.
66 };
67
68
69 extern struct tty_struct tty_table[];
70 extern int fg_console; // tty write queue.
71 // 端末の番号です。
72 // 以下マクロは、端末に応じてtty_table[]に番号「nr」を取得します。
73 // 端末の種類に応じてtty_table[]に番号「nr」を入れる。
74 // 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。
75 // 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
76 // が使用されているので、端末番号「fg_console」をtty_table[]エントリのインデックスとして使
// 用することができます。
77 // でttyの構造を取得します。devが0より大きい場合は、2つのケースが考えられます。
78 // (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
79 // 数です。仮想端末の場合、tty_table[]のtty構造は、dev-1(0--)をインデックスとする。
80 // 63). その他の端末の場合は、tty構造のインデックスエントリがdevとなります。
81 // 例えば、シリアルターミナル1を意味するdev = 64の場合、そのtty構造は次のようにになります。
82 // ttb_table[dev].dev = 1 の場合、対応する端末の tty 構造体は tty_table[0] となる。
83 // tty_io.cプログラムの70~73行目をご覧ください。
84
85 #define TTY_TABLE(nr) \((^o)
86 (tty_table + ((nr) ?((nr) < 64)?(nr)-1:(nr)) : fg_console) )
87
88 // 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。
89 // 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
90 // 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
91 /* 定義されていない場合、qifitで|の値が _POSIX_VDISABLE とおかだせき、対応する特殊な
92 // 文字の使用は禁止されています。81行目の初期値では8進数で表されています。
93 start='Q'      ストップ='S'      susp='Z'      eol=0

```

```

78      reprint=^R          ディスカー      werase=^W      lnext=^V
79      eol2=0              ド=^U
80 */
81 #define INIT_C_CC "#>~*).
82
83 void rs_init(void);           //シリアル端子 init. (kernel/chr_drv/serial.c)
84 void con_init(void);         //制御端末 init. (kernel/chr_drv/console.c)
85 void tty_init(void);         // (kernel/chr_drv/tty_io.c)
86
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
90 void con_write(struct tty_struct *tty);        // (kernel/chr_drv/console.c)
91 void rs_write(struct tty_struct *tty);       // (kernel/chr_drv/serial.c)
92 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

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

		説明
param.h	パラメータ名	カーネル名
resource.h	1809 バイト	1992-01-03 18:52:56
stat.h	1376 バイト	1992-01-11 18:42:48
time.h	1799 バイト	1992-01-09 03:51:28
times.h	200 バイト	1991-09-17 15:03:06
type.h	928 バイト	1992-01-14 13:50:35
utsname.h	272 バイト	1992-01-04 15:05:42
wait.h	593 バイト	1991-12-22 15:08:01

33. param.h

1. 機能性

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。

2. コードアノテーション

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           // システムクロックの周波数で、1秒間に100回です。
5 #define EXEC_PAGESIZE 4096 // 実行可能なページサイズ。
6
7 #define NGROUPS        32  /* 1ユーザあたりの最大グループ数 */
8 #define NOGROUP         -1
9
10#define MAXHOSTNAMELEN 8   // ホスト名の最大長、8バイト。
11
12#endif
13
```

34. resource.h

1. 機能性

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getusage()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、プロセスマスク構造体のrlim[]配列にあります。この配列にはRLIM_NLIMITSという項目があり、それぞれの項目は制限事項を定義したrlimit構造体です。

図14-5に示すように、リソースの使用に関する制限を設定します。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、このヘッダーファイルの41～46行目に定義されています。

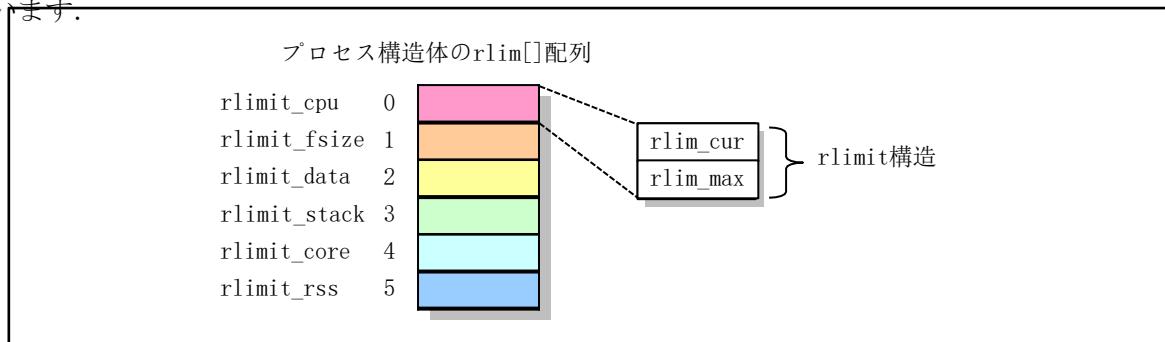


図14-5 プロセスマスク構造体のrlim[]配列の使い方

14.34.2 コードアノテーション

プログラム 14-30 linux/include/sys/resource.h

```

1 /*
2 * linux用の資源管理/会計ヘッダーファイル 3 \*/.
3
4
5 #ifndef _SYS_RESOURCE_H
6 #define _SYS_RESOURCE_H
7
8 // 以下のシンボル定数および構造体がgetusage()で使用されます。の 412 行目を参照してください
9 // kernel/sys.c フィル。
10 // struct rusage の定義は BSD 4.3 Reno 10 \*/ からの引用
11 // です。
12 // まだすべてをサポートしているわけではありませんが、あったほうがいいかもしれません....
13 // そうしないと、新しいアイテムを追加するたびに、このプログラムに依存するプログラムは
14 // 構造が負けてしまう。これにより、その可能性を減らすことができます。
15 */
16 // 以下は、getusage()のパラメータ'who'で使用されるシンボル定数です。
17 // 現在のプロセスのリソース使用状況を返します。

```

```

// 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。
15 #define RUSAGE_SELF_0          // 現在のプロセスのリソース使用率を返します。
16 #define RUSAGE_CHILDREN -1    // 子プロセスのリソース使用率を返します。
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。
// (include/sys/time.h)を参照してください。ru_utimeフィールドは、実行時間の統計情報を格納す
るために使用されます。
18構造体一覧の状態; ru_stimeフィールドは、実行時間の統計を保存するために使用されます。
カーネル状態のプロセスの //。
19     構造体 timeval_ru_utime;      /* 使用されたユーザータイム */
20     構造体 timeval_ru_stime;      /* 使用されたシステム時間 */
21     ロング RU_MAXRSSです。        /* レジデントセットの最大サイズ */
22     ロング ru_ixrssです。         /* 積算共有メモリサイズ */
23     ロング RU_IDRSSです。         /* 統合された非共有データのサイズ
*/
24     ロング ru_isrssです。         /* 積算非共有スタッカサイズ */
25     ロング ru_minflt;           /* page reclaims */
26     ロング ru_majflt;           /* ページフォルト */
27     ロング ru_nswap;            /* スワップ */
28     ロング ru_inblock;          /* 入力操作をブロックする */
29     ロング ru_oublock;          /* ブロック出力操作 */
30     ロング RU_MSGSNDです。       /* 送られたメッセージ */
31     ロング ru_msgrcvです。       /* 受信したメッセージ */
32     ロング ru_nssignalsです。     /* 受信した信号 */
33     ロング ru_nvcs。             /* 任意のコンテキストスイッチ */
34     ロング ru_nvcsさん          /* 不本意ながら */
35 } // getrlimit()とsetrlimit()で使用されるシンボル定数と構造体を以下に示します。
36 /*
37 * リソースの制限
38 */
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()とsetrlimit()の最初のパラメータリソースの値の //。実際には、これらの
// シンボリック定数は、プロセスタスク内のrlim[]配列の項目のインデックスである
// 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
41 #define RLIMIT_CPU      0      /* CPU時間(ミリ秒) */
42 #define RLIMIT_FSIZE     1      /* 最大ファイルサイズ */
43 #define RLIMIT_DATA      2      /* 最大データサイズ */
44 #define RLIMIT_STACK     3      /* 最大スタッカサイズ */
45 #define RLIMIT_CORE      4      /* コアの最大ファイルサイズ
*/
46 #define RLIMIT_RSS       5      /* レジデントセットの最大サ
イズ */
47 #ifdef notdef
48 #define RLIMIT_MEMLOCK   6      /* 最大ロックインメモリーアドレス空間 */
49 #define RLIMIT_NPROC     7
50 #define RLIMIT_OFILE     8      /* 最大プロセス数 */
51 #endif
52
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。この定数は
ここで定義されているリソースタイプの//は6つなので、最初の6つの項目だけが有効です。

```

```

54 #define RLIM_NLIMITS      6
55
56 #define RLIM_INFINITY     0x7fffffff // リソースは無制限であるか、または変更できません。
57
58 // リソースリミット構造
59 struct rlimit {
60     int    rlim_cur;           // 現在のリソース制限、またはソフトリミット。
61     int    rlim_max;          // ハードリミットです。
62
63#endif /* _SYS_RESOURCE_H */
64

```

35. stat.h

1. 機能性

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マクロと関数プロトタイプが示されています。

2. コードアナリティクス

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
6 // 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
7 struct stat {
8     dev_t   st_dev;           // ファイルが格納されているデバイス番号。
9     ino_t   st_ino;          // ファイルのi-node番号。
10    mode_t  st_mode;         // ファイルタイプとモード（下記参照）。
11    nlink_t st_nlink;        // ファイルへのリンクの数です。
12    uid_t   st_uid;          // ファイルのユーザーIDです。
13    gid_t   st_gid;          // ファイルのグループIDです。
14    dev_t   st_rdev;         // デバイス番号（特殊なチャーやブロックファイルである場合
15    off_t   st_size;         // ）。
16    time_t  st_atime;        // ファイルのサイズ（バイト）。
17    time_t  st_mtime;        // ラストアクセスタイム
18    time_t  st_ctime;        // 最終修正時刻。
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

```

```

// は類推していくことができます。
// ファイルタイプ
20 #define S_IFMT 00170000 // ファイルタイプのビットマスク（8進数）。
21 #define S_IFLNK 0120000 // シンボリックリンクです。
22 #define S_IFREG 0100000 // 通常のファイルです。
23 #define S_IFBLK 0060000 // harddisk dev/hd0のような特別なデバイスファイルをブロックします。
24 #define S_IFDIR 0040000 // ディレクトリです。
25 #define S_IFCHR 0020000 // Charのデバイスファイルです。
26 #define S_IFIFO 0010000 // FIFOのスペシャルファイルです。
// ファイルモードのビットです。// FIFOのスペシャルファイルです。
// 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調べるために使用されます。フラグが設定されている場合は
// プロセスの効率的なユーザーIDが、ファイル所有者のユーザーIDに設定されます。
27 #define S_ISUID 00004000 // 実行時にユーザIDを設定する（set-user-ID）。
28 #define S_ISGID 0002000 // 実行時にグループIDを設定する（set-group-ID）。
29 #define S_ISVTX 0001000 // ディレクトリの場合は、制限付き削除フラグ。
30
31 #define S_ISLNK(m) (((m) & S_IFMT) == S_IFLNK) // シンボリックリンクファイルであるか
32 #define S_ISREG(m) (((m) & S_IFMT) == S_IFREG) // どうかをテストします。
// 通常のファイルです。
33 #define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR) // ディレクトリを作成します。
34 #define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR) // charのデバイスファイルです。
35 #define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK) // ブロックデバイスファイル。
36 #define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO) // は、FIFO特殊ファイルです。
37 // ファイルアクセス許可:
38 #define S_IRWXU 00700 // 所有者は、読み取り、書き込み、実行/検索（Uはユーザー）できます。
39 #define S_IRUSR 00400 // オーナーが読める。
40 #define S_IWUSR 00200 // オーナーが書き込める。
41 #define S_IXUSR 00100 // オーナーが実行/検索できる
42
43 #define S_IRWXG 00070 // グループメンバーは、読み取り、書き込み、実行/検索ができます
44 #define S_IRGRP 00040 // (GはGroupの略)。
45 #define S_IWGRP 00020 // グループメンバーが読める
46 #define S_IXGRP 00010 // グループメンバーが書き込める。
47
48 #define S_IRWXO 00007 // グループメンバーが実行/検索できる
49 #define S_IROTH 00004 // 他の人は読む、書く、実行する、検索する（OはOtherの略）。
50 #define S_IWOTH 00002 // Others can read (最後の3文字がOtherを表す)。
51 #define S_IXOTH 00001 // 他の人も書ける。
52
53 extern int chmod(const char *_path, mode_t mode); 54 // ファイルモードの変更
extern int fstat(int fildes, struct stat *stat_buf); 55 // fhandleでファイルの状態情報を取
extern int mkdir(const char *_path, mode_t mode); 56 得します。
extern int mkfifo(const char *_path, mode_t mode); // ディレクトリを作る。
57 extern int stat(const char *filename, struct stat *stat_buf); // ファイルの状態情報を取
58 // mode_t umask(mode_t mask); // モードマスクを
59 // 設定します。
60 #endif
61

```

36. time.h

1. 機能性

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。

2. コードアノテーション

プログラム 14-32 linux/include/sys/time.h

```

1 #ifndef _SYS_TIME_H
2 #define _SYS_TIME_H
3
4 /* gettimeofday はこれを返します */
5 構造体 timeval {
6     ロン tv_sec;           /* 秒 */
7     グロ tv_usec;         /* マイクロ秒 */ /*
8 };    ング             マイクロ秒
9
// タイムゾーンの構造
// TZはTime Zoneの略で、DSTはDaylight Saving Timeの略です。
10 struct timezone {
11     int tz_minuteswest; /* グリニッジから西への分數 */ (注)
12     int tz_dsttime; /* dst補正の種類 */ (int)
13 };
14
15 #define DST_NONE      0      /* DSTではありません。*/
16 #define DST_USA        1      /* USAスタイルのDST */
17 #define DST_AUST       2      /* オーストラリアスタイルのDST */
18 #define DST_WET        3      /* 西ヨーロッパのDST */
19 #define DST_MET        4      /* 中欧のDST */
20 #define DST_EET        5      /* 東欧のDST */
21 #define DST_CAN        6      /* カナダ */
22 #define DST_GB         7      /* 英国・アイルランド */
23 #define DST_RUM        8      /* ルーマニア */
24 #define DST_TUR        9      /* トルコ */
25 #define DST_AUSTALT   10     /* 1986年にシフトしたオーストラリアスタイル */
26
// 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。
27 #define FD_SET(fd, fdsetp)  (*(fdsetp) |= (1 << (fd))) // fd セットに fd を設定します。
28 #define FD_CLR(fd, fdsetp)  (*(fdsetp) &= ~(1 << (fd))) // セット内のfdをクリアする。
29 #define FD_ISSET(fd, fdsetp) ((*(fdsetp) >> fd) & 1) // fdはセットに入っていますか?
30 #define FD_ZERO(fdsetp)    (*(fdsetp) = 0) // セット内のすべてのfdsを消去します。
31 * タイムバルの操作。 34 *
32 * NB: timercmp は >= や <= では動作しません
33 *
// timeval時間構造体の操作マクロです。
34 #define timerisset(tvp) ((tvp)->tv_sec || (tvp)->tv_usec)

```

```

38 #define timercmp(tvp, uvp, cmp) ↗ ↗ ↗
39     ((tvp)->tv_sec cmp (uvp)->tv_sec || `)
40     (tvp)->tv_sec == (uvp)->tv_sec && (tvp)->tv_usec cmp (uvp)->tv_usec)
41 #define timerclear(tvp)           ((tvp)->tv_sec = (tvp)->tv_usec = 0)
42
43 /*
44 * インターバルタイマーの名称と構造
45* タイマー設定の定義 46 */
46
47 #define ITIMER_REAL 0           // リアルタイムでの減少
48 #define ITIMER_VIRTUAL 1        // プロセスの仮想時間の減少
49 #define ITIMER_PROF 2          // プロセスの仮想時間またはランタイムの減少。
50
51 // 内部の時間構造
52 構造体 timeval_{
53     struct timeval_it_interval; /* タイマーの間隔 */
54     struct timeval_it_value;    /* 現在の値 */
55
56 #include <time.h>
57 #include <sys/types.h>
58
59 int gettimeofday(struct timeval* tp, struct timezone* tz);
60 int select(int width, fd_set* readfds, fd_set* writefds,
61 fd_set* exceptfds, struct timeval* timeout); 62
63 #endif /*_SYS_TIME_H*/
64

```

37. times.h

1. 機能性

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。

2. コードアノテーション

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です。基本的なシステムデータ型が定義されています。
5
6 struct timespec tms_utime; // ユーザーが使用したCPU時間。
7     time_t tms_stime; // システム（カーネル）が使用するCPU時間
8

```

```

9      time_t tms_cutime; // 終了した子プロセスが使用したユーザCPU時間 time_t
10     tms_cstime; // 終了した子プロセスが使用したシステムCPU時間
11 };
12
13 extern time_t times(struct tms * tp);
14
15 #endif
16

```

38. type.h

1. 機能性

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。

2. コードアノテーション

プログラム 14-34 linux/include/sys/types.h

```

1 ifndef _SYS_TYPES_H
2 define _SYS_TYPES_H
3
4 ifndef _SIZE_T
5 define _SIZE_T
6 typedef unsigned int size_t;           // オブジェクトのサイズ（長さ）に使用されま
7 endif                               す。
8
9 ifndef _TIME_T
10 define _TIME_T
11 typedef long time_t;                 // 時間（秒単位）に使用されま
12 endif                               す。
13
14 ifndef _PTRDIFF_T
15 define _PTRDIFF_T
16 typedef long ptrdiff_t;
17 endif
18
19 ifndef NULL
20 define NULL ((void *) 0)
21 endif
22
23 typedef int pid_t;                  // プロセスIDとプロセスグループIDに使用され
24 typedef unsigned short uid_t;       // ユーザーIDに使用されます。
25 typedef unsigned char gid_t;        // グループIDに使用されます。
26 typedef unsigned short dev_t;       // デバイス番号に使用します。
27 typedef unsigned short ino_t;        // inode番号に使用されます。
28 typedef unsigned short mode_t;      // いくつかのファイルモードで使用されます。

```

```

29 typedef unsigned short umode_t; // 
30 typedef unsigned char nlink_t; // ファイルリンクのカウントに使用し
31 typedef int daddr_t; // ます。
32 typedef long off_t; // ファイル内のオフセットに使用さ
33 typedef unsigned char u_char; // れます。
34 typedef unsigned short ushort; // unsigned char.
35 // unsigned short.
36 typedef unsigned char cc_t;
37 typedef unsigned int speed_t;
38 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設定。各ビットは1つのディス
41 // クリプターを表します。
42 typedef struct { int quot, rem; } div_t; // DIV操作に使用される。
43typedef struct { long quot, rem; } ldiv_t; // 長いDIV操作に使用される。 44
// ustat()関数のファイルシステムパラメータ構造です。最後の2つのフィールドは使用されません
// そして常にNULLを返します。
45 struct ustat {
46     daddr_t f_tfree; // システム内の総空きブロック数
47     ino_t f_tinode; // トータルフリーinode。
48     char f_fname[6]; // ファイルシステム名です。
49     char f_fpack[6]; // パックされたファイルシステム名。
50 };
51
52 #endif
53

```

39. utsname.h

1. 機能性

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。

2. コードアノテーション

```

1 #ifndef _SYS_UTSNAME_H プログラム 14-35 linux/include/sys/utsname.h
2 #define _SYS_UTSNAME_H
3
4 #include <sys/types.h> // 基本的なシステムデータタイプが定義されています。
5 #include <sys/param.h> // 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ます。

```

```

6 struct utsname {
7     char sysname[9];           // システム名です。
8     char nodename[MAXHOSTNAMELEN+1]; // ネットワーク上のノード名です
9     char release[9];          .
10    char version[9];          // リリースレベル
11    char machine[9];          // のバージョンです。
12};                           // ハードウェアの種類。
13
14 extern int uname(struct utsname * utsbuf);
15
16#endif
17

```

40. wait.h

1. 機能性

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。

2. コードアナリシジョン

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
6 define LOW(v)      ((v) & 0377)           // ローバイト(8進法)を取得します。
7 define HIGH(v)     (((v) >> 8) & 0377)        // ハイバイトの取得
8 /* waitpid のオプション、WUNTRACED はサポートされていません */。
// [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].
10 #define WIFSIGNALED(s) // 使用されないマクロ。関数waitpid()で使用される掛け括弧で走る。
11 define WUNTRACED    2                  // 停止していたチャイルドの状態を報告します。
12
// 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。
13 define WIFEXITED(s)   (!((s)&0xFF)) // 子供が正常に終了していれば真。
14 define WIFSTOPPED(s)  (((s)&0xFF)==0x7F) // 子供が停止していれば真。
15 define WEXITSTATUS(s) ((s)>>8)&0xFF // 出口の状態です。
16 define WTERMSIG(s)    ((s)&0x7F)       // プロセスを終了させたシグナル
17 define WCOREDUMP(s)   ((s)&0x80)       // コアダンプが実行されたかどうかを確認します。
18 define WSTOPSIG(s)    ((s)>>8)&0xFF
// 19 define WIFSIGNALLED(s) (((unsigned int)(s)&0x1&0x20)>0)止させたシグナル
0xFF) 20 define WIFSIGNALED(s) (((unsigned int)(s)-1 & 0xFFFF)
< 0xFF) 20
// wait()およびwaitpid()関数は、プロセスが自分の

```

```
// 子プロセス。この関数の様々なオプションにより、ステータス情報を取得することができます。  
終了または停止した子プロセスの //。ステータス情報がある場合  
// 2つ以上の子プロセスの場合、レポートの順序は指定されません。  
// wait()は、子プロセスの1つが終了するまで、現在のプロセスを中断します。  
// が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ります  
。  
// Waitpid() は、pid で指定された子プロセスが終了するか、または  
// プロセスの終了を要求するシグナル、またはシグナルハンドラを呼び出す必要があります。  
// pid=-1、オプション=0の場合、waitpid()はwait()関数と同じ動作をし、そうでない場合は  
// 挙動はpidとoptionsのパラメータによって異なります（kernel/exit.c, 142参照）。  
// パラメータの'pid'はプロセスID、'*stat_loc'はステータスの位置を示すポインタです。  
// 情報; 'options' は wait オプションで、上記の 10, 11 行目を参照してください。  
21 pid_t wait(int *stat_loc);  
22 pid_t waitpid(pid_t pid, int *stat_loc, int options);  
23  
24 #endif  
25
```

14.41 まとめ

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。

15 ライブラリファイル (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()などが実装されています。

execve()、メモリ割り当て関数malloc()、子プロセスの状態を待つ関数wait()、セッションを作成するシステムコールsetsid()、そしてinclude/string.hで実装されているすべての文字列操作関数です。

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

ファイル	サイズ	最終更新時刻(GMT)	説明
Makefile	2602バイト	1991-12-02 03:16:05	
exit.c	198バイト	1991-10-02 14:16:29	
close.ct	131バイト	1991-10-02 14:16:29	
ype.c	1202バイト	1991-10-02 14:16:29	
dup.c	127バイト	1991-10-02 14:16:29	
errno.c	73バイト	1991-10-02 14:16:29	
execve.c	170バイト	1991-10-02 14:16:29	
malloc.c	7469バイト	1991-12-02 03:15:20	
open.c	389 バイト	1991-10-02 14:16:29	
setid.c	128 バイト	1991-10-02 14:16:29	
文字列.c	177 バイト	1991-10-02 14:16:29	
wait.c	253 バイト	1991-10-02 14:16:29	
を書く	160バイト	1991-10-02 14:16:29	

→ とが
できま

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ライブラリにビルトされ、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、含まれている関数は少なく、非常にシンプルです。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。

```
ar -rc libmine.a a.o b.o c.o d.o
```

関数モジュールdup.oをこのライブラリファイルに追加するには、次のコマ

ンドを実行します。

1. _exit.c

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。

2. コードアノテーション

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> で定義された関数が標準ライブラリです。標準ライブラリで定義された関数が定義されません。
// システム割り込みを直接呼び出す int 0x80, 関数番号 NR_exit。
// パラメータ: exit_code - 終了コード。
// 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、さらに重要なことは
// 誤った警告を避けるために、このキーワードを使用します。
10 volatile void _exit(int exit_code)
11 {
12 // %0 - eax(%NR_exit)$0x80":: ebx(%NR_exit),%b" (exit_code));
13 }
14

```

15.1.3 情報

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。

2. close.c

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。

2. コードアノテーション

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標準のヘッダーファイルです。様々なシンボル定数や型が定義されており
// 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 __syscall0() も含まれます。
7#define __LIBRARY
8#include <unistd.h> (英語)
9
// ファイル機能を閉じる。
// 以下のマクロは、関数プロトタイプ: int close(int fd)に対応しています。を直接呼び出します
//
// システムのint 0x80、パラメータはNR_closeです。fdはファイルディスクリプターです。
10 __syscall1(int, close, int, fd)
11

```

3. ctype.c

1. 機能性

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に提供するするために使用されます。

2. コードアノテーション

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いくつかのマクロを定義しています。 7
#include <ctype.h>
8
9 char __ctmp; // ctype.hファイルで文字を変換するマクロのためのtmp変数です。

```

1 機能性

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。

2. コードアノテーション

プログラム 15-4 linux/lib/dup.c

```
1 /*  
2 * linux/lib/dup.c
```

```

3 /*
4 * (C) 1991 Linus Torvalds
5 */
6
// <unistd.h> Linux標準のヘッダーファイルです。様々なシンボル定数や型が定義されており
// 様々な関数が宣言されています。LIBRARY'が定義されている場合は、システムコール番
// 号とインラインアセンブリ _syscall10() も含まれます。
7 #define _ライブラリー
8 #include <unistd.h> (英語)
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 int dup(int fd)に対応しています。これは、直接
// システムの int 0x80、パラメータはNR_dupです。fdはファイルディスクリプターです。
10 syscall1(int, dup, int, fd)
11

```

5. errno.c

1. 機能性

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。

2. コードアノテーション

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

```

6. execve.c

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。

2. コードアノテーション

プログラム 15-6 linux/lib/execve.c

```

1 /*

```

```

2 * linux/lib/execve.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <unistd.h> Linux標準のヘッダーファイルです。様々なシンボル定数や型が定義されており
// 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 _syscall0() も含まれます。
7 #define _LIBRARY
8 #include <unistd.h> (英語)
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 - 実行可能なファイル名 argv - コマンドライン引数ポインタの配列
// envp - 環境変数のポインターの配列。システムのint 0x80を直接呼び出します。
// パラメータはNR_execveです。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

```

7. malloc.c

1. 機能性

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()に改名されています。

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テ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stackoverflow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。このプロセステ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()が呼ばれている。kernel/sys.cプログラムの228行目を参照のこと。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ストを使用してメモリブロックが割り当てられる。また、基本的な

構造体の概要を図15-1に示します。この関数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、4096バイトである1メモリページです。

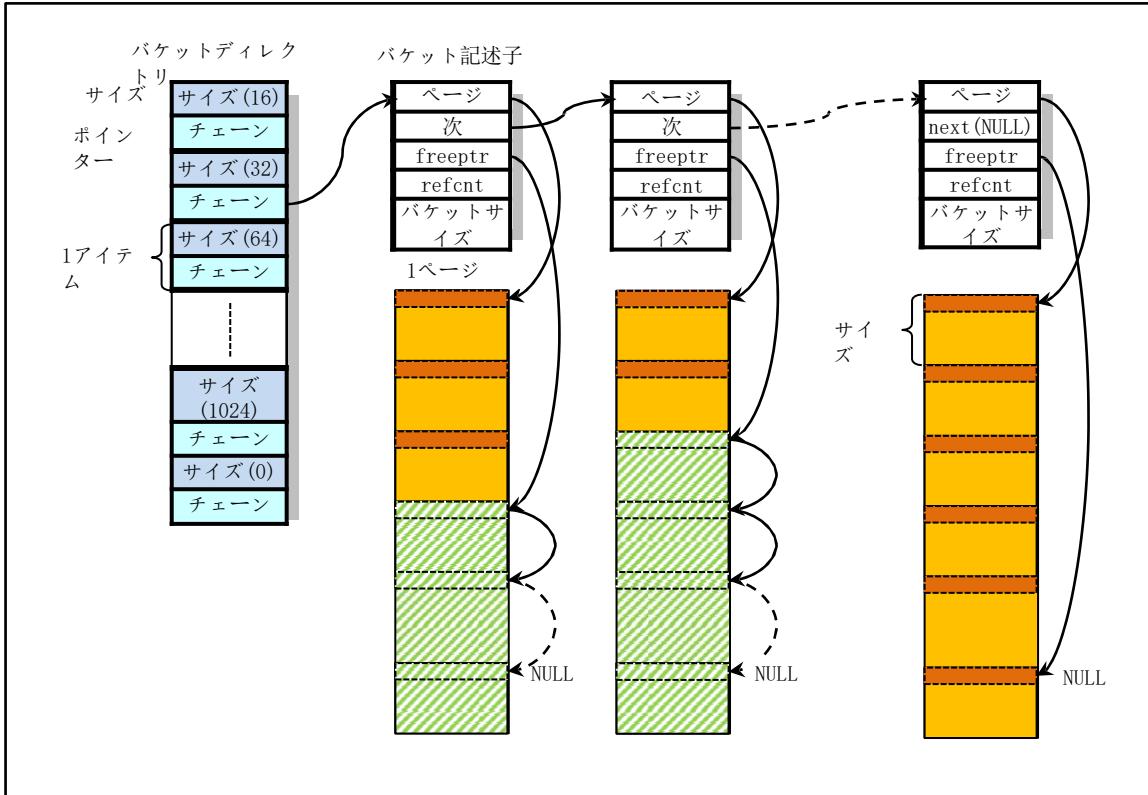


図15-1 バケットの原理を利用したメモリの割り当て管理

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。

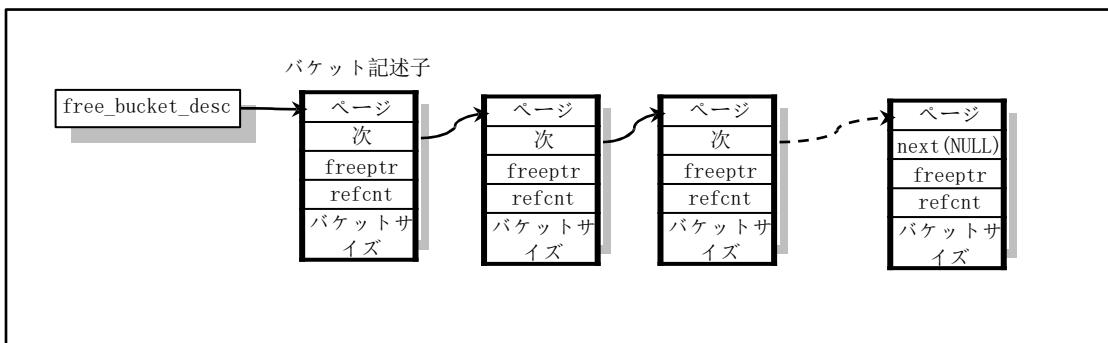


図15-2 フリーバケット記述子のチェーンテーブル構造

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になります。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、プログラムは再びページを申請し、新しいフリーバケット

図のように、その上に記述子のリストが表示されます。

malloc()関数の基本的な手順は以下の通りです。

1. まず、ディレクトリを検索し、要求されたメモリブロックのサイズと一致するディレクトリエントリに対応する記述子リストを探します。ディレクトリエントリのメモリサイズが要求されたバイトサイズよりも大きい場合、対応するディレクトリエントリが見つかります。ディレクトリ全体を検索しても適切なディレクトリエントリが見つからない場合は、ユーザーが要求したメモリブロックが大きすぎるということになります。
2. ディレクトリエントリに対応するディスクリプタリストの中から、空き領域のあるディスクリプタを探す。ディスクリプタの空きメモリポインタfreeptrがNULLでなければ、対応するディスクリプタが見つかったことを意味する。空き領域のあるディスクリプターが見つからない場合は、新しいディスクリプターを作成する必要があります。新しいディスクリプターを作成する手順は以下の通りです。
 - a. アイドルディスクリプタリストのヘッダポインタがまだNULLの場合は、malloc()関数が初めて呼び出されたか、空のバケットディスクリプターがすべて使い切られたことを意味します。この場合、関数 `init_bucket_desc()` を使用してアイドルディスクリプターのリストを作成する必要があります。
 - b. 次に、アイドルディスクリプタチェーンのヘッダからディスクリプタを取得し、ディスクリプタを初期化して、オブジェクト参照カウントを0にし、オブジェクトサイズをディレクトリエントリと同じにして、メモリページを申請し、ディスクリプタのページポインタがメモリページを指し、ディスクリプタのフリーメモリポインタもページの先頭を指すようにする。
 - c. このディレクトリエントリで使用されているオブジェクトのサイズに応じて、メモリページを初期化し、すべてのオブジェクトのリンクリストを確立する。すなわち、各オブジェクトの先頭には次のオブジェクトへのポインタを格納し、最後のオブジェクトには先頭にNULLを格納する。
 - d. そして、その記述子を、対応するディレクトリエントリの記述子リストの先頭に挿入します。
3. 記述子のフリーメモリポインタfreeptrをユーザに返すメモリポインタにコピーし、記述子に対応するメモリページの次のフリーオブジェクトの位置を指すようにfreeptrを調整し、記述子の参照カウントを1つ増加させる。

`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リストに格納することができる。

15.7.2 コードアノテーション

プログラム 15-7 linux/lib/malloc.c

```

1 /*
2 * malloc.c — Linux用の汎用カーネルメモリマネージャーです。
3 *
4 * 執筆: Theodore Ts'o (tytso@mit.edu), 11/29/91
5 *
6 * このルーチンは、可能な限り高速になるように書かれています。

```

```

7 /*は割込みレベルからの呼び出しが可能です。
8 *
9 * 制限: このルーチンを使用して割り当てることができるメモリの最大サイズ
10 *は、Linuxの1ページのサイズである4kです。
11 *
12 * 一般的なゲームプランでは、各ページ（バケットと呼ばれる）には、以下のもの
13 *だけが格納されます。
14 * 特定のサイズのオブジェクト。 ページ上のオブジェクトがすべて解放されたと
15 *き。
16 * そのページは一般のフリーピールに戻すことができます。 malloc()が
17 *が呼び出されると、自分の要求を満たす最小のバケットサイズを探します。
18 * リクエストに応じて、そのバケットピールからメモリを確保することができます
19 *。
20 *
21 * 各バケットは、その制御ブロックとして、バケット記述子を持ちます。
22 * そのページで何個のオブジェクトが使われているかを追跡し、フリーリスト
23 * バケット記述子は、バケット自体と同様に、そのページの
24 * get_free_page()でリクエストされたページに保存されます。 ただし、バケット
25 *とは異なります。
26 * バケットディスクリプターページに充てられたページは、決して元には戻されま
27 *せん。
28 * 幸いなことに、1つのシステムに必要なバケットの数は1つか2つです。
29 * ページは256個のバケット記述子を保持することができますため、記述子ページは
30 *は1メガバイト分のバケットページに相当します。カーネルが
31 *これをメモリを確保しているといううちは何か悪いことをしているのかも
32 *しませんね。しかし、get_free_page()とfree_page()が割り
33 *さされるとどうではなくなるかもしれません。その場合は、malloc()を修
34 *正して、いくつかの未使用的のページを「事前に確保」しておき、以下の
35 *Note: 正して、いくつかの未使用的のページを「事前に確保」しておき、以下の
36 *ように場合にそれらのページを安全に利用できるようにする必要があります
37 *ます。
38 * は、割り込みルーチンから呼び出されます。
39 *
40 *
41 *
42 */
43/*<linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプ
44ロトタイプ定義が含まれています。
45/* カーネルの使用する機能
46/*<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
47// 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
    // 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。
48 #include <linux/kernel.h> (日本語)
49 #include <linux/mm.h>
50 #include <asm/system.h>
51
    // バケットディスクリプターモジュール構造
52 struct bucket_desc { /* 16 バイト */.

```

```

53     ポイド          *ページ      // メモリページポインタ。
54     struct bucket_desc *次のページ // 次の記述子のポインタです。
55     ポイド          *freeptr;   // 解放されたメモリへのポインタです。
56     符号付きショート    refcnt;    // リファラנסカウント
57     符号付きショート    bucket_size; // バケットのサイズです。
58 };
59
// バケットディスクリプターのディレクトリ構造
60構造体 bucket_dir { /* 8 バイト */
61     int             の大きさになります。 // このバケットのサイズ（バイト単位）です。
62     struct bucket_desc *chain // バケットディスクリプタのリストポインタ。
63 };
64 /* 以下は、最初のバケットへのポインタを格納する場所です。
65 * 指定されたサイズの記述子です。
66 *
67 * Linux カーネルが多くのオブジェクトを割り当てていることが判明した場合は
68 * 特定のサイズがある場合は、そのサイズをこのリストに追加することができます。
69 * その方が、より効率的にメモリを割り当てることができるからです。
70 * ただし、各サイズごとにページを割かなければならぬので
71 * このリストにあるように、ここではある程度の節制が必要です。
72 *
73 * このリストは順番に並べる必要があることに注意してください。
74 *
75 * */
76 */
77// バケットのディレクトリリストです。
78     {... *) 0}, // 16バイトのメモリブロック
79     { 16, (struct bucket_desc) // です。
80     { 32, (struct bucket_desc) * 0}, // 32バイトのメモリブロック
81     { 64, (struct bucket_desc) * 0},
82     { 128, (struct bucket_desc) * 0},
83     { 256, (struct bucket_desc) * 0},
84     { 512, (struct bucket_desc) * 0},
85     { 1024, (struct bucket_desc) * 0},
86     { 2048, (struct bucket_desc) * 0}, // 4096バイトのメモリブロック。
87     { 4096, (struct bucket_desc) * 0},
88 * 空いているバケットディスクリプターブロックのリストの終がここであります。
89 */
90 struct bucket_desc *free_bucket_desc = (struct bucket_desc) 0;
91 */
92 /*
93 * このルーチンは、バケットの説明ページを初期化します。
94 */
95 /**
96 * このルーチンは、バケットの説明ページを初期化します。
97 static inline void init_bucket_desc()
98 {
99     struct bucket_desc *bdesc, *first;
100    int i;

```

101

// 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します。そして、数を計算します。

メモリのページに格納できるバケットディスクリプターの//を設定して、一方通行の
// そのリンクポインタ

102 first = bdesc = (struct bucket_desc *) get_free_page();

103 if (!bdesc)

104 panic ("Out of memory in init_bucket_desc()")が発生します。105 for (i = PAGE_SIZE/sizeof(struct bucket_desc); i > 1; i--) {

106 bdesc->next = bdesc+1;

107 bdesc++です。

108 }

109 /*

110 * これは最後に行われるもので、万が一のレースコンディションを避けるためです。

111 * get_free_page()がスリープして、このルーチンが再び呼ばれる....

112 */

// 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。

113 bdesc->next = free_bucket_desc;114 } free_bucket_desc = first;

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。

// 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

117 ます。*malloc(unsigned int len)

ド

118 {

119 構造体 bucket_dir *bdir;120 struct bucket_desc *bdesc;

121 ポイド *retval;

122

123 /*

124 * 最初に検索するのは bucket_dir で適切なバケットの変更を見つけることができます。

125 // を適用するのに適したバケット記述子リストを、バケットディレクトリで検索します。

126 // メモリブロックのサイズ。ディレクトリエントリのバケットサイズが、メモリブロックの

// 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が見つかる

127 for (bdir = bucket_dir; bdir->size; bdir++)

128 if (bdir->size >= len)

129 break;

// ディレクトリ全体を検索しても該当するディレクトリエントリが見つからない場合は

// サイズを超えている場合は、要求されたメモリブロックサイズが大きすぎて、アロケーション

// プログラムの限界(1ページまで)。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

130 ます。 if (!bdir->size) {

131 printk ("malloc called with impossibly large argument (%d)\n")

.

132 len)を使用しています。

133 panic ("malloc: bad arg").

134 }

135 /*

136 * 空き容量のあるバケット記述子を検索します。

137 */

138 // 対応するQケイド/ディレクトリをシミュレーションリストを検索して、バケットを見つける

// 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

```

// が空でなければ、対応するバケット記述子が見つかったことを示します。
139     for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)
140         if (bdesc->freeptr)
141             ブレークします。
142     /*
143     * 空き容量のあるバケットが見つからなかった場合には
144     * 新しいものを割り当てます。
145     */
146     if (!bdesc) {
147         char *cp;
148         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
// ページを申請し、その上にアイドル記述子リストを構築して初期化する。 free_bucket_desc
// は最初の空きバケットディスクリプターを指します。
150         if (! free\_bucket\_desc)
151             init\_bucket\_desc()を行います。

// 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。
// 参照数は0に等しい；バケットサイズは対応するバケットのサイズに等しい
// ディレクトリ；メモリページを適用し、ページポインタにページを指定させる；フリーメモリポイ
152     // bdesc = free\_bucket\_desc;
153     // free\_bucket\_desc = bdesc->next;
154     bdesc->refcnt = 0;
155     bdesc->bucket_size = bdir->size;
156     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。
// 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で指定されたバケットサイズで割る
// 最後のオブジェクトへの各オフセットの最初の4字节が次のオブジェクトへのポインタに設定されま
157     // す。ポインタは if (!cp)
158         panic ("Out of memory in kernel malloc()")が
159         // 発生します。
160         /* 解放されたオブジェクトのチェーンを設定 */
161         for (i=PAGE\_SIZE/bdir->size; i > 1; i--) {...}
162             *((char **) cp) = cp + bdir->size;
163             cp += bdir->size;
164     // そして、バケット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
165     // プタの
166     // 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
167     // チェイン
168     // バケットの記述子を指すbdir->chain = bdesc;
169     // チェインヘッダbdir->chain = bdesc;
170     }

// 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。
// ディスクリプタの対応するページが1つ増加します。最後に割込みを有効にする
// retval = (void *) bdesc->freeptr;
171

```

```

169     bdesc->freeptr = *((void **) retval);
170     bdesc->refcnt++;
171     sti(); /* OK, we're safe again */ (英
172     語)
173 } return(retval)になります。
174
175 /*
176 * ここでは、無料のルーチンを紹介します。
177 * を解放している場合、free_s()はその情報を利用して高速化します。
178 * バケットディスクリプターを検索します。
179 */
180 * "free(x)" が "free_s(x, 0)" になるように、マクロを #define します。
181 */
182 ///////////////////////////////////////////////////////////////////
183 void free_s(void *obj, int size)
184 {
185     void *page;
186     構造体 bucket_dir      *bdir;
187     構造体 bucket_desc     *bdesc, *prev;
188
189     /* このオブジェクトがどのページに存在するかを計算する */
190     page = (void *) ((unsigned long) obj & 0xfffff000);
191     /* そのページを探すためにバケットを検索する */。
192     for (bdir = bucket_dir; bdir->size; bdir++) {
193         prev = 0;
194         /* サイズがゼロの場合、この条件式は常に偽となります。
195         if (bdir->size < size)
196             を続けています。
197         // 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
198         // ような
199         // ディスクリプタのページポインタが'page'と等しい場合、対応するディスクリプタが
200         // が見つかったので、ラベル「found」にジャンプします。ディスクリプタに対応するページがない
201         // 場合は
202         // そうすると、ディスクリプタのポインター'prev'が指し示されます。
203         // 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が、そのディレクトリエ
204         // ントを見つけられなかった場合(bdesc->page == page)
205         // 指定されたページでは、エラーメッセージが表示され、コンピュータがクラッ
206         // シュします。for( bdesc = bdir->chain; bdesc; bdesc =
207             bdesc->next != bdesc;
208         }
209     }
210     panic("Bad address passed to kernel free_s()")
211 件見つ が発生しました。
212 かりに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
213 。 // 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、オブジェクトの参照カ
214 ウントが
215 そのディクク();アメモリブロックをアドレスを避けてます。
216     *(obj ***)obj) = bdesc->freeptr;
217     bdesc->freeptr = obj;
218     bdesc->refcnt--;
219
220 // 参照カウントが0になった場合、対応するメモリページを解放して
221 // バケットの記述子。

```

```

208     if (bdesc->refcnt == 0) {...  

209         /*  

210          * prevがまだ正確であることを確認する必要があります。  

211          * 誰かが私たちの邪魔をしたら、そうではないかもしませ  

212          ん。....  

213 // '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  

// 現在のディスクリプターが再検索される。 if ((prev &&  

213     (prev->next != bdesc)) ||  

214     (!prev && (bdir->chain != bdesc)))  

215     for (prev = bdir->chain; prev; prev = prev->next)  

216     if (prev->next == bdesc)  

217         ブレークします。  

218 // 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。  

219 // 鎖です。prev==NULL の場合は、現在のディスクリプタが最初のディスクリプタであることを意味  

220 // します。  

221 // ディレクトリ・エントリ、つまり、ディレクトリ・エントリ内のチェーンが直接、現在の  

222 // それ以外の場合は、リンクリストに問題があることを示します。  

223 // 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  

224 // 次の記述子を指す「チェーン」 if (prev)  

225     prev->next = bdesc->next;  

226 else {  

227     if (bdir->chain != bdesc)  

228         panic("malloc bucket chains corrupted");  

229     bdir->chain = bdesc->next;  

230 }  

231     // 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、ディスクリプターが  

232 // free_page((unsigned long) bdesc->page).....アイドルディ  

233 // スクリプタリストの先頭に挿入される。  

234     bdesc->next = free_bucket_desc;  

235     free_bucket_desc = bdesc;  

236 }  

237     sti(); // 割り込みを有効にして戻る return;  

238 }
239 }
```

8. open.c

1. 機能性

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。¹⁰³²

15.8.2 コードアノテーション

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標準のヘッダーファイルです。様々なシンボル定数や型が定義されており
// 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 _syscall1() も含まれます。
// <stdarg.h> 標準パラメータのヘッダーファイルです。変数パラメータのリストを以下の形式で定義し
// ます。 のマクロを紹介します。主に、vsprintf、vprintf、vfprintf関数の1つの型 (va_list) と3つ
// のマクロ (va_start、va_arg、va_end) について説明しています。
7 #define LIBRARY
8 #include <unistd.h>
9 #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, ...
// 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。
11 int open(const char * filename, int flag, ...)
12 { ... レジスタ int res;
13     va_list arg;
14
15     // マクロ関数 va_start() を使用して、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、ファイルを開く。
// %0 - eax (返されたディスクリプターまたはエラーコード); %1 - eax (システムコール関数
NR_open)。
16     // %2 - ebx (ファイル名); %3 - ecx (ファイルオープンフラグ); %4 - edx (ファイルモードの
17     追従). va_start(arg, flag);
18     asm ("int $0x80"
19          :"=a"(res)
20          :"d"(aNArg(arg), int"(filename)"(filename), "r"(flag)),
// システム割り込みコールが0以上の値を返した場合は、次のことを示します。
// それがファイルディスクリプターであれば、直接返されます。それ以外の場合は、戻り値が
// 0の場合は、エラーコードです。そこで、エラーコードを設定し、-1を返
21     if (res>0)
22         return res;
23     errno = -res;
24     return -1;
25 }
26

```

9. setid.c

1. 機能性

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。

2. コードアノテーション

プログラム 15-9 linux/lib/setsid.c

```

1 /*
2 * linux/lib/setsid.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6 // unistd.h は Linux標準で定義されています。標準的な機能が定義されています。
// 記号とインラインアセンブリ _syscall10() も含まれます。
7 #define LIBRARY
8 #include <unistd.h>
9
10 //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。
11 //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。
12 //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。
13 _syscall10(pid_t, setsid)
14

```

10. 文字列.c

1. 機能性

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ヘッダーファイルをインクルードしています。`include/string.h`ヘッダーファイルの前の説明を参照してください。

2. コードアノテーション

プログラム 15-10 linux/lib/string.c

```

1 /*
2 * linux/lib/string.c

```

```

3 /*
4 * (C) 1991 Linus Torvalds
5 */
6
7 #ifndef GNUC 8 #エ
8 ラー gcc が欲しい!
9 #endif
10
11 #define extern
12 #define inline
13 #define LIBRARY
14 #include <string.h>
15

```

11. wait.c

1. 機能性

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）。

2. コードアナリティクス

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 // 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
7 // <unistd.h> Linux標準のシグナルアイドルでも含まれます。
8 // <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
9 // 定数シンボル。 7
#define LIBRARY
8 #include <unistd.h>
9 #include <sys/wait.h>

```

```

10 /////
10 // プロセスが終了するのを待ちます。
11 // pid_t waitpid(pid_t pid, int * wait_stat, int options)
11 // Parameters: pid - 終了を待つプロセスのプロセスID、またはその他の特定の
11 // wait_stat - 状態情報の保存に使用されます。
11 // オプション - WNOHANG or WUNTRACED or 0.
12 11 syscall3(pid_t, waitpid, pid_t, pid, int *, wait_stat, int, options)
12 /////
12 // wait() システムコールです。waitpid()関数を直接呼び出します。
13 11 pid_t wait(int * wait_stat)
13 11 {。
14 11     waitpid(-1, wait_stat, 0) を返しま
14 11     す。
15 11
16 11
17 11

```

12. write.c

1. 機能性

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。

2. コードアノテーション

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標準のヘッダーファイルです。様々なシンボル定数や型が定義されており
// 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 _syscall0() も含まれます。
7 #define LIBRARY
8 #include <unistd.h>
9
/////
10 // ファイルの書き込みを行います。
10 // このマクロは、以下の関数に対応しています。
10 // パラメータ: fd - ファイルディスクリプター; buf - 書き込みバッファポインタ; count - 書き
10 // 込みバイト数。
10 // 戻り値: 成功したときに書き戻されたバイト数(0は0バイトを意味する); -1の場合は
10 // エラー時に返され、エラーコードが設定されます。
10 syscall3(int, write, int, fd, const char *, buf, off_t, count)
11

```

15.13 まとめ

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 カーネル（ツール）の構築

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アセンブラーを使って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。

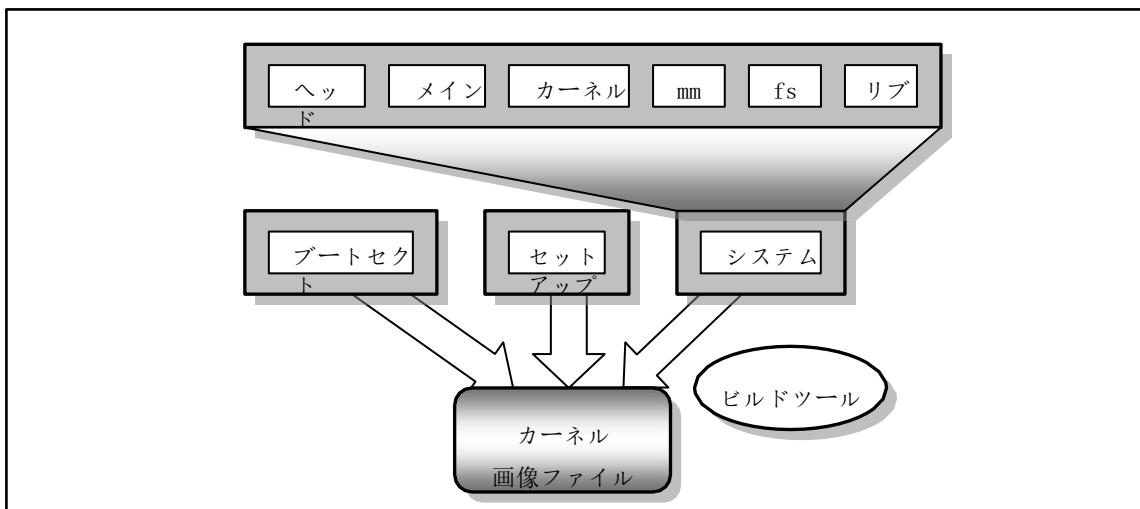


図16-1 カーネルのコンパイル・リンク・結合構造

1. build.c

1. 機能性

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、オプションのルートfsデバイス名ROOT_DEV、オプションのスワップデバイス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(プログラマリストの説明を参照)。また、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とリンクされており、GNU a.outの実行ファイル形式は使っていません。の主な仕事は

ビルドプログラムは、bootsectとsetupのMINIX実行ファイルのヘッダ情報を削除し、systemモジュールのa.outのヘッダ情報を削除し、それらのコード部分とデータ部分のみを残し、それらを順に結合して、「Image」という名前のファイルに順次書き込んでいきます。

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。

- 第1セクターにはブートセクトコードが格納されており、ちょうど512バイトの長さになります。
- 第2セクタから4セクタ（2～5セクタ）には、セットアップコードが格納され、サイズは4セクタ以内となります。
- システムモジュールは第6セクターから格納され、その長さはbuild.cの37行目で定義されたサイズ（128KB）を超えない。

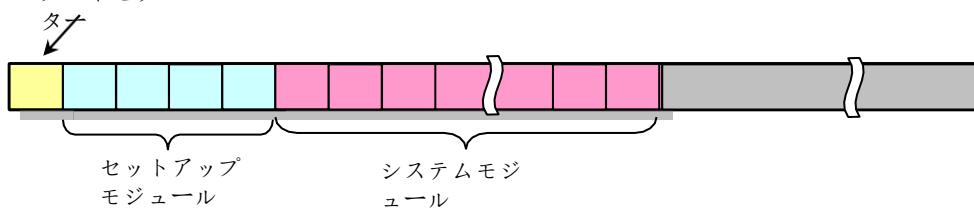


図 16-2 ディスク上の Linux カーネルのフォーマット

16.1.2 コードアノテーション

プログラム 16-1 linux/tools/build.c

```

1 /*
2 * linux/tools/build.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * このファイルは、3つの異なるファイルからディスクイメージを構築します。
9 *
10 * - bootsect: 最大510バイトの8086マシンコード、残りをロードする
11 * - setup: 最大4セクターの8086マシンコード、システムパルムの設定

```

```

12 * - system: 80386 実際のシステムのコード
13 *
14 * すべてのファイルが正しいタイプであるかどうかのチェックを行います
15 .
16 * 結果を標準出力に書き出すだけで、ヘッダを取り除き、パディングして
17 * 適切な量です。また、いくつかのシステムデータを標準エラーに書き込
18 * みます。
19 */
20 /*
21 * tytsoによる、ルートデバイスの指定を可能にする変更
22 * swap-device の仕様を追加しました。Linux 20.12.91
23 #include <stdio.h>      fprintf */
24 #include <string.h>      fprintf
25 #include <stdlib.h>      /* 出口を含む
26 #include <sys/types.h>    /* unistd.h はこれを必要とします */
27 #include <sys/stat.h>    // ファイル状態情報構造
28 #include <linux/fs.h>
29 #include <unistd.h>      /* 読み込み/書き込みを含む */
30 #include <fcntl.h>        // ファイル操作モードの定数
31
32
33
34 #define MINIX_HEADER_32          // MINIXのオブジェクトファイルのヘッダサイズは32
35 #define GCC_HEADER_1024          // バイトです。
36                                     // GCCのヘッダー情報のサイズは1024バイトです。
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
38                                     // す。
39                                     // 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
40                                     // バイス
41                                     // 番号0x0306）となっています。これは、リーナス氏がLinuxを開発した際に、最初のハードディスク
42 #define DEFAULT_MAJOR_ROOT_3     // メジャーデバイス番号-3、ハードディスク。
43 #define MINIX_MINOR_ROOT_0       // MINIXのルートデバイス名として、2番目のハードディスク番号を1番目のディスクの1番目の
44                                     // ディスクとして使用します。
45                                     // イション。
46 #define DEFAULT_MAJOR_SWAP_0    // スワップデバイスの
47 #define DEFAULT_MINOR_SWAP_0    // 番号です。
48
49 /* セットアップの最大セクタ数: 変更しない限り、変更しないでください。
50 */
51 // セットアップの最大サイズは4セクタ (2KB) です
52 .
53 #define STRINGIFY(x) #x
54
55 void die(char * str)
56 {
57     printf(stderr, "%s\n", str);
58     exit(1);
59 }
60
61 // プログラムの使用状況を表示して終了します。
62 void usage(void)
63 {
64     die("Usage: build bootsect setup system [rootdev] [> image]");
65 }

```

```

60 }
61
62 ///////////////////////////////////////////////////////////////////
63 // メインプログラムが始まる...
64 // 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
65 // デバイス番号とスワップデバイス番号を入力し、bootsect、setup、および
66 // それぞれのシステムモジュールファイルを標準出力に書き込んでいます。
67 // Imageファイルにリダイレクトされます。.
68
69 int main(int argc, char ** argv)
70 {
71     int i, c, id;
72     char buf[1024];
73     char major_root, minor_root;
74     char major_swap, minor_swap;
75     struct stat_sb;
76
77     // (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
78     // パラメーターの数に応じて設定されます。ビルドプログラムでは、4~6
79     // パラメーターを表示します。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は
80     // プログラムの使用状況を表示して終了します。
81     // 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
82     // "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
83     // 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。
84     // ルートデバイスが現在のデバイスであることを示すメジャーデバイス番号とマイナーデバイス番号
85     // を0で設定する旨を使用しています。
86
87     if ((argc < 4) || (argc > 6))
88         usage();
89     if (argc > 4) {
90         if (strcmp(argv[4], "FLOPPY")) {
91             if (stat(argv[4], &sb)) {
92                 perror(argv[4])のようになります。
93                 die("Couldn't stat root device.");
94             }
95             major_root = MAJOR(sb.st_rdev); // デバイス番号の取得
96             minor_root = MINOR(sb.st_rdev);
97         } else {
98             major_root = 0;
99             minor_root = 0;
100         }
101     }
102
103     // 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と同じにする
104
105     // デフォルトのルートデバイス番号。
106     } else {
107         major_root = DEFAULT_MAJOR_ROOT;
108         minor_root = DEFAULT_MINOR_ROOT;
109     }
110
111     // 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
112     // 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
113     // 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番号として扱われます。もし、
114     // 最後のパラメータが "NONE" の場合は、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が0となり、次のこ
115     // とを示します(argc == 6) {
116         if (strcmp(argv[5], "NONE")) {
117             if (stat(argv[5], &sb)) {
118                 perror(argv[5])のよ
119                 うになります
120             }
121         }
122     }

```

```

92                     die ("Couldn't stat root device.");
93             }     となります。
94             major_swap = MAJOR(sb.st_rdev);
95             minor_swap = MINOR(sb.st_rdev)
96         } else {となります。
97             major_swap = 0;
98             minor_swap = 0;
99         }
// パラメータが6個ではなく5個の場合は、コマンドにスワップデバイス名がないことを意味する
// の行になります。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
// デフォルトであるスワップ
100 // デバイス番号{
101             major_swap = DEFAULT_MAJOR_SWAP;
102             minor_swap = DEFAULT_MINOR_SWAP;
103         }

// 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。
// 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リダイレクトされます。
// をファイル「Image」に出力するので、保存したカーネルコードとデータを使って
104 // カーネルイメージファイル。
105     fprintf(stderr, "Root device is (%d, %d)\n", major_root, minor_root);
106     fprintf(stderr, "Swap device is (%d, %d)\n", major_swap, minor_swap);
107     if ((major_root != 2) && (major_root != 3) &&
108         (major_root != 0)) {
109         fprintf(stderr, "Illegal root device (major = %d)\n",
110             die ("Bad root device --- major #")
111        となります。
112     }    も (major_swap && major_swap != 3) {
113     }    し fprintf(stderr, "Illegal swap device (major = %d)\n",
114             major_swap).
115     die ("Bad root device --- major #")
116    となります。
117 // (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。
// コピー処理を行います。まず1KBのバッファを初期化し、次にパラメータ
// 1（ブートセクト）をリードオンリーモードで読み込み、32バイトのMINIX実行ヘッダー構造体（参
照
118 // 以下のリスト）をバッファbufに格納します。
119     for (i=0; i<sizeof buf; i++) buf[i]=0;
120     if ((id=open(argv[1], O_RDONLY, 0))<0)
121         die ("Unable to open 'boot'")となります。
122     if (read(id, buf, MINIX HEADER) != MINIX HEADER)
123         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コードとデータがファイルから読み込まれます。このとき、値
// 0x0301 - MINIXヘッダマジック値 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。
// 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
// 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、ヒープが
// a_bss フィールド (long) が0の場合、実行ポイントのa_entry フィールド (long) が0かどうかを確認し
// ます。
1043
// シンボルテーブルサイズフィールド a_sym が0であるかどうかを確認します。

```

```

122     if (((long *) buf)[0] != 0x04100301)
123         die("Non-Minix header of 'boot'");
124     if (((long *) buf)[1] != MINIX_HEADER)
125         die("Non-Minix header of 'boot'");
126     if (((long *) buf)[3] != 0)
127         die("Illegal data segment in 'boot'");
128     もし (((long *) buf)[4] != 0)
129         die("Illegal bss in 'boot'");
130         となります。
131     もし (((long *) buf)[5] != 0)
132     if (((long *) buf)[7] != 0)
133         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
// が正しい場合、読み取りバイト数は512バイトになります。ブートセクタファイルには
// 1つのセクターのブートセクタコードとデータ、そして最後の2バイトはブート可能な
// 旗 0xAA55です。
134     i = read(id, buf, sizeof buf) となります。
135     fprintf(stderr, "Boot sector %d bytes. in", i)
136         となります。
137     if (i != 512) die("Boot Block must be exactly 512 bytes");
138     if ((*unsigned short *) (buf + 510)) != 0xAA55
139         die("Boot Block hasn't got boot flag (0xAA55)") となります。
// 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に格納します。
// buf[506] = (char) minor_swap;
140     buf[507] = (char) major_swap;
141     buf[508] = (char) minor_root;
142     buf[509] = (char) major_root
143     。
// 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。
// linux/Makefileでは、ビルドプログラムが標準出力をカーネルイメージにリダイレクトする
// ファイル名 ">" インジケータを使用したイメージなので、ブートセクタのコードとデータが
// Imageの最初の512バイト。
144     i = write(1, buf, 512) とする。
145     if (i != 512)
146         die("Write call failed")
147         となります。
148 // (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、その内容を
32バイトのMINIX実行ファイルヘッダの//がバッファbufに読み込まれる。次に
// MINIXヘッダー構造では、セットアップが有効なMINIX実行ファイルであるかどうかがチェックされ
ます。
// その場合は、ヘッダー情報に対して一連のチェックが行われます。
149 // 上記のよ (fd=open(argv[2], O_RDONLY, 0))<0)
150         die("Unable to open 'setup'");
151     もし (read(id, buf, MINIX_HEADER) != MINIX_HEADER)
152         die("Unable to read header of 'setup'");
153     if (((long *) buf)[0] != 0x04100301)
154         die("Non-Minix header of 'setup'");
155     if (((long *) buf)[1] != MINIX_HEADER)          // ヘッダーサイズ(32バ
156         die("Non-Minix header of 'setup'");        ト)
157     if (((long *) buf)[3] != 0)           // データサイズフィールド
158         die("Illegal data segment in 'setup'"); a_data
159     if (((long *) buf)[4] != 0)           // a_bssフィー
ルド。

```

```

160         die("Illegal bss in 'setup'");
161    も (((long *) buf)[5] != 0) // a_entry point.
162    し die("Non-Minix header of 'setup'");
163     if (((long *) buf)[7] != 0)
164         die("Illegal symbol table in 'setup'");
165 // 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。
166 // 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
167 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
168 // 書き込み操作のコードとデータのサイズが、(SETUP_SECTS
169 // * 512) バイトでなければ、セットアップセットが占有するセクタ数を再編集しなければなりません。
170 ビルドプログラム、ブートセクトプログラム、セットアッププログラムで // カーネルを再コンパイル
171 してください。すべてが問題なければ
172 // セットアップの実際の長さの値が表示されます。
173     for (i=0; if (write(id,buf,sizeof(buf))>0 ; i+=c )
174             die("Write call failed")と
175             なります。
176     開じる(i >id)であるsectors*512) // セットアップファ
177     イルを閉じる。
178     die("Setup exceeds " STRINGIFY(SETUP_SECTS)
179     " sectors - rewrite build/boot/setup")
180     fprintf(stderr, "Setup is %d bytes.\n", i);
181 // バッファbufをクリアした後、実際の書き込み設定長と
182 // (setup_sects * 512). セットアップサイズがサイズ(4 * 512バイト)よりも小さい場合は、ファイル
183 // NULL文字で4 *512バイトになるように設定。
184     for (c=0 ; c<sizeof(buf) ; c++)
185         buf[c] = '\0';
186     while (i< SETUP_SECTS*512) {...}
187         c = SETUP_SECTS*512-i;
188         if (c > sizeof(buf))
189             c = sizeof(buf)となります。
190         if (write(1,buf,c) != c)
191             die("Write call failed")となります。
192         i += c.
193     }
194
// (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
// そのため、GNU a.out オブジェクトファイルフォーマットを採用しています。
// 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
// 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セクションを超える場合(
128KB
// バイト)になると、エラーメッセージを表示して終了します。エラーがない場合は0を返し、次のこ
とを示します。
// 正常終了。
195     if ((id=open(argv[3],O_RDONLY,0))<0)
196         die("Unable to open 'system'")となります。
197     if (read(id,buf,GCC_HEADER) != GCC_HEADER)
198         die("Unable to read header of 'system'")となります。
199     if (((long *) buf)[5] != 0) // エントリーロケーションは0であるべき。
200         die("Non-GCC header of 'system'")となります。
201     for (i=0 ; (c=read(id,buf,sizeof(buf))>0 ; i+=c )
202             if (write(1,buf,c)!=c)
203                 die("Write call failed")となります。
204     close(id)です。
205     fprintf(stderr, "System is %d bytes.\n", i);

```

```

195     if (i > SYS_SIZE*16)
196         die ("System is too big");
197     return(0);
198 }
199

```

3. インフォメーション

1. MINIXモジュールと実行ファイルのヘッダデータ構造

MINIXのコンパイラとリンク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。

構造体exec {。

符号付き char a_magic[2];	// マジックナンバー、0x0301のはずです。
符号付き char a_flags; 符号付	// Flags (下記参照)。
号付き char a_cpu; 符号付	// マシンのCPU識別子です。
き char a_hdrlen; 符号付	// 予約されたヘッダーサイズ、32または48バイ
き char a_unused; 符号付	ト。
き short a_version;	// Reserved.
ロン a_text;a	// バージョン情報 (現在は使用されていません
グロ _a_data;a_	。)
ング bss;a_en	// コードセクションのサイズ (バイト)。
ロン try;a_to	// データセクションのサイズ (バイト単位)。
グロ tal;a_sy	// スタックサイズ (バイト)。
ング ms;	// エントリポイン트を実行します。
ロン // ヘッダーサイズが32バイ	ドの割合当構造体はそれを総量まで
グロす。 a_trsize;	// シンボルセグメントの再配置テーブルサイ
グロ a_drsize;	ズ。
ログ a_tbase;a	// データセクションの再配置テーブルサイ
グロ _dbase.	ズ
};グロ	// コードセクションの再配置ベースアドレ
ング	ス。
ロン	// データセクションのリロケーションベー
グ	ク。その中でも、MINIX実行ファ
	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	// アンマップされた0ページ (ページ)。
A_PAL 0x02	// ページの境界でアジャストされます。
A_NSYM 0x04	// 新しいタイプのシンボルテーブル。
A_EXEC 0x10	// 実行可能なファイル
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	// 不明です。
A_I8086 0x04	// Intel i8086/8088.
A_M68K 0x0B	// モトローラのm68000。
A_NS16K 0x0C	// ナショナル・セミコンダクター社 16032
A_I80386 0x10	// Intel i80386.
A_SPARC 0x17	// Sun SPARC。

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 まとめ

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。

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・コードに少數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オープンソースPCシミュレーションパッケージ(<https://sourceforge.net/projects/bochs/>)です。
- Notepad++エディタ。バイナリファイル(<https://sourceforge.net/projects/notepad-plus/>)の編集に使用します。
- HxDヘックスエディター。バイナリファイルやディスクファイル、さらにはインメモリデータ(<https://mh-nexus.de/en/hxd/>)の編集に使用。
- WinImage DOSフォーマットフロッピーイメージファイル編集ソフト(<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ます。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ウェアでも、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、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には、VMware WorkstationやVirtualBoxを選択するのが良いでしょう。しかし、低レベルのシステムソフ

Bochsは良い選択だと思います。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、シミュレート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シミュレーションソフトウェア

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マ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・ウイ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
パスワー
ド
root# rpm -i bochs-2.6.i386.rpm
root# exit
ユーザー$ _
```

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¹⁴⁹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サイトから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

Linuxシステムは、[oldlinux.org](http://oldlinux.org/Linux.old/bochs/sls-1.0.zip)から直接ダウンロードすることができます。<http://oldlinux.org/Linux.old/bochs/sls-1.0.zip>

<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に修正する。

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の説明を参照してください。

1. Bochsシステムのセットアップ

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とbochsdbgの実行ファイルです。
 - BIOSイメージファイル（一般的には「BIOS-bochs-latest」と呼ばれています。）
 - VGABIOS イメージファイル（例：VGABIOS-lgpl-latest）。
 - 少なくとも1つのブートイメージファイル（フロッピー、ハードディスク、CDROMのイメージ）ファイル）。
-

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ができますが、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。

2. *.bxrc 設定ファイル

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。

1. メグス

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。

メガ: 128

2. floppyya (floppyb)

floppyaは1台目のフロッピードライブ、floppyb¹⁰⁵⁰は2台目のフロッピードライブを表しています。を起動する必要がある場合は

フロッピーディスクからシステムを起動する場合は、floppyaが起動可能なディスクを指している必要があります。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します。
floppya: 1_44=b:, status=inserted                # Winでは1.44MBのBドライブにアクセスします。
floppya: 1_44=bootimage.img, status=inserted      # イメージファイルbootimage.imgを使用します。
floppyb: 1_44=...Linux$rootimage.img, status=inserted
                                                # Use image ...Linux_rootimage.img.
```

3. アタ0、アタ1、アタ2、アタ3

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
```

4. ATA0-MASTER (ATA0-SLAVE)。

ata0-masterは、シミュレートされたシステムの最初の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デバイスを示します。以下に例を示すが、デバイス構成のオプションは表17-1の通りである。

```
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
```

オプション	説明	利用可能な値
タイプ	接続されているデバイスの種類	[disk cdrom]。
パス	画像ファイルのパス名	
モード	画像ファイルの種類、ディスクにのみ有効	[フラット コンкат 外部 DLL スペース]。 vmware3 undoable growing volatile] です。

シリンダー	ディスクの場合のみ有効	
頭部	ディスクの場合のみ有効	
spt	ディスクの場合のみ有効	
ステータス	ディスクの場合のみ有効	[Insulated Ejected] (挿入)
バイオデテクト	バイオス検出タイプ	[none auto], ata0のディスクに対してのみ有効 [cmos].
翻訳	バイオスコンバージョンのタイプ (int13)、有効 ディスクのみ	[none lba large rechs auto]のいずれかになります。
モード	デバイスから返された文字列を確認する ATAコマンド	

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: 翻訳の必要がなく、528MB (1032192セクタ) 以下の容量のハードディスクに適しています。
- ◆ large: 4.2GB(8257536セクタ)以下のハードディスクの標準的なビットシフトアルゴリズム。
- ◆ rechs: 修正シフトアルゴリズムは、7.9GB(15482880セクタ)以下の容量のハードディスクに対して、15ヘッドという疑似物理的なハードディスクのパラメータを使用しています。
- ◆ lba: 標準的なlbaアリストアルゴリズムを採用しています。8.4GB (16,450,560セクタ) 以下の容量のハードディスクに適しています。
- ◆ auto: 最適な変換方式を自動的に選択します（システムが起動しない場合は変更する必要があります）。

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: フラットなシーケンシャルファイルのこと。
- ◆ concat: 複数のファイル。
- ◆ external: 開発者が指定し、C++クラスが指定します。
- ◆ dll: 開発者が使用するもので、DLLが使用するものです。
- ◆ sparse: Stackable, Identifiable, retractable;
- ◆ vmware3: vmware3のハードディスクフォーマットに対応。
- ◆ undoable: REDOログが確認されたフラットファイル。
- ◆ growing: 大容量のスケーラブルな画像ファイル。
- ◆ volatile: REDOログが可変のフラットファイル。

上記オプションのデフォルト値は

mode=flat, biosdetect=auto, translation=auto, model="Generic 1234"

5. ブート

「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
```

6. 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え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	cpuスルペース=5000000モード=reset_on_triple_fault	典型的なIPS
2.4.6	3.4Ghz	インテル Core i7 2600、Win7x64/g++ 4.5.2搭載
2.3.7	3.2Ghz	インテル Core 2 Q9770、WinXP/g++ 3.4搭載
2.3.7	2.6Ghz	インテル Core 2 Due with WinXP/g++ 3.4
2.2.6	2.6Ghz	インテル Core 2 Due with WinXP/g++ 3.4
2.2.6	2.1Ghz	Athlon XP、Linux 2.6/g++ 3.4搭載

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。

ログ: bochsout.txt

17.2 BochsでのLinux 0.1xシステムの稼働

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-0.12-080324.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 パッケージ内のファイルの説明

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。
total 256916
-rwxr--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
[root@www linux-0.12-080324]# ls -o -g bochsout.txt
total 1054

```
-rw-r--r-- 1 1774 Mar 24 20:13 bochssrc-0.12-fd.bxrc
。
-rw-r--r-- 1 5903 Mar 24 17:56 bochssrc-0.12-hd.bxrc
。
-rw-r--r-- 1 35732 Dec 24 20:15 bochssrc-sample.txt
。
-rw-r--r-- 1 150016 Mar 6 2004 bootimage-0.12-fd
。
-rw-r--r-- 1 154624 2006年8月27日 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
[root@www linux-0.12-080324]#.
-rw-r--r-- 1 1474432 2006年8月27日 diskB.img
```

-rw- 本パッケージには91個のBochs 1.32 Linux-0.12 README, 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ます。

- Bochs-2.3.6-1.i586.rpm は、Linux オペレーティングシステム用の Bochs インストーラです。
最新のプログラムを再ダウンロードすることができます。
- Bochs-2.3.6.exe は、Windows オペレーティングシステムのプラットフォーム用 Bochs インストーラです。最新バージョンの Bochs ソフトウェアは、<http://sourceforge.net/projects/bochs/> からダウンロードできます。Bochs は改良を続けているため、一部の新しいバージョンでは互換性の問題が発生する可能性があります。これは、.bxrc 設定ファイルを修正することで解決する必要があります、一部の問題は、Linux 0.1x カーネルコードを修正することで解決する必要があります。
- bochsout.txt は、Bochs システムの実行時に自動的に生成されるログファイルです。このファイルには、Bochs ランタイムの様々なステータス情報が含まれています。Bochs の実行中に問題が発生した場合、このファイルの内容を確認することで、問題の原因を事前に判断することができます。
- bochssrc-0.12-fd.bxrc は、システムをフロッピーディスクから起動するための設定ファイルである。この設定ファイルは、Linux 0.12 システムを Bochs 仮想 A ドライブ (/dev/fd0) から起動するために使用されます。つまり、カーネルイメージファイルは仮想ディスク A に設定され、それに続くルートファイルシステムは現在の仮想ブートドライブに挿入される必要があります。起動時にルートファイルシステムのディスク (rootimage-0.12-fd) を仮想 A ドライブに「挿入」するように指示されます。この設定ファイルが使用するカーネルイメージとブートファイルは bootimage-0.12-fd です。Bochs が正しくインストールされた後、この設定ファイルをダブルクリックすると、設定された Linux 0.12 システムが実行されます。
- bochssrc-0.12-hd.bxrc は、A ドライブから起動するように設定された設定ファイルでもあります
が、ルートファイルである
システムをハードドライブのイメージファイル (rootimage-0.12-hd) に保存します。この設
定ファイルは bootimage-0.12-hd で起動します。同様に、Bochs を正しくインストールした後、
この設定ファイルをダブルクリックすると、設定された Linux 0.12 システムが実行されます
。
- bootimage-0.12-fd は、コンパイルされたカーネルによって生成されたイメージファイルです
。フロッピーブートセクターのコードを含む、カーネル全体のコードとデータが含まれてい
ます。設定された Linux 0.12 システムは、関連する設定ファイルをダブルクリックするこ
とで実行できます。
- bootimage-0.12-hd は、仮想ハードディスク上のルートファイルシステムを使用するためのカ
ーネルイメージファイルです。つまり、ファイルの 509 バイト目と 510 バイト目のルートファ
イルシステムのデバイス番号は、C ハードディスクの第 1 パーティション (/dev/hd1) に設定
されています。

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とdiskB.imgは、DOS形式のフロッピーメディアファイルです。いくつかのユーティリティが含まれています。Linux 0.12では、mcopyなどのコマンドを使って、この2つのイメージファイルにアクセスすることができます。もちろん、アクセスする前に、対応する「フロッピーディスク」を動的に「挿入」する必要がある。bochsrc-0.12-fd.bxrcまたはbochsrc-0.12--hd.bxrcの設定ファイルをダブルクリックしてLinuxを実行すると0.12システムでは、BドライブにdiskB.imgディスクが「挿入」されるように設定されています。
- rootimage-0.12-hdは、前述の仮想ハードディスクイメージファイルで、3つのパーティションを含んでいます。1つ目のパーティションはMINIXファイルシステムタイプ1.0のルートファイルシステムで、他の2つのパーティションはまた、MINIX 1.0のファイルシステムタイプや、テスト用のソースコードファイルなども格納されています。これらのスペースは、mountコマンドで読み込んで使用することができます。
- rootimage-0.12-fdは、フロッピーディスク上のルートファイルシステムです。このルートファイルシステムのディスクは、bochsrc-0.12-fd設定ファイルを使ってLinux 0.12システムを実行する際に使用されます。
- SYSTEM.MAPファイルは、Linux 0.12カーネルのコンパイル時に生成されるカーネルのメモリ格納位置情報ファイルです。このファイルの内容は、カーネルをデバッグする際に非常に役立ちます。

2. Bochsのインストール

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プラットフォームでのBochsの使用方法を紹介しています。

3. Linux 0.1xシステムの起動

1056

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、

```
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
ブート: 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のフロッピーア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ア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Linux 0.12システムの起動方法では、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の基本的な設定である。

ブートイメージファイルだけが bootimage-0.12-fd に置き換わっています。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。

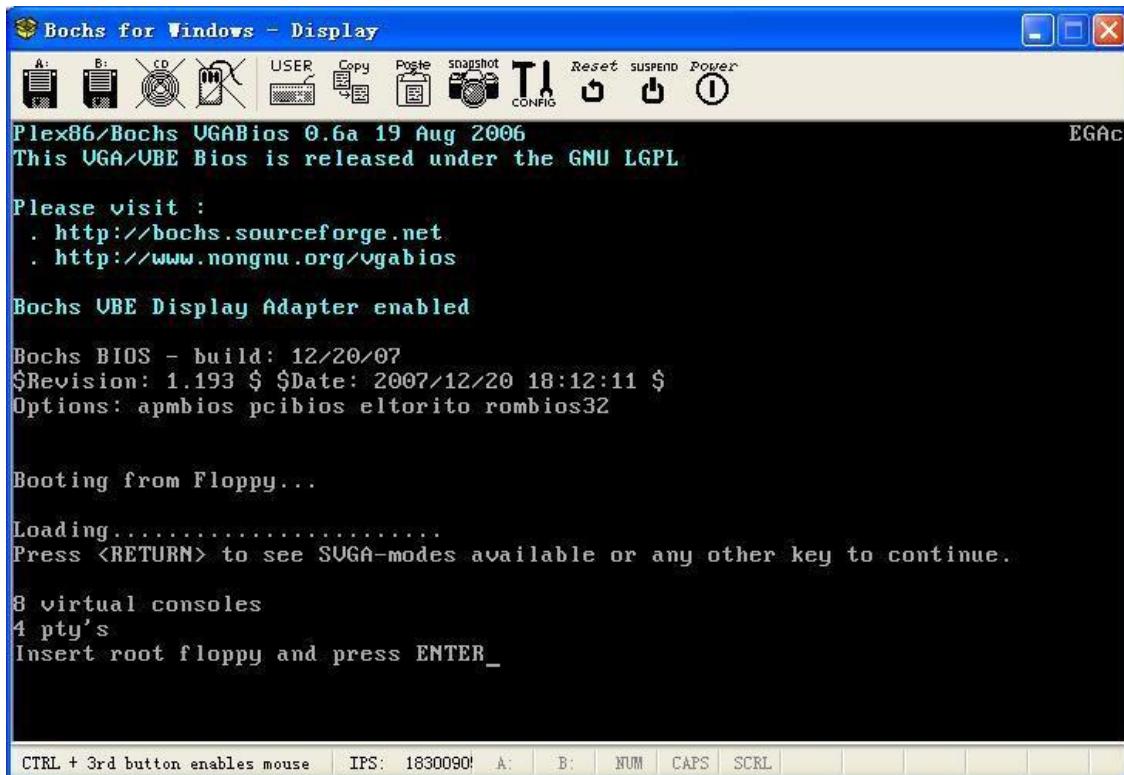


図17-1 フロッピーディスクからの起動とフロッピーディスク内のルートfsの使用

bochssrc-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。

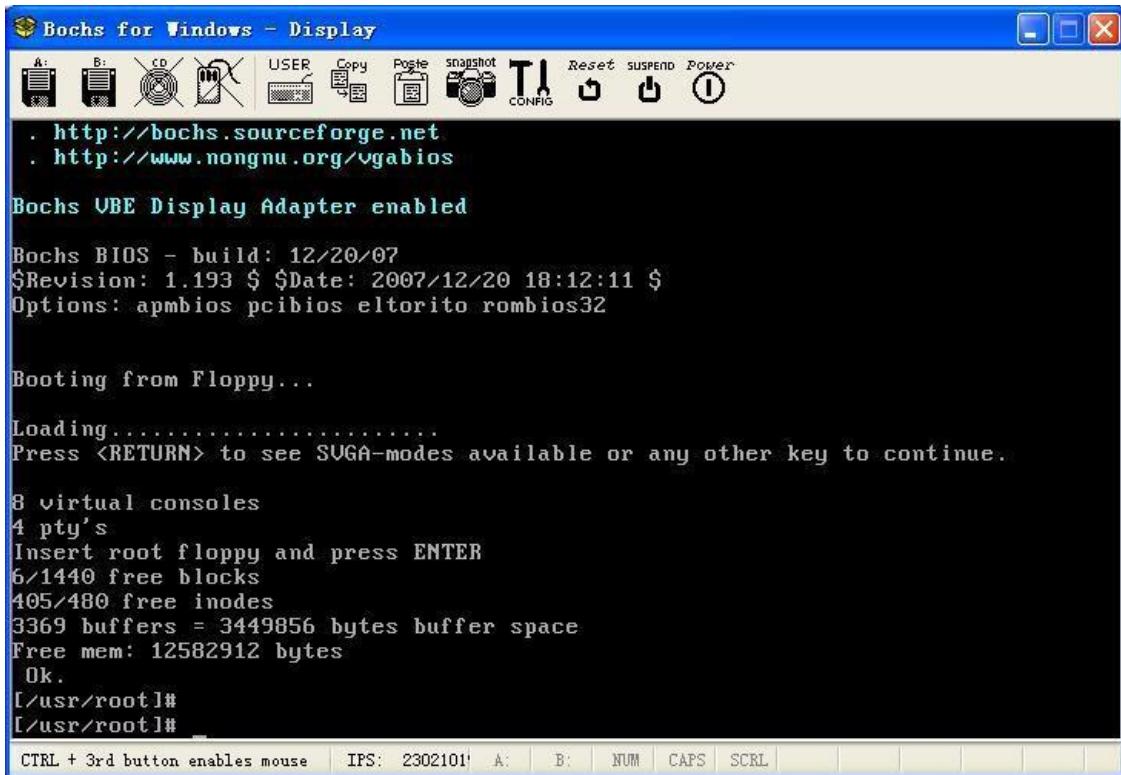


図17-2 フロッピー・ディスクを「交換」し、Enterを押して実行を続ける

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）に設定されているので、カーネルが初期化されると自動的に仮想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る。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 ディスクイメージファイルのアクセス情報

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ルールが必要です。¹⁰⁵⁹

一般的に、シミュレートされたシステムと交換するファイルサイズが小さい場合は、フロッピーのImageファイルを

を交換媒体として使用しています。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必要がある大規模なバッチファイルがある場合は、既存のLinuxシステムを使用し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つの側面から使用できるいくつかの方法について説明します。

- ディスクイメージツールを使って、フロッピーアイメージファイル内の情報（スマートファイルや分割ファイル）にアクセスします。
- Linuxでハードディスクのイメージファイルにアクセスするには、ループデバイスを使用します。（大量の交換）を行います。
- 情報交換（大量交換）のためにiso形式のファイルを使う。

17.3.1. WinImageソフトウェアの使用

フロッピーア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ファイルシステム内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、前述の方法で1.44MBのImageファイル（ファイル名はdiskb.imgとする）を用意し、Linuxの設定ファイルboochs.bxrcを変更する必要があります。

0.12.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 枚目のフロッピーアイメージに書き込みます。フロッピーアイメージが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。

```
HELLO C 74 4-30-104 4:47p
[/usr/root]# mcop hell 147452バイトの空
HELLO.Cをs)コピーする き
[/usr/root]# mdirc b:

```

ドライブBのボリュームにラベル

がなきにBでのBochsシミュレーターを終了し、WinImageでdiskb.imgファイルを開きます。WinImageのメインウインドウに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 既存のLinuxシステムの利用

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。フロッピーデバイス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形式のフロッピーデバイスの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、一般的に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を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、開始セクタのオフセット位置をパーティションの情報は、Imageファイルの中でアクセスすることができます。ハードディスクのImage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-0.12-080324.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 for help): x エキス
パーティション(m for help): p
ディスク /dev/loop1: 16 ヘッド, 63 セクタ, 487 シリンダ
```

```

1 80   1   1    0  15  63  130          1    132047 81
2 00   0   1  131  15  63  261        132048 132048 81
3 00   0   1  262  15  63  392        264096 132048 81
4 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swap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 vmlinuz
[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 シンプルカーネルのコンパイルと実行

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下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。

<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` - 圧縮されたソースファイル。
 2. `linux-0.00-rh9.tar.gz` - 圧縮されたソースファイル。
 - カーネルベースのイメージファイル。
 3. イメージ
 - Bochsの設定ファイルです。
 4. `bochsrc-0.00.bxrc`
 - WindowsでImageをフロッピーディスクに書き込むためのプログラム
 5. `rawrite.exe`
 - 。
 6. `README`
 - パッケージのドキュメント。
-

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 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フロッピーア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によって、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、ここのか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ます。

上記のか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することができます。

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下さい。ルート 487 Jun 12 19:25 Makefile
`-rw----- 1 ルート 4096 1557 Jun 12 18:55 boot.s`
`-rw----- 1 ルート ルート 5243 Jun 12 19:01 head.s`
`[/usr/root/linux-0.0]# ls -l`
`[root@root/linux-0.0]# make`

```
as86 -O -a -o boot.o boot.s
ld86 -O -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 record in
16+0 レコードアウト

dd bs=512 if=system of=Image skip=2 seek=1
16+0 records in
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      システム.マップ          の頭をし        システ
boot.o          。                     ています       ム
[/usr/root/linux-0.0]# make disk
Makefile:92: bootImage of fs/dev/fd0   の頭をし
1+1 records in           ています
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。

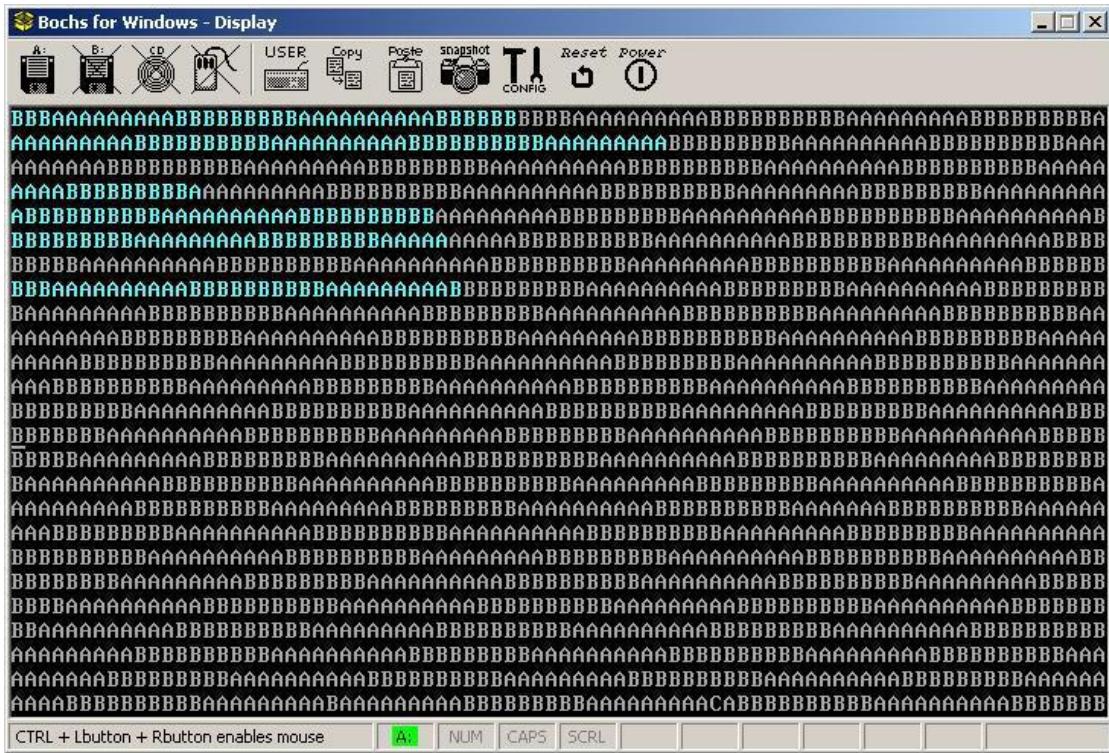


図17-3 シンプルなカーネルランタイムの画面表示

17.5 Bochsを使ったカーネルのデバッグ

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んだ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 Bochsデバッガの実行

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「bochssrc-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 bochssrc-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:confidence-0.12>"C:confidence-2.3.6\$bochsdbg" -q -f bochsrc-0.12-hd.bxrc  
000000000000i 【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  
=====
```

```
000000000000i[ ] bochsrc-hd-new.bxrcから設定を読み込む。  
000000000000i[ ] BOCHSのGUIとしてwin32モジュールをインストール  
000000000000i[ ] する  
次は t=0で [ ] ログファイルbochsout.txt使用  
(0) [0xfffffff0] f000:ffff0 (unk. ctxt): jmp far f000:e05b ea5be000f0  
<bochs:1>
```

この時点では、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します。  
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, 1b|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。

stepi [カウン
t] [count] count」命令の実行、デフォルトは1 lbid.
step [count] lbid.
s [count] 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ことです。
。 lbid.
n (または次 実行を停止し、コマンドラインプロンプトに戻ります。
) Ctrl-C 空のコマンドラインプロンプトでコマンドを入力した場合、Bochsを終了し
Ctrl-D ます。 デバッグを終了します。
quit, q

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 lbid.
vb seg:off リニアアドレスに命令ブレークポイントを設定 lbid.
lbreak addr lb 物理アドレスに命令ブレークポイントを設定します。ここで「*」はGDBとの互換性のためのオプションです。
addr
pbreak [*] addr lbid.
lbid.
pb [*] addr lbidです。
break [*] addr 現在のすべてのブレークポイントの状態を表示しま
す。 ブレークポイントを削除します。
b [*] addr lbid.
info break lbid.
delete n lbid.
DEL n
D n

3. メモリ操作コマンド

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 物理アドレス「addr」のメモリ内容を確認します。
オプションのパラメーター「n」、「u」、「f」は、次のようにになります。
n 表示するメモリユニットの数です。初期値は1です。
u ユニットサイズを示し、初期設定では以下の「w」が選択されています。
b(Bytes), 1バイト、h(Halfwords), 2バイト、w(Words), 4バイト、g(Giantwords), 8バイト。
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 (16進数)、d (10進数)、u (符号なし)、o (8進数)、t (2進数)。
c (char) チャーを表示します。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。
インフォダ 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
一ティー 初の20バイトのみが表示されます。

4. 情報表示・CPUレジスタ操作コマンド

情報プログラ プログラムの実行状況を表示します。
ム

```
info registers CPU レジスタを表示します（浮動小数点レジスタはありません）。
info break 現在のブレークポイントの設定状態を表示します。
set $reg = val CPUのレジスタ(セグメントとフラグ・レジスタを除く)の内容を変更します。
      例えば、$eax = 0x01234567を設定し、$edx = 25を設定します。
dump_cpu    CPUの全ステータス情報を表示します。 CPUの
set_cpu     全てのステータス情報を設定します。
```

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$"
"ebx:0x%x$"
"ecx:0x%x%""
"edx:0x%x%""
"ebp:0x%x%""
"esi:0x%x%""
"edi:0x%x%""
"esp:0x%x%""
"eflag:0x%x%""
"eip:0x%x%""
"eflags:0x%x%""
"eip:0x%x%""
"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%xdows"
"cr2:0x%xdows"
"cr3:0x%xdows"
"cr4:0x%xdows"
```

- 「s」はセレクターを意味します。
- 「dl」は、セレクタ・シャドウ・レジスタ内のセグメント・ディスクリプタの下位4バイトです。
- 「dh」は、セレクタ・シャドウ・レジスタ内のセグメント・ディスクリプタの上位4バイトです。
- 「valid」は、有効なシャドウ記述子がセグメント・レジスタに保存されているかどうかを示します。
- 「inhibit_int」は、命令遅延割り込みフラグです。セットされていれば、直前に実行された命令が、CPUが割り込みを受け入れるのを遅延させる命令であることを意味します（例えば、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るか、最後に「done」が表示された後に表示されます。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。

\$ disassembly \$start end 与えられたリニアアドレス範囲内の命令をディスアセンブルすることができます。
disas

u

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* -- 実行時に入力コマンド列をファイル「*filename*」に書き込みます。ファイルには「%s %d %x」の形式の行が含まれます。最初のパラメータはイベントタイプ、2番目はタイムスタンプ、3番目は関連イベントのデータです。
- playback *filename* -- 実行コマンドは、ファイル'*filename*'の内容を使って再生されます。他のコマンドをコントロールウィンドウに直接入力することもできます。ファイル内の各イベントが再生され、再生時間はコマンドが実行された時間を基準にしてカウントされます。
- print-stack [num words] -- スタックの先頭に num 個の 16 ビットワードを表示します。num のデフォルト値は 16 です。スタックセグメントのベースアドレスが 0 の場合、このコマンドはプロテクトモードでのみ正常に使用できます。
- load-symbols [global] *filename* [offset] -- ファイル '*filename*' からシンボル情報をロードします。キーワード global が与えられた場合、すべてのシンボルは、シンボルがロードされる前のコンテキストで表示されます。 offset(デフォルトは0)は各シンボルアイテムに追加されます。シンボル情報は、現在実行中のコードのコンテキストに読み込まれます。シンボルファイルファイル名の各行のフォーマットは「%x %s」です。最初の値はアドレス、2番目の値はシンボル名です。

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」を使ってデバッグを追いかけることができます（サブルーチンへの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 /1 # Disassemble 10 instructions.
00007c00: ( 0 ) : mov ax, 0x7c00 b8c007
00007c03: ( ) : mov ds, 軸 8ED8
00007c05: ( ) : mov ax, 0x9000 ; b80090
00007c08: ( ) : mov es, ax 8ec0
00007c0a: ( ) : mov cx, 0x100 ; b90001
00007c0d: ( ) : サブシ si ; 29f6
00007c0f: ( ) : サブデ ディ 29ff
    イ
00007c11: ( ) : rep movs word ptr [di], word ptr [si] ; f3a5
00007c13: ( ) : jmp 9000:0018 ; ea18000090
00007c18: ( ) : mov ax, cs 8cc8
<bochs:9> 情報 r # 現在のレジスタの内容を見る
eax      0xaa55 43605
ecx      0x110001 1114113
edx      0x0 0
ebx      0x0 0
esp      0xffffe 0xffffe
ebp      0x0 0x0
esi      0x0 0
edi      0xffe4 65508
eip      0x7c00 0x7c00
eflags   0x282 642
cs       0x0 0
ss       0x0 0
ds       0x0 0
es       0x0 0
fs       0x0 0
gs       0x0 0
<bochs:10> print-stack # 現在のスタックを表示する
0000ffffe [0000ffffe] 0000
です。
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:0xffffe
eflags:0x282
eip:0x7c00
cs:s=0x0, dl=0xfffff, dh=0x9b00, valid=1      # s-selector;dl,dh - descendの下位と上位の4バイ
ss:s=0x0, dl=0xfffff, dh=0x9300, valid=7      ト。
ds:s=0x0, dl=0xfffff, dh=0x9300, valid=1
es:s=0x0, dl=0xfffff, dh=0x9300, valid=1
fs:s=0x0, dl=0xfffff, dh=0x9300, valid=1
gs:s=0x0, dl=0xfffff, 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
dr6:0xfffff0ff0
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
済
<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0x00000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。ヘルプ」を参照してください。

コマンドを使用しています。

17.5.2 カーネル内の変数やデータ構造の配置

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> 1b 0xd1ec          # リニアアドレスのブレークポイント  
<bochs:13> c                  を設定します。 # 連続実行します  
(0) ブレークポイント 2, 0xd1ec in .  
?() 次は t=16689666です。  
(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 ディスクイメージファイルの作成

ディスクイメージファイルとは、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是データを格納するために使用され、表面全体が

ディスクは、同心円状のシリンドラーによってトラックに分けられている。ディスク上のデータを読み書きするためには、各ディスクの両側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トラック上を移動することができ、ディスク表面のすべての有効な位置にアクセスすることができる。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。

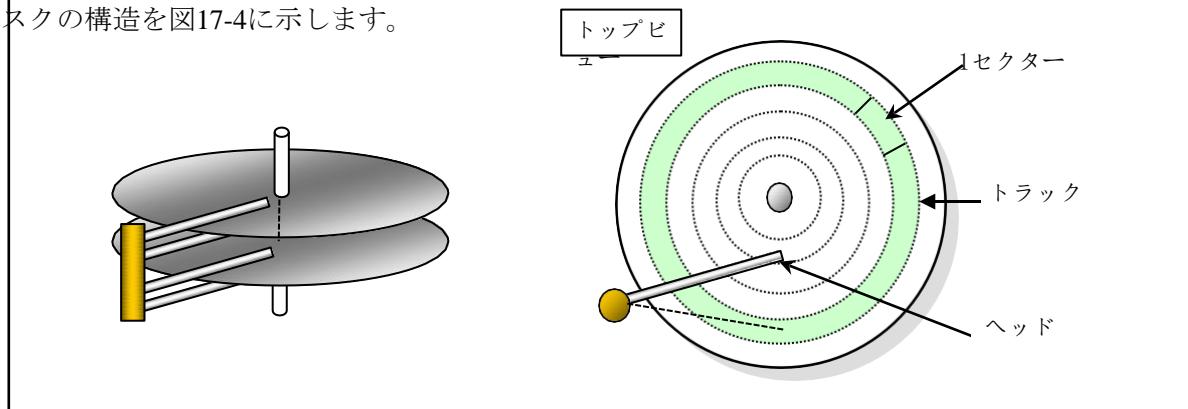


図17-4 典型的なハードディスクの内部構造

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（またはシリンドラー）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

$$\text{総セクタ数} = \text{物理トラック数} * \text{物理ヘッド数} * \text{トラックあたりのセクタ数}$$

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、現在一般的に使用されている救済策では、オペレーティング

システムやマシンのBIOSは、ハードディスクの総セクタ数が一定になるようにしながら、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、互換性やパラメータ表現の制約を確保します。ハードディスク・デバイス・パラメータの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とことと、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、 $64 \times 4 = 256\text{MB}$ 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（シリンドー数）、ヘッド数、およびディスクサイズが固定された、分割されていない超小型のハードディスクと考えることができます。

1トラックあたりのセクタ数。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18セクタ、1セクタ512バイトである。総セクタ数は2880で、総容量は $80 \times 2 \times 18 \times 512 = 1474560\text{バイト}$ となる。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 BOCHS独自の画像作成ツールを使う

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（シリンドー数（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]と入力してください。

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メガバイト単位で、1~32255の間で入力する [10] **256**

1074

cyl=520の「フラット」なハードディスク・イメージを作成します。

ヘッド=16

```
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。

bochs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

必要な容量のハードディスクのイメージファイルがすでにある場合は、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ア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。

2. Linuxでのddコマンドによるイメージファイルの作成

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トラックのシリンドー数が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のフロッピーア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になります。

```
dd if=/dev/zero of=diska. img bs=512 count=2880
```

3. WinImageによるDOS形式のフロッピーディスクイメージファイルの作成

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。

- a) WinImageを起動します。オプション」→「設定」メニューを選択し、「画像設定」ページを選択します。圧縮」を「なし」に設定します（つまり、インジケータを左端に引きます）。

1.44MBの容量のフォーマットをお選びください。

- c) ブートセクターのプロパティのメニュー項目「イメージ」→「ブートセクターのプロパティ」を選択し、ダイアログボックスの「MS-DOS」ボタンをクリックします。
- d) ファイルを保存します。

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。

- ImageファイルをNotepad++で開き、プラグインのHex-Editorを実行します。ディスクイメージファイルの511、512バイトは55、AAの2つの16進数によると、512バイトを押し戻し、すべての前のバイトになります。この時点では、MSDOS 5.0をブートに使用するディスクの場合、ファイルの最初の数バイトは「EB 3C 90 4D ...」のようになります。
- その後、右のスクロールバーを引いて、imgファイルの最後まで移動します。"...F6 F6 F6"以降のデータをすべて削除します。通常は、0x168000から始まるすべてのデータを削除します。操作が完了したときの最後の行は、「F6 F6 F6...」という完全な行になっているはずです。保存して終了すると、Imageファイルが使用できます。

7. ルートファイルシステムの作成

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つのフロッピーア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。

1. ルートファイルシステムとルートファイルデバイス

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。

- ◆ bin/ システム実行プログラムを格納する。sh、mkfs、fdiskなど、ライブラ
- ◆ usr/ リ機能やマニュアルなどのファイルを格納する。
- ◆ usr/bin/ ユーザーがよく使うコマンドを保存する。
- ◆ var/ システム稼働中のデータや、ログなどの情報を保存するために使用します。

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とき、ルートファイルシステムのデバイス番号を示すワード (ROOT_DEV) にしたがって、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 (509バイト目と510バイト目)。

は、起動ディスクのブート・セクタである。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は、指定されたハードディスク・パーティションからルート・ファイル・システムがロードされます。Linux 0.1X カーネルでサポートされている

表17-3 ハードディスクの論理デバイス番号		
デバイス番号	デバイスファイル名	説明
0x0300	/dev/hd0	初代ハードドライブ全体を表す
0x0301	/dev/hd1	第1ディスクの第1パーティション
0x0302	/dev/hd2	第1ディスクの第2パーティション
0x0303	/dev/hd3	1枚目のディスクの3番目のパーティション
0x0304	/dev/hd4	第1ディスクの第4パーティション
0x0305	/dev/hd5	2枚目のハードドライブ全体を表す
0x0306	/dev/hd6	2枚目のディスクの最初のパーティション
0x0307	/dev/hd7	2枚目のディスクの2番目のパーティション
0x0308	/dev/hd8	2枚目のディスクの3番目のパーティション
0x0309	/dev/hd9	2枚目のディスクの4番目のパーティション

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

デバイスNr	デバイスファイル	説明
0x0208	/dev/at0	1.2MB A ドライブ
0x0209	/dev/at1	1.2MB B ドライブ
0x021c	/dev/fd0	1.44MB A ドライブ
0x021d	/dev/fd1	1.44MB B ドライブ

17.7.2 ファイルシステムの作成

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って新しいイメージファイルを処理します。以下は、SLS Linuxのエミュレーションがインストールされていることを前提としています。

システムで、SLS-Linuxというサブディレクトリに格納されています。これを使って、上記で作成した256MBのハードディスクのイメージファイルhdc.imgをパーティション化し、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。

1. SLS-Linuxディレクトリ内にLinux-0.12というサブディレクトリを作成し、そこにhdc.imgファイルを移動します。
2. SLS-Linuxのディレクトリに移動し、SLS Linuxシステム用のBochs設定ファイル'bochsrc.bxrc'を編集します。~~オプション'ata0-master'の下に、私たちのハードディスク・イメージ・ファイルの設定パラメータ行を追加します。~~

```
ata0-slave:type=disk, path=...linux-0.12hdc.img, cylinders=520, heads=16, spt=63
```

3. エディタを終了します。`bochsrc.bxrc` ファイルのアイコンをダブルクリックして、SLS Linuxエミュレーションシステムを起動します。ログインプロンプトで「root」と入力し、Enterキーを押します。この時点でBochsが正常に動作しない場合、一般的には設定ファイルの情報が間違っていることが多いので、設定ファイルを再編集してください。
4. fdiskを使って、hdc.imgファイルに1つのパーティションを作成します。以下は、1つ目のパーティションを作成するための一連のコマンドです。残りの3つのパーティションも同様に作成します。SLS Linuxがデフォルトで確立しているパーティショントラップは、MINIX2.0ファイルシステムをサポートする81タイプ（Linux/MINIX）なので、fdisk tコマンドを使って80（Old MINIX）タイプに変更する必要があります。ここで注意していただきたいのは、SLS Linuxシステム下の2台目のハードディスクに、hdc.imgをフックしたことです。Linux-0.1Xのハードディスクの命名規則によると、ハードディスクの全体のデバイス名は/dev/hd5になるはずです。しかし、Linuxカーネルバージョン0.95以降、ハードディスクの命名規則は現在使用されている規則に変更されたため、SLS Linux下の第2ハードディスクのデバイス名は/dev/hdbとなっています。

```
[/]# fdisk /dev/hdb
マンド (m for help): n
コマンドアクション
e.g.
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セクタ番号 (1~5208) : 1
最後のセクタ番号 (1~5208) : 80
ヘッダーレコード (Lを入力するとコードが一覧表示されます) : L
4 DOS 16ビット <32M 51 ノベル? : 83
16進数コード (タイプL~リストコードボート: 80) : 80
6 DOS 16ビット >-32M 63 GNU HURD : 94
コマンド (mはヘルプ) : p
デバイス /dev/hdb: 1664ヘッド, 63セクタ, 520ブロック
                ブロックサイズ = 1008 * 512 バイトのシリングダ
                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のコマンド(m for help):w  
パーティションテーブルが変更されています  
。 再起動してから作業を行ってください。  
[/]#
```

5. このパーティションのデータブロック数（ここでは65015）を覚えておき、ファイルシステムの作成時に使用します。パーティションの設定が完了したら、SLS Linux カーネルが新しく追加されたパーティションを正しく認識できるように、通常通りシステムを再起動する必要があります。
 6. 再びSLS Linuxのエミュレーションシステムに入った後、mkfsコマンドを使って、先ほど作成した最初のパーティションにMINIXファイルシステムを作成します。コマンドと情報は以下の通りです。これで作成されるのは
パーティションには64,000個のデータブロック（1ブロックは1KB）があります。
-

```
[/]# mkfs /dev/hdb1 64000  
21333 inodes  
64000ブロック  
Firstdatazone=680 (680)  
Zonesize=1024  
Maxsize=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 Linux-0.12用Bochs設定ファイル

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  
ログ: 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

を上記と同じ内容に変更してください。特に注意が必要なのは「floppya」「ata0-master」「boot」です。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。

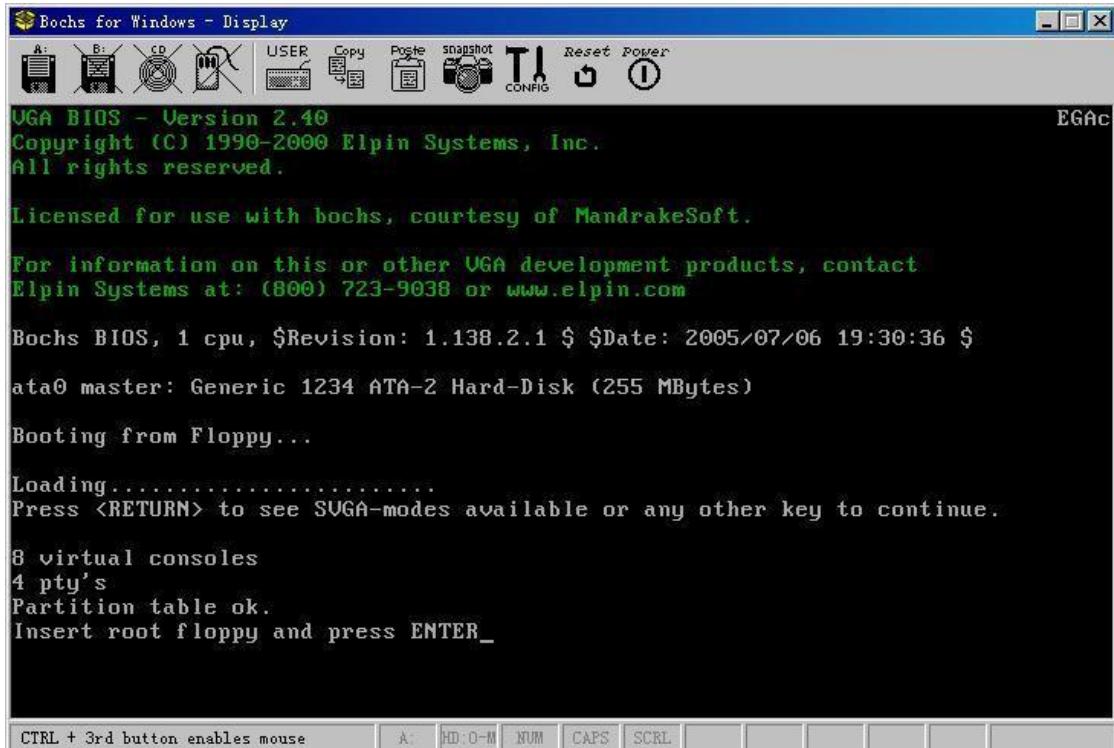


図 17-5 Bochs システムの実行ウィンドウ

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。

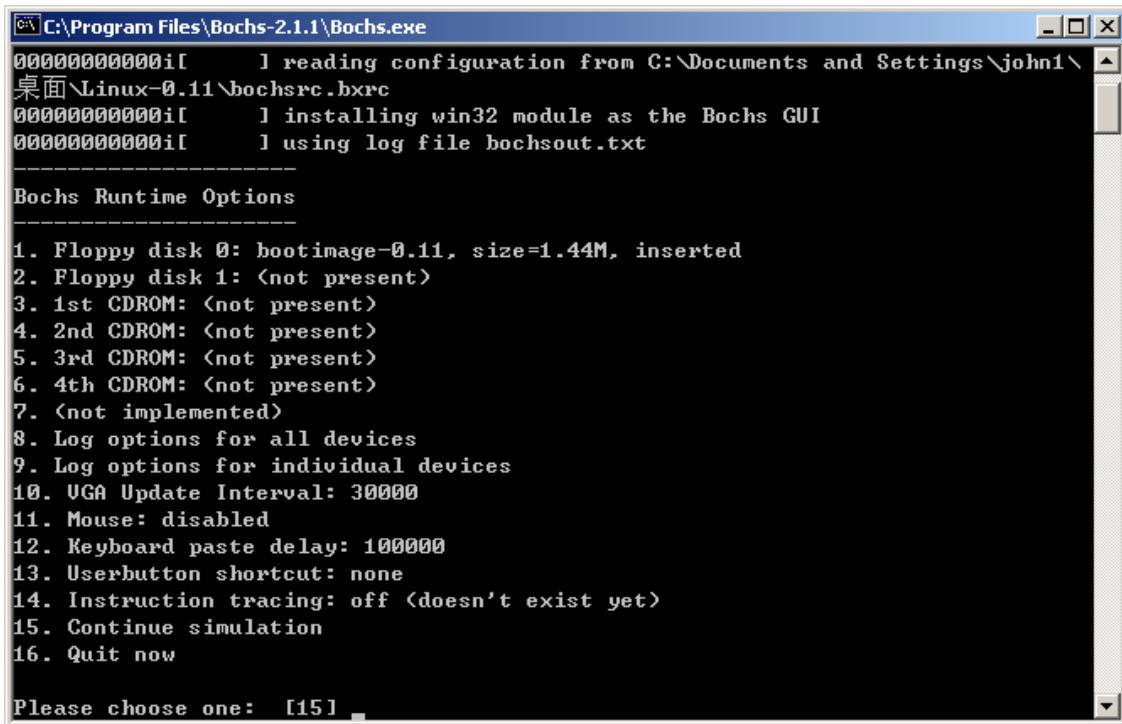


図 17-6 Bochs システム・コンフィギュレーション・ウィンドウ

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。

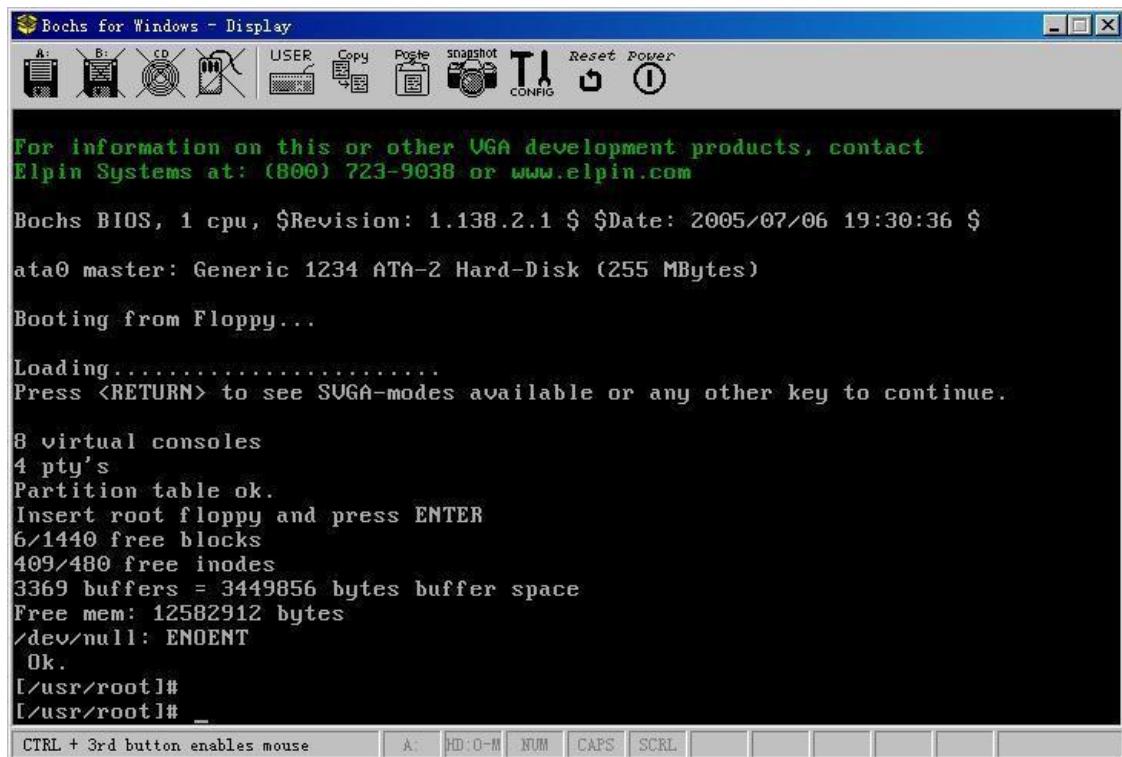


図17-7 Bochsで動作するLinux 0.12システム

17.7.4 hdc.imgでのルートファイルシステムの構築

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。前節では、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 マスター。汎用1234 ATA-2ハードディスク (255Mバイト)"

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
> ド
> cp -recursive +verbose /$i $i
> 濟
```

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
```

child 4 died with code 0000

17.7.5 ハードディスクイメージのルートファイルシステムの利用

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。

1. まず、bootimage-0.12とbochssrc.bxrcの2つのファイルをコピーして、bootimage-0.12-hdとbochssrc-hd.bxrcのファイルを生成します。
 2. bochssrc-hd.bxrc設定ファイルを編集し、「floppya:」オプションのファイル名を'bootimage-0.12-hd'に変更して、保存する。
 3. bootimage-0.12-hdのバイナリをNotepad++などのバイナリエディタで編集し、509番目と510番目のバイト(つまり0x1fc, 0x1fd)を修正します。元の値は00, 00ですが、これを01, 03に修正すると、ルートファイルシステムのデバイスがハードディスクのイメージの第1パーティションにあることを示しますので、保存して終了します。ファイルシステムを別のパーティションにインストールしている場合は、1バイト目を自分のパーティションに対応するように修正する必要があります。

000001f0h:00 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。

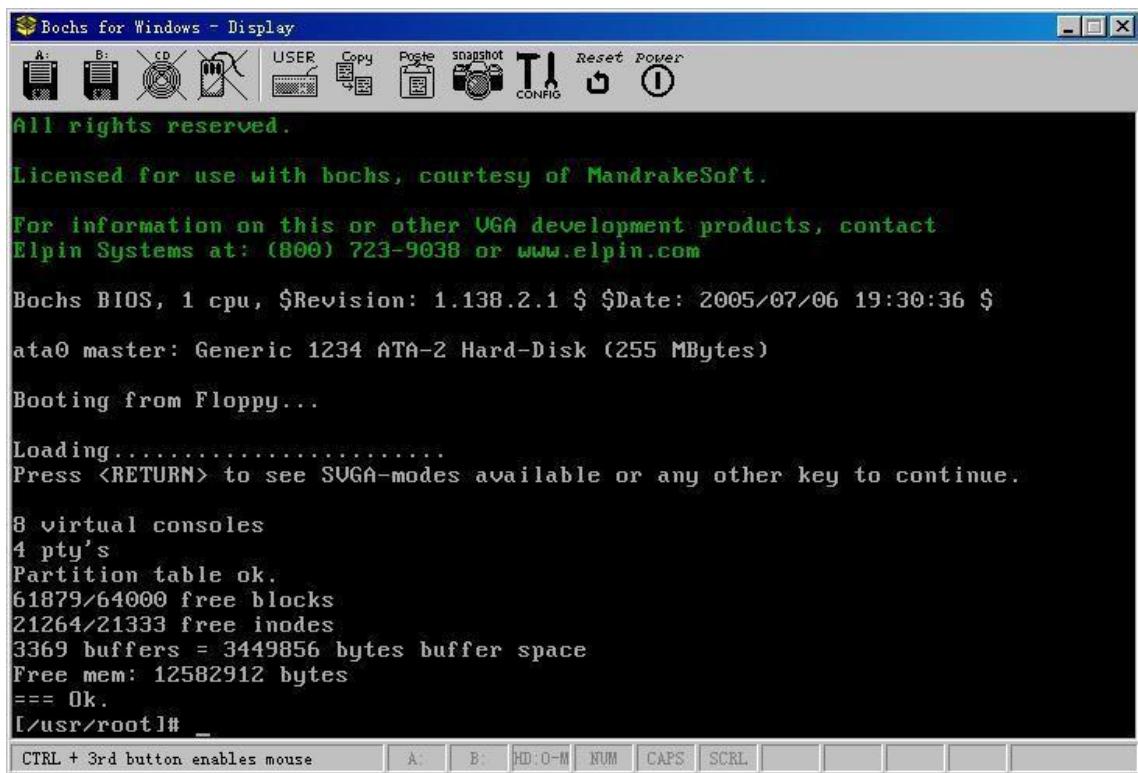


図17-8 ハードディスク・イメージ・ファイルのファイル
・システムの使用

17.8 Linux 0.12 システムでのカーネルのコンパイル

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-0.12-080324.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]# dd bs=8192 if=Image of=/dev/fd0
```

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番目のフロッピーアイメジファイル「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]# mc当地 画像bです。  
IMAGEのコピー  
[/usr/src/linux]# mc当地 System.mapのb:  
SYSTEM.mapをコピーする  
[/usr/src/linux]# mdir b:  
|のディスクのパーティションはB  
|です。 /  
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 4 File(s) 3829764 bytes 3-29-104 8:03p  
Free  
[/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。文字列を生成する際のgccの最適化に使用されます。

の指示に従います。この問題を解決するためには、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考えられます。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 Redhatシステムでのカーネルのコンパイル

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す。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マ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bootimage¹⁰⁸⁵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。

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

... システムの読み込
み ...

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。

1. Makefileの変更

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。

- a. gas」を「as」に、「gld」を「ld」にリネームします。なぜなら今、「gas」と「gld」は直接「as」と「ld」に名前を変えているからです。
- b. as」（オリジナルのガス）には「-c」オプションを使う必要がないので、カーネルのホームディレクトリLinuxのMakefileから「-c」コンパイルオプションを削除する必要があります。
- c. gccのコンパイルフラグオプションを削除する。「fcombin-reg'、'-mstring-insns'、そしてこれら2つのオプションをすべてのサブディレクトリのMakefileで削除する。fcombin-reg'オプションは1994年のgccマニュアルには見当たりませんでした。また、「-mstring-insns'はLinusがgccの修正に追加したオプションですので、この最適化オプションは間違いなくあなたのgccには含まれていません。
- d. gccのコンパイルオプションで、「-m386」オプションを追加します。こうすることで、RedHat 9でコンパイルされたカーネルイメージファイルには、80486以上のCPUの命令が含まれなくなり、80386マシンでもカーネルが動作するようになります。

2. アセンブリ言語プログラムのコメントの修正

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。

3. メモリ・アライメント・ステートメントのAlign値の変更

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りました。 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
```

4. インラインマクロのアセンブリ言語プログラムの修正

asのアセンブリは改良が重ねられ、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。

この修正には時々問題があります。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できるため、場所によっては、修正されるレジスター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。

5. アセンブリステートメントにおけるC変数の参照表現

Linux 0.1X カーネルの開発に使用されているアセンブリでは、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、アセンブリ内のすべての 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
```

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 プロテクトモードでのデバッグ表示機能

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とも、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ティリを使うだけで、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。

```
/*
 * 目的: 32ビットの整数を16進法で画面に表示する。
 * Params: value -- 表示する整数。
 *          pos ... 画面の位置を16文字の幅を単位として指定します。例えば、「2」とすると、左上
 *          から32文字の幅で表示を開始します。
 * 戻る。なし。
 * アセンブリ言語のプログラムで使用したい場合は、その関数がコンパイルされていることを確認してください。
 * 関数成形pos カーネルにリンクします。はcpu版は必ずこの値実際の以下の通置を換えてください。
 * プレッシャー値           // 'pos' と 'value' には、任意の合法的なアドレス指定方法を使用できます
 * call check_data32       。
 */
inline void check_data32(int value, int pos)
{
    asm volatile (
        "shl    $4, %%ebx\t"           // %0 -表示される値; EBX -画面の位置 pos.
        "addl   $0xb8000, %%ebx_n\t"  // posを16倍して、VGAメモリのスタートアドレスを加えます。
        "movl   $0xf0000000, %%eax_n\t" // 画面の左上からの位置をEBXで取得します。
        "movb   $28, %%cl\t"           // 4ビットのマスクを設定します。
        "1:n\t"
        "movl   %0, %%edx\t"           // 表示された値をEDXに入れる。
        "andl   %%eax, %%edx\t"        // EAXで指定された4ビットをEDXで取る。
        "shr    %%cl, %%edx\t"          // 28ビットを右にシフト、EDXは4ビット取った値。
        "add    $0x30, %%dxn\t"         // この値をASCIIコードに変換します。
        "cmp    $0x3a, %%dxn\t"         // 10未満の場合は、ラベル2にジャンプします。
        "jb2f\t"
        "add   $0x07, %%dxn\t"          // それ以外の場合は7を加えてA-Fに変換します。
    )
}
```

```
"2:n¥¥"
"add"    $0x0c00, %%dx¥n¥t"      // 表示属性を設定します。
"movw"   %%dx, (%ebx)¥¥t"      // この値を表示メモリに入れる。
"sub"    $0x04, %%cln¥¥"        // シフト数から4を引いた次の16進数を用意する。
"shr"    $0x04, %%eaxn¥t"      // マスクは4ビットだけ右にシフトされます。
"add"    $0x02, %%ebxn¥t"      // 表示メモリの位置を更新する。
"cmpl"   $0x0, %%eaxn¥t"       // マスクが右から移動した(8ヘックス表示)?
"jnz1b¥"                         // 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
:: "m"(value)、"b"(pos)          // します。
) .
}
```

17.10 統合されたブートディスクとルート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のフロッピーメディア用の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 統合ディスクの構築原理

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。

1. ブートプロセスの原理

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場合（サイズが0の場合）、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や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、ルートファイルシステムが存在する場合は一般的な起動処理を行います。

仮想ディスクはありません。

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。

2. 統合ディスクの構造

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
カーネルコ
ード」
256 257
ブートブロック スーパーブ
ロック

図17-9 統合ディスクのコード構造

前述の通り、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枚目のディスクの先頭と定義している

ブロックを作成しました。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です。したがって、ルートファイルシステムをディスクの256番目のディスクブロックの先頭に置くことは、確かに問題はありませんが、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 $1440 - 256 = 1184$ KB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。

2. 統合されたディスク構築プロセス

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。

1. カーネルのリコンパイル

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
```

2. 一時的なルートファイルシステムの作成

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。

- (1) この章の前に説明した方法で、1024KBの大きさの空のイメージファイルを作ります。ファイルの名前は「rootram.img」と指定します。次のコマンドで、現在の下に生成されるLinuxシステムです。
-

```
dd bs=1024 if=/dev/zero of=rootram.img count=1024
```

- (2) Bochsでlinux-0.1Xシステムを実行します。そして、Bochs のメインウィンドウでドライバーディスクを設定します。ディスク A は rootimage-0.1X (0.11 カーネルは rootimage-0.11-origin)、ディスク B は rootram.img です。
- (3) 以下のコマンドで、rootram.imgディスクに1024KBのサイズの空のファイルシステムを作成します。¹⁰⁹¹ その後、AディスクとBディスクをそれぞれ/mntと/mnt1ディレクトリにマウントします。ディレクトリ /mnt1 が存在しない場合は、作成します。

```
mkfs /dev/fd1 1024
mkdir /mnt1
mount /dev/fd0 /mnt
mount /dev/fd1 /mnt1
```

- (4) cp コマンドを使って、rootimage-0.1X の /mnt ファイルを選択的に /mnt1 ディレクトリにコピーし、/mnt1 に root ファイルシステムを作成します。エラーメッセージが表示される場合は、通常 1024KB以上のコンテンツがあることを意味します。まず/mnt/内のファイルを減らして、1024KB 以下という容量要件を満たします。そのためには、/binと/usr/binの下にあるいくつかのファイルを削除します。容量については、「df」コマンドを使って確認することができます。例えば、残すことを選択できるファイルは以下の通りです。

```
[/mnt/bin]# ll
total 495
-rwx--x--x 1 ルート ルート 29700 4月29日 20:15 mkfs
-rwx--x--x 1 ルート ルート 21508 4月29日 20:15 mknod
-rwx--x--x 1 ルート ルート 25564 Apr 29 20:07 Mount
-rwxr-xr-x 1 ルート ルート 283652 9月28日 10:11 sh
-rwx--x--x 1 ルート ルート 25646 4月29日 20:08 umount
-rwxr-xr-x 1 ルート 4096 116479 2004年3月3日 vi
[/mnt/bin]# cd /mnt/usr/bin
[/mnt/usr/bin]# ll
合計 364
-rwxr-xr-x 1 ルート ルート 29700 ジャ 15 1992年の猫
                                         ン
-rwxr-xr-x 1 ルート ルート 29700 Mar 4 2004 chmod
-rwxr-xr-x 1 ルート ルート 33796 Mar 4 2004 chown
-rwxr-xr-x 1 ルート ルート 37892 Mar 4 2004cp
-rwxr-xr-x 1 ルート ルート 29700 Mar 4 2004年 dd
-rwx--x--x 1 ルート 4096 36125 Mar 4 2004年 df
-rwx--x--x 1 ルート ルート 46084 9月 28 10:39 ls
-rwxr-xr-x 1 ルート ルート 29700 ジャ 15 1992年
                                         ン
                                         mkdir
                                         ン
                                         mv
                                         ン
                                         rm
[/mnt/usr/bin]# rm -r ./*
                                         ン
```

- (5) その後、以下のコマンドでファイルをコピーしました。さらに、必要に応じて/mnt/etc/fstabや /mnt/etc/rcの内容を変更します。この時点で、fd1(/mnt1/)に1024KB以下のサイズのファイルシステムを作成しました。

```
cd /mnt1
for i in bin dev etc usr tmp
do
cp -recursive +verbose /mnt/$i $i
done
シンク
```

- (6) 「umount」コマンドで/dev/fd0と/dev/fd1のファイルシステムをアンマウントし、「dd」コマンドで /dev/fd1のファイルシステムをLinux-0.1Xシステムにコピーし、「rootram- 0.1X root file system Image file」という名前のファイルを作成します。

```
dd bs=1024 if=/dev/fd1 of=rootram-0.1X count=1024
```

この時、BochsのLinux-0.1Xシステムでは、新しくコンパイルされたカーネルイメージファイルを /usr/src/linux/Image と、1024KB以下の容量を持つシンプルなルートファイルシステムのイメージファイルrootram-0.1Xです。

17.10.2.3 統合ディスクの作成

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。のAディスクの構成を変更します。

を起動し、先に用意した1.44MBのイメージファイル「bootroot-0.1X」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 統合ディスクシステムの起動

まず、統合ディスク用の簡単なBochs設定ファイル、bootroot-0.1X.bxrcを作成してみましょう。主な設定内容は以下の通りです。

```
floppya: 1_44=bootroot-0.1X
```

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。

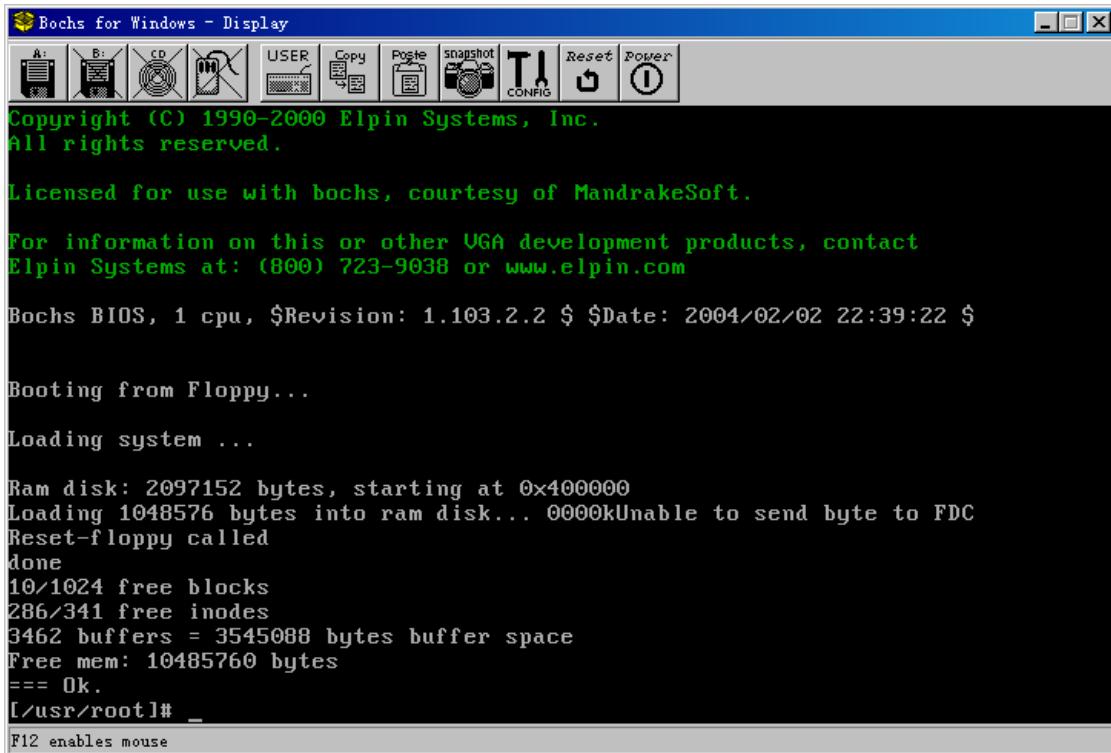


図17-10 統合ディスク実行インターフェース

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>

11. GDBとBochsによるカーネルコードのデバッグ

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。

1. gdbstubによるBochsシステムのコンパイル

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システムのソースをダウンロードします。

のコードを以下のサイトからダウンロードしてください（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 Linux 0.1X カーネルのデバッグ情報付きコンパイル

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
CFLAGS = -Wall -O -g -fomit-frame-pointer	します。
.	// '-g' ノフグを追加
	します。

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
      シンク
```

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。

3. デバッグの方法と手順

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。

1. 最新のLinuxでコンパイルされたLinux 0.11カーネルのデバッグ

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つのルート          root          18055 Jun 18 15:07 bochsrc-fd1-gdb.bxrc
drwxr-xr-x       10ルート        root          4096 Jun 18 22:55 linux
-rw-r--r--       10ルート        root        1474560 Jun 18 20:21 rootimage-0.11-for-orig
。
。
[root@plinux linux-gdb-rh9]# ls -l
1つのルート          root          35 Jun 18 16:54 ラン
```

最初のファイル「bochssrc-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
```

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。

1. Xウィンドウシステムで2つのターミナルウィンドウを開く。
2. ターミナルウィンドウの1つで、作業ディレクトリをlinux-gdb-rh9ディレクトリに切り替え、プログラム'./run'を実行します。この時点で、gdbの接続を待つメッセージが表示されます。
"Wait for gdb connection On localhost:1234 "と表示され、Bochsのメインウィンドウが作成されます（この時点では内容はありません）。
3. 別のターミナルウィンドウで、作業ディレクトリをカーネルのソースディレクトリであるlinux-gdb-rh9/linux/に切り替え、コマンドを実行します。"gdb tools/system"と入力します。
4. gdbが実行されているウィンドウに、「break main」と「target remote localhost:1234」というコマンドを入力します。この時、gdbはBochsに接続されている情報を表示します。
5. gdb環境でコマンド「cont」を実行します。しばらくすると、gdbはプログラムがinit/main.cのmain()関数で停止していることを示します。

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

起動して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でカバーされ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を行うことができます。 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。詳細は "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) ブレークメイン // main()関数にブレークポイントを設定します
0x6621でブレークポイント1: ファイル init/main.c,

line 110. (gdb) ターゲットリモート ローカルホスト // Bochsにつなぐ。

:1234

0x000006ff01284をmkdir (pathname=0x0, mode=0) at namei.c:481

481 namei.cです。そのようなファイルやディレクトリはありません。

(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
                                                // ソースコードを見る。
105 {
106 /*                      /* スタートアップ・ルーチンは、以下のことを前提と
107 * 割り込みは無効のままで。必要な設定を行ってから
108 * それを可能にする
109 */
110     root_dev = orig_root_dev;
111     drive_info = DRIVE_INFO;
112     memory_end = (1<<20) + (EXT_MEM_K<<10);
113     memory_end &= 0xfffff000;
114     if (memory_end > 16*1024*1024)
(gdb) 次の                                // シングルステップ実行。
111     drive_info = DRIVE_INFO;
(gdb) 次の                                // シングルステ
112     ップ。 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
(gdb) quit                                です。
プログラムを実行してい    とりあえず終了? (y また    // gdbを終了します。
ます。 [root@plinux      は n) y
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 linker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。

2. Linux 0.1Xシステムでコンパイルされた0.1Xカーネルのデバッグ

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カーネルの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、mc当地コマンドを使ってBochsの2番目のフロッピーメディアに書き込み、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は次のとおりです。

以下の通りです。

1. Bochsの下でLinux-0.1Xを起動し、/usr/src/ディレクトリに入り、「linux-gdb」ディレクトリを作成します。
2. まず 0.1X カーネルのソースツリー全体をコピーするために、「cp -a linux-gdb/」というコマンドを使います。その後、linux-gdb/linux/ディレクトリに入り、上記のようにすべてのMakefileを修正し、カーネルをコンパイルします。
3. usr/src/ ディレクトリに戻り、「tar」コマンドを使って linux-gdb/ ディレクトリを圧縮し、「linux-gdb.tgz」ファイルを取得します。
4. 圧縮ファイルを2枚目のフロッピー（b ドライブ）のイメージファイルにコピーします。
"mc当地 linux-gdb.tgz b:".b ディスクの空き容量が足りない場合は、ファイル削除コマンド「mdelete b: file name」を使用して、b ディスクの空き容量を確保してください。
5. ホスト環境がWindows OSの場合は、WinImageを使って圧縮された

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。

6. コピーした圧縮ファイルを最新のLinuxシステムで解凍すると、0.1Xカーネルのソースツリーを含むlinux-gdbディレクトリが生成される。linux-gdbディレクトリに行き、bochs設定ファイル'bochsrc-fd1-gdb.bxrc'を作成する。また、「linux-0.11-devel」パッケージから「bochsrc-fdb.bxrc」設定ファイルの内容を取り出し、「gdbstub」パラメータ行を自分で追加することもできる。次に、oldlinux.orgのウェブサイトから'rootimage-0.11'ルートファイルシステムフロッピーアイメージファイルをダウンロードし、これもlinux-gdbディレクトリに保存する。

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  
[/usr/src]# mkdir linux-gdb  
[/usr/src]# cp -a linux linux-gdb/  
[/usr/src]# cd linux-gdb/linux  
[/usr/src/linux-gdb/linux]# vi Makefile  
...  
[/usr/src/linux-gdb/linux]# make clean; make  
...  
[/usr/src/linux-gdb/linux]# cd ../../  
[/usr/src]# tar zcvf linux-gdb.tgz linux-gdb  
...  
[/usr/src]# mdisk b: ドラ  
イブBのボリュームはBt B  
のディレクトリ:/です。  


|              |        |           |        |
|--------------|--------|-----------|--------|
| 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  |
| ショエラ~1       | 101767 | 9-19-104  | 1:24p  |
| Zシステムマ       | 17771  | 10-05-104 | 11:22p |
| ップ           | 90624  | バイトの空     |        |

  
[/usr/src]# mdisk b:linux-gdb.tgz  
[/usr/src]# mcopy linux-gdb.tgz b:  
LINUX-GD.TGZをコピー  
する [/usr/src]#  
// ソースコードのディレクトリを入力  
します。  
// Create ディレクトリ linux-gdb/  
// ソースコードをlinux-ddb/にコピー  
Makefileを修正する。  
  
// カーネルを補完する  
  
// 圧縮ファイルを作成しま  
す。  
// bディスクの内容を確認します。  
  
// スペースが足りないので、ファイルを  
削除します。  
// linux-gdb.tgzをbディスクにコピーし  
ます。
```

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。

```
-rwxr-xr-x  1  ルート   ルート    17771年10月  5  2004年 システム
                .マップ
[roo[r@plinu[b.11]# cp /mnt/d4/linux-gd.tgz . // ファイルをコピーし
[roo[r@plinu[0.11]# umount /mnt/d4                         ます。
[roo[r@plinu[0.11]# tar zxvf linux-gd.tgz // Bディスクをアンマウ
...                                              ントします。
[roo[r@plinu[0.11]# cd linux-gd // ファイルをアンタリ
[roo[r@plinu[linux-gd]# ls -l ングする。
total 4
drwxr-x--x 10 15806 root 4096 Jun 19 [root@plinu[ 2005年 リ
linux-gd]#. ナックス
```

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、フロッピーディスク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 まとめ

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ることを祈っています。ありがとうございました。

リファレ ンス

- 1 インテル株式会社 INTEL 80386 Programmer's Reference Manual 1986, INTEL CORPORATION,1987.
- 2 インテル株式会社 IA-32 インテル・アーキテクチャー・ソフトウェア開発者向けマニュアル Volume.3:システム・プログラミング・ガイド <http://www.intel.com/>, 2005.
- 3 ジェームズ・L・ターリー Advanced 80386 Programming Techniques.Osborne McGraw-Hill,1988.
- 4 Brian W. Kernighan, Dennis M. Ritchie.The C programming Language.Prentice-Hall 1988.
- 5 Leland L. Beck.システムソフトウェア:システムプログラミング入門,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 (株)デジタルVT100ユーザーガイド.<http://www.vt100.net/>
- 11 クラーク・L・コールマンUsing Inline Assembly with gcc.<http://oldlinux.org/Linux.old/>.
- 12 ジョン・H・クロフォード, パトリック・P・ゲルシング.80386のプログラミング.シベックス社、1988年
- 13 FreeBSD オンラインマニュアル.<http://www.freebsd.org/cgi/man.cgi>
- 14 Andrew S.Tanenbaum.Operating Systems:設計と実装.Prentice-Hall-International Editions.1990.4
- 15 モーリス・J・バッハThe Design of the UNIX Operating System.Prentice Hall.1990
- 16 ジョン・ライオンズライオンズのUNIX解説 第6版 ソースコード付き.ピアツーピア・コミュニケーションズ社1996
- 17 Andrew S. Tanenbaum, Albert S. Woodhull.オペレーティング・システム：設計と実装（第2版）.Prentice Hall.1997.
- 18 アレッサンドロ・ルビーニ、ジョナサンLinuxデバイスドライバー.O'Reilly & Associates.2001年
- 19 Daniel P. Bovet, Marco Cesati.Understanding The Linux Kernel.China Electric Power Press.2001.
- 20 张载鸿。微型机(pc 系列)接口控制教程, 清华大学出版社, 1992.
- 21 Smilax china (species of sarsaparilla) ms-dos 5.0 内核剖析.西安电子科技大学出版社, 1992.
- 22 RedHat 9.0 オンラインマニュアル。<http://www.plinux.org/cgi-bin/man.cgi>
- 23 W.Richard Stevens.UNIX環境での高度なプログラミング.中国機械出版.2000.2
- 24 Linux週刊誌のニュース <http://lwn.net/>
- 25 P.J. Plauger.The Standard C Library.プレンティスホール、1992
- 26 フリーソフトウェアファウンデーション。GNU C ライブライ. <http://www.gnu.org/> 2001
- 27 チャック・アリソンThe Standard C Library.C/C++ Users Journal CD-ROM, Release 6.
- 28 ボークスのシミュレーションシステム。<http://bochs.sourceforge.net/>
- 29 Brennan "Bas" Underwood.Brennan's Guide to Inline Assembly. <http://www.rt66.com/~brennan/>
- 30 ジョン・R・レヴィンリンカーズ&ローダーズ <http://www.iecc.com/linker/>
- 31 Randal E. Bryant, David R. O'Hallaron.コンピュータシステム A programmer's Perspective.電子工業界の出版社。2004.3
- 32 インテルデータシート:8254プログラマブル・インターバル・タイマ.1993.9
- 33 インテルデータシート:8259A プログラマブルインタラプトコントローラ.1988.12
- 34 インテルデータシート。82077A CMOSシングルチップフロッピーディスクコントローラ.1994.5
- 35 ロバート・ラブLinux Kernel Development.China Machine Press.2004
- 36 Adam Chapweske.The PS/2 Keyboard Interface. <http://www.computer-engineering.org/>

- 37 ディーン・エルスナー、ジェイ・フェンレイソン&フレンズとして使ってています。GNUアセンブラー。
<http://www.gnu.org/> 1998
- 38 スティーブ・チェンバレンldを使う。GNUリンクエラー。
<http://www.gnu.org/> 1998
- 39 マイケル・K・ジョンソン。The Linux Kernel Hackers' Guide. <http://www.tldp.org/> 1995年
- 40 リチャード・F・フェラーロEGA, VGA, Super VGAカードのプログラマーズガイド.第3版.Addison-Wesley, 1995.

付録

A1 ASCIIコード表

10進法	16進文字	10進法	Hex	キャラクター	10進法	Hex	キャラクター
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 (スペース)	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

共通C0制御文字表

ニーモニック	コード	対策	
NUL	0x00	Null -- 受信時に無視される（入力バッファに保存されない）。	
ENQ	0x05	問い合わせ・・・返信メッセージを送信します。	
BEL	0x07	ベル...音が鳴る。	
BS	0x08	Backspace -- カーソルを1文字分左に移動します。カーソルがすでに左端にある場合。アクションはありません。	
HT	0x09	Horizontal Tabulation -- カーソルを次のタブストップに移動させることができます。右側にタブストップがない場合。が右端に移動します。	
LF	0x0a	Linefeed -- このコードは、キャリッジリターンまたはラインフィード操作を引き起こす（ラインフィードモードを参照）。	
VT	0x0b	垂直方向のタブレーションは、LFと同様に動作します。	
FF	0x0c	フォームフィード -- LFのように動作します。	
CR	0x0d	Carriage Return -- カーソルを現在の行の左端に移動させる。	
SO	0x0e	Shift Out -- SCS 制御シーケンスで選択された G1 文字セットを使用します。G1は次の5つのうちの1つを指定できます。 キャラクタセット。	
SI	0x0f	Shift In -- SCS 制御シーケンスで選択された G0 文字セットを使用します。G0は次の5つのうちの1つを指定できます。 キャラクタセット。	
DC1	0x11	デバイスコントロール1 -- XON.端末に送信を再開させる。	
DC3	0x13	デバイスコントロール3 -- XOFF。XOFFとXONの送信以外のコードの送信を停止する。	
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。また、エラー文字も表示されます。	
SUB	0x1a	Substitute -- CANと同じ働きをします。	
ESC	0x1b	Escape -- エスケープコントロールシーケンスを生成します。	
DEL	0x7f	Delete -- 入力時、無視する（バッファには保存されない）。	
ニーモニック	コード	7B seq.	対策
IND	0x84	ESC D	Index -- カーソルが同じ列の1行下に移動します。すでにカーソルが存在する場合は下の行では、スクロール操作を行います。
NEL	0x85	ESC-H	Next Line -- カーソルは次の行の最初の列に移動します。もしカーソルがすでに下の行では、スクロール操作が行われます。
HTS	0x88	ESC E	Horizontal Tab Set -- カーソル位置に水平方向のタブストップを設定します。
RI	0x8d	ESC M	Reverse Index -- カーソルが同じ列の1行上に移動します。既にカーソルが存在する場合は上の行では、スクロール操作が行われます。
SS2	0x8e	ESC N	Single Shift G2 -- GLのG2文字セットを一時的に使用し、次の表示を行う。文字になります。G2は、SCS (Selective Character Set) 制御シーケンスで指定されます（付録3のエスケープシーケンスと制御シーケンスの表を参照）。
SS3	0x8f	ESC O	Single Shift G3 -- GLのG3文字セットを一時的に呼び出し、次の文字になります。G3は、SCS (Selective Character Set) 制御シーケンスで指定されています（付録3のエスケープシーケンスと制御シーケンスの表を参照）。

			付録3のエスケープシーケンスとコントロールシーケンスの表)。
DCS	0x90	ESC P	デバイスコントロールストリング -- デバイスコントロールストリングの開始クオリファイアとして使用される。
CSI	0x9b	ESC [Control Sequence Introducer • • • 制御シーケンスのリーダーコードとして使用される。
ST	0x9c	ESC eldest	String Terminator -- DCS文字列の終了修飾子として使用される。

A3 エスケープとコントロール・シー ケンス

配列と名称	説明
ESC (Ps or ESC) Ps 文字セットの選択	SCS>Select Character Set)・・・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を指定できます。 セットを指定します。 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。 端末は最大で254種類の文字を表示することができますが、端末のROMには127種類の表示文字しか保存されていません。 残りの127文字を表示するためには、文字セットROMを追加インストールする必要があります。 ある時点で、端末は94文字（1文字セット）を選択できるようになります。 したがって、端末は5つの文字セットのいずれかを使用することができます、その中には複数の文字セットに含まれるものもある。 任意の時点で、端末は2つのアクティブな文字セットを使用することができます。 コンピュータは、SCSシーケンスを使用して、任意の2つの文字セットをG0およびG1として指定することができます。 そして、1つの制御文字を使って、これらの2つのキャラクタセットを切り替えることができます。 G0文字セットの選択にはシフトイン-SI (14) 制御文字を、G1文字セットの選択にはシフトアウト-SO (15) 制御文字を使用することができます。 指定された文字セットは、端末が別のSCSシーケンスを受信するまで、現在の文字セットとして使用されます。
ESC [Pn A カーソルアップ (ターミナル<->ホスト)	Cursor Up (CUU) -- CUU 制御シーケンスは、カーソルを上に移動させますが、列の位置は変更していません。 移動する文字位置の数はパラメータで決まります。 パラメータが 'Pn' の場合、カーソルは 'Pn' 行目まで移動します。 カーソルは最大で最上段まで移動します。 なお、'Pn' は ASCII の数値変数です。 パラメータを選択しない場合や、パラメータ値が 0 の場合は、端末はパラメータ値を 1 とみなします。
ESC [Pn B または ESC [Pn e カーソルダウン(タ ーミナル<->ホス ト)	Cursor Down (CUD) -- CUD 制御シーケンスは、カーソルを下に移動させますが、列の位置は変更されません。 移動する文字位置の数はパラメータで決まります。 パラメータが 1 または 0 の場合、カーソルは 1 行下に移動します。 パラメータが 'Pn' の場合、カーソルは 'Pn' 行下に移動します。 カーソルは最大で最下行まで移動します。
ESC [Pn C または ESC [Pn a カーソル移動 (タ ーミナル<->ホス ト)	Cursor Forward (CUF) -- CUF 制御シーケンスは、現在のカーソルを右に移動させます。 これは移動位置の数は、パラメータで決定されます。 引数が「1」または「0」の場合、1 文字分の位置を移動します。 引数の値が 'Pn' の場合、カーソルは 'Pn' 文字位置分移動します。 カーソルは最大で右ボーダーまで移動します。
ESC [Pn D カーソルを後方に 移動 (ターミナル <->ホスト)	Cursor Backward (CUB) -- CUB 制御シーケンスは、現在のカーソルを左に移動させます。 これは移動位置の数は、パラメータで決定されます。 引数が「1」または「0」の場合、1 文字分の位置を移動します。 引数の値が 'Pn' の場合、カーソルは 'Pn' 文字位置分移動します。 カーソルのは最大でも左のボーダーまでしか動きません。
ESC [Pn E カーソルが下に移動	Cursor Next Line (CNL) -- この制御シーケンスは、カーソルを 「Pn」 の最初の文字に移動させます。 の行を下に表示します。
ESC [Pn F カーソルが上に移動	Cursor Last Line (CLL) -- この制御シーケンスは、カーソルを 「Pn」 の最初の文字まで移動させます。 のラインになります。 1106
ESC [Pn G または ESC [Pn ` カーソルがライン上を 後方に移動	Cursor Horizon Absolute (CHA) -- この制御シーケンスは、カーソルを現在の行の 'Pn' 文字の位置に移動させます。

ESC [Pn;Pn f カーソルの位置	は、現在のカーソルをパラメータで指定された位置に移動させます。2つのパラメータは、行と列の値をそれぞれ示しています。値が0の場合は1と同じで、1ポジション移動したことを示します。パラメータのないデフォルトの状態では、カーソルをホームポジションに移動させる（つまり ESC [H] ）ことと同じです。
ESC [Pn d 行位置の設定	Vertical Line Position Absolute -- カーソルを現在の列の'Pn'行に移動させます。もしをクリックして最終行の下に移動すると、カーソルは最終行に留まります。
ESC [s カーソル位置の保存	Save Current Cursor Position -- このコントロールシーケンスは、DECSCと同じ効果を持ちますが、以下の点が異なります。 カーソルに表示されているページ番号は保存されません。
ESC [u カーソル位置の復元	Restore Saved Cursor Position -- この制御シーケンスは、DECRCと同じ効果を持ちますが、次の点が異なります。 の場合、カーソルは同じ表示ページにあり、カーソルが保存されている表示ページに移動していません。
ESC D インデックス	Index (IND) -- この制御シーケンスは、カーソルを1行下に移動させますが、列番号は変わりません。カーソルが下の行にある場合は、画面が1行上にスクロールします。
ESC M リバースインデックス	Reverse Index (RI) -- この制御シーケンスは、カーソルを1行上に移動させますが、列番号は変化しません。カーソルが一番上の行にある場合は、画面が1行下にスクロールすることになります。
ESC E 1行下に移動	Next Line (NEL) -- このコントロール・シーケンスは、カーソルを左端の次の行に移動します。カーソルが下の行にある場合は、画面を1行分上にスクロールさせます。
ESC 7 カーソルの保存	Save Cursor (DECSC) -- この制御シーケンスは、カーソル位置、グラフィックスを再現し、保存する文字セットを決定します。
ESC 8 カーソルの復元	Restore Cursor (DECRC) -- この制御シーケンスは、以前に保存されたカーソル位置を復帰させます。 グラフィックスを再現し、文字セットを復元することができます。
ESC [Ps; Ps; ... ; Ps m 文字属性の設定	セレクト・グラフィック・レンディション (SGR) - キャラクターの再表示と属性が影響する特性です。 文字コードを変更することなく、文字を表示することができます。制御シーケンスは、パラメータに応じて文字の表示属性を設定する。今後、端末に送信されるすべてのキャラクターは、制御シーケンスが再びキャラクターの属性をリセットするまで、ここで指定された属性を使用する。パラメータ「Ps」。 0 - 無属性（デフォルトの属性）、1 - 太くて明るい、4 - 下線、5 - 閃光、7 - 反転、22 - 太くない、24 - 下線なし、25 - 閃光なし、;30-38 前景色の設定、39 - デフォルトの前景色（白）、40-48 - 背景色の設定、49 - デフォルトの背景色（黒）。30-37、40-47は色に対応しています。黒、赤、緑、黄、青、マゼンタ、シアン、白。
ESC [Pn L ラインの挿入	Insert Line (IL) -- この制御シーケンスは、カーソルに1つ以上の空行を挿入します。カーソルは の位置は、操作完了後も変わりません。空白行を挿入すると、カーソルの下のスクロール エリアの行が下に移動します。表示ページの外にスクロールしている行は失われます。
ESC [Pn M ラインの削除	Delete Line (DL) -- この制御シーケンスは、カーソルのある行から1つまたは複数の行を削除します。 がスクロールエリアに配置されます。行が削除されると、スクロールエリアの削除された行の下の行が上に移動し、最下行に1行の空白行が追加されます。Pn」が表示ページに残っている行数よりも大きい場合、このシーケンスはこれらの残りの行を削除するだけで、スクロールエリアの外では動作しない。
ESC [Pn @ キャラクターの挿入	Insert Character (ICH) -- この制御シーケンスは、1つまたは複数のスペース文字を現在のカーソルに通常の文字属性を使用しています。Pn」は挿入される文字の数です。カーソルは、最初に挿入されたスペース文字の位置に留まります。カーソルと右の境界にある文字は右に移動します。右の境界線を超える文字は失われます。
ESC [Pn P 文字の削除	Delete Character (DCH) -- この制御シーケンスは、カーソルから「Pn」文字を削除します。の場合は 文字が削除されると、カーソルの右にあるすべての文字が左に移動します。これにより

	ヌル文字を右端に配置。その特性は、最後の左シフト文字と同じです。
ESC [Ps J 文字を消す	<p>Erase In Display (ED) -- 表示されている文字の一部または全部を消去する制御シーケンスです。</p> <p>パラメータに応じて消去操作は、他の文字に影響を与えることなく、画面上の文字を消去します。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。文字を消去する際、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」。</p> <p>0 - カーソルから画面下部の全文字までを消去、1 - 画面上部から全文字を除くカーソルまでを消去、2 - 画面全体を消去することができます。</p>
ESC [Ps K ラインでの消去	<p>Erase In Line (EL) -- カーソルのある行の一部または全部の文字を、指定した方法で消去します。</p> <p>パラメーターを表示します。消去操作は、他の文字に影響を与えることなく、画面上の文字を消去します。消去された文字は破棄されます。消去してもカーソル位置は変わりません。</p> <p>文字や線を消すことです。文字を消去すると、その文字の属性も消去されます。パラメータ「Ps」。</p> <p>0 - カーソルから行末のすべての文字まで消去、1 - 左ボーダーからカーソルのすべての文字まで消去、2 - 行全体を消去します。</p>
ESC [Pn ; Pn r 上下のマージンを設定	<p>Set Top and Bottom Margins (DECSTBM) -- この制御シーケンスは、上と下の領域をスクロール画面に表示します。スクロールマージンとは、元の文字を画面から取り去ることで、新しい文字を受け取ることができる画面上の領域のことです。この領域は、画面の上端と下端の境界線によって定義されます。最初のパラメータは、スクロール領域の開始の最初の行で、2番目のパラメータは、スクロール領域の最後の行です。デフォルトでは、画面全体が対象となります。最小のスクロールエリアは2行で、上の境界線が下の境界線よりも小さくなければなりません。カーソルは、ホームポジションに配置されます。</p>
ESC [Pn c または ESC Z デバイスの属性 (ターミナル<--> ホスト)	<p>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、デバイスマ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。<u>デバイスマトリビュート(DA) -- ホストは、パラメータなしまたはパラメータ0のデバイスマトリビュート(DA)制御シーケンス(ESC Zと同じ)を送信する。</u> 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。</p> <p>端子オプション属性 送信シーケンス なし、VT101ESC [?1;0c</p> <p>プロセッサオプション (STP) ESC [?1;1c</p> <p>アドバンス・ビデオ (AVO) VT100ESC [?1;2c</p> <p>AVOとSTPESC [?1;3c</p> <p>グラフィック・プロパティ・オプション (GPO) ESC [?1;4c</p> <p>GPOとSTPESC [?1;5c</p> <p>GPOとAVO、VT102ESC [?1;6c</p> <p>GPO、STP、AVOESC [?1;7c</p>
ESC c 初期状態へのリセット	<p>Reset To Initial State (RIS) -- 端末を初期状態（電源を入れたばかりの状態）に戻すことができます。すべてのリセットフェーズで受信した文字は失われます。これを避けるには2つの方法があります。</p> <p>1.(Auto XON/XOFF) 送信後、ホストは端末がXOFFを送信したと見なします。ホストは、XONを受信するまで、文字の送信を停止します。2.10秒以上遅らせて、端末のリセット操作が完了するのを待つ。</p>

A4 キーボードスキャンコードの最初のセ ット

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
A	1E	9E	9	0A	8A	[1A		9A
B	30	B0	`	29	89	INSERT E0, 52		E0, D2
C	2E	AE	-	0C	8C	HOME E0, 47		E0, 97
D	20	A0	=	0D	8D	PG UP E0, 49		E0, C9
E	12	92	¥	2B	AB	DELETE E0, 53		E0, D3
F	21	A1	BKSP	0E	8E	END E0, 4F		E0, CF
G	22	A2	スペース	39	B9	PG DN E0, 51		E0, D1
H	23	A3	TAB	0F	8F	上矢印 E0, 48		E0, C8
I	17	97	CAPS	3A	BA	左矢印 E0, 4B		E0, CB
J	24	A4	左SHFT	2A	AA	下矢印 E0, 50		E0, DO
K	25	A5	左 CTRL	1D	9D	右矢印 E0, 4D		E0, CD
L	26	A6	左側のGUI	E0, 5B	E0, DB	NUMロック 45		C5
M	32	B2	左ALT	38	B8	KP / E0, 35		E0, B5
N	31	B1	右SHFT	36	B6	KP * 37		B7
O	18	98	右 CTRL	E0, 1D	E0, 9D	KP - 4A		CA
P	19	99	右のGUI	E0, 5C	E0, DC	KP + 4E		CE
Q	10	90	右ALT	E0, 38	E0, B8	KP ENTER E0, 1C		E0, 9C
R	13	93	APPS	E0, 5D	E0, DD	KP . 53		D3
S	1F	9F	ENTER	1C	9C	KP 0 52		D2
T	14	94	ESC	01	81	KP 1 4F		CF
U	16	96	F1	3B	BB	KP 2 50		D0
V	2F	AF	F2	3C	BC	KP 3 51		D1
W	11	91	F3	3D	BD	KP 4 4B		CB
X	2D	AD	F4	3E	BE	KP 5 4C		CC
Y	15	95	F5	3F	BF	KP 6 4D		CD
Z	2C	AC	F6	40	C0	KP 7 47		C7
0	0B	8B	F7	41	C1	KP 8 48		C8
1	02	82	F8	42	C2	KP 9 49		C9
2	03	83	F9	43	C3] 1B		9B
3	04	84	F10	44	C4	; 27		A7
4	05	85	F11	57	D7	' 28		A8
5	06	86	F12	58	D8	, 33		B3
6	07	87	PRNT SCRN	E0, 2Aで す。 E0, 37	E0, B7で す。 EO, AA	. 34		B4
7	08	88	スクロール	46	C6	/ 35		B5
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。

注2：表中のKP - KeyPadは、テンキー上のキーを表す。 注3：表中の色付き部分は全て拡張キーです。