

# 13 Memory Management (mm)

In the Intel 80X86 architecture system, the Linux kernel's memory management program uses paging management. It uses the page directory and page table structure to handle the application and release of memory from other parts of the kernel. Memory management is performed in units of memory pages, and a memory page refers to 4K bytes of physical memory with consecutive addresses. The page directory entry and page table entry allow you to address and manage the usage of the specified memory page. There are three files in the memory management directory of Linux 0.12, as shown in Listing 13-1:

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

Among them, the page.s assembly file is relatively short, it only contains the memory page exception interrupt service program (INT 14), mainly to achieve the processing of page fault and page write protection. Memory.c is the core program for memory page management. It is used for memory initialization operations, page directory and page table management, and other parts of the kernel for memory application processing. The swap.c program is used for memory page exchange management, which mainly includes exchange mapping bitmap management functions and switching device access functions.

## 13.1 Main Functionalities

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

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

メモリのアドレスを指しているのかを明確に区別する必要があります。

Intel 80X86 CPU Protected Mode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。については

13.1.1 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。

### 13.1.2 Memory paging mechanism

インテル80X86システムでは、図13-1に示すように、メモリページディレクトリテーブルとページテーブルからなる2階層のテーブルでメモリページング管理を行っています。ページディレクトリテーブルとページテーブルは、以下の図13-4に示すように同じ構造をしており、テーブルの項目構造も同じです。ページディレクトリテーブルの各エントリ（ページディレクトリエントリと呼ぶ、4バイト）は、ページテーブルのアドレスに使用され、各ページテーブルエントリ（4バイト）は、物理メモリのページを指定するのに使用される。したがって、ページディレクトリエントリとページテーブルエントリが指定されると、対応する物理メモリページを一意に決定することができる。ページディレクトリテーブルは1ページ分のメモリを占有するため、最大1024個のページテーブルを指定することができ、各ページテーブルも1ページ分のメモリを占有するため、1つのページテーブルも最大1024個の物理メモリページを指定することができます。したがって、32ビットの80X86CPUでは、ページディレクトリテーブルでアドレス指定されたすべてのページテーブルは、合計で $1024 \times 1024 \times 4096 = 4G$ 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。

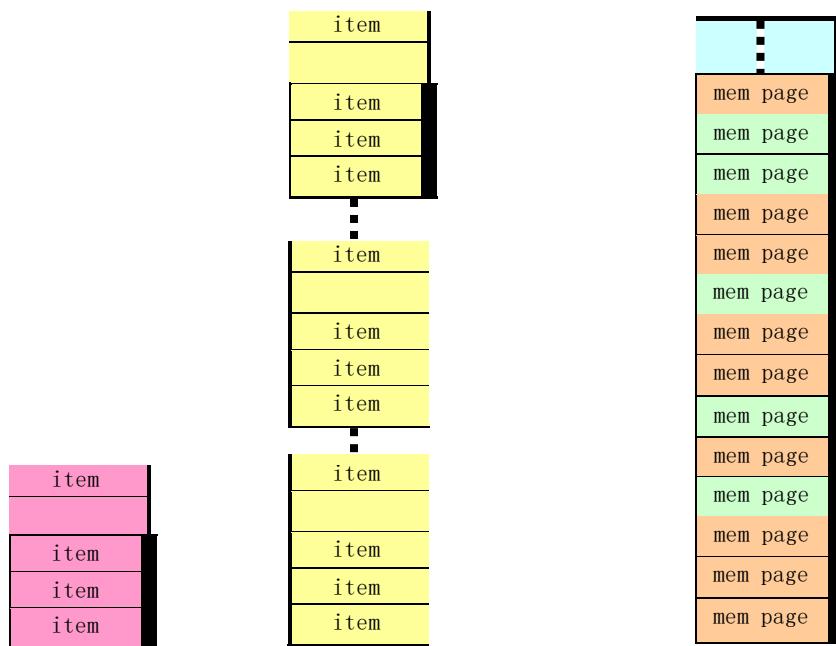


図13-1 ページディレクトリテーブルとページテーブルの構造図

For user processes or other parts of the kernel, a linear address is used when applying for memory. So, how does a linear address use these two tables to map to a physical address? In order to use the paging mechanism, a 32-bit linear address is divided into three parts, which are used to specify a page directory entry, a page table

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。

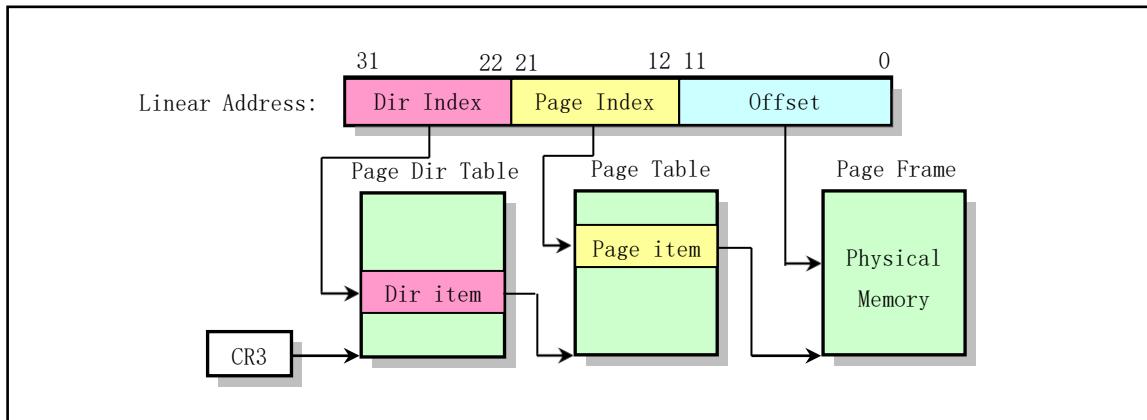


図13-2 リニアアドレス変換の模式図

Bits 31-22 of the linear address are used to determine the directory entries in the page directory; bits 21-12 are used to address the page table entries in the page table specified by the page directory entry, and the last 12 bits are used as the offset address in the physical memory page specified by the page entry.

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。

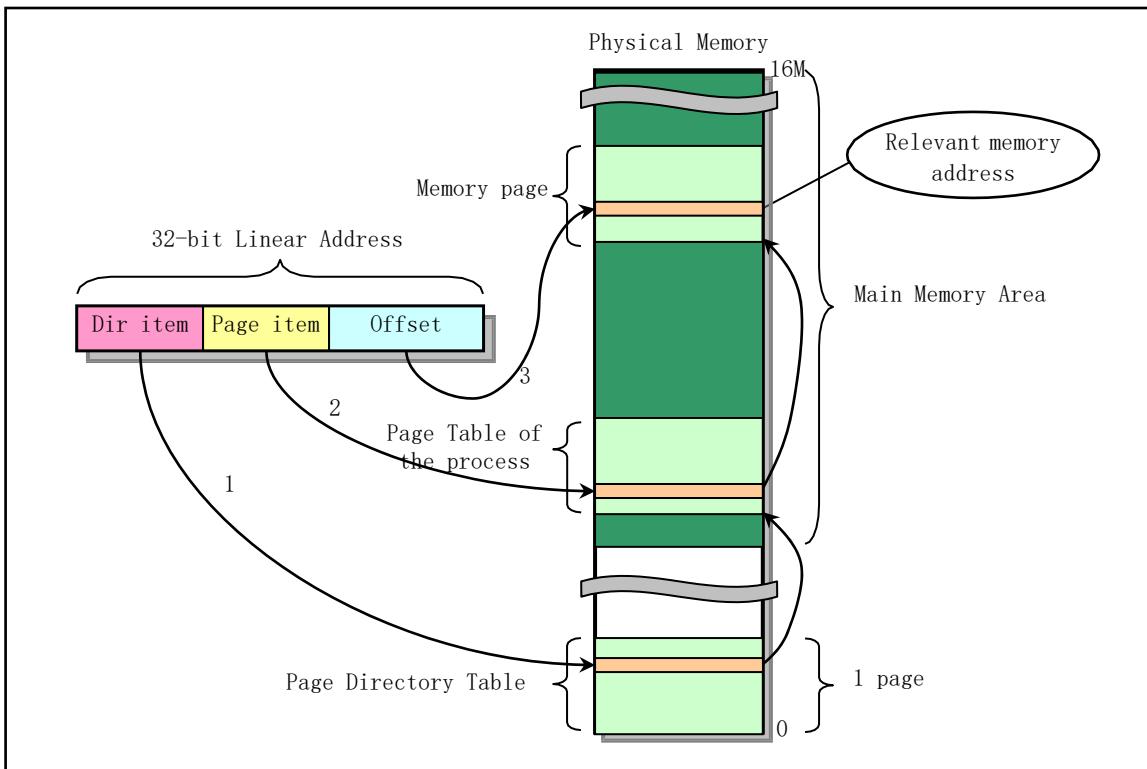


図13-3 リニアアドレスに対応する物理アドレス

Multiple page directory tables can exist simultaneously in a system, and only one page directory table is available at a time. The currently used page directory table is determined by the CPU's register CR3, which stores the physical memory address of the current page directory table. But in the Linux kernel discussed in this book, the kernel code and all processes share a single page directory table.

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

The page frame address specifies the physical start address of a page of memory. Because the memory page is on the 4K address boundary, its lower 12 bits are always 0, so the lower 12 bits of the entry can be used for other purposes. In a page directory table, the page frame address of the table entry is the start address of a page table; in the second level page table, the page frame address of the page table entry contains the physical memory page address of the desired memory operation.

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。

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

### 13.1.3 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。

#### 13.1.4 Physical Memory Allocation and Management

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

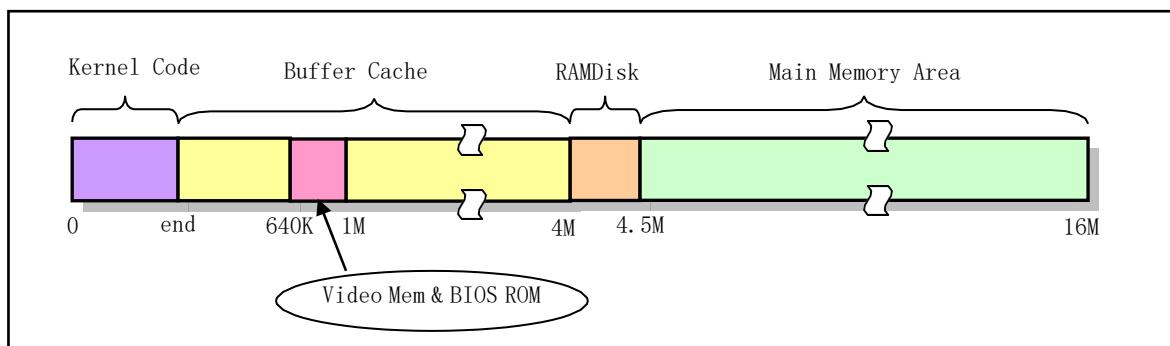


図13-5 メインメモリ領域の模式図

In Chapter 6, Booting System, we already know that Linux's page directory and page table are set in the program head.s. The head.s program stores a page directory table at physical address 0, followed by four page tables. These four page tables will be used for mapping operations in the memory area occupied by the kernel. Since the code and data for task 0 are included in the kernel region, task 0 also uses these page tables. Other derived processes will request memory pages in the main memory area to store their own page tables. The two programs in this chapter are used to manage these tables to achieve the allocation of memory pages in the main memory area.

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

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

### 13.1.6 Linear address space allocation

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

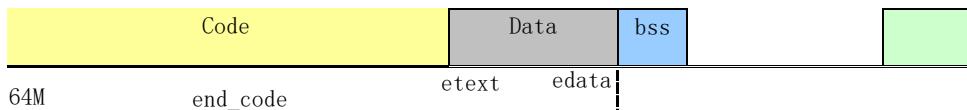


図13-6 論理アドレス空間におけるプロセスの分布

The logical address space occupied by each process starts from the location of  $nr*64MB$  in the linear address space ( $nr$  is the task number), and the logical address space occupies a range of 64MB. The last part of the environment parameter data block is up to 128K in length, and its left side is the starting stack pointer. 另外, In the figure, bss is the data segment that is not initialized by the process. The first page of the bss segment is initialized to all zeros when the process is created.

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

カーネルのメモリ管理コードでは、プログラムの論理アドレス（または仮想アドレス）、CPUのリニアアドレス、物理メモリアドレスの間の変換操作が頻繁に発生します。例えば、ページテーブルを

コピーする際には、与えられたページディレクトリエントリ（PDE）を線形変換して、対応するページテーブル（PT）の物理メモリアドレスを得る必要があります。コピーオンライト操作に関しては、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. Conversion of address to corresponding page address

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

---

```
Page_addr = addr & 0xfffff000;
```

---

In these three address spaces, their page address calculation process is the same. Below we explain the transformation calculation method for the page address.

### 2. Linear address and logical address decomposition

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

---

~~ページディレクトリエントリのエントリ番号のPDE\_No = (addr >> 22) & 0x3ff, ページテーブルエントリのPTE\_No = (addr >> 12) & 0x3ff, ページ内オフセットのOffset = (addr & 0x1ff)~~

---

Similarly, when performing the address translation operation, we can also regard the logical address 'Vaddr' in the process logical address space as being composed of these three parts "logically". Just when mapping them to the CPU linear address space, you need to add the corresponding part of the program code base address 'Base' in the linear address space:

---

~~Logic\_Base\_No = (Vaddr >> 22) & 0x3ff, Logic\_Offset = (Vaddr & 0x1ff) です。~~

---

Since in this kernel, each process allocates a linear address space in units of 64 MB in length, the page table entry number and the in-page offset value corresponding to the code base address of each process are both 0, so we only need to consider the page directory entry number of the code base address during the conversion process. In the actual code, the code base address 'Base' of the process 'p' is 'p->start\_code', so when a logical address 'Vaddr' in the program corresponds to the linear address space, the three parts of the address are:

---

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. Program page logical address to physical address conversion

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

---

```

PDE = PDT_Base + (PDE_No << 2);
= 0 + (PDE_No << 2).
= (((p->start_code >> 22) + (Vaddr >> 22)) << 2) となります。
= ((p->start_code >> 20) & 0xffc) + ((Vaddr >> 20) & 0xffc) となります。

```

Referring to Figure 13-4, we can get the physical address PT of the corresponding page table from the contents of the directory entry, and add the offset of the entry in the page table to obtain the physical address (pointer) of the page table entry PTE. :

---

```

PT = (*PDE) & 0xfffff000;
PTE = PT + (PTE_No <<
2)
= PT + (((Vaddr >> 12) & 0x3ff) << 2).
= ((*PDE) & 0xfffff000) + ((Vaddr >> 10) & 0xffc) となります。

```

Bits 31-12 of the page table entry are the physical page frame addresses. Therefore, the physical page address corresponding to the logical address 'Vaddr' of the final program is:

---

PPAddr = (\*PTE) & 0xfffff000です。

### 4. Linear address to physical address conversion

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。

We can get the physical address PT of the corresponding page table from the contents of the directory entry, and add the offset value of the entry in the page table to obtain the physical address PTE (pointer) of the page table entry:

---

```
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);
```

Therefore, the actual physical page address corresponding to the linear address 'Laddr' is 'PPaddr', and the corresponding physical address 'Paddr' is the physical page address plus the offset within the page, as shown below:

---

```
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;
```

### 13.1.8 Page-fault exception handling

In the state where the paging mechanism (PG=1) is enabled, if the CPU detects the following conditions during the conversion of the linear address to the physical address, it will cause a page-fault exception interrupt INT 14:

- The presence bit (P) in the page directory entry or page table entry used in the address translation process is equal to 0, indicating that the page table or the page containing the operand does not exist in physical memory;
- The current execution code does not have sufficient privileges to access the specified page, or the user mode code writes a read-only page, and so on.
- ページ例外処理手順は、中断されたプログラムやプロセスをページが存在しない状態から回復して再起動することができ、プログラムの実行の継続性に影響を与えません。また、特権侵害を受けたプログラムやタスクを再起動することもできますが、特権侵害の原因となった問題が修正されない場合があります。このとき、CPUはページ・フォルト例外ハンドラに以下の2つの側面を提供し、エラーの診断と修正を行っています。
  - An error code on the stack. The format of the error code is a 32-bit long word, but only the lowest 3

bits are useful. Their names are the same as the last three bits in the page table entry (U/S, W/R, P). Their meanings and roles are:

- ◆ Bit 0 (P), the exception is caused by a not-present page or violating access privileges. P=0, indicating that the page does not exist; P=1 indicates that the page-level protection privilege is

- ◆ を破った。
- ◆ Bit 1 (W/R), the exception is due to a memory read or write operation. W/R=0, indicating that it is caused by a read operation; W/R=1, indicating that it is caused by a write operation.
- ◆ Bit 2 (U/S), the code level at which the CPU executes when an exception occurs. U/S=0, the CPU was executing at supervisor mode; U/S=1, CPU was executing at user mode.
- The linear address in control register CR2. The CPU will store the linear address that generated the exception in CR2. The page fault exception handler can use this address to locate the relevant page directory and page table entry. If another page exception is allowed to occur during the execution of the page exception handler, the handler should push CR2 onto the stack.

13.1.9 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。

### 13.1.10 Copy-on-write mechanism

Copy-on-writeは、データのコピーを延期または回避する方法です。このとき、カーネルはプロセスのアドレス空間全体のデータをコピーするのではなく、親プロセスと子プロセスが同じコピーを共有できるようにします。プロセスAが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・オン・ライトの仕組みです。

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

そのため、プロセスが自身の仮想アドレス範囲内で書き込みを行う場合には、「書き込み操作→ページフォルト例外→書き込み保護例外の処理→書き込み操作命令の再実行」という、上記のパッシュなコピー・オン・ライトの操作が行われます。カーネルコードの場合、プロセスがシステムコールを呼び出すなど、プロセスの仮想アドレス範囲内で書き込み操作が行われた場合、システムコールがプロセスのバッファにデータをコピーすると、カーネルは関数verify\_area()を呼び出します。この関数は、

まずメモリページ検証関数 `write_verify()`を積極的に呼び出し、ページ共有状態があるかどうかを判断します。ページ共有状態があれば、そのページのコピーオンライト操作が行われます。

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

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

### 13.1.11 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。

### 13.1.12 Load on demand mechanism

`execve()`システムコールでファイルシステム上の実行形式イメージファイルをロードする際、カーネルはCPUの4Gリニアアドレス空間で対応するプロセスに64MBの連続した空間を割り当て、その環境やコマンドラインパラメータのために一定量の物理メモリページを適用・確保します。それ以外に、実行ファイルに割り当てられた物理メモリページは実はありません。もちろん、実行イメージファイルのコードやデータをファイルシステムから読み込むこともできません。したがって、エントリー実行ポイントからプログラムが実行を開始すると、すぐにCPUにページフォルト例外（実行ポインタがあるメモリページが存在しない）が発生する。このとき、カーネルのページ・フォ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ブロックデバイスのI/O操作を待つ必要がなく、`execve()`システムコールを呼び出した直後に実行プログラムの実行を開始することができる。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かう。ただし、この手法では、オブジェクトファイルをロードする形式に一定の要件があります。それは、実行されるファイルのオブジェクトフォーマットがZMAGICタイプであること、つまりデマンドページングフォーマットのオブジェクトファイルフォーマットであることです。このオブジェクトファイル形式では、プログラムのコードセグメントとデータセグメントがページ境界から格納され、カーネルがコードやデータの内容を1ページ単位で読み取れるようになっている。

## 13.2 memory.c

### 13.2.1 Function

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

メモリ管理の初期化の過程で、カーネルコードはまずメモリの数を計算します。

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ことはできない。2944項目の値は0にクリアされ、メモリマネージャによる割り当てが可能となる。

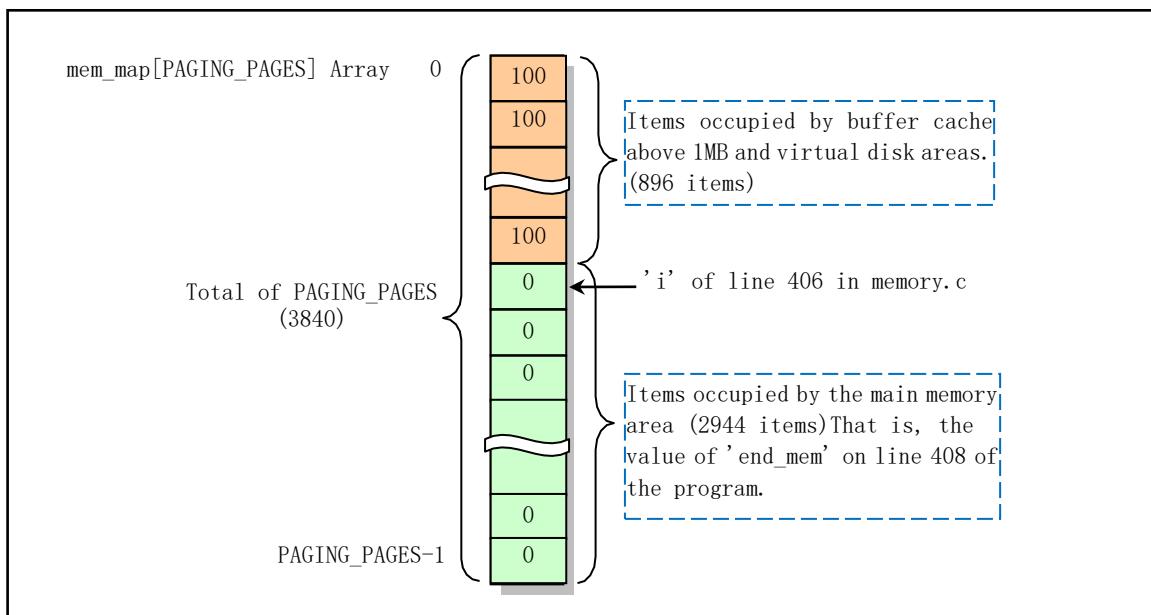


図13-7 16MBのメモリと512KBのvdiskを使用した`Mem_map[]`アレイの初期化

For the management of the process virtual address (or logical address), the kernel uses the segment management mechanism of the processor to implement, and the mapping relationship between the physical memory page and the linear address is handled by modifying the contents of the page directory and the page table entry. The following is a detailed description of several main functions provided in the program.

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

開始アドレスが返されます。

free\_page()は、指定されたアドレスの物理メモリのページを解放するために使用されます。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、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トリの内容を変更するだけでなく、各ページテーブルのすべてのページテーブルアイテムに対応する物理メモリページを解放または占有する。

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

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

`put_page()`関数は、指定された物理メモリページを指定されたリニアアドレスにマッピングする

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

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

do\_no\_page()は、ページ例外発生時に呼び出されるページフォルト関数です。do\_no\_page()は、ページ例外発生時に呼び出されるページフォルト関数で、まず、プロセス空間のプロセスベースアドレスに対する指定されたリニアアドレスのオフセット長値を決定します。それがコード+データの長さよりも大きい場合や、プロセスの生成が始まったばかりの場合は、物理メモリのページを申請し、プロセスのリニアアドレスにマッピングします。そうでなければ、ページ共有操作を試みます。そうでなければ、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加えた分だけ、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 Code annotation

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 * demand-loading started 01.12.91 - seems it is high on the list of
9 * things wanted, and it should be easy to implement. - Linus
10 */
11
12 /*
13 * Ok, demand-loading was easy, shared pages a little bit trickier. Shared
14 * pages started 02.12.91, seems to work. -Linus.
15 *
16 * Tested sharing by executing about 30 /bin/sh: under the old kernel it
17 * would have taken more than the 6M I have free, but it worked well as
18 * far as I could see.
19 *
20 * Also corrected some "invalidate()"s - I wasn't doing enough of them.
21 */
22
23 /*
24 * Real VM (paging to/from disk) started 18.12.91. Much more work and
25 * thought has to go into this. Oh, well..
26 * 19.12.91 - works, somewhat. Sometimes I get faults, don't know why.
27 *           Found it. Everything seems to work now.
28 * 20.12.91 - Ok, making the swap-device changeable like the root.
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
// manipulation function prototypes.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロです。
// descriptors/interrupt gates, etc. is defined.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// of the initial task 0, and some embedded assembly function macro statements about the
// descriptor parameter settings and acquisition.
// <linux/head.h> ヘッ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。

// along with several selector constants.

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

// used functions of the kernel.

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>
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
50 49 /*
51 * Free a page of memory at physical address 'addr'. Used by
52 * 'free_page_tables()'
52 */
// 物理アドレス「addr」を起点に1ページ分のメモリを解放します。
// 物理アドレスの1MB以下のメモリ空間は、カーネルのプログラムやバッファに使用されており
// をアロケーションページのメモリ空間として使用することはできません。したがって、パラメータ
'addr' は
// を1MB以上に設定する必要があります。
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
// カーネルのプログラムやキャッシュの範囲内であることを意味し、処理されません；'addr'
// がシステム内の物理メモリの最上位よりも大きいか同じであれば
// エラーメッセージが表示され、カーネルが動作しなくなりました。
55
56     if (addr < LOW_MEM) return; // LOW_MEM = 1MB, defined in include/linux/mm.h, 30.
57     if (addr >= HIGH_MEMORY)

```

---

58                  panic("trying to free nonexistent page");  
  バラメーター'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ます。  
  // 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。  
  // "Page number = (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

```
addr -= LOW MEM;  
/  
その後、エラーメッセージが表示され、オーバルカ停止します。  
。  
  
58  
59     addr >= 12;                      // divided by 4096.  
60     if (mem_map[addr]--) return;  
61     mem_map[addr]=0;  
62     panic("trying to free free page");  
63 }  
64  
65 /*  
66 * This function frees a continuos block of page tables, as needed  
67 * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.  
68 */  
    ////////////////////////////////////////////////////////////////////  
    // 与えられたリニアアドレスとリミットレンジス (ページテーブルの数) に応じて、フリーの  
    // メモリブロックを使用し、テーブルのエントリを空にします。  
    // ページディレクトリテーブルは、物理アドレス0の先頭にあります。そこには、1024個の  
    // のエントリーがあり、それぞれが4バイトで、合わせて4Kバイトになります。各ディレクトリエントリ  
    // は  
    // は、1つのページテーブルに対応しています。カーネルのページテーブルは、物理アドレス0x1000から  
    // 始まる  
    // (ディレクトリのテーブル空間に続く) で、4つのページテーブルを持っています。各ページテーブル  
    // には1024のエントリがあります。  
    // それぞれが4バイトであるため、4KB (1ページ) のメモリを占めることになります。ただし  
    // カーネルコードのプロセス0と1では、他のプロセスのページテーブルが占めるページは  
    // プロセスが作成されると、カーネルはメインメモリー領域から // プロセスを要求します。  
    // ページテーブルの各エントリは物理メモリの1ページに対応しているので、ページテーブルは以下のようにマッピングできます。
```

// 4MBの物理メモリを搭載しています。

// パラメータ : from - 線形ベースアドレス、 size - 解放されるバイト数。

71 69 int free\_page\_tables(unsigned long from,unsigned long size) 70 {.

72        unsigned long \*pg\_table;

73        unsigned long \* dir, nr;

73

// まず、パラメータ'from'で指定されたリニアベースアドレスが4MB境界にあるかどうかをチェックします。

// 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  
// 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。

// 次に、以下の長さに対応するページディレクトリエントリの数を計算します。

// 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。

// 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。

// 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

// キャリー付き整数倍の結果。

// 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。

// 次に、与えられたリニアベースアドレスに対応する開始ディレクトリエントリは

// 算出されます。

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

// 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

// ディレクトリエントリポインタは「ディレクトリエントリ番号 << 2」で、「(from >> 20)」となり

ます。"&0ffc"

// ディレクトリエントリポインタが有効範囲内にあることを確認します。

74        if (from & 0xffff)

panic ("free\_page\_tables called with wrong alignment");

75        if (!from)

panic ("Trying to free up swapper memory space");

76        size = (size + 0x3fffff) >> 22;

77        dir = (unsigned long \*) ((from>>20) & 0xffc); /\* pg\_dir = 0 \*/

// この時点での「サイズ」は、解放する必要のあるページテーブルの数、つまり

ページのディレクトリエントリの//、'dir'は開始ディレクトリエントリポインタです。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
イレクトリ
// 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ン
トリが
// 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
// 次のページディレクトリエントリが処理されます。最後に、ページ変換キャッシュを更新し
// 0を返します。
80     for ( ; size-->0 ; dir++) {
81         if (!(1 & *dir))
82             continue;
83         pg_table = (unsigned long *) (0xfffff000 & *dir); // get page table addr.
84         for (nr=0 ; nr<1024 ; nr++) {
85             if (*pg_table) {
86                 if (1 & *pg_table) // free the page if valid.
87                     free\_page(0xfffff000 & *pg_table);
88                 else // free the page in swap device.
89                     swap\_free(*pg_table >> 1);
90                 *pg_table = 0; // reset the page content.
91             }
92             pg_table++; // points to next page.
93         }
94         free\_page(0xfffff000 & *dir); // free the page of the page table.
95         *dir = 0; // reset the directory entry.
96     }
97     invalidate(); // refresh CPU page transform cache.
98     return 0;
99 }

100 /*
101 */
102 /*
103 * Well, here is one of the most complicated functions in mm. It
104 * copies a range of linear addresses by copying only the pages.
105 * Let's hope this is bug-free, 'cause this one I don't want to debug :-
106 */
107 /*
108 * Note! We don't copy just any chunks of memory - addresses have to
109 * be divisible by 4Mb (one page-directory entry), as this makes the
110 * function easier. It's used only by fork anyway.
111 */
112 /*
113 * NOTE 2!! When from==0 we are copying kernel space for the first
114 * fork(). Then we DONT want to copy a full page-directory entry, as
115 * that would lead to some serious memory waste - we just copy the
116 * first 160 pages - 640kB. Even that is more than we need, but it

```

115 \* doesn't take any more memory - we don't copy-on-write in the low  
116 \* 1 Mb-range, so the pages can be shared with the kernel. Thus the  
117 \* special case for nr=xxxx.  
117 \*/

//// ページディレクトリエントリとページテーブルエントリをコピーします。

// この関数は、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。

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 {

121     unsigned long \* from\_page\_table;  
122     unsigned long \* to\_page\_table;  
123     unsigned long this\_page;  
124     unsigned long \* from\_dir, \* to\_dir;  
125     unsigned long new\_page;  
126     unsigned long nr;

// コードはまず、ソースアドレス「from」とデスティネーションアドレスの有効性を検出します。  
// パラメータで与えられた「to」。送信元と送信先の両方のアドレスが  
// 4Mbのメモリ境界アドレス。この要件は、ページテーブルの1024エントリが  
// 4Mbのメモリを管理できます。送信元アドレスと送信先アドレスは、この条件を満たすだけで  
// ページテーブルのエントリが、ページの最初のエントリからコピーされることを保証する要件  
// テーブルを取得し、新しいページテーブルの元のエントリがすべて有効になります。そして、スタート  
を取得します。  
// 送信元アドレスと送信先アドレスのディレクトリエンタリーポインター (from\_dir と to\_dir)。その後  
// が占めるページテーブルの数 (つまりディレクトリエンタリの数) を計算します。

//       if ((from&0x3fffff) || (to&0x3fffff))

パラメータで指定された「サイズ」に応じて、コピーされるメモリブロ

127  
 ック  
 。  
 128        panic("copy\_page\_tables called with wrong alignment");  
 129        from\_dir = (unsigned long \*) ((from>>20) & 0xffff); /\* pg\_dir = 0 \*/  
 130        to\_dir = (unsigned long \*) ((to>>20) & 0xffff);  
 131        size = ((unsigned) (size+0x3fffff)) >> 22;  
  
 // 送信元の開始ディレクトリエンタリポインタfrom\_dirを取得した後、送信先の開始  
 // ディレクトリエンタリポインタ to\_dir と、コピーするページテーブルの数を指定すると  
 // 各ページのディレクトリエンタリに1ページ分のメモリを適用して対応する保存を開始  
 // ページテーブル、およびページテーブルエンタリのコピー操作を開始します。で指定されたページテーブルが存在しない場合は  
 // 宛先ディレクトリ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。ソース  
 ディレクトリの  
 // エントリーが無効、つまり指定されたページテーブルが存在しない (P=0) 場合は、次のページ  
 // デ  
 イ  
 レ  
 ク  
 ト  
 リ  
 エ  
 ン  
 ト  
 リ  
 は  
 ル  
 一  
 プ  
 し  
 続  
 け  
 ま  
 す  
 。  
 132  
 133        if (1 & \*to\_dir)  
 134            panic("copy\_page\_tables: already exist");  
 135        if (!(1 & \*from\_dir))  
 136            continue;  
  
 // 現在のソースディレクトリエンタリとデスティネーションエンタリが正常であることを確認した後。  
 // ソースのディレクトリエンタリにあるページテーブルアドレス from\_page\_table を取得します。するために  
 // 宛先のディレクトリエンタ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。  
 // 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  
 // 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。  
 // その後、-1の値を返して終了します。  
 138            from\_page\_table = (unsigned long \*) (0xffff000 & \*from\_dir);  
 139            if (!(to\_page\_table = (unsigned long \*) get\_free\_page()))  
 140                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  
// がよい。

// 640KBの物理メモリの先頭に対応しています。

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。

```
142         *to_dir = ((unsigned long) to_page_table) | 7;  
141         nr = (from==0)?0xA0:1024;                                // 0xA0 = 160
```

// この時点では、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。

// 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。

// 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ーされない

// そして、次のエントリーが処理されます。

```
143         for ( ; nr-- > 0 ; from_page_table++, to_page_table++) {  
144             this_page = *from_page_table;  
145             if (!this_page)  
146                 continue;
```

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

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

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

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

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

// ページエントリになります。それ以外の場合は、ページテーブルエントリでR/Wフラグ（ビット1は0に設定）がリセットされます

//つまり、ページテーブルエントリに対応するメモリページが読み取り専用に設定され、その後

```
// ページ                     if (!(1 & this_page)) {  
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
```

```

148                     return -1;
149             read_swap_page(this_page>>1, (char *) new_page);
150             *to_page_table = this_page;
151             *from_page_table = new_page | (PAGE_DIRTY | 7);
152             continue;
153         }
154         this_page &= ~2;
155         *to_page_table = this_page;

// ページテーブルエンtries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行目のコメントの意味は、ソース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  
if (this\_page > LOW MEM) {

```

156
157                         *from_page_table = this_page; // set source read-only too.
158                         this_page -= LOW MEM;
159                         this_page >>= 12;
160                         mem map[this_page]++;
161                     }
162                 }
163             }
164             invalidate(); // refresh CPU page transform cache.
165             return 0;
166         }
167
168 /*
169  * This function puts a page in memory at the wanted address.
170  * It returns the physical address of the page gotten, 0 if
171  * out of memory (either when trying to access page-table or
172  * page.)
173 */

```

//// 物理的なメモリページを、リニアアドレス空間の指定された位置にマッピングする。  
// あるいは、リニアアドレス空間の指定されたアドレスのページがメインメモリにマッピングされる  
// エリアページです。この関数を実装する主な仕事は、指定されたページの情報を設定することです。  
// 関連するページディレクトリエントリとページテーブルエントリの中のページ。この関数が呼び出さ  
れるのは

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

// 現在では、ページ不在のためにページテーブルが変更された場合には  
// CPUのページ変換バッファ (TLB) をリフレッシュします。  
// ページテーブルエントリーフラグPが0から1に変更された場合、無効なページエントリーは  
// バッファリングされていないので、無効なページテーブルのエントリが変更されても、リフレッシュ  
する必要はありません。

ここでは、Invalidate()関数を呼び出さずに、 // を表示しています。パラメータの 'page' は、次のような  
ポインタです。

// 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。

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

```

176     unsigned long tmp, *page_table;
177
178 /* NOTE !!! This uses the fact that _pg_dir=0*/
179 
```

```

// ここではまず、パラメータで指定された物理メモリページ「page」の有効性を判断します。
// ページ位置がLOW_MEMよりも低い場合や、システムが実際に
// メモリハイエンドHIGH_MEMORYを含む。LOW_MEM=1MB」が定義されています。
// include/linux/mm.hファイルでは、メインメモリの最低開始位置である
// の領域になります。システムの物理メモリが6MB以下の場合、メインメモリ領域の
// は、LOW_MEMから直接始まります。次に、そのページが適用されたものであるかどうかを確認しま
す。
// 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。
180 // そうでない場合は、警告が必要です。
181     if (page < LOW MEM || page >= HIGH MEMORY)
182         printk( "Trying to put page %p at %p\n", page, address);
183     if (mem_map[(page-LOW MEM)>>12] != 1)
184         printk( "mem_map disagrees with %p at %p\n", page, address);

// 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
// ページディレクトリテーブルの//を算出し、そこからページテーブルアドレスを取得する。

```



```
217     else {
218         if (!(tmp=get free page()))
219             return 0;
220         *page_table = tmp|7;
221         page_table = (unsigned long *) tmp;
222     }
```

```

223     page_table[(address>>12) & 0x3ff] = page | (PAGE_DIRTY | 7);
224 /* no need for invalidate */
225     return page;
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（コピーオンライト）時には
// カーネルがプロセスを生成する際に、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。
// 入力パラメータは、ページテーブルエントリポインタです。
228 void un_wp_page(unsigned long * table_entry)
229 {
230     unsigned long old_page, new_page;
231
// まず、パラメータのページテーブルエントリの物理ページの位置（アドレス）を取得します。
// をクリックし、そのページが共有ページかどうかを判断します。元のページのアドレスがより大きい
場合
// メモリLOW_MEM（メインメモリエリア内）の下限よりも、ページ内のその値が
// マッピングバイト配列が1（ページが1回しか参照されないことを示す。
// 共有されている）の場合は、ページテーブルエントリにR/Wフラグ（書き込み可能）が設定され、ペ
ージ変換
// 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使用
されていない場合は
// 現時点では、カーネル内のプロセスではないため、属性を直接変更して
// 書き込み可能であり、新しいページを再適用する必要はありません。
232
233     old_page = 0xfffffff000 & *table_entry; // get physical page address in the entry.
234     if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
235         *table_entry |= 2; // set R/W flag.
236         invalidate();
237     }
// そうでない場合は、プロセスのためにメインメモリ領域の空きページを申請する必要があります。
// 書込み操作を行った後に、ページ共有をキャンセルした場合。元のページ位置が
// がメモリの下限よりも大きい場合（mem_map[] > 1、ページが共有されていることを意味します）に
は
// 元のページのバイト配列の値を1だけデクリメントし、内容を更新する
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。

// (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  
// が最終的に新しいページにコピーされます。

```

238     if (! (new_page = get_free_page()))
239         oom();                                // Out of Memory.
240     if (old_page >= LOW_MEM)
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 * This routine handles present pages, when users try to write
249 * to a shared page. It is done by copying the page to a new address
250 * and decrementing the shared-page counter for the old page.
251 */

```

```

251 *
252 * If it's in code space we exit with a segment error.
253 */
    //// 共有ページの書き込み保護処理を行います。
    // これは、ページの例外処理中に呼び出されるC言語の関数です。
    // ページ.sのプログラムです。関数のパラメータ error_code と address は自動的に
    // プロセスが書き込み保護されたページを書き込んだときにCPUが生成します。パラメータ error_code
    // はエラーの種類、address は例外が発生したページのリニアアドレスです。
    // 共有ページを書き込む際には、ページをコピーする必要があります（コピーオンライン）。
254 void do_wp_page(unsigned long error_code,unsigned long address) 255
{
    // 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。
    //     if (address < TASK_SIZE)
エラーメッセージが表示されます。
.
256
257         printk( "|n|rBAD! KERNEL MEMORY WP-ERR! |n|r";
258         if (address - current->start_code > TASK_SIZE) {
259             printk( "Bad things happen: page error in do_wp_page|n|r";
260             do_exit(SIGSEGV);
261         }
262 #if 0
263 /* we cannot do this yet: the estdio Library writes to code space */
264 /* stupid, stupid. I really want the libc.a from GNU */
    // リニアアドレスがプロセスのコードスペースにある場合、実行プログラムは
265 // が終了します。コードが読み取り専用のため
    if (CODE_SPACE(address))
266         do_exit(SIGSEGV);
267 #endif
    // そして、上記の関数un_wp_page()を呼び出して、ページ保護の解除を処理します。しかし、最初に
    // そのためのパラメータを用意する必要があります。そのパラメータは、ページのエントリポインタ
    // 与えられたリニアアドレス「address」のテーブルで、次のように計算されます。

```

```

// (1) 「((アドレス>>10) & 0xffc)」。のページテーブルエントリのオフセットアドレスを計算します。
// ページテーブルのリニアアドレスを指定します。なぜなら、リニアアドレス構造による。
// "(address>>12)"はページテーブルエントリのインデックスですが、各エントリは4バイトを占めるので
// 4倍した後は、「(アドレス>>12)<<2 = (アドレス>>10)&0xffc」となり、オフセットアドレスを得ることができます。
テーブル内のページテーブルエントリの//。AND "演算 &0xffc は、アドレスを制限するために使用されます。
// 1ページ内の範囲です。また、10ビットだけがシフトされるので、最後の2ビットが最上位の
// リニアアドレスの下位12ビットのうちの2ビットであり、マスクアウトされるべきものです。したがって、より多くの
// リニアアドレスのページテーブルエントリのオフセットを見つけるための明白な方法は
//     "((address>>12) & 0x3ff)<<2)"。
// (2) 「(0xfffff000 &*((アドレス>>20) &0xffc)」。のページテーブルのアドレスを取得するために使用します。
// ディレクトリのエントリーです。ここで、"((address>>20) &0xffc)"は、エントリーのオフセットを得るために使用されます。
// リニアアドレスのディレクトリテーブル内のインデックスです。アドレス>>22」はディレクトリなので
// エントリーインデックスと各アイテムは4バイトなので、4倍します：「(アドレス>>22)<<2 = (アドレス>>20)」。
// これは、ディレクトリテーブルの指定されたエントリのオフセットアドレスです。"&0xffc"が使用される
// ディレクトリエントリのインデックス値の最後の2ビットをマスクアウトするために、"*((アドレス>>20) &0xffc)"
// 指定されたページテーブルのコンテンツの中で、対応するページテーブルの物理アドレスを取得します。
// ディレクトリエントリになります。最後に、0xfffff000との"AND"は、以下のフラグビットのいくつかをマスクするために使用されます。
// ページのディレクトリ・エントリの内容（ディレクトリ・エントリの下位12ビット）です。これには

```

```

// は、より直感的に次のように表されます。
//      "(0xffffffff000 & *((unsigned long *) (((address>>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         (((address>>10) & 0xffc) + (0xffffffff000 &
270             *((unsigned long *) ((address>>20) & 0xffc))));  

271
272 }
273
    //// 書き込みページの検証。
    // ページが書き込み可能でない場合は、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
    // 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オンライトを行わず、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。
    // 次にコードは、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リー
    // 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
    // のアドレスを入力してください。
278
279     if (!(page = *((unsigned long *) ((address>>20) & 0xffc)) )&1)
280         return;
281     page &= 0xffffffff000;
282     page += ((address>>10) & 0xffc);
    // そして、ページテーブルエントリのビット1 (R/W) とビット0 (P) のフラグをチェックします。もし、ページ
    // 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
    // (コピーオンライト) が実行された場合は、何もせずにそのまま終了します。
283
284     if ((3 & *(unsigned long *) page) == 1)      /* non-writeable, present */
285         un_wp_page((unsigned long *) page);
286     return;
285 }
286

```

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

287 void get\_empty\_page(unsigned long address)

288 {

289 unsigned long tmp;

290

```
// 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。
```

```

// は問題ではなく、関数はそれを無視して正常に戻ります。
291     if (!(tmp=get_free_page()) || !put_page(tmp, address)) {
292         free_page(tmp);           /* 0 is ok - ignored */
293         oom();
294     }
295 }
296
297 /*
298 * try_to_share() checks the page at address "address" in the task "p",
299 * to see if it exists, and if it is clean. If so, share it with the current
300 * task.
301 *
302 * NOTE! This assumes we have checked that p != current, and that they
303 * share the same executable or library.
304 */
305
306 ///////////////////////////////////////////////////////////////////
307 // 現在のプロセスの与えられたアドレスのページを、与えられたプロセスと共有しようとする。
308 // 現在のプロセスは、プロセス'p'と同じ実行コードを持っています。と考えることもできます。
309 // 現在のプロセスが、fork()を実行した'p'プロセスによって生成された子プロセスであること。
310 // の操作を行うため、それらのコードの内容は最初から同じです。もし、データの内容が
311 // セグメントの内容は変更されていないので、内容も同じである必要があります。
312 // パラメータの'address'は、プロセス内の論理的なアドレスであり、論理的なページである
313 // 現在のプロセスが'p'プロセスと共有したいと考えているアドレス、pはプロセス
314 // を共有することになります。p' プロセスの 'アドレス' にあるページが存在し、まだ
315 // 修正して、現在のプロセスにpプロセスと共有させます。また、それと同時に
316 // 指定されたアドレスが申請されてページを取得したかどうかを確認する必要があります。
317 // そうすると、カーネルがおかしくなってクラッシュします。ページ共有が成功した場合は1を、失敗し
318 // た場合は0を返します。
319
320 static int try_to_share(unsigned long address, struct task_struct * p) {
321     unsigned long from;
322     unsigned long to;
323     unsigned long from_page;          // page directory entry of process 'p'
324     unsigned long to_page;           // page directory entry of current process.
325     unsigned long phys_addr;
326
327     // 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。
328     // 与えられたプロセスpと現在のプロセスをそれぞれ取得します。便宜上
329     // 計算の際には、まず、与えられたアドレスの「論理的な」ページ・ディレクトリ・エントリを見つけて
330     // います。
331     // 「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
332     // (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
333     // 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
334     // 対応する実際のページディレクトリのエントリーアドレス'from_page'と'to_page'を取得します。
335     // をそれぞれアドレス 'address' のページに設定します。
336     from_page = to_page = ((address>>20) & 0xffc);
337     from_page += ((p->start_code>>20) & 0xffc);           // page dir entry of process p.
338     to_page += ((current->start_code>>20) & 0xffc);       // page dir entry of current.
339
340     return 1;
341 }

```

```
// 2つのプロセスに対応するディレクトリ・エントリを取得した後、それらを処理する
// を別々に行います。pプロセスのエントリーの動作について説明します。その目的は
// pプロセスの'アドレス'に対応する物理ページアドレスを取得して
// 物理ページが存在し、クリーン（変更されていない、ダーティでない）であるかどうかを判断します。
このメソッドは
// は、まずディレクトリエントリの内容を取得し、次に物理的なアドレスを取得します。
// に対応するペジテーブルエントリポインタを計算して、その中のペジテーブルの
// 論理アドレス「アドレス」で、ペジテーブルエントリの内容を抽出し、一時的に
```

```

317 // physic_addrに保存されます。
318 /* is there a page-directory at from? */
319     from = *(unsigned long *) from_page;           // get content of page dir entry.
320     if (!(from & 1))                                // page table not exist ?
321         return 0;
322     from &= 0xfffffff000;                          // page table address.
323     from_page = from + ((address>>10) & 0xffc); // page table entry pointer.
324     phys_addr = *(unsigned long *) from_page;      // content of the entry.

```

// 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し、ク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。

// ページアドレスです。これは、マシン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 /\* is the page clean and present? \*/

```

324     if ((phys_addr & 0x41) != 0x01)
325         return 0;
326     phys_addr &= 0xfffffff000;                      // physical page address.
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

```

//      to = *(unsigned long *) to_page;           // content of the entry of current.

```

デ  
イ  
レ  
ク  
ト  
リ  
エ  
ン  
ト  
リ  
'to  
\_p  
ag  
e'  
の  
内  
容  
を

ページを指すように変更します。  
。

```

329
330     if (!(to & 1))
331         if (to = get_free_page())
332             *(unsigned long *) to_page = to | 7;
333         else
334             oom();
// 次に、ディレクトリエントリのページテーブルアドレスを'to'に取り、さらにオフセットアドレスを取ります。
テーブルのエントリーの // を調べ、ページテーブルのエントリーアドレスを 'to_page' に取得します。これに対して
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。
// 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
//      to &= 0xfffff000;                                // page table address.
// それをすると、カーネルが間違っている、クラッシュする
335
336     to_page = to + ((address>>10) & 0xffc);          // page table entry pointer.
337     if (1 & *(unsigned long *) to_page)
338         panic("try_to_share: to_page already exists");
// アドレス'address'に対応するクリーンで既存の物理ページを見つけた後
// をプロセスpで使用し、さらに現在のプロセスのページテーブルエントリーアドレスを決定すると
// 今からそれらを共有し始めます。使用する方法は非常に簡単で、まずページテーブルを変更します。

```

pプロセスの // エントリで、書き込み保護 (R/W=0、読み取り専用) フラグを設定してから  
// 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。  
339 pプロセスの//。  
340 /\* share them: write-protect \*/

```

341     *(unsigned long *) from_page &= ~2;
342     *(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
343     // 共有処理が成功した場合は1を返します。
344     invalidate();
345     phys_addr -= LOW_MEM;
346     phys_addr >>= 12;                      // obtain the page numberin main mem area.
347     mem_map[phys_addr]++;
348     return 1;
347 }
348
350 /* 
351 * share_page() tries to find a process that could share a page with
352 * the current one. Address is the address of the wanted page relative
353 * to the current data space.
354 */
355 * We first check if it is at all feasible by checking executable->i_count.
356 * It should be >1 if there are other tasks sharing this inode.
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。
    // ページです。'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 {
359     struct task_struct ** p;
360
    // まず、パラメータで与えられたメモリのi-node参照カウント値をチェックします。もし、i-node
    // 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、共有は一切なく、関数を直接終了するだけです。  
362     if (inode->i_count < 2 || !inode)  
363         return 0;  
// それ以外の場合は、タスク配列のすべてのタスクを検索し、ページを共有できるプロセスを見つけています。  
// 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。  
// 同じでない場合は、検索を続けてください。  
// プロセス'p'の'executable'または'library'フィールドが指定されたものと同じであれば  
// '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

364

// 共有操作が成功した場合は1、そうでない場合は0を返します。

```

365     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
366         if (!*p)                                // continue search if the item is null.
367             continue;
368         if (current == *p)                      // continue search if it's the current.
369             continue;
370         if (address < LIBRARY_OFFSET) {
371             if (inode != (*p)->executable)    // i-node of the executable file.
372                 continue;
373         } else {
374             if (inode != (*p)->library)        // i-node of the library file.
375                 continue;
376         }
377         if (try_to_share(address,*p))
378             return 1;
379     }
380 }
```

//// Page-not-present processing.

// 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、例外が発生し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。

// メモリページになります。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できません。

// 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。

381 void do\_no\_page(unsigned long error\_code,unsigned long address) 382

{

384 int nr[4];

385 unsigned long tmp;

```
386     unsigned long page;
387     int block, i;
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。
// エラーメッセージが表示されます。
389     if (address < TASK_SIZE)
```

```

390         printk( "|n|rBAD!! KERNEL PAGE MISSING|n|r");
391     if (address - current->start_code > TASK_SIZE) {
392         printk( "Bad things happen: nonexistent page error in do_no_page|n|r");
393         do_exit(SIGSEGV);
394     }
// 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。
// それにより、ページテーブルエントリの内容を取得します。エントリの内容が
// 0だが、ビットP=0であれば、ページテーブルエントリで指定された物理ページが
// swap device.
page = *(unsigned long *) ((address >> 20) & 0xffc); // content of page dir entry.

```

```

395
396     if (page & 1) {
397         page &= 0xfffff000;           // page table address.
398         page += (address >> 10) & 0xffc;    // page table entry pointer.
399         tmp = *(unsigned long *) page;      // content of page entry.
400         if (tmp && !(1 & tmp)) {
401             swap_in((unsigned long *) page); // read in from swap device.
402             return;
403         }
404     }

// 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
// プロセスベースアドレスに対するプロセス空間内のアドレスのオフセット 'tmp'、すなわち
// 対応する論理アドレスです。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
406     address &= 0xfffff000;           // page address
407     tmp = address - current->start_code; // logical address of the page.

// 論理アドレス'tmp'がライブラリイメージファイルの開始位置よりも大きい場合
// プロセスの論理空間では、欠落したページはライブラリのイメージファイルにあります。したがって、
その
// 現在のプロセスタスクからライブラリ画像ファイルのi-node 'library'を取得できる
// 構造、およびライブラリファイル内の欠落したページの開始ブロック番号'block'
// が算出されます。論理アドレス'tmp'が実行イメージの終端よりも小さければ
プロセスの//ファイルであれば、欠落しているページはプロセスの実行ファイルイメージにあるので
// 実行ファイルのi-node番号「executable」を現在のプロセスから取得可能
// タスク構造の中で欠落しているページの開始ブロック番号「block」を計算します。
// 実行ファイルイメージ。論理アドレス'tmp'が、実行ファイルイメージのアドレス範囲に含まれていな
い場合は
// 実行ファイルでもライブラリファイル空間でも、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ヘッダなので
// 構造になっているため、ファイルを読み取る際に最初のデータブロックをスキップする必要があります。
なぜなら、各

```

// データブロックの長さがBLOCK\_SIZE = 1KBである場合、1ページのメモリには4ブロックのデータが格納できます。

//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

408 // 実行画像ファイルの欠損ページの開始ブロック番号「block」。

```
409     if (tmp >= LIBRARY_OFFSET) {
410         inode = current->library;           // inode & block no. of library file.
411         block = 1 + (tmp-LIBRARY_OFFSET) / BLOCK_SIZE;
412     } else if (tmp < current->end_data) {
```

```

411         inode = current->executable;           // inode & block no of executable file.
412         block = 1 + tmp / BLOCK SIZE;
413     } else {
414         inode = NULL;                      // for dynamically applied page.
415         block = 0;
416     }
// プロセスがその動的アプリケーションのページにアクセスした場合、またはページフォルト例外が発生した場合
// スタック情報を格納することで、物理メモリのページを直接申請します。
// そしてそれをリニアアドレス「address」にマッピングする。それ以外の場合は、行方不明のページがスコープ内の
プロセス実行ファイルやライブラリファイルの//を共有しようとするので、ページ操作をして終了します。
// 成功した場合。失敗した場合は、物理メモリのページを申請する必要があります。
// その後、実行ファイルの対応するページをデバイスから読み込んで配置（マッピング）する
// プロセスページの論理アドレス'tmp'へ。
417     if (!inode) {                           // it's a dynamic applied page.
418         get_empty_page(address);
419         return;
420     }
421     if (share_page(inode, tmp))          // try to share the page at 'tmp'.
422         return;
423     if (!(page = get_free_page()))       // apply for a free page.
424         oom();
425 /* remember that 1 block is used for header */
// 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。
// 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
// for (i=0 ; i<4 ; block++, i++)
ビ
ツ
ト
マ
ッ
プ
を
作
成
し
、
br
ea
d_
pa
ge()
使
つ
て
4
つ
の
論
理
ブ
ロ
ッ
ク

```

を物理ページに読み込みます。  
。

426  
427           nr[i] = bmap(inode, block);  
428        bread\_page(page, inode->i\_dev, nr);

// デバイス・ロジック・ブロックの読み出し時には、読み出しへ位置が  
// は、ファイルの最後から1ページ以下の長さになることがあります。そのため、いくつかの無駄な情報  
// を読み取ることができます。以下の操作は、実行を超えてこの部分をクリアするためのものです。  
// 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  
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/が読み  
込まれるので

//       i = tmp + 4096 - current->end\_data;                    // the excess byte length.

クリア操作を行う必要はありません。  
。

429  
430       if (i>4095)    // more than 1 page from the end.  
431              i = 0;  
432       tmp = page + 4096;    // points to the end of the page.  
433       while (i-- > 0) {    // i bytes cleared start from page end.  
434              tmp--;  
435              \*(char \*)tmp = 0;  
436       }

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

```
437             return;
438         free_page(page);
439         oom();
440     }
441 }
442 //// Memory management initialization.
```

```

// 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アイテム ((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
void mem_init(long start_mem, long end_mem)

```

```

443
444 {
445     int i;
446
// The function first sets the memory mapped byte array items corresponding to all pages in
// the range of 1MB to 16MB to the occupied state, that is, all bytes are set to USED (100).
// PAGING_PAGES is defined as (PAGING_MEMORY>>12), which is the number of all physical memory
// pages above 1MB (15MB/4KB = 3840).
447     HIGH_MEMORY = end_mem;                                // set the memory top (16MB).
448     for (i=0 ; i<PAGING_PAGES ; i++)                  // PAGING_PAGES = 3840.
449         mem_map[i] = USED;
// 次に、開始アドレスのページに対応するバイト配列のアイテム番号'i'を見つけます。
// 'start_mem'で、メインメモリ領域のページ数を計算します。この時点では
// mem_map[]のi番目の項目は、メインメモリ領域の最初のページに対応しています。最後に

```

// 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[]内のメモリ領域がクリアされます。

```

451        i = MAP_NR(start_mem);                            // page number at the beginning of the MMA.
452        end_mem -= start_mem;
453        end_mem >>= 12;                                  // total number of pages in MMA.
454        while (end_mem-->0)
455              mem_map[i++]=0;                            // bytes of MMA pages are reset.
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
// First, according to the byte array mem_map[], we count the total number of pages in the main
// memory area 'total', and the number of free pages 'free' and the number of shared pages
// 'shared', and display these information.
463     printk("Mem=info: |n|r");
464     for(i=0 ; i<PAGING_PAGES ; i++) {
465         if (mem_map[i] == USED)           // pages not for allocation.
466             continue;
467         total++;
468         if (!mem_map[i])
469             free++;                   // free pages in the main mem ares.
470         else
471             shared += mem_map[i]-1;    // shared pages (byte value > 1).
472     }
473     printk("%d free pages of %d|n|r", free, total);
474     printk("%d pages shared|n|r", shared);

```

// 次に、CPUのページング管理ロジックのページ数を数えます。最初の4項目は  
ページディレクトリテーブルの // は、カーネルコードが使用するもので、統計的には  
// の範囲になります。この方法では、項目5から始まるすべてのページディレクトリエントリをループし  
ます。もし  
// 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  
になります。  
// ページテーブル自体がまずカウントされ (484行目) 、次に対応する物理メモリページがカウントされ  
ます。  
ページテーブルのすべてのページエントリに対する // がカウントされます。

```

475     k = 0;                      // statistics of pages occupied by a process.
476     for(i=4 ; i<1024 ; ) {
477         if (1&pg_dir[i]) {
//(ページディレクトリエントリ内の対応するページテーブルのアドレスが
//マシンの最高物理メモリアドレスであるHIGH_MEMORYに問題があります。
//ディレクトリ・エントリを表示します。ディレクトリ・エントリの情報が表示され、次のディレクト
リ
// エントリーの処理を続けます)
479         if (pg_dir[i]>HIGH_MEMORY) {          // content is abnormal.
480             printk("page directory[%d]: %08X|n|r",
481                     i, pg_dir[i]);
482             continue;                  // needs "i++;" before it.
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
//つまり1MB) 、1つのプロセスの物理メモリページ統計値'k'がインクリメントされます。
// システムが占有しているすべての物理メモリページの統計値が「free」であること。
//を1つずつ増やしていきます。次に、対応するページテーブルアドレス'pg_tbl'を取得し、カウント
// 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。示される物理ページが
// 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。
//の場合           if (pg_dir[i]>LOW_MEM)

```

は、ページ  
テーブルエ  
ントリの対  
応するペー  
ジが統計値  
に含まれま  
す。

```

483
484           free++, k++;           // pages occupied in the page table.
485           pg_tbl=(unsigned long *) (0xfffff000 & pg_dir[i]);
486           for(j=0 ; j<1024 ; j++)
487               if ((pg_tbl[j]&1) && pg_tbl[j]>LOW MEM)
// (物理ページのアドレスが本機の最上位の物理メモリアドレスよりも大きい場合は
// マシンのHIGH_MEMORYは、ページテーブルの内容に問題があることを示しています。
// の項目があるので、ページテーブルの項目の内容が表示されます。それ以外の場合は、対応するペー
ジ

```

ページテーブル項目の//は統計値に含まれます)。

```

489         if (pg_tbl[j]>HIGH_MEMORY)
490             printk("page_dir[%d][%d]: %08X\n|r",
491                   i, j, pg_tbl[j]);
492         else
493             k++, free++; // pages of page table items.
493     }

```

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

// 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ページテーブルが

```

// タスク構造がカウントされます。この時点でき=0であれば、タスク構造に対応するプロセスが
// the current 16 directory entries does not exist in the system (not created or terminated).
// Then, after the corresponding process number and the physical page statistics value k are
// displayed, k is cleared to count the pages occupied by the next process.
494     i++;
495     if (!(i&15) && k) { // k !=0 indicates that the process exists.
496         k++, free++; /* one page/process for task_struct */
497         printk("Process %d: %d pages\n|r", (i>>4)-1, k);
498         k = 0;
499     }
500 }

```

// Finally, the memory page being used in the system and the total number of pages in the main  
// memory area are displayed.

```

501     printk("Memory found: %d (%d)\n|r", free-shared, total);
502 }
503

```

## 13.3 page.s

### 13.3.1 Function

The page.s file includes page fault exception interrupt handler (interrupt 14), which is handled primarily in two cases. First, due to page fault exceptions caused by page not present, this needs to be handled by calling do\_no\_page(error\_code, address); second, page exceptions caused by page write protection. At this point, the page write protection handler do\_wp\_page(error\_code, address) is called for processing. The error code (error\_code) in the function parameter is automatically generated by the CPU and pushed onto the stack. The linear address (address) accessed when an exception occurs is taken from the control register CR2. CR2 is specifically used to store linear addresses when a page fails.

### 13.3.2 Code annotation

7 /\*

---

```

1 /*
2 * linux/mm/page.s
3 *
4 * (C) 1991 Linus Torvalds
5 */
6

```

プログラム 13-2 linux/mm/page.s

```

8 * page.s contains the low-level page-exception code.
9 * the real work is done in mm.c // memory.c
10 */
11
// この変数は、kernel/traps.c でページ例外記述子を設定するために使用されます。
12 .globl _page_fault # declared as a global variable.
13
14 _page_fault:
15     xchgl %eax, (%esp) # take the error code to EAX.
16     pushl %ecx
17     pushl %edx
18     push %ds
19     push %es
20     push %fs
21     movl $0x10,%edx # set the kernel data segment selector.
22     mov %dx,%ds
23     mov %dx,%es
24     mov %dx,%fs
25     movl %cr2,%edx # get the linear address that caused the page exception.
26     pushl %edx # the linear address and error code are pushed onto the stack
27     pushl %eax # as arguments to the function to be called.
// "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
// caused by a page fault. Otherwise, the page write protection function do_wp_page() is called.
28     testl $1,%eax # Test flag P (bit 0), jump if it is set.
29     jne 1f
30     call _do_no_page # mm/memory.c, line 381.
31     jmp 2f
32 1:    call _do_wp_page # mm/memory.c, line 254.
// Discard the two arguments pushed onto the stack, pop the registers and exit the interrupt.
33 2:    addl $8,%esp
34     pop %fs
35     pop %es
36     pop %ds
37     popl %edx
38     popl %ecx
39     popl %eax
40     iret

```

## 13.4 swap.c

### 13.4.1 Function

Starting with version 0.12, Linux has added virtual memory swapping functionality to the kernel. This function is mainly implemented by this program. When the physical memory capacity is limited and the usage is tight, the program saves the temporarily unused memory page contents to the disk (switching device) to free up memory space for the programs that are in urgent need. If we later need to use the memory page content already stored on the swap device again in the future, the program is responsible for taking them back and putting them back into memory. Memory swap management uses a mapping technique similar to that of main memory area management, using bit maps to determine the specific save location and map location of the

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がない場合、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));
```

---

The ll\_rw\_page() function is a low-level page read and write function of the block device. The code is implemented in the kernel/blk\_drv/ll\_rw\_blk.c file. It can be seen that the swap device access functions are essentially the device page access functions that specifies the device number.

カーネルの初期化処理において、システムがスワップデバイス番号SWA\_DEVを定義している場合、カーネルはスワップ処理の初期化関数init\_swapping()を実行します。この関数はまず、システム内のブロックの配列に基づいて、システムにスワップデバイスがあるかどうか、デバイスのスワップパーティションが有効かどうかをチェックします。そして、メモリページを要求し、スワップパーティションの最初のスワップ管理ページ（ページ0）をメモリページに読み込みます。ページ0には、スワップページのビットマップマッピング情報が格納されており、各ビットがスワップページを表します。ビットが0の場合は、対応するデバイスのスワップページが使用されている（占有されている）か、利用できないことを示し、ビットが1の場合は、対応するスワップページが利用可能であることを示します。1ページはSWAP\_BITS ( $4096 \times 8 = 32768$ ) 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

ページの数は「swap\_size - 1」であり、そのページ番号の範囲は[1 -- swap\_size-1]であるため、ビットマップ内の対応するビットはすべて1（アイドル）となります。ビットマップの[swap\_size -- SWAP\_BITS]の範囲のビットは、デバイス上に対応する swapping ページがないため、利用できない状態（すべて0）に初期化されます。したがって、初期の正常なケースでは、ビットマップのビット状態は以下のようになるはずです（斜線部分は swapping に利用できません）。

---

bits in the bitmap:	0, 1, 1, 1, ... , 1,	0, ... , 0.
swap page number :	0, 1, 2, 3, ... , swap_size-1,	swap_size, ... , SWAP_BITS-1.

---

### 13.4.2 Code annotation

プログラム 13-3 linux/mm/swap.c

```

1 /*
2 * linux/mm/swap.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 /*
8 * This file should contain most things doing the swapping from/to disk.
9 * Started 18.12.91
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
// release function prototypes.
// <linux/sched.h> スケジューラーのヘッダーファイルでは、タスク構造体task_struct、データ
// of the initial task 0, and some embedded assembly function macro statements about the
// descriptor parameter settings and acquisition.
// <linux/head.h> ヘッ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。
// along with several selector constants.
// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロト
// タイプ定義が含まれています。
// used functions of the kernel.
12 #include <string.h>
13
14 #include <linux/mm.h>
15 #include <linux/sched.h>
16 #include <linux/head.h>
17 #include <linux/kernel.h>
18
// 各バイトは8ビットなので、1ページ（4096バイト）は合計32768ビットです。もし、1つのビットが
// 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ていると、対応するスワップ
// ページはアイドルです。
19 #define SWAP_BITS (4096<<3)           // define swap total bits in a page (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・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
```

```

21 // 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。
22 #define bitop(name, op) \
23 static inline int name(char * addr, unsigned int nr) \
24 { \
25     int res; __asm volatile ("bt" op " %1,%2; adcl $0,%0" ) 26 :"=g" \
26     (res) \
27 : "r" (nr), "m" (*(addr)), "0" (0)); \
28     return res; \
29 }
30
31 // Here we define 3 inline functions according to different op characters.
32 bitop(bit, "")           // define bit(char * addr, unsigned int nr)
33 bitop(setbit, "s")        // define setbit(char * addr, unsigned int nr)
34 bitop(clrbit, "r")        // define clrbit(char * addr, unsigned int nr)
35 static char * swap_bitmap = NULL;
36 int SWAP_DEV = 0;          // The swap device number set when the kernel is initialized.
37
38 /*
39 * We never page the pages in task[0] - kernel memory.
40 * We page all other pages.
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 (count from 0).
46
47 // 応募して、スワップのページ番号をゲット。
48 // 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。
49 // 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。
50     int nr;

```

```

48
49
50     if (!swap_bitmap)
51         return 0;
52     for (nr = 1; nr < 32768 ; nr++)
53         if (clrbit(swap_bitmap, nr))
54             return nr;           // return the current free swap page number.
55     return 0;
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。
60
58 void swap_free(int swap_nr)
59 {
60     if (!swap_nr)
61         return;

```

```

63     if (swap_bitmap && swap_nr < SWAP_BITS)
64         if (!setbit(swap_bitmap, swap_nr))
65             return;
66         printk("Swap-space bad (swap_free())|n|r");
67     return;
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する。
// そして、対応するフラグを設定します。
71 void swap_in(unsigned long *table_ptr) 70
{
72     int swap_nr;
73     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
// エントリーはスワップページ番号*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。
74     // 以下の関数 try_to_swap_out()で。
75     if (!swap_bitmap) {
76         printk("Trying to swap in without swap bit-map");
77         return;
77     }
78     if (1 & *table_ptr) {
79         printk("trying to swap in present page|n|r");
80         return;
81     }
82     swap_nr = *table_ptr >> 1;
83     if (!swap_nr) {
84         printk("No swap page in swap_in|n|r");
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、書き込み
// 可能性と
87     // 存在フラグ(Dirty, U/S, R/W, P)。

```

```
88     if (! (page = get_free_page ()))
89         oom ();
90     read_swap_page (swap_nr, (char *) page);      // defined in file nclude/linux/mm.h
91     if (setbit (swap_bitmap, swap_nr))
92         printk ("swapping in multiply from same page\n|r");
93     *table_ptr = page | (PAGE_DIRTY | 7);
93 }
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  
// bit P = 0. The parameter 'table_ptr' is a pointer to a page table entry. Returns 1 if the  
// 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 {
```

```
98         unsigned long page;  
99         unsigned long swap_nr; 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  
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
```

// 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。

// (15MB)でも終了しています。

```
100     page = *table_ptr;
101     if (!(PAGE_PRESENT & page))
102         return 0;
103     if (page - LOW MEM > PAGING MEMORY)
104         return 0;
```

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

// このようなページはスワップアウトされるべきではないので、この関数は

// 0で存在する。それ以外の場合は、スワップページ番号を取得し、それをページテーブルのエントリに保存してから

// ページをスワップアウトし、対応する物理メモリページを解放します。

```
106     if (PAGE DIRTY & page) {
107         page &= 0xfffffff000;                                // get physical page address.
108         if (mem map[MAP NR(page)] != 1)
109             return 0;
110         if (!(swap_nr = get swap page()))           // get a swap page number.
111             return 0;
```

```
// スワップされるページについては、スワップページ番号 (swap_nr << 1) が、対応する  
// ページテーブルエントリ。2をかけるのは、ページテーブルエントリのプレゼンスピット (P、ビット0)  
// を解放するためです。
```

// ページテーブルエントリ。ビットP=0で、ページテーブルエントリの内容が0以外のページのみが  
// がスワップデバイスにあることを示しています。インテルのマニュアルでは、テーブルエントリに  
// 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

// write\_swap\_page(nr, buffer) は、ll\_rw\_page(WRITE, SWAP\_DEV, (nr), (buffer)) として定義されています

//linux/mm.h ファイルの12行目です

```
113     *table_ptr = swap_nr<<1;  
114     invalidate(); // refresh the CPU transform cache.  
115     write_swap_page(swap_nr, (char *) page);
```

```
114         free_page(page);
115         return 1;
116     }
117     // Otherwise, it means that the page has not been modified, so you don't have to swap it out,
118     // but you can release it directly.
119     *table_ptr = 0;
120     invalidate();
121     free_page(page);
122     return 1;
123 }
124 /*
125 * Ok this has a rather intricate logic - the idea is to make good
126 * and fast machine code. If we didn't worry about that, things would
```

```

126 * be easier.
127 */
// スワップデバイスにメモリページを入れる。
// リニアに対応するページディレクトリエントリ(FIRST_VM_PAGE>>10)から開始します。
// 64MBのアドレスでは、4GBのリニアー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 {
130     static int dir_entry = FIRST_VM_PAGE>>10; // 16, first directory entry of task 1.
131     static int page_entry = -1;
132     int counter = VM_PAGES; // 1032192, see line 44.
133     int pg_table;
134
135     // 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
136     pg_table
137     // 見つかった場合はループを終了します。
138     // 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
139     // 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
140     // 検索してもエントリーが見つからなかった場合、コードは0を返します。
141
142     while (counter>0) {
143         pg_table = pg_dir[dir_entry]; // content of the directory entry.
144         if (pg_table & 1) // exit the loop if the table exists.
145             break;
146         counter -= 1024; // One page table has 1024 items.
147         dir_entry++; // next directory entry.
148         if (dir_entry >= 1024)
149             dir_entry = FIRST_VM_PAGE>>10;
150     }
151
152     // 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
153     // 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
154     // 試みます。
155     // 出します。ページがスワップデバイスに正常にスワップアウトされた場合は1を返します。もしすべて
156     // のページテーブル
157     // すべてのディレクトリエントリの処理に失敗した場合は、警告メッセージを表示し、0を返します。
158
159     pg_table &= 0xfffff000; // page table pointer (address).
160     while (counter-- > 0) {
161         page_entry++; // Page table entry (initially -1).
162
163         // 現在のページテーブルのすべてのアイテムを処理しようとしたが、まだ処理できない場合
164         // 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
165         // が1024以上の場合は、前の135～143行と同じように使用します。
166         // 次のページ・ディレクトリ・エントリでセカンダリ・ページ・テーブルを選択します。
167
168         if (page_entry >= 1024) {

```

```
150         page_entry = 0;
151     repeat:
152         dir_entry++;
153         if (dir_entry >= 1024)
154             dir_entry = FIRST VM PAGE>>10;
155         pg_table = pg_dir[dir_entry]; // content of page directory entry.
156         if (!(pg_table&1))
155             if ((counter -= 1024) > 0)
156                 goto repeat;
157             else
158                 break;
```

```

159             pg_table &= 0xfffff000;           // get page table 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|r");
165     return 0;
166 }
167 /*
168 * Get physical address of first (actually last :-) free page, and mark it
169 * used. If no free pages left, return 0.
170 */
171 */

//// メインメモリ領域の空き物理ページを取得します。
// 利用可能な物理メモリページがない場合は、ページスワップ処理を行ってから
// 申し込みをして、再度ページを取得しようとします。
// 入力: %1(ax = 0) ; %2(LOW_MEM) バイトビットマップで管理されるメモリの開始位置。
// %3(cx = PAGING_PAGES) ; %4(edi = mem_map + PAGING_PAGES - 1).
// この関数は、新しいページのアドレスを %0(ax = 物理ページの開始アドレス)で返します。
// 上記の%4レジスタは、実際にはメモリバイトビットマップmem_map[].Thisの最後のバイトを指しています。
// 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ヤンします（ページの合計数は
// //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は0）。
// 注意！この関数は、単にメインメモリ領域の空き物理ページを指定するだけですが、これは
// //は、どのプロセスのアドレス空間にもマッピングされていません。memory.cのput_page()関数は以下のようになっています。
// プログラムは、指定したページをプロセスのアドレス空間にマッピングするために使用されます。もちろん、このプログラムで
// // 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。
// 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。
// 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。
// //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、主にインラインの
// //アセンブリファイルです。
172 unsigned long get_free_page(void)
173 {
174 register unsigned long res asm("ax"); 175
    // まず、メモリバイトのビットマップで値が0のバイトを調べて、クリアします。
    // 対応する物理メモリページ。結果として得られるページアドレスが、実際の
    // //物理メモリの容量を、もう一度探してみてください。空きページが見つからない場合は、スワップを実行します。
176 // の操作をしてから、もう一度調べてみましょう。最後に、空いている物理ページのアドレスを返します。

```

```

177 repeat:
178 // メモリバイトのビットマップを使って空きページを探す mem_map[].
179 __asm ("std ; repne ; scash|n|t"    // al(0) compared with content of each page (di).
180     "jne 1f|n|t"                  // jump to label 1 if none of it equals 0.
181     "movb $1, 1(%edi)|n|t"      // set byte [1+edi] to 1 of the page.
182     "sall $12, %%ecx|n|t"      // pages * 4K = relative address of the page.
183     "addl %2, %%ecx|n|t"       // plus LOW_MEM to get the absolute page address.
184 // メモリページをゼロにします。
185     "movl %%ecx, %%edx|n|t"    // load edx register with page start address.
186     "movl $1024, %%ecx|n|t"    // load ecx with counting number 1024.
187     "leal 4092(%edx), %%edi|n|t" // load edi with page end address (4092 + edx).
188     "rep ; stosl|n|t"          // reset (clear) each byte in the page.
189     "movl %%edx, %%eax|n"      // load eax with the page start address.
187     "1:"                      // loop label.
188     : "=a" (_res)

```

```

189      :"O" (0), "i" (LOW_MEM), "c" (PAGING_PAGES),
190      "D" (mem_map+PAGING_PAGES-1)
191      :"di", "cx", "dx");
192      if (_res >= HIGH_MEMORY) // search again if the page is out of main page area.
193          goto repeat;
194      if (!_res && swap_out()) // do swapping if no free page found and search again.
195          goto repeat;
196      return _res;           // return the address of the free page.
197  }
198

// メモリページスワッピングの初期化。
// この関数はまず、デバイスのパーティションに基づいて、デバイスにスワップパーティションがあるかどうかを確認します。
// の配列（ブロックの配列）を取得し、スワップパーティションが有効かどうかをチェックします。その後、ページを取得するために申請
// スワップページ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。
    // メジャーデバイス番号。その各項目は、所有するデータブロックの総数に対応しています。
    // 1つのサブデバイスで、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。
    //     extern int *blk_size[];           // defined in blk_drv/11_rw_blk.c, line 49.

```

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  
。

```

201
202     int swap_size, i, j;
203
204     if (!SWAP_DEV)
205         return;
206     if (!blk_size[MAJOR(SWAP_DEV)]) {
207         printk("Unable to get size of swap device |n|r");
208         return;
209     }
// 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
// swap_size = blk_size[MAJOR(SWAP_DEV)][MINOR(SWAP_DEV)];
// の
// メッセージが表示され、終了します。
// 終了します。
210
211     if (!swap_size)
212         return;
213     if (swap_size < 100) {
214         printk("Swap device too small (%d blocks) |n|r", swap_size);
215         return;
216     }
// 次に、スワップブロックの合計数を、対応するスワップの合計数に変換します。
// 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ジを
// 取得します。
217 // 各ビットが1つのスワップページを表しています。
218     swap_size >>= 2;
219     if (swap_size > SWAP_BITS)
220         swap_size = SWAP_BITS;
221     swap_bitmap = (char *) get_free_page();
222     if (!swap_bitmap) {
223         printk("Unable to start swapping: out of memory :-|n|r");
224         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）がクリアされます。
// マクロread_swap_page(nr, buffer)は、ファイルinclude/linux/mm.hの11行目で定義されています。
225     read_swap_page(0, swap_bitmap);
226     if (strncmp("SWAP-SPACE", swap_bitmap+4086, 10)) {
227         printk("Unable to find swap-space signature\n|r");
228         free_page((long) swap_bitmap);
229         swap_bitmap = NULL;
230         return;
231     }
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。
// 対応するスワップページです。以下、ビットマップをチェックする場合、2つのステップでチェックし
// ます。
// 利用できない部分と利用できる部分に基づいて
// 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。
// 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
//      for (i = 0 ; i < SWAP_BITS ; i++) {
// 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。

```

233
234     if (i == 1)
235         i = swap_size;
236     if (bit(swap_bitmap, i)) {
237         printk("Bad swap-space bit-map\n|r");
238         free_page((long) swap_bitmap);
239         swap_bitmap = NULL;
240         return;
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。
// space are displayed.

```

```
243     j = 0;
244     for (i = 1 ; i < swap_size ; i++)
245         if (bit(swap_bitmap, i))
246             j++;
247     if (!j) {
248         free_page((long) swap_bitmap);
249         swap_bitmap = NULL;
250         return;
251     }
252     printk("Swap device ok: %d pages (%d bytes) swap-space |n|r", j, j*4096);
253 }
```

254

---

## 13.5 Summary

This chapter describes how the kernel manages and accesses physical and virtual memory in the system. It focuses on the memory allocation management mechanism of Intel CPU, and how the Linux kernel divides the memory space. At the same time, it gives the principle of copy-on-write mechanism and demand loading mechanism. Finally, we illustrate and describe the bitmap-swap page management and processing mechanism of the swap partition on the device.

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定義などを完全に説明します。

## 14 Header Files (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 <filename>
```

---

Of course, the filename 'filename' cannot contain > and newline characters, as well as the ", ', \, or /\* characters. The compiler will search for this file in a set of pre-set places. Similarly, the following form of control line will cause the compiler to first search for the 'filename' file in the directory where the source program is located:

---

```
# include "filename"
```

---

If the file is not found, the compiler will perform the same search process as above. In this form, the file name 'filename' also cannot contain newline characters and ", ', \, or /\* characters, but the > character is allowed. In general application source code, the header files are inextricably linked to the library files in the development environment. Each function in the library needs to be declared in the relevant header file. The header files in the application development environment (usually placed in the /usr/include/ directory) can be thought of as an integral part of the functions in the libraries they provide (eg libc.a), and are instructions or interface statements for library functions. After the compiler converts the source code program into an object module, the linker combines all the object modules of the program, including the modules in any library files

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については、プラウガー氏の著書「The Standard C Library」に詳しく書かれている。

本書で紹介するカーネルのソースコードにおいて、関係するヘッダーファイルは、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（ブ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。

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 Files in the include/ directory

カーネルが使用するヘッダーファイルは、カーネルのソースツリーの include/ ディレクトリに格納されています。このディレクトリ内のファイルはリスト14-1に示されています。ここで注意すべき点は、使いやすさと互換性のために、Linusはカーネルプログラムのヘッダーファイルをコンパイルする際に、標準的なカーネルCのヘッダーファイルに対して同様の命名規則を使用していることです。多くのヘッダーファイルの名前や、ファイルの内容の一部も、基本的には標準Cライブラリのものと同じです。しかし、これらのヘッダーファイルは、やはりカーネル固有のもの、あるいはカーネルと密接に結びついたプログラム固有のものです。Linuxシステムでは、標準ライブラリのヘッダ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。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、主に使用しているコンピュータアーキテクチャに密接に関連する関数宣言やデータ構造のヘッダーファイルを格納するために使用されます。例えば、Intel CPUポートIOアセンブリマクロファイルio.h、割り込みディスクリプターセットアセンブリマクロヘッダーファイルsystem.hなどがあります。linux/ ディレクトリには、スケジューラが使用するヘッダーファイル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ファイルが格納されている。sys/ディレクトリには、カーネルリソースに関連するいくつかのヘッダファイルが格納されています。ただし、バージョン0.98からは、カーネルディレクトリツリーの下のsys/ディレクトリにあるヘッダファイルは、すべてlinux/ディレクトリに移動しています。

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

リスト 14-1 linux/include/ディレクトリ内のファイル

Filename	Size	Last Modified Time	Description
asm/		1992-01-09 16:46:04	
linux/		1992-01-12 19:43:55	
sys/		1992-01-09 16:46:03	
a.out.h	6047 bytes	1991-09-17 15:10:49	
const.h	321 bytes	1991-09-17 15:12:39	
ctype.h	1049 bytes	1991-11-07 17:30:47	

---

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

---

## 14.2 a.out.h

### 14.2.1 Function

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

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

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

a.out の実行ファイルは、先頭の exec ヘッダー部と、それに続くコード、データなどの部分で構成されています。実行ファイルのヘッダー部には、主に exec 構造体があり、マジックナンバーフィールド、コードとデータの長さ、シンボルテーブルの長さ、コード実行開始位置などの情報が含まれています。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、リンカー (ld) はこれらのパラメータを使っていくつかのモジュールファイルを1つの実行ファイルにまとめます。これが

オブジェクトファイルの唯一の必要な構成要素である。

0.96カーネルから、LinuxシステムはGNUのヘッダファイルa.out.hを直接使用するようになります。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。

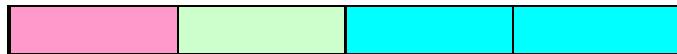


図14-1 exec構造体の最初のフィールド a\_magic(a\_info)

In the Linux 0.9x system, for executable files linked with static libraries, the values in parentheses in the figure are the default values for each field. The 4 bytes at the beginning of this binary executable file are:

---

0x0b, 0x01, 0x64, 0x00

---

The a.out header file in this kernel only defines the magic number field. Therefore, the first 4 bytes of a binary executable file in a.out format on a Linux 0.1x system are:

---

0x0b, 0x01, 0x00, 0x00

---

It can be seen that the difference between the executable file in the a.out format of GNU and the executable file compiled on the Linux 0.1x system is only in the machine type field. So we can clear the machine type field (3rd byte) of the a.out format executable file on Linux 0.9x and run it on the 0.1x system. As long as the system-call invoked by the ported executable file is already implemented in the 0.1x system. The author used this approach when starting to rebuild many of the commands in the Linux 0.1x root file system. In other respects, GNU's a.out.h header file is no different from a.out.h here.

## 14.2.2 Code annotation

プログラム14-1 linux/include/a.out.h

---

```

1 #ifndef A_OUT_H
2 #define A_OUT_H
3
4 #define GNU_EXEC_MACROS
5
// 6～108行目は、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 // 以下は、オブジェクトファイルのヘッダ構造で、プログラムの後に詳細な説明があります。6 struct
exec {
8     unsigned long a_magic;          /* Use macros N_MAGIC, etc for access */
9     unsigned a_text;                /* length of text, in bytes */

```

```

10 unsigned a_data;           /* length of data, in bytes */
11 unsigned a_bss;            /* length of uninitialized data area for file, in bytes */
12 unsigned a_syms;           /* length of symbol table data in file, in bytes */
13 unsigned a_entry;          /* start address */
14 unsigned a_trsize;         /* length of relocation info for text, in bytes */
15 unsigned a_drsize;         /* length of relocation info for data, in bytes */
15 };
16
17 // 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。
18 #ifndef N_MAGIC
19 #define N_MAGIC(exec) ((exec).a_magic)
19 #endif
20
21 #ifndef OMAGIC
22 /* Code indicating object file or impure executable. */
// 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
// (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
実行ファイルのヘッダー構造の先頭にあった // 0407番 (0x107)。
// 元々はPDP-11のジャンプ命令で、先頭にジャンプすることを示していました。
次の7つの単語の後にコードの//を入力します。このようにして、ローダは直接先頭にジャンプすることができます。
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。
// 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）として残されています。
23 // 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。
24 #define OMAGIC 0407
25 /* Code indicating pure executable. */
// 1975年以降に使用されたNMAGIC (New Magic)。仮想記憶のメカニズムに関わるものです。
25 #define NMAGIC 0410          // 0410 == 0x108
26 /* デマンドページングされた実行ファイルを示すコード*/
// 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。
27 #define ZMAGIC 0413          // 0413 == 0x10b
28 #endif /* not OMAGIC */
29 // There is also a QMAGIC, which is used to save disk capacity and store the header structure
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。
// 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。
30 // のフィールドに表示されます。マジックナンバーが認識できない場合は、trueを返します。
31 #ifndef N_BADMAG
32 #define N_BADMAG(x)           \
33   (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC) \
34   && N_MAGIC(x) != ZMAGIC) \
34 #endif
35
36 #define N_BADMAG(x)           \
37   (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC) \
38   && N_MAGIC(x) != ZMAGIC) \
39
// 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。

```

//と1KBの位置関係になっています。

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

41

// 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。

// モジュールファイルと実行ファイル。

// 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。
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
47 // 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。
48 #ifndef N_DATOFF
49 #define N_DATOFF(x) (N_TXTOFF(x) + (x).a_text)
50 #endif
51
52 // 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。
53 #ifndef N_TRELOFF
54 #define N_TRELOFF(x) (N_DATOFF(x) + (x).a_data)
55 #endif
56
57 // 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。
58 #ifndef N_DRELOFF
59 #define N_DRELOFF(x) (N_TRELOFF(x) + (x).a_trsize)
60 #endif
61
62 // 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。
63 #ifndef N_SYMOFF
64 #define N_SYMOFF(x) (N_DRELOFF(x) + (x).a_drsize)
65 #endif
66
67 // 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。
68 /* 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。*/
69 #ifndef N_TXTADDR
70 #define N_TXTADDR(x) 0 // The code segment begins at address 0.
71
72 /* Address of data segment in memory after it is loaded.
73 Note that it is up to you to define SEGMENT_SIZE
74 on machines not listed here. */
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 /* Sony. */

```

```
84 #ifdef is68k  
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。
92 // なので、上記の定義は使われていません。
93 #define PAGE_SIZE 4096
94 #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なります。
// 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。
99 ZMAGICタイプのファイルの場合は、// (1KB境界のアライメント) となります。
100 #ifndef N_DATADDR
101 #define N_DATADDR(x) \
102     (N_MAGIC(x)==OMAGIC? (_N_TXTENDADDR(x)) \
103      : (_N_SEGMENT_ROUND (_N_TXTENDADDR(x)))) \
103 #endif
104
105 /* Address of bss segment in memory after it is loaded. */
106 // 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。
107 #ifndef N_BSSADDR
108 #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トリー（レコード）構造です。
112 #ifndef
N_NLIST_DECLARED 111
struct nlist {
113     union {
114         char *n_name;
115         struct nlist *n_next;
116         long n_strx;
117     } n_un;
118     unsigned char n_type;           // The byte is divided into 3 fields, and
119     char n_other;                 // lines 146-154 are the mask code for each field.

```

```
120     short n_desc;
121     unsigned long n_value;
121 };
122 #endif
123
124 // 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。
125 #ifndef N_UNDF
126 #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
145
146 // 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。
147 #ifndef N_EXT
148 #define N_EXT 1           // 0x01 (0b0000,0001) symbol is external (global) ?
149 #endif
150 #ifndef N_TYPE
151 #define N_TYPE 036        // 0x1e (0b0001,1110) The type bits of the symbol.
152 #endif
153 #ifndef N_STAB
154 #define N_STAB 0340       // STAB -- Symbol table types.
155 #endif
156 /* The following type indicates the definition of a symbol as being
157    an indirect reference to another symbol. The other symbol
158    appears as an undefined reference, immediately following this symbol.
159
160 Indirection is asymmetrical. The other symbol's value will be used
161 to satisfy requests for the indirect symbol, but not vice versa.
162 If the other symbol does not have a definition, libraries will
163 be searched to find a definition. */
164 #define N_INDR 0xa
165
166 /* The following symbols refer to set elements.
167 All the N_SET[ATDB] symbols with the same name form one set.
168 Space is allocated for the set in the text section, and each set
169 element's value is stored into one word of the space.
170 The first word of the space is the length of the set (number of elements).
171
172 The address of the set is made into an N_SETV symbol
173 whose name is the same as the name of the set.
174 This symbol acts like a N_DATA global symbol
175 in that it can satisfy undefined external references. */
176
177 /* These appear as input to LD, in a.o file. */

```

```

178 #define N_SETA 0x14      /* Absolute set element symbol */
179 #define N_SETT 0x16       /* Text set element symbol */
180 #define N_SETD 0x18       /* Data set element symbol */
181 #define N_SETB 0x1A       /* Bss set element symbol */
182
183 /* This is output from LD. */
184 #define N_SETV 0x1C       /* Pointer to set vector in data area. */
185
186 #ifndef N_RELOCATION_INFO_DECLARED
187
188 /* This structure describes a single relocation to be performed.
189   The text-relocation section of the file is a vector of these structures,
190   all of which apply to the text section.
191   Likewise, the data-relocation section applies to the data section. */
192
193 struct relocation_info
194 {
195     int r_address;
196     /* The meaning of r_symbolnum depends on r_extern. */
197     unsigned int r_symbolnum:24;
198     /* Nonzero means value is a pc-relative offset
199       and it should be relocated for changes in its own address
200       as well as for changes in the symbol or section specified. */
201     unsigned int r_pcrel:1;
202     /* Length (as exponent of 2) of the field to be relocated.
203       Thus, a value of 2 indicates 1<<2 bytes. */
204     unsigned int r_length:2;
205     /* 1 => relocate with value of symbol.
206       r_symbolnum is the index of the symbol
207       in file's the symbol table.
208       0 => relocate with the address of a segment.
209       r_symbolnum is N_TEXT, N_DATA, N_BSS or N_ABS
210       (the N_EXT bit may be set also, but signifies nothing). */
211     unsigned int r_extern:1;
212     /* Four bits that aren't used, but when writing an object file
213       it is desirable to clear them. */
214     unsigned int r_pad:4;
215 };
216 }
217 #endif /* no N_RELOCATION_INFO_DECLARED. */
218
219
220#endif /* _A_OUT_GNU_H_ */
221

```

## 14.2.3 Information

### 14.2.3.1 a.out executable file format

The Linux kernel version 0.12 only supports the format of a.out (Assembler output) executable files and object files. Although this format has been gradually not used, but the ELF (Executable and Link Format)

format is more fully used, due to its simplicity, it is suitable as a material for learning. Below we give a comprehensive introduction to the a.out format.

- a) 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
- b) **exec header** -- This section contains some parameters (exec structure) that the kernel uses to load the executable file into memory and execute it, and the linker (ld) uses these parameters to combine some of the module files into one executable file. This is the only necessary component of the object file.
  - c) **text segment** -- Contains the instruction code (text) and related data that is loaded into memory when the program is executed. It can be loaded in read-only form.
  - d) **data segment** -- This section contains data that has already been initialized and is always loaded into readable and writable memory.
  - e) **text relocations** -- This section contains records data for use by the linker. Used to locate and update a pointer or address in a code segment when combining object module files.
  - f) **data relocation** -- Similar to the role of the code relocation section, but for the relocation of pointers in the data segment.
  - g) **symbol table** -- This section also contains records data for use by the linker to cross-reference named variables and functions (symbols) between binary object module files.
  - h) **string table** -- This part contains character strings corresponding to the symbol names.

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

---

構造体exec {。

```
unsigned long a_magic          // Use macros for access, such as N_MAGIC.
unsigned a_text                // length of code, in bytes.
unsigned a_data                // length of data, in bytes.
unsigned a_bss                 // length of the uninitialized data area for file, in bytes.
unsigned a_syms                // length of the symbol table data in the file, in bytes.
unsigned a_entry               // Execution start address.
unsigned a_trsize              // length of the relocation info for text, in bytes.
unsigned a_drsize              // length of the relocation info for data, in bytes.
};
```

---

The functionality of each field is as follows:

- **a\_magic** -- This field contains three subfields, the flag field, the machine type id field, and the magic number field, as shown in Figure 14-1. However, for the Linux 0.12 system, its object file only uses the magic number subfield and is accessed using the macro N\_MAGIC(), which uniquely determines the difference between the binary executable file and other loaded files. This subfield must contain one of the following values:
  - ◆ **OMAGIC** -- Indicates that the text and data segments are immediately following the execution header and are stored consecutively. The kernel loads both text and data segments into readable and writable memory. The magic number of the object file compiled by the compiler is

- ◆ OMAGIC (octal 0407).
- ◆ NMAGIC -- Like OMAGIC, text and data segments follow the execution header and are stored continuously. However, the kernel loads the text into read-only memory and loads the data

- ◆ 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 -- The kernel loads separate pages from the binary executable when necessary. The execution header, text segment, and data segment are all processed by the linker into blocks of multiple page sizes. The text page loaded by the kernel is read-only, and the page of the data segment is writable. The magic number of the executable file generated by the linker is ZMAGIC (0413, ie 0x10b).
- a\_text -- This field contains the size of the text segment, the number of bytes.
- a\_data -- This field contains the size of the data segment, the number of bytes.
- a\_bss -- Contains the length of the 'bss segment' that the kernel uses to set the initial break(brk) after the data segment. When the kernel is loading the program, this writable memory appears to be behind the data segment and is initially all zeros.
- a\_syms -- Contains the size in bytes of the symbol table section.
- a\_entry -- The memory address of the program execution start point after the kernel has loaded the executable file into memory.
- a\_trsize -- This field contains the size of the text relocation table, in bytes.
- a\_drsiz -- This field contains the size of the data relocation table, in bytes.

a.out.hヘッダーファイルにはいくつかのマクロが定義されています。これらのマクロはexec構造を利用して整合性をテストしたり、実行ファイル内の様々なセクションオフセットを見つけたりします。これらのマクロは

N_BADMAG(exec)	Returns a non-zero value if the a_magic field cannot be recognized.
N_TXTOFF(exec)	The starting byte offset of the code segment.
N_DATOFF(exec)	The starting byte offset of the data segment.
N_DRELOFF(exec)	The starting byte offset of the data relocation table.
N_TRELOFF(exec)	The starting byte offset of the text relocation table.
N_SYMOFF(exec)	The starting byte offset of the symbol table.
N_STROFF(exec)	The starting byte offset of the string table.

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

---

#### 構造体relocation\_info

```
{
    int r_address;           // The address that needs to be relocated within the segment.
    unsigned int r_symbolnum:24; // The meaning is related to r_extern.
                                // Specifies a symbol or a segment in the symbol table.
    unsigned int r_pcrel:1;   // A pc-related flag.
    unsigned int r_length:2;  // Length (as exponent of 2) of the field to be relocated.
    unsigned int r_extern:1;  // 1:relocate with value of symbol, 0:with address of segment.
    unsigned int r_pad:4;    // 4 bits are not used, but it is best to clear them.
};
```

---

The meaning of each field in the structure is as follows:

- r\_address -- This field contains the byte offset of the pointer that the linker needs to process (edit).

The offset of the text relocation is counted from the beginning of the text segment, and the offset of the data relocation is calculated from the beginning of the data segment. The linker adds the value

- 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 -- This field contains the ordinal number (not the byte offset) of a symbol structure in the symbol table. After the linker calculates the absolute address of the symbol, it adds the address to the pointer being relocated. (If the r\_extern bit is 0, then the situation is different, see below.)
- r\_pcrel -- If this bit is set, the linker considers that a pointer is being updated, which uses the pc-related addressing mode and is part of the machine code instruction. When the running program uses this relocated pointer, the address of the pointer is implicitly added to the pointer.
- r\_length -- This field contains the power of 2 of the length of the pointer: 0 means 1 byte long, 1 means 2 bytes long, 2 means 4 bytes long.
- r\_extern -- If set, it indicates that the relocation requires an external reference; the linker must use a symbol address to update the pointer. When the bit is 0, the relocation is "local"; the linker updates the pointer to reflect the changes in the load addresses of the various segments, rather than reflecting changes in the value of a symbol. In this case, the contents of the r\_symbolnum field are an n\_type value; such field tells the linker what segment the relocated pointer points into.
- r\_pad -- These 4 bits are not used in Linux systems. It is best to set all 0 when writing a object file.

シンボルは名前とアドレスを対応させます（より一般的には文字列と値を対応させます）。リン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。

---

```
struct nlist
{ union {
    char *n_name; struct
    nlist *n_next; long
    n_strx;
} n_un;
unsigned char n_type;           // divided into 3 fields, and lines 146-154 are their masks.
char       n_other;
short      n_desc;
unsigned long n_value;
};
```

---

The meaning of each field is:

- n\_un.n\_strx -- Contains the byte offset of the symbol name in the string table. When a program accesses a symbol table using the nlist() function, the field is replaced with the n\_un.n\_name field, which is a pointer to a string in memory.
- n\_type -- Used by the linker to determine how to update the value of the symbol. The 8-bit wide n\_type field can be divided into three subfields using the bitmasks code starting at line 146--154, as shown in Figure 14-2. For symbols of N\_EXT type location bits, the linker treats them as "external" symbols and allows other binary object files to reference them. The N\_TYPE mask is used to select the bits of interest to the linker:
  - ◆ N\_UNDEF -- An undefined symbol. The linker must locate an external symbol with the same

name in another binary object file to determine the absolute data value of the symbol. In special cases, if the n\_type field is non-zero and no binary file defines this symbol, the linker resolves the symbol into an address in the BSS segment, reserving bytes of length equal to n\_value. If the

- ◆ 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 -- An absolute symbol. The linker does not update an absolute symbol.
  - ◆ N\_TEXT -- A text (code) symbol. The value of this symbol is the text address, and the linker updates its value when it merges the binary object files.
  - ◆ N\_DATA -- A data symbol; similar to N\_TEXT, but for data addresses. The value of the corresponding text and data symbol is not the offset of the file but the address; In order to find the offset of the file, it is necessary to determine the address at which the relevant section starts loading and subtract it, and then add the offset of the section.
  - ◆ N\_BSS -- A BSS symbol; similar to a text or data symbol, but without a corresponding offset in the binary object file.
  - ◆ N\_FN -- A file name symbol. When merging a binary object file, the linker inserts the symbol before the symbol in the binary file. The name of the symbol is the file name given to the linker, and its value is the address of the first text segment in the binary file. File name symbols are not required for linking and loading, but are very useful for debugging programs.
  - ◆ N\_STAB -- The mask code is used to select the bits of interest to the symbolic debugger (eg gdb); its value is specified in stab().
- n\_other -- This field provides symbol independence information about the symbol relocation operation according to the segment determined by n\_type. Currently, the lowest 4 bits of the n\_other field contain one of two values: AUX\_FUNC and AUX\_OBJECT. AUX\_FUNC associates symbols with callable functions, and AUX\_OBJECT associates symbols with data, regardless of whether they are in text segments or data segments. This field is mainly used by the linker ld for the creation of dynamic executable programs.
- n\_desc -- Reserved for use by the debugger; the linker does not process it. Different debuggers use this field for different purposes.
- n\_value -- Contains the value of the symbol. For text, data, and BSS symbols, this is an address; for other symbols (such as debugger symbols), the value can be arbitrary.

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

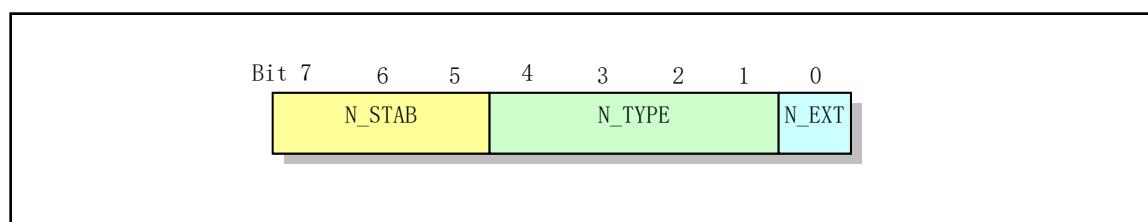


図 14-2 シンボルタイプフィールド n\_type

## 14.3 const.h

### 14.3.1 Function

The const.h file defines some of the flag constants used by the file modes and type field i\_mode in the file i-node.

### 14.3.2 Code annotation

1 プログラム 14-2 linux/include/const.h

```

2 #ifndef CONST_H
3 #define CONST_H
3
4 #define BUFFER_END 0x200000          // the memory end used by the buffer (not used).
5
// i-node構造体のi_modeフィールドの各フラグビット。
6 #define I_TYPE      0170000    // Indicate the i-node type (type mask).
7 #define I_DIRECTORY 0040000    // Is a directory file.
8 #define I_REGULAR   0100000    // Is a regular file, not a dir or a special file.
9 #define I_BLOCK_SPECIAL 0060000 // Is a block device special file.
10 #define I_CHAR_SPECIAL 0020000 // Is a character device special file.
11 #define I_NAMED_PIPE 0010000    // Is a named pipe node.
12 #define I_SET_UID_BIT 0004000  // Set efficient user ID type at execution time.
13 #define I_SET_GID_BIT 0002000  // Set efficient group ID type at execution time.
14
15 #endif
16

```

## 14.4 ctype.h

### 14.4.1 Function

The ctype.h file is a header file for character testing and processing, and is one of the header files for the standard C library. It defines some macros for character type checking and conversion. It defines some macros for character type checking and conversion. For example, determine if a character c is a numeric character (isdigit(c)) or a space character (isspace(c)). During the check process, an array or table (defined in lib/ctype.c) is used, which defines the properties and types of all the characters in the ASCII table. When a macro is used, the character code is taken as an index value in the table \_ctype[], and a byte is obtained from the table, so that the relevant bit is obtained.

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。

### 14.4.2 Code annotation

プログラム14-3 linux/include/ctype.h

---

```

1 #ifndef _CTYPE_H
2 #define _CTYPE_H
3
7 4 #define _U 0x01 /* upper */ // 大文字[A-Z]に使用されます。5 #define _L 0x02 /* lower */ //
小文字[a-z]に使用されます。6 #define _D 0x04 /* digit */ // デジタル文字[0-9]に使用されます。
8 #define _C 0x08 /* cntrl */ // used for control characters.
9 #define _P 0x10 /* punct */ // used for punctuation characters.
10 #define _S 0x20 /* white space (space/lf/tab) */
11 #define _X 0x40 /* hex digit */ // used for hexadecimal digits.
12 #define _SP 0x80 /* hard space (0x20) */ // used for the space character (0x20).
12

// 各キャラクターの属性を定義した、キャラクター属性の配列（テーブル）です。
// 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。
13 extern unsigned char _ctype[]; 14
extern char _ctmp;
15

// 以下に、文字の種類を決めるマクロを紹介します。
16 #define isalnum(c) (((ctype+1)[c]&(_U|_L|_D)) // is a character or digital.
17 #define isalpha(c) (((ctype+1)[c]&(_U|_L)) // is a character.
18 #define iscntrl(c) (((ctype+1)[c]&(_C)) // is a control character.
19 #define isdigit(c) (((ctype+1)[c]&(_D)) // is a digital.
20 #define isgraph(c) (((ctype+1)[c]&(_P|_U|_L|_D)) // is a graphic character.
21 #define islower(c) (((ctype+1)[c]&(_L)) // is a lowercase character.
22 #define isprint(c) (((ctype+1)[c]&(_P|_U|_L|_D|_SP)) // is a printable character.
23 #define ispunct(c) (((ctype+1)[c]&(_P)) // is a punctuation mark.
24 #define isspace(c) (((ctype+1)[c]&(_S)) // is a space, \f, \n, \r, \t, \v.
25 #define isupper(c) (((ctype+1)[c]&(_U)) // is an uppercase character.
26 #define isxdigit(c) (((ctype+1)[c]&(_D|_X)) // is a hexadecimal number.
27

// 以下の2つのマクロ定義では、マクロパラメータはプレフィックス（符号なし）なので、'c'
// が括弧で囲まれているはずなので、(c)であることがわかります。の複雑な表現である可能性があるた
// め、'c'は
// 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）
// a + b のマクロ定義は、明らかに間違っています。括弧書きの後、正しくは
28 // (符号なし) (a + b)と表されます。
29 #define isascii(c) (((unsigned) c)<=0x7f) // is an ASCII character.
30 #define toascii(c) (((unsigned) c)&0x7f) // convert to ASCII character.
30

// 以下の2つのマクロ定義で一時変数「_ctmp」を使用する理由は
// は、マクロのパラメータは、マクロ定義の中で一度しか使用できません。しかし、これは
// 複数のスレッドがこのパブリックな一時変数を使用する可能性があるため、マルチスレッドでは安全
// ではない
// を同時に使用するようになりました。そこで、カーネル2.2.xからは、これらの2つのマクロ定義を変更
// し、2つの
// 機能です。
31 #define tolower(c) (_ctmp=c,isupper(_ctmp)?_ctmp-('A'-'a'):_ctmp) // 小文字の文字にします。32
#define toupper(c) (_ctmp=c,islower(_ctmp)?_ctmp-('a'-'A'):_ctmp) // 大文字の文字にします。33
34 #endif

```

35

---

## 14.5 errno.h

### 14.5.1 Function

There is a variable called 'errno' in the UNIX type system or the standard C language. Whether this variable is needed in the C standard has caused a lot of controversy in the C standardization organization (X3J11). But the result of the debate was that the 'errno' was not removed, instead a header file named "errno.h" was created. Because the standardization organization wants each library function or data object to be declared in a corresponding standard header file.

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

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

### 14.5.2 Code annotation

プログラム 14-4 linux/include/errno.h

---

```

1 #ifndef _ERRNO_H_
2 #define _ERRNO_H_
3
4 /*
5 * ok, as I hadn't got any other source of information about
6 * possible error numbers, I was forced to use the same numbers
7 * as minix.
8 * Hopefully these are posix or something. I wouldn't know (and posix
9 * isn't telling me - they want $$$ for their f***ing standard).
10 *
11 * We don't use the _SIGN cludge of minix, so kernel returns must
12 * see to the sign by themselves.
13 */

```

14 \* 注意！このファイルを変更した場合は、strerror()を変更することを  
忘れないでください。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見ただけです。型を知る必要がある場合は  
// エラーが発生した場合、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
#define ERROR          99           // General error.

```

19		
20	#define EPERM	1           // The operation is not permitted.
21	#define ENOENT	2           // The file or directory does not exist.
22	#define ESRCH	3           // The specified process does not exist.
23	#define EINTR	4           // Interrupted system-call.
24	#define EIO	5           // Input/output error.
25	#define ENXIO	6           // Specified device or address doesn't exist.
26	#define E2BIG	7           // The parameter list is too long.
27	#define ENOEXEC	8           // The format of executable file is incorrect.
28	#define EBADF	9           // File handle (descriptor) is incorrect.
29	#define ECHILD	10          // The child process does not exist.
30	#define EAGAIN	11          // The resource is temporarily unavailable.
31	#define ENOMEM	12          // No enough memory.
32	#define EACCES	13          // No access permissions.
33	#define EFAULT	14          // The address is wrong.
34	#define ENOTBLK	15          // Not a block device file.
35	#define EBUSY	16          // The resource is busy.
36	#define EEXIST	17          // File already exists.
37	#define EXDEV	18          // Illegal connection to device.
38	#define ENODEV	19          // The device does not exist.
39	#define ENOTDIR	20          // Not a directory file.
40	#define EISDIR	21          // Is a directory file.

---

```

41 #define EINVAL      22          // Invalid argument.
42 #define ENFILE       23          // The system has too many open files.
43 #define EMFILE       24          // Too many open files.
44 #define ENOTTY       25          // Inappropriate IO (no tty terminal).
45 #define ETXTBSY     26          // (No longer use).
46 #define EFBIG        27          // File size too big.
47 #define ENOSPC       28          // The device is full (the device has no space).
48 #define ESPIPE        29          // Invalid file pointer relocation.
49 #define EROFS        30          // The file system is read only.
50 #define EMLINK       31          // Too many links.
51 #define EPIPE         32          // The pipe is wrong.
52 #define EDOM          33          // Domain error.
53 #define ERANGE        34          // The result range error.
54 #define EDEADLK      35          // Resource deadlocks.
55 #define ENAMETOOLONG 36          // The filename is too long.
56 #define ENOLCK        37          // No locks are available.
57 #define ENOSYS        38          // Not yet implemented.
58 #define ENOTEMPTY    39          // The directory is not empty.
59
60 /* Should never be seen by user programs */
61 #define ERESTARTSYS   512         // Re-execute the system-call.
62 #define ERESTARTNOINTR 513         // Re-execute the system-call, no interrupt.
63
64 #endif
65

```

---

## 14.6 fcntl.h

### 14.6.1 Function

The fcntl.h file is the file control option header file. It mainly defines the file control function fcntl () and some of the options used in the file creation or open function. The fcntl() function is implemented in the linux/fs/fcntl.c file, and is used to perform various specified operations on the file descriptor (handle). The specific operation is specified by the function parameter cmd (command). .

### 14.6.2 Code annotation

プログラム 14-5 linux/include/fcntl.h

---

```
1 ifndef _FCNTL_H
2 define _FCNTL_H
3
4 #include <sys/types.h>      // Type header file. The basic system data types are defined.
5
6 /* open/fcntl - NOCTTY, NDELAY isn't implemented yet */
7 define O_ACCMODE      00003          // File access mode mask.
8     define O_RDONLY       00           // Open file in read-only mode.
```

```

8 #define O_WRONLY          01           // Open file in write-only mode.
9 #define O_RDWR             02           // Open file in read-write mode.
10 // 以下は、open()のファイル作成および操作フラグです。上記の
11 // アクセスマードを「bit or」モードにする。
12 #define O_CREAT            00100      /* not fcntl */ // Create if file does not exist.
13 #define O_EXCL             00200      /* not fcntl */ // Exclusive use of file.
14 #define O_NOCTTY            00400      /* not fcntl */ // No control terminal.
15 #define O_TRUNC             01000      /* not fcntl */ // truncated to zero if write operation.
16 #define O_APPEND            02000      // Open file in append mode.
17 #define O_NONBLOCK          04000      /* not fcntl */ // Open file in non-blocking manner.
18
19 /* Defines for fcntl-commands. Note that currently
20 * locking isn't supported, and other things aren't really
21 * tested.
22 */
23 // フ F_DUPFD            0           /* dup */           // duplicate a file handle.
24 // イ
25 // ルデ
26 // イス
27 // クリ
28 // プタ
29 // ーの
30 // 操作
31 // 関数
32 // fcntl(
33 // )で使
34 // 用す
35 // るコ
36 // マン
37 // ド
38 // (cmd
39 // )で
40 // す。
41 #define F_GETFD            1           /* get f_flags */ // get file handle flags (FD_CLOEXEC).
42 #define F_SETFD             2           /* set f_flags */ // set file handle flags.
43 #define F_GETFL             3           /* more flags (cloexec) */
44 #define F_SETFL             4           // get/set file state flag & access mode.
45 // Below are file lock commands. The parameter 'lock' of fcntl() points to flock structure.
46 #define F_GETLK             5           /* not implemented */ // get flock that blocks the lock.
47 #define F_SETLK             6           // Set (F_RDLCK or F_WRLCK) or clear lock (F_UNLCK).
48 #define F_SETLKW            7           // Set or clear the lock in wait mode.

```

```

31
32 /* for F_[GET|SET]FL */
33                                     // 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。
34 #define FD_CLOEXEC      1      /* actually anything with low bit set goes */
35
35 /* Ok, these are locking features, and aren't implemented at any
36 * level. POSIX wants them.
37 */
38 #define F_RDLCK          0      // Share or read file lock.
39 #define F_WRLCK          1      // Exclusive or write file lock.
40 #define F_UNLCK          2      // File unlock.
41
42 /* Once again - not implemented, but ... */
43                                     // 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。
44                                     // 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
43 // (l_len)、そしてロックを実装するpidです。
44 struct flock {
45     short l_type;           // Lock type (F_RDLCK, F_WRLCK, F_UNLCK).
46     short l_whence;         // Start offset (SEEK_SET, SEEK_CUR or SEEK_END).
47     off_t l_start;          // The beginning of the lock. Relative offset (in bytes).
48     off_t l_len;            // The size of the lock; if 0, it is the end of the file.
49     pid_t l_pid;            // The process id of the lock.
49 };
50
50                                     // 上記のフラグやコマンドを使った関数のプロトタイプを以下に示します。
51                                     // 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
52                                     // 作成されるファイルの//、'mode'はプロパティ (include/sys/stat.h参照)。
51 extern int creat(const char * filename, mode_t mode);
52                                     // 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
53                                     // 'fildes' は
54                                     // 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。
55                                     // 関数は次のような形になります。
56                                     // int fcntl(int fildes, int cmd).
57                                     // int fcntl(int fildes, int cmd, long arg)。
58 // int fcntl(int fildes,int cmd, struct flock *lock); 52 extern int
59 fcntl(int fildes,int cmd, ...);
60                                     // ファイルを開く。ファイルとファイルハンドルの間の接続を確立するために使用されます。
61                                     // パラメータ'flags'は、上記の7~17行目のフラグを組み合わせたものです。
62
62 extern int open(const char * filename, int flags, ...); 54
63 #endif
64

```

## 14.7 signal.h

### 14.7.1 Functionality

Signals provide a way to handle asynchronous events, and signals are also known as soft interrupts. By sending a signal to a process, we can control the execution state of the process (pause, resume, or terminated). The signal.h file defines the names and basic operational functions of all the signals used in the kernel. The

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。

14.7.2 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。

### 14.7.3 Code annotation

[1](#) プログラム 14-6 linux/include/signal.h

```

2 #ifndef _SIGNAL_H_
3 #define _SIGNAL_H_
4
5 #include <sys/types.h>           // Type header file. The basic system data types are defined.
6
7 typedef int __sig_atomic_t;       // define signal atomic operation type.
8 typedef unsigned int __sigset_t;  /* 32 bits */ // define signal set type.
9
10#define _NSIG                 32      // number of signals.
11#define NSIG                  _NSIG    // NSIG = _NSIG
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れています
13#define SIGHUP                1      // Hang Up      -- Hang up the control terminal or process.
14#define SIGINT                2      // Interrupt   -- Interrupt from the keyboard.
15#define SIGQUIT               3      // Quit        -- Exit command from the keyboard.
16#define SIGILL                4      // Illeagle    -- Illegal instruction.
17#define SIGTRAP               5      // Trap        -- Track breakpoints.
18#define SIGABRT               6      // Abort       -- Abort
19#define SIGIOT                6      // IO Trap     -- IO Trap
20#define SIGUNUSED              7      // Unused      -- Unused
21#define SIGFPE                8      // FPE         -- Coprocessor error.
22#define SIGKILL               9      // Kill        -- Force the process to terminate.
23#define SIGUSR1               10     // User1       -- User defined signal 1.
24#define SIGSEGV               11     // Segment Violation -- Invalid memory reference.
25#define SIGUSR2               12     // User2       -- User defined signal 2.
26#define SIGPIPE               13     // Pipe        -- Pipe write error, no reader.
27#define SIGALRM               14     // Alarm       -- Real-time timer alarm.

```

```

27 #define SIGTERM      15    // Terminate -- Process terminated.
28 #define SIGSTKFLT   16    // Stack Fault -- Stack error (coprocessor).
29 #define SIGCHLD      17    // Child      -- Child process is stopped or terminated.
30 #define SIGCONT      18    // Continue   -- Resume the execution of process.
31 #define SIGSTOP      19    // Stop       -- Stop the execution of process.
32 #define SIGTSTP     20    // TTY Stop   -- Stop sig sent by process, can be ignored.
33 #define SIGTTIN     21    // TTY In     -- Background process requests input.
34 #define SIGTTOU     22    // TTY Out    -- Background process requests output.
35
36 /* Ok, I haven't implemented sigactions, but trying to keep headers POSIX */
// 0.12カーネルではsigaction()が実装されているので、上記のコメントは廃止されています。
// 以下は、'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。
// Σ(° Ⅲ)
// 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。
// SA_INTERRUPT - シグナルによって中断された後、システムコールは再開されません。
// SA_NOMASK   - This signal is not blocked from being received in its signal handler.

```

```

37 // 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。
38 #define SA_NOCLDSTOP      1
39 #define SA_INTERRUPT     0x20000000
40 #define SA_NOMASK        0x40000000
41 #define SA_ONESHOT       0x80000000
41

// 以下の定数はsigprocmask(how, )で使用されます -- 与えられたシグナルを追加/削除するために。
// ブロッキングシグナルセットへの // の出入りや、ブロッキングシグナルセット（マスクコード）の変更
// を行います。
42 // この関数の動作を変更するために使用します。
43 #define SIG_BLOCK        0 /* for blocking signals */
44 #define SIG_UNBLOCK       1 /* for unblocking signals */
45 #define SIG_SETMASK      2 /* for setting the signal mask */
45

// 以下の3つの定数記号は、いずれも戻り値のない関数ポインタを表しています。
// の値を持ち、INTの整数パラメータを持つ。これらの3つのポインタは、論理的にはアドレス
// 実質的に不可能な機能の//。の2番目のパラメータとして使用することができます。
// 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
//      SIG_DFL          ((void (*) (int))0)    /* default signal handling */
kerne
l/sign
al.c
の
156-
158
行目
をご
覧く
ださ
い。
46 #define
47 #define SIG_IGN         ((void (*) (int))1)    /* ignore signal */
48 #define SIG_ERR        ((void (*) (int))-1)   /* error return from signal */
49

50 // 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。
51 #ifdef notdef
52 #define sigemptyset(mask) ((*(mask) = 0), 1)    // Clear mask.
53 #define sigfillset(mask) ((*(mask) = ~0), 1)   // All bits of the mask are set.
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もできます。
// '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。

```

```
55 // SA_NOMASKフラグを使用します。
56 struct sigaction {
57     void (*sa_handler)(int);
58     sigset_t sa_mask;
59     int sa_flags;
60     void (*sa_restorer)(void);
61 };
62 // 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
63 // ます。
64 // この関数は、2つの引数を取ります：1つは、シグナル _sig を
65 // 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。
66 // 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
67 // (int))となります。戻り値はなく、シグナルの元のハンドルとなります。
68 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。
64 // これは kill(getpid(), sig)と同じです。kernel/exit.c, line 205 を参照してください。
65 int raise(int sig);
66 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
67 // セット (1 - はい、0 - いいえ、-1 - エラー)。
68 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
70 // プロセス中にブロックされたシグナルセット。
71 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'ポインタが
72 // 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。
73 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

74 //呼び出しの前に

75 int sigsuspend(sigset\_t \*sigmask);

// 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。

76 // kernel/signal.c プログラムです。

77 int sigaction(int sig, struct sigaction \*act, struct sigaction \*oldact);

74

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

76

---

## 14.8 stdarg.h

### 14.8.1 Function

One of the biggest features of the C language is that it allows programmers to customize functions with

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。

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

### 14.8.3 Code annotation

## 1 プログラム 14-7 linux/include/stdarg.h

```
2 #ifndef STDARG_H
3 #define STDARG_H
4
5
6 /* Amount of space required in an argument list for an arg of type TYPE.
7   TYPE may alternatively be an expression whose type is used. */
8
9 // 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
10 // intの長さ(4)の
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 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" の項を参照し
22 // てください。
23 // マニュアル「Target Description Macros」で説明しています。
24 #define va_start(AP, LASTARG)
```

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14 (AP = ((char *) &(LASTARG) + va_rounded_size (LASTARG)))
15 #else
16 #define va_start(AP, LASTARG) \
17   (_builtin_saveregs (), \
18   AP = ((char *) &(LASTARG) + va_rounded_size (LASTARG)))
19 #endif
20
// 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 は、以
下のように変更できます。
// 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なり
ません。
21 // はすべてのパラメータを読みました。
22 void va_end (va_list);           /* Defined in gnulib */
23 #define va_end(AP)
24
// 次のマクロは、式を拡張して、同じタイプと値を持つ
// 次に渡されるパラメータ。デフォルトの値としては、va_arg は、文字、符号なし

```

```

// 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へは、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
24 // 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。
25 #define va_arg(AP, TYPE) \
26   (AP += __va_rounded_size (TYPE), \
27    *((TYPE *) (AP - __va_rounded_size (TYPE)))) \
27
28#endif /* _STDARG_H */
29

```

## 14.9 stddef.h

### 14.9.1 Functionality

The stddef.h header file is created by the C standardization organization (X3J11), and the meaning of the file name is the standard (std) definition (def). It is mainly used to store "standard definitions" such as some commonly used constant symbols and function prototypes in UNIX-like systems. Another confusing header file is stdlib.h, which is also created by the standardization organization, and is mainly used to declare various function prototype declarations that are not related to other header file types. But the contents of these two header files often make it impossible to figure out which declarations are in which header file.

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

- float.h Describe the floating point representation feature;
- limits.h Describe the integer representation characteristics;
- stdarg.h Provides macro definitions for accessing variable parameter list.

スタンダードアロン環境で使用されるその他の型やマクロの定義は、stddef.hファイルに置くべきですが、後の組織メンバーがこの制限を緩和したため、いくつかの定義が複数のヘッダーファイルに存在することになりました。例えば、NULLマクロの定義は、他の4つのヘッダ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 Code annotation

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;           // The type of result of subtracting two pointers.
7 #endif
8
9 #ifndef SIZE_T
10 #define SIZE_T
11 typedef unsigned long size_t;     // The type of result returned by sizeof().
12 #endif
13
14 #undef NULL
15 #define NULL ((void *)0)          // null pointer.
16

// 以下は、型の中のメンバーのオフセット位置を計算するマクロを定義しています。
// このマクロを使うと、メンバー（フィールド）の先頭からのバイトオフセットを求めることができます

// を含む構造体タイプの中で、その構造体のこのマクロの結果は、整数
// size_t型の定数表現。ここではトリック的な使い方をします。((TYPE *)0)は、整数の
// 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
// 結果。

17 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)-
>MEMBER) 18
19 #endif
20

```

---

## 14.10 string.h

### 14.10.1 Functionality

The string.h header file defines all string manipulation functions as inline functions, and uses inline assembly language to improve execution speed. In addition, a NULL macro and a SIZE\_T type are defined at the beginning of the file.

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

「extern」と「inline」を空に定義し、次にstring.hというヘッダーファイルを入れています。したがって、string.cプログラムは、実際にはstring.hヘッダーファイルで宣言された関数の実装コードを含んでいます。

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

## 14.10.2 Code annotation

プログラム14-9 linux/include/string.h

```

1 #ifndef STRING_H
2 #define STRING_H
3
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
16 15 /*
17 * This string-include defines all string functions as inline
18 * functions. Use gcc. It also assumes ds=es=data space, this should be
19 * normal. Most of the string-functions are rather heavily hand-optimized,
20 * see especially strtok,strstr,str[c]spn. They should work, but are not
        は、完全にレジスタリーン内で行われま
        す。ストリングス命令はライになってい
        る」という不明確なコード:-)
26
    //// コピー元の文字列をコピー先の文字列にNULL文字に遭遇するまでコピーします。
    // Params: dest - コピー先の str ポインタ, src - コピー元の str ポインタ。
    // 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。
29 27 extern inline char * strcpy(char * dest,const char *src) 28 {
30     __asm ("cld\n"           // Clear direction flag.
31         "1:|tlodsb|n|t"      // Load 1 byte from DS:[ESI] into AL and update ESI.
32         "stosb|n|t"          // Store the byte in AL to ES:[EDI] and update EDI.
33         "testb %%al,%%al|n|t" // The byte just stored is 0?
34         "jne 1b"              // If not, jump backward to label 1, else end.
35         :: "S" (src), "D" (dest):"si","di","ax");
36     return dest;            // Returns the destination str pointer.
37
    //// コピー元の文字列のカウントバイトをコピー先にコピーします。
    // source strの長さがcountバイトに満たない場合、null charsをdest stringに追加します。
    // パラメータ : dest - コピー先の str ポインタ, src - コピー元の str ポインタ, count - 数字
// コピーされたバイト数。0 - esi(src), %1 - edi(dest), %2 - ecx(count). 38
extern inline char * strncpy(char * dest,const char *src,int count) 39 {
40     asm ("cld\n"           // Clear direction flag.
41         "1:|tdec1 %2|n|t"    // Register ECX-- (count--).
42         "js 2f|n|t"          // If count<0 then jump forward to label 2 and end.

```

```

43      "lodsb|n|t"           // Get 1 byte from DS:[ESI] to AL, and ESI++.
44      "stosb|n|t"           // Store the byte to ES:[EDI], and EDI++.
45      "testb %%al, %%al|n|t" // Is this byte 0?
46      "jne 1b|n|t"           // If not, jump forward to label 1 to continue copy.
47      "rep|n|t"              // Else, store remain number of NULLs into dest str.
48      "stosb|n"
49      "2:""
50      :: "S" (src), "D" (dest), "c" (count): "si", "di", "ax", "cx";
51  return dest;           // Returns the dest string pointer.
52 }
53

```

//// コピー元の文字列をコピー先の文字列の末尾にコピーします。

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

// アセンブリコードでは、 %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1)。

[54 extern inline char \\* strcat\(char \\* dest,const char \\* src\) 55 {...](#)

```

57 __asm ("cld|n|t"
58     "repne|n|t"           // Compare AL and ES:[EDI] byte and update EDI++,,
59     "scasb|n|t"           // Until byte in dest is 0, the EDI points to last byte.
60     "dec1 %1|n"           // Let ES:[EDI] points to the location of 0.
61     "1:|tlodsb|n|t"       // Take source str byte DS:[ESI] to AL, and ESI++.
62     "stosb|n|t"           // Store this byte to ES:[EDI] and EDI++.
63     "testb %%al, %%al|n|t" // Is this byte 0?
64     "jne 1b"               // If not, jump back to label 1 to continue, else end.
65     :: "S" (src), "D" (dest), "a" (0), "c" (0xffffffff): "si", "di", "ax", "cx";
66  return dest;           // Returns the dest str pointer.
67 }

```

//// 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。

// 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。

// アセンブリコードでは、 %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1), %4 - (count) となります。

[70 68 extern inline char \\* strncat\(char \\* dest,const char \\* src,int count\) 69 {...](#)

```

71 __asm ("cld|n|t"
72     "repne|n|t"           // Compare AL and ES:[EDI], and update EDI++,,
73     "scasb|n|t"           // Until 0 in dest, the EDI points to last byte.
74     "dec1 %1|n|t"         // Let ES:[EDI] points to the location of 0.
75     "movl %4, %3|n"       // Place the bytes to be copied into ECX.
76     "1:|tdec1 %3|n|t"     // ECX-- (counts from 0).
77     "js 2f|n|t"           // Ecx < 0 ?, yes, jump forward to the label 2.
78     "lodsb|n|t"           // Otherwise take the bytes at DS:[ESI] to AL, ESI++.
79     "stosb|n|t"           // Store to ES:[EDI], and EDI++.
80     "testb %%al, %%al|n|t" // The byte value is 0?
81     "jne 1b|n"             // If not, jump back to label 1 and continue copying.
82     "2:|txorl %2, %2|n|t" // Clear AL to zero.
83     "stosb"                // Save to ES:[EDI].
84     :: "S" (src), "D" (dest), "a" (0), "c" (0xffffffff), "g" (count)
85     : "si", "di", "ax", "cx";
86  return dest;           // Returns the dest string pointer.
87 }

```

//// 2つの文字列を比較します。

// パラメータ： cs-文字列1、ct-文字列2。

```
// %0 - eax(res) の戻り値、%1 - edi(cs) の str1 ポインタ、%2 - esi(ct) の str2 ポインタ。  
extern inline int strcmp(const char * cs, const char * ct)
```

88  
89 {  
90 register int \_res\_asm ("ax"); // \_res is a register variable (eax).  
91 \_asm ("cld\n")

```

92      "1:|tlodsb|n|t"           // Get str2 byte DS:[ESI] to AL, and ESI++.
93      "scasb|n|t"             // Compare AL with str1 byte ES:[EDI], and EDI++.
94      "jne 2f|n|t"             // If they are not equal, jump forward to label 2.
95      "testb %%al,%%al|n|t"    // Is this byte 0 (end of string)?
96      "jne 1b|n|t"             // No, jump back to label 1, and continue comparison.
97      "xorl %%eax,%%eax|n|t"   // Yes, EAX is cleared, jumps forward to label 3, end.
98      "jmp 3f|n"
99      "2:|tmovl $1,%%eax|n|t"  // EAX is set to 1.
100     "jl 3f|n|t"              // If str2 < str1, returns a positive value, end.
101     "negl %%eax|n"          // Else EAX = -EAX, return a negative value, and end.
102     "3:""
103     : "=a" (__res): "D" (cs), "S" (ct): "si", "di";
104     return __res;            // Returns the comparison result.
105 }
106

//// 文字列1は、文字列2の最初のカウント文字と比較されます。
// パラメータ：cs - str1, ct - str2, count - 比較する文字数。
// If str1 > str2, returns 1; str1 = str2 returns 0; str1 < str2 returns -1.
extern inline int strncmp(const char * cs, const char * ct, int count)

```



```

107 {
108 {
109 register int __res__asm__("ax"); // _res is a register variable (eax).
110 __asm__("cld\n"
111     "1: | tdec1 %3|n|t"           // Register ECX-- (count--).
112     "js 2f|n|t"                 // If count<0, then jump forward to label 2.
113     "lodsb|n|t"                // Load str2 char DS:[ESI] to AL, and ESI++.
114     "scasb|n|t"                // Compare AL with str1 char ES:[EDI], and EDI++.
115     "jne 3f|n|t"                // If they are not equal, jump forward to label 3.
116     "testb %%al, %%al|n|t"       // Is it a NULL char?
117     "jne 1b|n"                  // No, jump back to label 1, continue the comparison.
118     "2: | txorl %%eax, %%eax|n|t" // Yes, then EAX is cleared (return value).
119     "jmp 4f|n"                  // Jump forward to label 4 and end.
120     "3: | tmovl $1, %%eax|n|t"   // Set EAX to 1.
121     "jl 4f|n|t"                  // If the result is str2 < str1, 1 is returned.
122     "negl %%eax|n"              // Else EAX = -EAX, return a negative value and end.
123     "4:"
124     : "=a" (__res) : "D" (cs), "S" (ct), "c" (count) : "si", "di", "cx");
125 return __res; // Returns the comparison result.
126 }
127
128 ///////////////////////////////////////////////////////////////////
129 // Look for the first matching character in the string.
130 // Parameters: s - the string, c - the character to be found.
131 // In the assembly code, %0 - eax(__res), %1 - esi (str pointer s), %2 - eax (char c).
132 // Returns a pointer to the first occurrence of a matching character in the string. If no
133 // matching characters are found, a null pointer is returned.
134 extern inline char * strchr(const char * s, char c)
135 {

```

32

130 register char \* res asm ("ax"); 131 asm ("cld\\$\n\\$t")

```

133      "movb %%al, %%ah|n"          // Move the char to be compared to AH.
134      "1: |tlodsb|n|t"           // Get a string char DS:[ESI] to AL, and ESI++.
135      "cmpb %%ah, %%al|n|t"       // The char AL is compared with the specified char AH.
136      "je 2f|n|t"                // If they are the same, jump forward to the label 2.
137      "testb %%al, %%al|n|t"       // Is the char in AL a NULL char? (End of string?)
138      "jne 1b|n|t"                // If not, back to label 1 and continue comparison.
139      "movl $1, %1|n"             // If yes, no match char is found, and set ESI to 1.
140      "2: |tmovl %1, %0|n|t"       // Put pointer at the next byte of match char into EAX.
141      "dec1 %0"                  // Adjust the pointer to point to the matching char.
141      : "=a" ( res ) : "S" ( s ), "0" ( c ) : "si" ;
142 return_ res;                      // Returns the pointer.
143 }
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 \* strrchr(const char \* s,char c) 146 {。

147 register char \* res asm ("dx"); 148 asm ("cld\\$\n\\$t")

```

150      "movb %%al, %%ah|n"          // Move the char to be compared to AH.
151      "1: |tlodsb|n|t"           // Get a string char DS:[ESI] to AL, and ESI++.
152      "cmpb %%ah, %%al|n|t"       // The char AL is compared with the specified char AH.
153      "jne 2f|n|t"                // If not the same, jump forward to the label 2.
154      "movl %%esi, %0|n|t"         // Store the char pointer to EDX.
155      "dec1 %0|n"                  // Adjust the pointer to point to the matching char.
156      "2: |ttestb %%al, %%al|n|t"    // Is the char in AL a NULL char? (End of string?)
157      "jne 1b"                     // If not, back to label 1 and continue comparison.
157      : "=d" ( res ) : "0" ( 0 ), "S" ( s ), "a" ( c ) : "ax", "si" ;
158 return_ res;                      // Returns the pointer.
159 }
160

```

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

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

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

// str2に含まれるstr1の最初の文字列の長さを返します。

161 extern inline int strspn(const char \* cs, const char \* ct) 162 {。

163 register char \* res asm ("si"); 164 asm ("cld\\$\n\\$t")

```

166      "movl %4, %%edi|n|t"        // Calc the len of str2, store its pointer to EDI.
167      "repne|n|t"                 // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
168      "scasb|n|t"                  // If they are not equal, continue to compare (ECX--).
169      "not1 %%ecx|n|t"              // Each bit in ECX is inversed.
170      "dec1 %%ecx|n|t"              // ECX--, The length of str2 is obtained.
171      "movl %%ecx, %%edx|n"         // Put the length of str2 into EDX temporarily.
172      "1: |tlodsb|n|t"               // Put the str1 char DS:[ESI] to AL, and ESI++.
173      "testb %%al, %%al|n|t"         // Is the char equal to 0 (end of str1)?
174      "je 2f|n|t"                  // If yes, jump forward to label 2 and end.

```

49

65

```

174      "movl %4, %%edi|n|t"          // Get the str2 pointer into EDI.
175      "movl %%edx, %%ecx|n|t"      // Put the str2 length into ECX.
176      "repne|n|t"                // Compare AL and str2 char ES:[EDI], and EDI++.
177      "scasb|n|t"                // If they are equal, continue to compare.
178      "je 1b|n"                  // If they are equal, jump backward to label 1.
179      "2: |tdecl%0"              // ESI--, points to the char in str1 contained in str2.
180      : "=S" (_res): "a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
181      : "ax", "cx", "dx", "di");
182 return __res-cs;                 // Returns the length of the sequence of chars.
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。
// 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)。
extern inline int strcspn(const char * cs, const char * ct)

```

```

185 {
186
187 register char *_res_asm_("si");
188 __asm__("cld|n|t"
189         "movl %4, %%edi|n|t"          // Calc the len of str2, store its pointer to EDI.
190         "repne|n|t"                // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
191         "scasb|n|t"                // If they are not equal, continue to compare (ECX--).

```

```

192     "notl %%ecx|n|t"           // Calc the length of str2 by inverting each bits
193     "dec1 %%ecx|n|t"           // in ECX and decreasing ECX by one.
194     "movl %%ecx, %%edx|n|"    // Put str2 length into EDX temporarily.
195     "1:|tlodsb|n|t"          // Take the str1 char DS:[ESI] to AL, and ESI++.
196     "testb %%al, %%al|n|t"    // Is the char equal to 0 (end of str1)?
197     "je 2f|n|t"              // If yes, jump forward to label 2 and end.
198     "movl %4, %%edi|n|t"      // Get the str2 pointer into EDI.
199     "movl %%edx, %%ecx|n|t"    // Put the str2 length into ECX.
200     "repne|n|t"              // Compare AL and str2 char ES:[EDI], and EDI++.
201     "scasb|n|t"              // If they not are equal, continue to compare.
202     "jne 1b|n"               // If they not are equal, jump backward to label 1.
203     "2:|tdec1 %0"            // ESI--, points to the char in str1 contained in str2.
204     :="S" (_res):"a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
205     :"ax", "cx", "dx", "di";
206 return __res-cs;           // Returns the length of the sequence of chars.
207 }
208
// In string1, look for any first character contained in string2.
// Parameters: cs - string1 pointer, ct - string2 pointer.
// %0 -esi(_res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(str1 ptr cs), %4 -(str2 ptr ct).
// Returns the pointer in string1 containing the first character in string2.
209 extern inline char * strpbrk(const char * cs, const char * ct)
210 {
211 register char *_res_asm__("si");
212 __asm__("cld|n|t"
213     "movl %4, %%edi|n|t"      // Calc the len of str2, store its pointer to EDI.
214     "repne|n|t"              // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
215     "scasb|n|t"              // If they are not equal, continue to compare (ECX--).
216     "notl %%ecx|n|t"          // Calc the length of str2 by inverting each bits
217     "dec1 %%ecx|n|t"           // in ECX and decreasing ECX by one.
218     "movl %%ecx, %%edx|n|"    // Store str2 length into EDX temporarily.

```

```

219      "1:|tlodsb|n|t"           // Get the str1 char DS:[ESI] to AL, and ESI++.
220      "testb %%al,%%al|n|t"    // Is the char equal to 0 (end of str1)?
221      "je 2f|n|t"             // If yes, jump forward to label 2.
222      "movl %4,%%edi|n|t"    // Get the str2 pointer into EDI.
223      "movl %%edx,%%ecx|n|t" // Put the str2 length into ECX.
224      "repne|n|t"             // Compare AL and str2 char ES:[EDI], and EDI++.
225      "scasb|n|t"             // If they not are equal, continue to compare.
226      "jne 1b|n|t"             // If they not are equal, jump backward to label 1.
227      "decl %0|n|t"            // ESI--, points to a char in str1 contained in str2.
228      "jmp 3f|n"               // Jump forward to label 3 and end.
229      "2:|txorl %0,%0|n"     // If no match is found, it will return NULL.
230      "3:"                     // 
231      : "=S" (_res): "a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
232      : "ax", "cx", "dx", "di";
233 return __res;                      // Returns the pointer in str1.
234 }
235

//// In string1, look for the first substring that matches the entire string2.
// Parameters: cs - string1 pointer, ct - string2 pointer.
// %0 -eax(_res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(str1 ptr cs), %4 -(str2 ptr ct).
// Returns the first substring pointer in string1 that matches entire string2.
236 extern inline char * strstr(const char * cs, const char * ct)
237 {
238 register char * __res_asm__("ax");
239 __asm__("cld|n|t" \
240         "movl %4,%%edi|n|t"           // Calc the len of str2, store its pointer to EDI.
241         "repne|n|t"                 // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
242         "scasb|n|t"                // If they are not equal, continue to compare (ECX--).
243         "notl %%ecx|n|t"           // Calc str2 len by inversing all bits and dec 1 in ECX.
244         "decl %%ecx\n\t" /* NOTE! This also sets Z if searchString== */
245         "movl %%ecx,%%edx\n\t"       // Store str2 length into EDX temporarily.
246         "1:|tmovl %4,%%edi|n|t"    // Get the str2 pointer into EDI.
247         "movl %%esi,%%eax|n|t"     // Copy the str1 pointer to EAX.
248         "movl %%edx,%%ecx|n|t"     // Put the str2 length into ECX.
249         "repe|n|t"                  // Comp str1,str2 char DS:[ESI],ES:[EDI], ESI++, EDI++.
250         "cmpsb|n|t"                // If the chars are equal, they continue to compare.
251         "je 2f\n\t" /* also works for empty string, see above */
252         "xchgl %%eax,%%esi|n|t"    // str1 ptr => ESI, comparison result str1 ptr => EAX.
253         "incl %%esi|n|t"           // str1 pointer points to the next char.
254         "cmpb $0,-(%eax)|n|t"     // Is the byte pointed to by ptr in str1 (EAX-1) 0?
255         "jne 1b|n|t"               // If not, go labell1, continue comp from 2nd char of str1.
256         "xorl %%eax,%%eax|n|t"     // Clear EAX, indicating that no match was found.
257         "2:"                     // 
258         : "=a" (_res): "0" (0), "c" (0xffffffff), "S" (cs), "g" (ct)
259         : "cx", "dx", "di", "si";
260 return __res;                      // Returns the result.
261 }
262

//// 文字列の長さを計算する。
// パラメータ:s - 文字列。
// %0 - ecx(res), %1 - edi(string pointer s), %2 - eax(0), %3 - ecx(0xffffffff) です。
// 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"); // res is a register variable (ECX).
266 __asm__ ("cld\n|t" // Clear the direction flag.
267     "repne\n|t" // AL(0) is compared with char es:[edi] in the string,
268     "scasb\n|t" // If they are not equal, continue to compare.
269     "notl %0\n|t" // Inverse each bit in ECX.
270     "dec1 %0" // ECX--, the length of the string is obtained.
271     : "=c" (res) : "D" (s), "a" (0), "0" (0xffffffff) : "di");
272 return_ res; // Returns the string length.
273 }
274

```

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

275 extern char \* strtok; // string token.

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、トークンが返される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をスキップし続けます。
// 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。

```

279 277 extern inline char \* strtok(char \* s,const char \* ct) 278 { ...

```

280 register char * res asm ("si");
281 __asm__ ("testl %1,%1\n|t" // First test if ESI (str1 pointer s) is NULL.
282     "jne 1f\n|t" // If not, means the first call, and jump label 1.
283     "testl %0,%0\n|t" // If NULL, means subsequent call, test EBX( strtok).
284     "je 8f\n|t" // If EBX is NULL, it cannot be processed, jump to end.
285     "movl %0,%1\n" // Copy the EBX pointer(current str1 ptr) to ESI.
286     "1:\txorl %0,%0\n|t" // Clear EBX.
287     "movl $-1,%%ecx\n|t" // Set ECX = 0xffffffff.
288     "xorl %%eax,%%eax\n|t" // Clear EAX.
289     "cld\n|t" // Clear direction flag.
290     "movl %4,%%edi\n|t" // Let's find the str2 length. EDI points to str2.
291     "repne\n|t" // Compare AL(0) with char at ES:[EDI], and EDI++.
292     "scasb\n|t" // Until end of str2 (null char), or count ECX = 0.
293     "notl %%ecx\n|t" // Inverse ECX and subtract 1, to get str2 length.
294     "dec1 %%ecx\n|t" // If str2 len is 0, go forward to label 7.
295     "je 7f\n|t" /* empty delimiter-string */
296     "movl %%ecx,%%edx\n" // Store str2 length to EDX temporarily.
297     "2:\tlodsb\n|t" // Get str1 char DS:[ESI] to AL, and ESI++.
298     "testb %%al,%%al\n|t" // Is the char 0 (end of str1)?
299     "je 7f\n|t" // If yes, jump forward to label 7.
300     "movl %4,%%edi\n|t" // EDI again points to first char of the str2.
301     "movl %%edx,%%ecx\n|t" // Copy str2 length into the counter ECX.

```

```
302      "repne |n|t"           // Compare str1 char AL with all chars in str2,  
303      "scash |n|t"           // to check if the char is a separator.  
304      "je 2b |n|t"           // If found it in str2, jump backward to label 2.
```

```

304     "dec1 %1|n|t"           // Else the str1 pointer ESI points to the char now.
305     "cmpb $0, (%1)|n|t"    // Is this a NULL character?
306     "je 7f|n|t"             // If yes, jump forward to label 7.
307     "movl %1, %0|n|"       // The char pointer ESI is stored in EBX.
308     "3: |tlodsb|n|t"        // Get next char of str1 to DS:[ESI], and ESI++.
309     "testb %%al, %%al|n|t"  // Is the char 0 (end of str1)?
310     "je 5f|n|t"             // If yes, means str1 ends and jump forward to label 5.
311     "movl %4, %%edi|n|t"   // EDI again points to first char of the str2.
312     "movl %%edx, %%ecx|n|t" // Copy str2 length into the counter ECX.
313     "repne|n|t"             // Compare str1 char AL with all chars in str2,
314     "scasb|n|t"             // to check if the char is a separator.
315     "jne 3b|n|t"             // If not, jump to label3 to detect the next char in str1.
316     "dec1 %1|n|t"             // If it is a separator, ESI--, points to the separator.
317     "cmpb $0, (%1)|n|t"    // Is this a NULL character?
318     "je 5f|n|t"             // If yes, jump forward to label 5.
319     "movb $0, (%1)|n|t"    // If not, replaced the separator with a NULL char.
320     "incl %1|n|t"             // ESI points to next char in str1 (remaining str).
321     "jmp 6f|n"               // Jump forward to label 6.
322     "5: |txorl %1, %1|n|"  // Clear ESI.
323     "6: |tcmpb $0, (%0)|n|t" // Does the EBX pointer point to a NULL char?
324     "jne 7f|n|t"             // If not, jump forward to label 7.
325     "xorl %0, %0|n|"        // If yes, let the EBX = NULL. (end of str1)
326     "7: |ttestl %0, %0|n|t"  // EBX is NULL?
327     "jne 8f|n|t"             // If not, jump forward to label 8, end return.
328     "movl %0, %1|n|"        // Set ESI to NULL, means no more token in str1.
329     "8:"                     // Clear ESI.
330     : "=b" (_res), "=S" ( strtok )
331     : "0" ( strtok ), "1" ( s ), "g" ( ct )
332     : "ax", "cx", "dx", "di" );
333 return _res;                                // Returns pointer to the new token.
334 }
335

//// メモリのコピーです。ソースアドレスsrcからデスティネーションアドレスdestにnバイトをコピーします。
// Params: dest - デスティネーション・アドレス、src - ソース・アドレス、n - バイト数。
// %0 - ecx(n), %1 - esi(src), %2 - edi(dest) です。
336 extern inline void * memcpy(void * dest,const void * src, int n) 337 {
337     __asm ("cld|n|t"           // Clear direction flag.
338         "rep|n|t"              // Repeatedly copying ECX bytes,
339         "movsb"                 // From DS:[ESI++] to ES:[EDI++].
340         :: "c" (n), "S" (src), "D" (dest)
341         : "cx", "si", "di" );
342     return dest;                // Returns the dest address.
343 }
344 }

//// メモリの移動です。メモリのコピー操作と同じですが、移動の方向を考慮します。
// Params: dest - デスティネーション・アドレス、src - ソース・アドレス、n - バイト数。
// If (dest < src) then: %0 - ecx(n), %1 - esi(src), %2 - edi(dest);
// それ以外の場合 : %0 - ecx(n), %1 - esi(src + n - 1), %2 - edi(dest + n - 1).
// この方向性の操作は、コピー時のデータの重なりを防ぐためのものです。
345 if (dest<src)
346 extern inline void * memmove(void * dest,const void * src, int n) 347 {
347

```

```

350 __asm ("cld\n\t"           // Clear direction flag.
351     "rep\n\t"             // From DS:[ESI++] to ES:[EDI++],
352     "movsb"               // Repeatedly copying ECX bytes..
352     :: "c" (n), "S" (src), "D" (dest)
353     : "cx", "si", "di");
354 else
355 __asm_ ("std\n\t"           // Set the direction and start copying from the end.
356     "rep\n\t"             // From DS:[ESI--] to ES:[EDI--],
357     "movsb"               // Repeatedly copying ECX bytes.
358     :: "c" (n), "S" (src+n-1), "D" (dest+n-1)
359     : "cx", "si", "di");
360 return dest;                // Returns the dest address.
361 }
362
363 ///////////////////////////////////////////////////////////////////
363 // Compare n bytes of two memory blocks, and don't stop comparing even if encounter NULL char.
363 // Params: cs - mem block1 address, ct - mem block2 address, count - number of bytes compared.
363 // %0 - eax(_res), %1 - eax(0), %2 - edi (mem block1), %3 - esi (mem block2), %4 - ecx(count).
363 // If block1 > block2 returns 1; block1 < block2, returns -1; block1 == block2, returns 0.
363 extern inline int memcmp(const void * cs, const void * ct, int count)
364 {
365 register int_res_asm_("ax");
366 __asm_ ("cld\n\t"
367     "repe\n\t"             // Compare DS:[ESI++] with ES:[EDI++].
368     "cmpsb\n\t"            // If equal, continue comparing. repeat ECX times.
369     "je 1f\n\t"            // If all the same, jump to label 1, return 0 (EAX).
370     "movl $1,%%eax\n\t"    // else set EAX to 1.
371     "jl 1f\n\t"            // If value of mem block2 < block1, jump to label 1.
372     "negl %%eax\n\t"       // else invert EAX value.
373     "1:"                  // Label for loop.
374     : "=a" (_res) : "0" (0), "D" (cs), "S" (ct), "c" (count)
375     : "si", "di", "cx");
376 return __res;                // Returns the compare result (in EAX).
377 }

378 ///////////////////////////////////////////////////////////////////
378 // nバイトのメモリブロックの中から、指定した文字を探す。
378 // 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。
378 // %0 - edi(res), %1 - eax(char c), %2 - edi(mem block address cs), %3 - ecx(num of bytes).
378 extern inline void * memchr(const void * cs, char c, int count)

```

```
379
380 {
381 register void *_res_asm_( "di" );
382 if ( !count )                                // If memory block size is 0, then NULL is returned.
383     return NULL;
384 __asm__( "cld\n" );
385     "repne\n"                                // Compare the char in AL with the ES:[EDI++],
386     "scasb\n"                                // Repeat if it is not equal (up to ECX times).
387     "je If\n"                                // If equal, jump forward to label 1.
388     "movl $1,%0\n"                            // else set EDI to 1.
389     "1:\tdecl %0"                            // Let EDI point to the char searched. (or NULL).
390     : "=D" (_res) : "a" (c), "D" (cs), "c" (count)
391     : "cx" );
392 return __res;                                // Return the char pointer.
393 }
```

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

---

```

395
396 {
397 __asm__ ("cld\n\t"
398         "rep\n\t"           // Repeat ECX times,
399         "stosb"             // Fill the char in AL into ES:[EDI++].
400         :: "a" (c), "D" (s), "c" (count)
401         : "cx", "di";
402 return s;                      // Returns the memory block pointer.
403 }
404
405 #endif
406

```

---

## 14.11 termios.h

### 14.11.1 Functionality

The termios.h file contains terminal I/O interface definitions, including termios data structures and some function prototypes for common terminal interface settings. These functions are used to read or set the properties of the terminal, line control, read or set the baud rate, and read or set the group ID of the terminal front end process. Although this is an early Linux header file, it is fully compliant with the current POSIX standard and has been appropriately extended.

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。

### 14.11.2 Code annotation

1 プログラム 14-10 linux/include/termios.h

```
2 #ifndef TERMIOS_H
3 #define TERMIOS_H
3
4 #include <sys/types.h>
5
6 #define TTY_BUF_SIZE 1024           // The buffer size in the tty (in bytes).
7
8 /* 0x54 は、これらを相対的に单一化するためのマジックナンバー ('T')です */
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。

```

10 // 端末のtermios構造体の情報を取得します (tcgetattr()参照)。
11 #define TCGETS          0x5401      // Terminal Command GET Settings.
12 // 端末のtermios構造体に情報を設定する (TCSANOWのtcsetattr()参照)。
13 #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。
   // affect the output (see tcsetattr(), TCSADRAIN option). (TCSETSW - TCSETS Wait)
14 #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
15 // 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)。
16 #define TCSETSF         0x5404
17 // termio構造体の属性情報を取る (tcgetattr()参照)。
18 #define TCGETA          0x5405
19 // termio構造体の属性情報を設定する (tcsetattr()、TCSANOWオプション参照)。
20 #define TCSETA           0x5406
   // termioの属性情報を設定する前に、すべてのデータを待つために
   // 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。
21 // 出力に影響を与えるパラメータ (tcsetattr()のTCSADRAINオプション参照)。
22 #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
23 // 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)。
24 #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
25 // (tcsendbreak()、tcdrain()参照)。
26 #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
27 // 入力は再び開かれます (tcflow()参照)。
28 #define TCFLSH          0x540B
   // 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
29 // 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)。
30 #define TIOCEXCL        0x540C      // Terminal I/O Control Exclude.
31 // 端末のシリアルラインの排他モードをリセットします。
32 #define TIOCNXCL        0x540D
33 // 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)。
34 #define TIOCSCTTY       0x540E
35 // 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)。
36 #define TIOCGPGRP        0x540F      // Terminal I/O Control Get PGRP.
37 // 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)。
38 #define TIOCSPGRP       0x5410

```

```
39 // 送信されていない出力キューの文字数を返します。
40 #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
41 // または制御端末の読み取り許可
42 #define TIOCSTI      0x5412
43 // 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)。
44 #define TIOCGWINSZ   0x5413
45 // 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)。
46 #define TIOCSWINSZ   0x5414
```

```

47 // 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)。
48 #define TIOCGET      0x5415
49 // 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。
50 #define TIOCMIS      0x5416
51 // 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。
52 #define TIOCMIC      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
53 MODEMへの//が有効になるように設定されます。
54 #define TIOCMSET      0x5418
    // ソフトウェアのキャリア検出フラグを取得します (1 - オン、0 - オフ)。ローカルに接続された端末
    // や
    // 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
55 // 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。
56 #define TIOCGSOFTCAR  0x5419
57 // ソフトウェアキャリア検出フラグを設定します (1 - オン、0 - オフ)。
58 #define TIOCSSOFTCAR 0x541A
59 // 入力キューの中で、まだフェッチされていない文字の数を返します。
60 #define FIONREAD     0x541B
61 #define TIOCINQ      FIONREAD
38
    // ウィンドウサイズ属性構造体。スクリーンベースのアプリケーションに使用することができます。
    TIOCGWINSZ の
    ioctl の // や TIOCSWINSZ コマンドを使って、この情報を読み取ったり設定したりすることができます。
        unsigned short ws_row;           // window character rows.

    ...
40
41     unsigned short ws_col;         // window character columns.
42     unsigned short ws_xpixel;      // window width in pixels.
43     unsigned short ws_ypixel;      // window height in pixels.
44 };
45
    // AT&Tシステムのターミオ構造 V.

```

```

46 #define NCC 8                                // the size of the control character array in termio.
47 struct _termio {
48     unsigned short c_iflag;                  /* input mode flags */
49     unsigned short c_oflag;                  /* output mode flags */
50     unsigned short c_cflag;                  /* control mode flags */
51     unsigned short c_lflag;                  /* local mode flags */
52     unsigned char c_line;                   /* line discipline */
53     unsigned char c_cc[NCC];                /* control characters */
54 };
55
// POSIXで定義されているtermios構造です。
56 #define NCCS 17                               // the size of the control character array in termios.
57 struct _termios {
58     tcflag_t c_iflag;                  /* input mode flags */
59     tcflag_t c_oflag;                  /* output mode flags */
60     tcflag_t c_cflag;                  /* control mode flags */
61     tcflag_t c_lflag;                  /* local mode flags */
62     cc_t c_line;                     /* line discipline */
63     cc_t c_cc[NCCS];                /* control characters */
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  
*/\* c\_cc characters \*/*

```

66
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]   = TIME    (\0), \0.  timer value (see below).
73 #define VMIN 6       // c_cc[VMIN]    = MIN     (\1), \1.  timer value.
74 #define VSWTC 7      // c_cc[VSWTC]   = SWTC    (\0), \0.  switch char.
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. suspend char.
78 #define VEOL 11           // c_cc[VEOL]     = EOL       (^O), \0.
79 #define VREPRINT 12        // c_cc[VREPRINT] = REPRINT   (^R), \022.
80 #define VDISCARD 13        // c_cc[VDISCARD] = DISCARD   (^O), \017.
81 #define VWERASE 14          // c_cc[VWERASE]  = WERASE    (^W), \027. word erase char.
82 #define VLNEXT 15           // c_cc[VLNEXT]   = LNEXT     (^V), \026.
83 #define VEOL2 16            // c_cc[VEOL2]    = EOL2      (^O), \0.
84

// 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。
8 #define IGNBRK 0000001      // Ignore the BREAK condition when input.
5
7
*
c
H
T
a
g
B
I
P
7
86
87 #define BRKINT 0000002      // Generates a SIGINT signal on BREAK.
88 #define IGNPAR 0000004      // Ignore the character of the parity error.
89 #define PARMRK 0000010      // Mark parity error.
90 #define INPCK 0000020       // Allow input to check parity.
91 #define ISTRIP 0000040      // Mask the 8th bit of the character.
92 #define INLCR 0000100       // Map line feed NL to a carriage return CR.
93 #define IGNCR 0000200      // Ignore the CR.
94 #define ICRNL 0000400       // Map CR to NL.
95 #define IUCLC 0001000       // Convert uppercase chars to lowercase chars.
96 #define IXON 0002000        // Allow start/stop (XON/XOFF) output control.
97 #define IXANY 0004000       // Allow any character to restart the output.
98 #define IXOFF 0010000       // Allow start/stop (XON/XOFF) input control.
99 #define IMAXBEL 0020000     // Rings when the input queue is full.
100

// The constants of various flags used in the output mode field c_oflag in the termios.
101 /* c_oflag bits */
102 #define OPOST 0000001       // Perform output processing.
103 #define OLCUC 0000002       // Convert lowercase chars to uppercase chars.
104 #define ONLCR 0000004       // Map NL to CR-NL.
105 #define OCRNL 0000010      // Map CR to NL.
106 #define ONOCR 0000020      // The CR is not output in the column 0.
107 #define ONLRET 0000040      // The NL performs the function of carriage return.
108 #define OFILL 0000100       // Use fill chars when deferred instead of time delays.
109 #define OFDEL 0000200       // The fill char is DEL. The default is NULL if not set.
110 #define NLDLY 0000400       // Choose a line feed delay.
111 #define NLO 0000000         // NL delay type 0.
112 #define NL1 0000400         // NL delay type 1.
113 #define CRDLY 0003000       // Choose a carriage return delay.
114 #define CRO 0000000         // CR delay type 0.

```

```

115 #define CR1 0001000 // CR delay type 1.
116 #define CR2 0002000 // CR delay type 2.
117 #define CR3 0003000 // CR delay type 2.
118 #define TABDLY 0014000 // Select horizontal TAB delay.
119 #define TAB0 0000000 // TAB delay type 0.
120 #define TAB1 0004000 // TAB delay type 1.
121 #define TAB2 0010000 // TAB delay type 2.
122 #define TAB3 0014000 // TAB delay type 3.
123 #define XTABS 0014000 // Replace TAB with spaces, it is the number of spaces.
124 #define BSDLY 0020000 // Select the backspace (BS) delay.
125 #define BS0 0000000 // BS delay type 0.
126 #define BS1 0020000 // BS delay type 1.
127 #define VTDLY 0040000 // Select vertical tabulation delay.
128 #define VT0 0000000 // VT delay type 0.
129 #define VT1 0040000 // VT delay type 1.
130 #define FFDLY 0040000 // Select Form feed delay.
131 #define FF0 0000000 // FF delay type 0.
132 #define FF1 0040000 // FF delay type 1.

133 // コントロールモードフィールドc_flagで使用されるフラグをtermios (8進数) で表したもの。
134 /* c_cflag bit meaning */
135 #define CBAUD 0000017 // Transmit baud rate bit mask.
136 #define B0 0000000 /* hang up */
137 #define B50 0000001 // baud rate 50.
138 #define B75 0000002
139 #define B110 0000003
140 #define B134 0000004
141 #define B150 0000005
142 #define B200 0000006
143 #define B300 0000007
144 #define B600 0000010
145 #define B1200 0000011
146 #define B1800 0000012
147 #define B2400 0000013
148 #define B4800 0000014
149 #define B9600 0000015
150 #define B19200 0000016
151 #define B38400 0000017 // baud rate 38400.
152 #define EXTA B19200 // Extended baud rate A.
153 #define EXTB B38400 // Extended baud rate B.

154 #define CSIZE 0000060 // Character bit width mask.
155 #define CS5 0000000 // 5 bits per character.
156 #define CS6 0000020 // 6 bits.
157 #define CS7 0000040 // 7 bits.
158 #define CS8 0000060 // 8 bits.
159 #define CSTOPB 0000100 // Set two stop bits instead of one.
160 #define CREAD 0000200 // Allow to receive.
161 #define PARENBT 0000400 // Enable parity operation.
162 #define PARODD 0001000 // The input check is odd check.
163 #define HUPCL 0002000 // Hang up after the last process is closed.
164 #define CLOCAL 0004000 // Ignore the MODEM control lines.
165 #define CIBAUD 03600000 /* input baud rate (not used) */

```

```

166 #define CRTSCTS 02000000000000 /* flow control */
167
168 /* c_lflag bits */
169 #define ISIG 0000001 // signal generated when device INTR, QUIT, SUSP or DSUSP .
170 #define ICANON 0000002 // Enable canonical mode (cooked mode).
171 #define XCASE 0000004 // If ICANON is set, terminal displays uppercase chars.
172 #define ECHO 0000010 // Echo the input characters.
173 #define ECHOE 0000020 // If ICANON, ERASE/WERASE erase the previous char/word.
174 #define ECHOK 0000040 // If ICANON, KILL char will erase the current line.
175 #define ECHONL 0000100 // If ICANON, NL is echoed even if ECHO is not enabled.

    SIGINT および SIGQUIT シグナルが発生しても、// 入出力キューはフラッシュされません。

176 // また、SIGSUSP信号が発生すると、入力キューがフラッシュされます。
177 #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シグナルを送信す
    る。
178 // 独自の制御端子を備えています。
179 #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。
    となり、X値は制御値+0x40となります。

    // If ICANON & IECHO, chars will be displayed when erasing.
180 #define ECHOKE 0004000 // If ICANON, KILL is echoed by erasing each char on the line.
181 #define FLUSHO 0010000 // Output is flushed. it can be toggled by DISCARD char.
182 #define PENDIN 0040000 // All chars in inqueue are reprinted when next char is read.
183 #define IEXTEN 0100000 // Enable implementation-defined input processing.

184
185 /* modem lines */
186 #define TIOCM_LE 0x001 // Line Enable.
187 #define TIOCM_DTR 0x002 // Data Terminal Ready.
188 #define TIOCM RTS 0x004 // Request to Send.
189 #define TIOCM_ST 0x008 // Serial Transfer.
190 #define TIOCM_SR 0x010 // Serial Receive.
191 #define TIOCM_CTS 0x020 // Clear To Send.
192 #define TIOCM_CAR 0x040 // Carrier Detect.
193 #define TIOCM RNG 0x080 // Ring indicate.
194 #define TIOCM_DSR 0x100 // Data Set Ready.

195 #define TIOCM_CD TIOCM_CAR
196 #define TIOCM RI TIOCM RNG
197

198 /* tcflow() and TCXONC use these */
199 #define TCOOFF 0 // Suspend output ("Terminal Control Output OFF").
200 #define TCOON 1 // Restart suspended output.
201 #define TCIOFF 2 // Send a STOP to stop device transmitting data to system.
202 #define TCION 3 // Send a START to start device transmitting to system.

203
204 /* tcflush() and TCFLSH use these */
205 #define TCIFLUSH 0 // Clear the received data but do not read it.
206 #define TCOFLUSH 1 // Clear the output data but not transmit it.
207 #define TCIOFLUSH 2 // Clear in/out data but not read/transmit it.

208
209 /* tcsetattr uses these */

```

```
210 #define TCSANOW      0      // Enable settings now.  
211 #define TCSADRAIN    1      // Change occurs after all output has been transmitted.
```

---

```

212 #define TCSAFLUSH      2          // Change occurs after input/output are all flushed.

213 // 以下の関数は、ビルド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。

214 // termios_p で参照される termios 構造体の受信ボーレートを返す。
215 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);

217 // termios構造体に送信ボーレートを設定する。
218 extern int cfsetospeed(struct termios *termios_p, speed_t speed);
219 // 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。
220 extern int tcdrain(int fildes);
221 // 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。
222 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

223 受信したが、まだ読まれていないすべてのデータとして、//。
224 extern int tcflush(int fildes, int queue_selector);
225 // 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。
226 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

227 // 一定時間連続して送信される。
228 extern int tcsendbreak(int fildes, int duration);
229 // 端末に関連するパラメータを設定するために、termios構造のデータを使用します。
230 extern int tcsetattr(int fildes, int optional_actions,
231         struct termios *termios_p);
225
226 #endif
227

```

---

## 14.11.3 Information

### 14.11.3.1 Control Character TIME、MIN

When the canonical mode flag ICANON is disabled, the input is in the non-canonical mode (raw mode). In non-canonical mode, input characters are not processed into lines, and input characters are immediately readable, without the need to enter (user type) carriage-return, line feed, etc. line-defining characters. Therefore, erasing and terminating processing will not occur. The settings of MIN (c\_cc[VMIN]) and TIME (c\_cc[VTIME]) in termios structure are used to determine how to handle the received characters, how many characters are read by the associated read() system-call, and when to return to the user program.

- ◆ 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（ $\text{MIN} > 0$ 、 $\text{TIME} > 0$ ）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。
  - ◆  $\text{MIN} > 0, \text{TIME} = 0$ :
- ◆ この場合、TIMEの値は0なので、タイマーは動作せず、MINだけが意味を持ちます。MIN文字を受信して初めて、待ち受けていた読み出し動作が満たされることになる（待ち受け動作はMIN文字を受信するまでスリープする）。これを用いてレコードベースのターミナルIOを読み取るプログラムは、読み取り動作が（任意に）無期限にブロックされることになる。
  - ◆  $\text{MIN} = 0, \text{TIME} > 0$ :
- ◆ この場合、 $\text{MIN}=0$ 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、 $\text{TIME} * 0.10$ 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。
  - ◆  $\text{MIN} = 0, \text{TIME} = 0$ :

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ニーに存在する文字数は、バッファに入力される文字数を待たずに返されます。

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

## 14.12 time.h

### 14.12.1 Functionality

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

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

#### 4.12.2

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。

#### 14.12.3 Code annotation

[1](#) プログラム 14-11 linux/include/time.h

```

2 #ifndef _TIME_H_
3 #define _TIME_H_
4
5 #ifndef _TIME_T_
6 #define _TIME_T_
7
8
9 #ifndef _SIZE_T_
10 #define _SIZE_T_
11 typedef unsigned int _size_t;
12
13
14 #ifndef NULL
15 #define NULL ((void *) 0)
16
17
18 #define CLOCKS_PER_SEC 100           // System clock tick frequency, 100HZ.
19
20 typedef long _clock_t;           // Ticks passed by system from the start of a process.
21
22 struct _tm {
23     int tm_sec;                  // Seconds [0, 59].
24     int tm_min;                  // Minutes [0, 59].
25     int tm_hour;                 // Hours [0, 59].
26     int tm_mday;                 // The number of days in a month [0, 31].
27     int tm_mon;                  // The number of months in a year [0, 11].
28     int tm_year;                 // The number of years since 1900.
29     int tm_wday;                 // One day of the week [0, 6] (Sunday =0).
30     int tm_yday;                 // One day in a year [0, 365].
31     int tm_isdst;                // Summer time sign. > 0 -in use; = 0 -not used; < 0 -invalid.
32 };
33

```

// うるう年かどうかをチェックするマクロです。  
34 #define \_isleap(year) \

---

```

35   ((year) % 4 == 0 && ((year) % 100 != 0 || (year) % 1000 == 0))
36
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。
37 // 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。
38 clock_t clock(void);
39 // 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。
40 time_t time(time_t * tp);
41 // 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。
42 double difftime(time_t time2, time_t time1);
43 // tm構造体で表される時間をカレンダー時間に変換する。
44 time_t mktime(struct tm * tp);
45
45 // 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。
46 char * asctime(const struct tm * tp);
47 // カレンダーの時間を "Wed Jun 30 21:49:08:1993\n" のように文字列に変換します。
48 char * ctime(const time_t * tp);
49 // 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。
50 struct tm * gmtime(const time_t *tp);
51 // 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。
52 struct tm * localtime(const time_t * tp);
53 // tm構造体が表す時間を、最大長の文字列'smax'に変換します。
54 // 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。
55 size_t strftime(char * s, size_t smax, const char * fmt, const struct tm * tp);
56 // 時間の変換情報を初期化し、変数znameの初期化を
57 // この関数は、時間変換機能で自動的に呼び出されます。
58 // タイムゾーンに関連付けられています。
59 void tzset(void);
60
60 #endif

```

---

## 14.13 unistd.h

### 14.13.1 Functionality

Standard symbol constants and types are defined in the unistd.h header file. There are many different symbol constants and types defined in this file, as well as some function declarations. If the symbol `_LIBRARY_` is defined in the program, it will also include the kernel system-call number and the inline assembly macro `_syscall0()`, `_syscall1()`, and so on.

### 14.13.2 Code annotation

[1](#) プログラム 14-12 linux/include/unistd.h

```

2 ifndef UNISTD_H
3 define UNISTD_H

```

3

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

// 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 // 整数值である。

```

5 #define _POSIX_VERSION 198808L
6
7 #define _POSIX_CHOWN_RESTRICTED /* rootのみがchownを行える（と思う） */
// Path names longer than NAME_MAX will generate an error and will not be automatically truncated.
8 #define _POSIX_NO_TRUNC /* no pathname truncation (but see in kernel) */
// _POSIX_VDISABLEは、端末の一部の特殊文字の機能を制御するために使用されます。
// 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
9 // は禁止です。
10#define _POSIX_VDISABLE '\0' /* character to disable things like ^C */
11 // 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。
12#define _POSIX_JOB_CONTROL
// 各プロセスは、保存されたset-user-IDと保存されたset-group-IDを持っています。
// 実装されました、どんな良いことがあっても */
// 標準入力ファイルのハンドル（記述子）番号。
// 標準出力ファイルのハンドル番号。
// 標準エラー出力ファイルのハンドル番号

18#define NULL ((void *)0) // Define the value of a null pointer.
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/* access */
22#define F_OK 0 // Check if the file exists.
23#define X_OK 1 // Check if it is executable (searchable).
24#define W_OK 2 // Check if it is writable.
25#define R_OK 4 // Check if it is readable.
26
27 // 以下のシンボル定数は、lseek() および fcntl() 関数で使用されます。
28/* lseek */
29#define SEEK_SET 0 // Set file read/write pointer to the offset.
30#define SEEK_CUR 1 // Set to the current value plus the offset.
31#define SEEK_END 2 // Set to the file length plus the offset.

// 以下のシンボル定数は、sysconf()関数で使用されます。
32/* _SC stands for System Configuration. We don't use them much */
33#define SC_ARG_MAX 1 // The maximum number of arguments.
34#define SC_CHILD_MAX 2 // The maximum number of child processes.
35#define SC_CLOCKS_PER_SEC 3 // Ticks per second.
36#define SC_NGROUPS_MAX 4 // The maximum number of groups.
37#define SC_OPEN_MAX 5 // The maximum number of opened files.
38#define SC_JOB_CONTROL 6 // Job control.
39#define SC_SAVED_IDS 7 // The saved identifier.
40#define SC_VERSION 8 // Version.

// pathconf()関数では、以下のシンボル定数を使用しています。
42/* more (possibly) configurable things - now pathnames */
43#define PC_LINK_MAX 1 // The maximum number of links.
44#define PC_MAX_CANON 2 // The maximum number of regular files.

```

```
45 #define PC_MAX_INPUT      3 // Maximum input length.  
46 #define PC_NAME_MAX       4 // The maximum length of the name.
```

```

47 #define PC_PATH_MAX      5 // The maximum length of a path.
48 #define PC_PIPE_BUF      6 // Pipe buffer size.
49 #define PC_NO_TRUNC       7 // The file name is not truncated.
50 #define PC_VDISABLE       8 // The specified control char is disabled.
51 #define PC_CHOWN_RESTRICTED 9 // Change the owner is restricted.
52
// <sys/stat.h> ファイル状態のヘッダファイルです。ファイルやファイルシステムの状態を表す構造体を
// 含む stat{}。
//     and constants.
// <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。
//     the process.
// <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。
//     resources used by processes.
// <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()の
//     prototype are defined.
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。
// システムコール関数表 (include/linux/sys.h参照) にある
62 #define 85 #define
63 #define 86 #define
64 #define 87 #define
65 #define
66 #define
67 #define
68 #define
69 #define
70 #define
71 #define
72 #define
73 #define
74 #define
75 #define
76 #define
77 #define
78 #define
79 #define
80 #define
81 #define
82 #define
83 #define
84 #define

```

---

N	<u>NR_exit</u>	1
R	<u>NR_fork</u>	2
-	<u>NR_read</u>	3
s	<u>NR_write</u>	4
e	<u>NR_open</u>	5
t	<u>NR_close</u>	6
u	<u>NR_waitpid</u>	7
p	<u>NR_creat</u>	8
0	<u>NR_link</u>	9
/	<u>NR_unlink</u>	10
*	<u>NR_execve</u>	11
u	<u>NR_chdir</u>	12
s	<u>NR_time</u>	13
e	<u>NR_mknod</u>	14
d	<u>NR_chmod</u>	15
o	<u>NR_chown</u>	16
n	<u>NR_break</u>	17
l	<u>NR_stat</u>	18
y	<u>NR_lseek</u>	19
b	<u>NR_getpid</u>	20
y	<u>NR_mount</u>	21
l	<u>NR_umount</u>	22
g	<u>NR_setuid</u>	23
e	<u>NR_getuid</u>	24
t	<u>NR_stime</u>	25

i  
n  
i  
t  
,t  
o

g  
e  
t

s  
y  
s  
t  
e  
m

g  
o  
i  
n  
g

\*

---

```

88 #define NR_ptrace 26
89 #define NR_alarm 27
90 #define NR_fstat 28
91 #define NR_pause 29
92 #define NR_utime 30
93 #define NR_stty 31
94 #define NR_gtty 32
95 #define NR_access 33
96 #define NR_nice 34
97 #define NR_ftime 35
98 #define NR_sync 36
99 #define NR_kill 37
100 #define NR_rename 38
101 #define NR_mkdir 39
102 #define NR_rmdir 40
103 #define NR_dup 41
104 #define NR_pipe 42
105 #define NR_times 43
106 #define NR_prof 44
107 #define NR_brk 45
108 #define NR_setgid 46
109 #define NR_getgid 47
110 #define NR_signal 48
111 #define NR_geteuid 49
112 #define NR_getegid 50
113 #define NR_acct 51
114 #define NR_phys 52
115 #define NR_lock 53
116 #define NR_ioctl 54
117 #define NR_fcntl 55
118 #define NR_mpx 56
119 #define NR_setpgid 57
120 #define NR_ulimit 58
121 #define NR_uname 59
122 #define NR_umask 60
123 #define NR_chroot 61
124 #define NR_ustat 62
125 #define NR_dup2 63
126 #define NR_getppid 64
127 #define NR_getpgrp 65
128 #define NR_setsid 66
129 #define NR_sigaction 67
130 #define NR_sgetmask 68
131 #define NR_ssetmask 69
132 #define NR_setreuid 70
133 #define NR_setregid 71
134 #define NR_sigsuspend 72
135 #define NR_sigpending 73
136 #define NR_sethostname 74
137 #define NR_setrlimit 75
138 #define NR_getrlimit 76
139 #define NR_getusage 77
140 #define NR_gettimeofday 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
149

// 以下は、システムコールの組み込みアセンブリマクロ関数の定義です。
// 引数のないシステムコールマクロ関数：type name(void).
// %0 - eax( res), %1 - eax( NR_##name). name はシステムコールの名前で、結合された
// 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。
// the error number errno is set and -1 is returned.
#define syscall0(type, name) \

```

```

150
151 type name(void) \
152 { \
153     long_res; \
154     __asm volatile ("int $0x80" \
155                     : "=a" (__res) \
156                     : "0" (__NR_##name)); \
157     if (__res >= 0) \
158         return (type) __res; \
159     __errno = -__res; \
160     return -1; \
161 }
162
// A system-call macro function with 1 parameter: type name(atype a)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a).
163 #define __syscall11(type, name, atype, a) \
164 type name(atype a) \
165 { \
166     long_res; \
167     __asm volatile ("int $0x80" \
168                     : "=a" (__res) \
169                     : "0" (__NR_##name), "b" ((long) (a))); \
170     if (__res >= 0) \
171         return (type) __res; \
172     __errno = -__res; \
173     return -1; \
174 }
175
// A system call macro function with 2 parameters: type name(atype a, btype b)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b).
176 #define __syscall12(type, name, atype, a, btype, b) \
177 type name(atype a, btype b) \
178 { \

```

```

179 long res; \
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; \
185 errno = - res; \
186 return -1; ↴ ↴ ↴ ↴ ↴
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は
// 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; \
193 __asm volatile ("int $0x80" \
194 : "=a" ( res) \
195      : "0" (_NR_##name), "b" ((long)(a)), "c" ((long)(b)), "d" ((long)(c))); \
196 if (_res>=0) \
197     return (type) res; \
198 errno=- res; \
199 return -1; .
200 }
201
202 #endif /* LIBRARY */
203
204 extern int errno; // Error number, a global variable.
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);
207 int alarm(int sec);
208 int brk(void * end_data_segment);
209 void * sbrk(ptrdiff_t increment);
210 int chdir(const char * filename);
211 int chmod(const char * filename, mode_t mode);
212 int chown(const char * filename, uid_t owner, gid_t group);
213 int chroot(const char * filename);
214 int close(int fildes);
215 int creat(const char * filename, mode_t mode);
216 int dup(int fildes);
217 int execve(const char * filename, char ** argv, char ** envp); 219 int
execv(const char * pathname, char ** argv);
218 int execvp(const char * file, char ** argv);
219

```

```
222 int exec1(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);

228 225 volatile void _exit(int status); 226 int
fcntl(int fildes, int cmd, ...); 227 int
fork(void);
229 int getpid(void);
230 int getuid(void);
231 int geteuid(void);
232 int getgid(void);
233 int getegid(void);
234 int ioctl(int fildes, int cmd, ...);
234 int kill(pid_t pid, int signal);

237 235 int link(const char * filename1, const char * filename2); 236 int
lseek(int fildes, off_t offset, int origin);
238 int mknod(const char * filename, mode_t mode, dev_t dev);
239 int mount(const char * specialfile, const char * dir, int rwflag);
239 int nice(int val);

242 240 int open(const char * filename, int flag, ...); 241 int
pause(void);
243 int pipe(int * fildes);
244 int read(int fildes, char * buf, off_t count);
244 int setpgrp(void);

247 245 int setpgid(pid_t pid, pid_t pgid); 246
int setuid(uid_t uid);
248 int setgid(gid_t gid);
249 void (*signal(int sig, void (*fn)(int)))(int);
250 int stat(const char * filename, struct stat * stat_buf);
250 int fstat(int fildes, struct stat * stat_buf);

253 251 int stime(time_t * ptr); 252
int sync(void);

254 time_t time(time_t * tloc);
255 time_t times(struct tms * tbuf);
255 int ulimit(int cmd, long limit);
256 mode_t umask(mode_t mask);

260 257 int umount(const char * specialfile); 258
int uname(struct utsname * name); 259 int
unlink(const char * filename);
261 int ustat(dev_t dev, struct ustat * ubuf);
262 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 getpgrp(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 getrusage(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 * writefds,
278           fd_set * exceptfds, struct timeval * timeout);
279
280 #endif
281

```

---

## 14.14 utime.h

### 14.14.1 Functionality

The utime.h file defines the file access and modification time structure `utimbuf{ }` , and the `utime()` function prototype, where time is in seconds.

### 14.14.2 Code annotation

[1](#) プログラム 14-13 linux/include/utime.h

```

2 ifndef UTIME_H
3 define UTIME_H
3
// <sys/types.h> 型のヘッダファイルです。基本的なシステムデータタイプとファイルシステムパラメー
タの
//      structure are defined.
6 #include <sys/types.h> /* 分かってますよ、こんなことしちゃいけない
    って。 */ 5
7 struct utimbuf {
8     time_t actime;          // File access time. seconds from 1970.1.1:0:0:0.
9     time_t modtime;         // File modified time. seconds from 1970.1.1:0:0:0.
9 };
10
// ファイルのアクセス時間や修正時間を設定する関数です。
11 extern int utime(const char *filename, struct utimbuf *times); 12
13 #endif
14

```

---

## 14.15 Files in the include/asm/ directory

linux/include/asm/ディレクトリ内のファイル一覧 14-2

Filename	Size	Last Modified Time (GMT)	Description
io.h	477 bytes	1991-08-07 10:17:51	
memory.h	507 bytes	1991-06-15 20:54:44	
segment.h	1366 bytes	1991-11-25 18:48:24	
 system.h	1707 bytes	1992-01-13 13:02:10	

## 14.16 io.h

### 14.16.1 Functionality

The embedded assembly macro functions that access the hardware IO port are defined in the io.h file: outb(), inb(), and outb\_p() and inb\_p(). The main difference between the first two functions and the latter two is that the jmp instruction is used in the latter code for time delay.

### 14.16.2 Code annotation

プログラム 14-14 linux/include/asm/io.h

```

// ハードウェアポートのバイト出力機能。
// 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。
5 // 入力バイトを返します。5
#define inb(port) ({ __asm__ __volatile__ ("inb %%dx,%%al": "=a" (_v) : "d" (port)); \
6 __asm__ __volatile__ ("movb %al,%w:_v" : : : "al"); \
7 _v; \
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" ) ￥
14           "|tjmp 1f|n"\           // Jump forward to label 1 (the next statement).
15           "1:|tjmp 1f|n"\        // Jump forward to label 1.
15           "1::: "a" (value), "d" (port))
16

```

//// 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。

```

19 // 入力バイトを返します。17
#define inb_p(port) ({ 18
    unsigned char _v; /*^w^*/
20     __asm volatile ("inb %%dx, %%al\n" \
21         "|tjmp lf\n" \
22         "1:|tjmp lf\n" \
22         "1:": "=a" (_v): "d" (port)); \
23     _v; \
24 })
25

```

## 14.17 memory.h

### 14.17.1 Functionality

The memory.h file contains a memory copy embedded assembly macro function memcpy(). It is identical to memcpy() defined in string.h, except that the latter is defined in the form of an embedded assembly C function.

### 14.17.2 Code annotation

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

```

1 /*
2  * NOTE!!! memcpy(dest, src, n) assumes ds=es=normal data segment. This
3  * goes for all kernel functions (ds=es=kernel space, fs=local data,
4  * gs=null), as well as for all well-behaving user programs (ds=es=
5  * user data space). This is NOT a bug, as any user program that changes
6  * es deserves to die if it isn't careful. -
7 */
8
9 ///////////////////////////////////////////////////////////////////
10 // %0 - edi (address dest), %1 - esi (address src), %2 - ecx (number of bytes n). 8 #define
11 // memcpy(dest,src,n) ({ /*^w^*/
12 //     void *_res = dest; \
13 //     __asm_ ("cld;rep;movsb" \
14 //             :: "D" ((long) (_res)), "S" ((long) (src)), "c" ((long) (n)) \
15 //             :"di", "si", "cx"); \
16 //     _res; eldest
17 })
18

```

## 14.18 segment.h

### 14.18.1 Functionality

The segment.h file defines some functions that access the Intel CPU segment registers, as well as some memory manipulation functions associated with the segment registers. In a Linux system, when the user program starts executing kernel code through a system-call, the kernel code first loads the kernel data segment descriptor (segment value 0x10) in the GDT into the registers DS and ES, that is, DS and ES are used for access the kernel data segment; and the task data segment descriptor (segment value 0x17) in the LDT is loaded into the FS, that is, the FS is used to access the user data segment. See lines 92--96 in kernel/sys\_call.s. Therefore, when executing kernel code, you need to use a special method to access the data in the user program (task). Functions such as get\_fs\_byte() and put\_fs\_byte() in this file are specifically used to access data in the user program.

### 14.18.2 Code annotation

プログラム 14-16 linux/include/asm/segment.h

```
//// FS セグメントの指定されたアドレスのバイトを取得します。
// パラメータ：addr - 指定されたメモリアドレス。
// %0 - (返されたバイト _v); %1 - (メモリアドレス addr).
// 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;      // a register variable for efficient access.
4
5     _asm_( "movb %%fs:%1,%0": "=r" (_v) : "m" (*addr));
6     return _v;
7 }
8
//// FS セグメントの指定されたアドレスのワードを取得します。
// %0 - (返されたワード _v); %1 - (メモリアドレス addr).
// メモリFS:[addr]のワードを返します。
9     unsigned short _v;
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13     __asm__( "movw %%fs:%1,%0": "=r" (_v) : "m" (*addr));
14     return _v;
15 }
16
17 // Get a long word at the specified address in the FS segment.
18 // %0 - (返されたロングワード _v); %1 - (メモリアドレス addr).
19 // メモリFS:[addr]のロングワードを返します。
20
21 extern inline unsigned long get_fs_long(const unsigned long *addr) 18 {.
```

```
19     unsigned long _v;  
20  
21     _asm_( "movl %%fs:%l,%0": "=r" (_v) : "m" (*addr)) ; \
```

```

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     extern inline void put_fs_byte(char val,char *addr) 26 {};
29
30     /// 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。
31     // パラメータ : val - ワード値、addr - メモリアドレス。
32     // %0 - レジスタ (ワード値 val); %1 - (メモリアドレス addr).
33     extern inline void put_fs_word(short val,short * addr) 31 {};
34
35     /// 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。
36     // パラメータ : val - ロングワードの値、addr - メモリアドレス。
37     // %0 - レジスタ (ロングワード値 val); %1 - (メモリアドレス addr).
38     extern inline void put_fs_long(unsigned long val,unsigned long * addr) 36 {};
39
40 /* 41
42 * Someone who knows GNU asm better than I should double check the followig.
43 * It seems to work, but I don't know if I'm doing something subtly wrong.
44 * --- TYT, 11/24/91
45 * [ Linusさん、ここは何も問題ありません ] 46 */
46 {

```

```
48
49     unsigned short _v;
50     __asm__( "mov %%fs, %%ax": "=a" (_v):);
51     return _v;
52 }
53
54 // Get DS segment register value.
55 extern inline unsigned long get_ds() {
56     unsigned short _v;
57     __asm__( "mov %%ds, %%ax": "=a" (_v):);
58     return _v;
59 }
60
61 // FSセグメントレジスタの設定
62 extern inline void set_fs(unsigned long val) {
```

```

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

```

## 14.19 system.h

### 14.19.1 Functionality

The system.h file contains embedded assembly macros that set or modify segment descriptors/interrupt gate descriptors. Among them, the function move\_to\_user\_mode() is used for the kernel to manually switch (move) to the initial process (task 0) when the kernel initialization ends, that is, to run from the code of the privilege level 0 code to the privilege level 3. The method used is to simulate the interrupt call return process, that is, use the IRET instruction to implement the privilege level change and the stack switch, thereby moving the CPU execution control flow to the environment of the initial task 0, as shown in Figure 14-3.

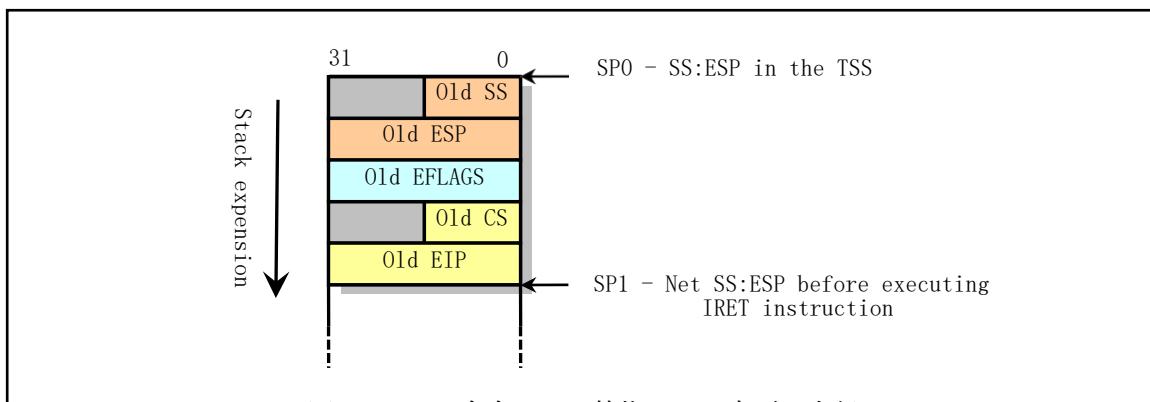


図14-3 IRET命令による特権レベル変更の実行

The use of this method for privilege level changes and control transfer is caused by the CPU protection mechanism. By call gate, interrupt or trap gate, the CPU allows low-level (such as privilege level 3) code to call or transfer control to high-level code, but not vice versa. So the kernel uses the method of simulating IRET instruction returns to low-level code from a high level.

タスク0のコードを実行する前に、カーネルはまずスタックを設定して、特権レベルスイッチが割り込み呼び出し手順に入ったばかりのときのスタック内のコンテンツ配置をシミュレートします。その後、IRET命令が実行され、システムがタスク0の実行に移行します。IRET命令が実行されると、スタックの内容は図14-3のようになります。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。

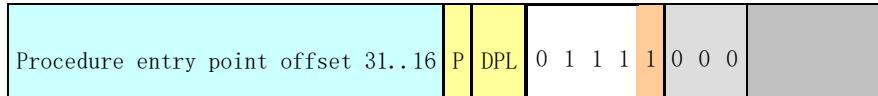
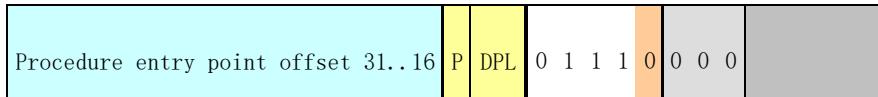


図14-4 テーブルIDTにおけるインタラプトゲートおよびトラップゲート記述子のフォーマット

Where P is the segment presence flag; DPL is the privilege level of the descriptor. The difference between the interrupt gate and the trap gate is the effect of the EFLAGS interrupt enable flag IF. The interrupt executed by the interrupt gate descriptor resets the IF flag, so this way prevents other interrupts from interfering with the current interrupt processing, and the subsequent interrupt end instruction IRET will restore the original value of the IF flag from the stack; The interrupt executed by the trap gate will not affect the IF flag.

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のディスクリプターフォーマットの6バイト目の下位4ビットに相当するので、`type=14(0x0E)`は割り込みゲートディスクリプタ、`type=15(0x0F)`はトラップゲートディスクリプタを示す。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 Code annotation

プログラム 14-17 linux/include/asm/system.h

//// ユーザーモードに移行して実行します。

```
// 1 #define move_to_user_mode() .^~^.)
2 _asm_ ("movl %%esp, %%eax\n\t" \
3       "pushl $0x17\n\t" \
4       "pushl %%eax\n\t" \
5       "pushfl\n\t" \
6       "pushl $0x0f\n\t" \
7       "pushl $1f\n\t" \
8       "iret\n\t" \
9       "1:\tmovl $0x17, %%eax\n\t" \
10      "movw %%ax, %%ds\n\t" \
11      "movw %%ax, %%es\n\t" \
12      "movw %%ax, %%fs\n\t" \
13      "movw %%ax, %%gs" \
14      ::: "ax")
15
16 #define sti() _asm_ ("sti"::) // enable interrupt.
17 #define cli() _asm_ ("cli"::) // disable interrupt.
18 #define nop() _asm_ ("nop"::) // no op.
19
20 #define iret() _asm_ ("iret"::) // interrupt return.
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 - descriptor privilege level; addr - offset address.
// %0 - (type flag word combined by 'dpl', 'type'); %1 - (descriptor low 4 byte address);
// %2 - (descriptor high 4 byte address); %3 - EDX (program offset address addr);
22 // %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)。
23 #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)
    // にします。
24 // 最後に、ゲートディスクリプターの下位4バイトと上位4バイトを別々に設定します。
25 __asm ("movw %%dx, %%ax\n\t" \
26       "movw %0, %%dx\n\t" \
27       "movl %%eax, %1\n\t" \
28       "movl %%edx,%2" \
: \
: "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
        // %0
```

```
29      "o" (*((char *) (gate_addr))), \  
30      "o" (*(4+(char *) (gate_addr))), \  
31      "d" ((char *) (addr)), "a" (0x00080000))           // %3, %4  
32
```

```

//// 割り込みゲートの設定（後続の割り込みを自動的にマスクする）。
// パラメータ : n - 割り込み番号、addr - 割り込みプログラムのオフセットアドレス。
// 「&idt[n]」は、割り込み番号nの対応するエントリのオフセット値です。
// 割り込みディスクリプターテーブルIDTの中で、割り込みディスクリプターのタイプは14、そして
33 // 特権レベルは0です。
34 #define set_intr_gate(n,addr) \
35     set_gate(&idt[n], 14, 0, addr)
35

//// トラップゲートの設定
// パラメータ : n - 割り込み番号、addr - 割り込みプログラムのオフセットアドレス。
// 「&idt[n]」は、IDT内の割り込み番号「n」に対応するエントリのオフセットです。
36 // 割り込みディスクリプターのタイプは15で、特権レベルは0です。
37 #define set_trap_gate(n,addr) \
38     set_gate(&idt[n], 15, 0, addr)
38

//// システムのトラップゲートを設定します。
// 上記のset_trap_gate()で設定されたディスクリプターは、特権レベルが0で、ここでは3です。
// したがって、set_system_gate()で設定した割り込み処理は、すべてのプログラムで実行することができます。
// シングルステップデバッグ、オーバーフローエラー、バウンダリアウトのエラー処理など。
// パラメータ : n - 割り込み番号、addr - 割り込みプログラムのオフセットアドレス。
// 「&idt[n]」は、IDT内の割り込み番号nに対応するエントリのオフセットです。
39 // 割り込みディスクリプターのタイプは15で、特権レベルは3です。
40 #define set_system_gate(n,addr) \
41     set_gate(&idt[n], 15, 3, addr)
41

//// セグメントディスクリプターを設定します（カーネルでは使用しません）。
// パラメータ : gate_addr - ディスクリプターのアドレス、type - ディスクリプターのタイプフィールドの値。
// dpl - ディスクリプタの特権レベル、base - セグメントのベースアドレス、limit - セグメントの制限。
// セグメントディスクリプターのフォーマットを参照してください。ここで割り当てオブジェクトが正しくないことに注意してください
// (逆) になっています。43行目は「*((gate_addr)+1)」とし、49行目は「*(gate_addr)」とします。
// ただし。
// このマクロはカーネルコードでは使用されていないので、Linusは知らない。)
42 #define _set_seg_desc(gate_addr,type,dpl,base,limit) {~~~!
43     *((gate_addr) = ((base) & 0xff000000) | \
44         (((base) & 0x00ff0000)>>16) | \
45         ((limit) & 0xf0000) | \
46         ((dpl)<<13) | \
47         (0x00408000) | \
48         ((type)<<8); \
49     *((gate_addr)+1) = (((base) & 0x0000ffff)<<16) | \
50         ((limit) & 0xffff); \
51

//// グローバルテーブルGDTにタスクステータスセグメント／ローカルテーブル記述子を設定。

```

```

// ステータスセグメントとローカルテーブルセグメントの長さは、ともに104バイトに設定されています。
// パラメータ：n - グローバルテーブルGDTのディスクリプターアイテムnに対応するアドレス。
// addr - ステートセグメント/ローカルテーブルが配置されているメモリのベースアドレス。
// type - ディスクリプタ内のフラグタイプバイトです。
// %0 - eax (アドレスaddr); %1 - (記述子項目nのアドレス); %2 - (のオフセット2)
// ディスクリプタ・アイテムn); %3 - (ディスクリプタ・アイテムnのオフセット4); %4 - (ディスクリ
// プタ・アイテムnのオフセット5)
// ディスクリプタ・アイテムn); %5 - (ディスクリプタ・アイテムnのオフセット6); %6 - (ディスクリ
// プタ・アイテムnのオフセット7)
52 // ディスクリプタ・アイテムn)。
53 #define set_tssldt_desc(n,addr,type) \
54 __asm__ ("movw$104,%1|n|t"\           // The TSS length is stored in length field (0-th byte).
55           "movw%%ax,%2|n|t"\           // Put tne low word of base into the 2-3rd byte.

```

```

56      "rorl $16, %%eax|n|t"\    // Rotate base high word into AX (low word to high).
57      "movb %%al, %3|n|t"\    // Move low byte of the base high word to the 4th byte.
58      "movb $" type ",%4|n|t"\ // Move flag type byte into the 5th byte.
59      "movb $0x00, %5|n|t"\    // The sixth byte of the descriptor is set to zero.
60      "movb %%ah, %6|n|t"\    // Move high byte of the base high word to the 7th byte.
61      "rorl $16, %%eax"\      // Loop 16 bits to the right, EAX restores the original.
61      :: "a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
62          "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
63      )
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
65 // のエントリーです。TSS記述子のタイプは0x89である。
66 #define set_tss_desc(n,addr) _set_tsslldt_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
    67 // 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。
68 #define set_ldt_desc(n,addr) _set_tsslldt_desc(((char *) (n)),addr,"0x82")
67

```

## 14.20 Files in the directory include/linux/

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

Filename	Size	Last Modified Time(GMT)	Description
config.h	1545 bytes	1992-01-11 00:13:18	
fdreg.h	2466 bytes	1991-11-02 10:48:44	
fs.h	5754 bytes	1992-01-12 07:00:20	
hdreg.h	1968 bytes	1991-10-13 15:32:15	
head.h	304 bytes	1991-06-19 19:24:13	
kernel.h	1036 bytes	1992-01-12 02:17:34	
math_emu.h	4924 bytes	1992-01-01 17:33:04	
mm.h	1101 bytes	1992-01-13 15:46:41	
sched.h	7351 bytes	1992-01-13 22:24:42	
sys.h	3402 bytes	1992-01-13 21:42:37	
tty.h	2801 bytes	1992-01-08 22:51:56	

## 14.21 config.h

### 14.21.1 Functionality

Config.hはカーネルコンフィグレーションのヘッダーファイルで、unameコマンドが使用するマシンコンフィグレーション情報や、使用するキーボード言語タイプやハードディスクタイプ(HD\_TYPE)のオプションを定義しています。

### 14.21.2 Code annotation

[1](#) プログラム 14-18 linux/include/linux/config.h

```

2 #ifndef CONFIG_H
3 #define CONFIG_H
3
4 /*
5 * Defines for what uname() should return
6 */
7 #define UTS_SYSNAME "Linux"
8 #define UTS_NODENAME "(none)" /* set by sethostname() */
9 #define UTS_RELEASE "" /* patchlevel */
10 #define UTS_VERSION "0.12"
11 #define UTS_MACHINE "i386" /* hardware type */
12
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。
#define DEF_INITSEG      0x9000      // The segment to which the boot sector be moved.



---


14
15 #define DEF_SYSSEG      0x1000      // The segment to which the system module loaded.
16 #define DEF_SETUPSEG     0x9020      // The segment where the setup program is located.
17 #define DEF_SYSSIZE       0x3000      // The maximum system module size (in units of 16).
18 /*
19 */
20 * The root-device is no longer hard-coded. You can change the default
21 * root-device by changing the line ROOT_DEV = XXX in boot/bootsect.s
22 */
23 /*
24 */
25 * The keyboard is now defined in kernel/chr_dev/keyboard.S
26 */
27 /*
28 */
29 * Normally, Linux can get the drive parameters from the BIOS at

```

```
30 * startup, but if this for some unfathomable reason fails, you'd
31 * be left stranded. For this case, you can define HD_TYPE, which
32 * contains all necessary info on your harddisk.
33 *
34 * The HD_TYPE macro should look like this:
35 *
36 * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
37 *
38 * In case of two harddisks, the info should be separated by
39 * commas:
40 *
41 * #define HD_TYPE { h, s, c, wpcom, lz, ctl }, { h, s, c, wpcom, lz, ctl }
42 */
43 */
44 This is an example, two drives, first is type 2, second is type 3:
45
46 #define HD_TYPE { 4,17,615,300,615,8 }, { 6,17,615,300,615,0 }
47
48 NOTE: ctl is 0 for all drives with heads<=8, and ctl=8 for drives
49 with more than 8 heads.
50
51 If you want the BIOS to tell what kind of drive you have, just
52 leave HD_TYPE undefined. This is the normal thing to do.
53 */
54
55 #endif
56
```

---

## 14.22 fdreg.h

### 14.22.1 Functionality

The fdreg.h header file is used to describe some of the parameters commonly used in floppy devices and the I/O ports used. Because the control of the floppy disk drive is cumbersome and there are many commands, it is best to refer to the book about the principle of the microcomputer interface before reading the code to understand how the floppy disk controller (FDC) works. Then you will think that the definition here is still reasonable and orderly.

フロッピーディスク装置をプログラミングする場合、各レジスタまたは複数のレジスタに1つずつ、計4つのポートにアクセスする必要があります。1.2Mのフロッピーディスクコントローラの場合、表14-1のようなポートがあります。

表14-1 フロッピーディスクコントローラのポート

I/O port	Read/Write	Register name
0x3f	Write only	Digital output (control) Register
2	Read only	FDC main status register
0x3f	Read/Write	FDCデータレジスタ
4		
0x3f		
5		
0x3f7	Read only	Digital input register
0x3f7	Write only	Floppy disk control register (rate control)

The digital output port (digital control port) is an 8-bit register that controls the drive motor on, drive selection, start/reset FDC, and enable/disable DMA and interrupt requests.

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、読み出し可能なりザルトレジスタ）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の取得動作を検出するために、FDCにステータスを送る必要があります。

### 14.22.2 Code annotation

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

```

1  /*
2   * This file contains some defines for the floppy disk controller.
3   * Various sources. Mostly "IBM Microcomputers: A Programmers
4   * Handbook", Sanches and Canton.
5   */
6 #ifndef FDREG_H // This definition is used to exclude duplicate header files in code.
7 #define FDREG_H
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す。
16 /* Fd controller regs. S&C, about page 340 */
17 #define FD_STATUS 0x3f4 // Main status register port.
18 #define FD_DATA 0x3f5 // Data port.
19 #define FD_DOR 0x3f2 /* Digital Output Register */
20 #define FD_DIR 0x3f7 /* Digital Input Register (read) */
21 #define FD_DCR 0x3f7 /* Diskette Control Register (write)*/
22
/* Bits of main status register */
23 #define STATUS_BUSYMASK 0x0F /* drive busy mask */ // (one bit per driver).
24 #define STATUS_BUSY 0x10 /* FDC busy */
25 #define STATUS_DMA 0x20 /* 0- DMA mode */
26 #define STATUS_DIR 0x40 /* 0- cpu->fdc */
27 #define STATUS_READY 0x80 /* Data reg ready */
28
29 /* Bits of FD_ST0 */
30 #define ST0_DS 0x03 /* drive select mask */
31 #define ST0_HA 0x04 /* Head (Address) */
32 #define ST0_NR 0x08 /* Not Ready */
33 #define ST0_ECE 0x10 /* Equipment chech error */
34 #define ST0_SE 0x20 /* Seek end */ // or recalitrate end.
// 割り込みコードビット(割り込み理由)、00 - コマンドが正常に終了、01 - コマンドが終了
// 異常; 10 - コマンドが無効; 11 - FDDの準備状態が変化。

```

```

41 #define ST1_OR          0x10      /* OverRun */ // Data transfer timeout.
42 #define ST1_CRC          0x20      /* CRC error in data or addr */
43 #define ST1_EOC          0x80      /* End Of Cylinder */

44
45 /* Bits of FD_ST2 */
46 #define ST2_MAM          0x01      /* Missing Address Mark (again) */
47 #define ST2_BC           0x02      /* Bad Cylinder */
48 #define ST2_SNS          0x04      /* Scan Not Satisfied */
49 #define ST2_SEH          0x08      /* Scan Equal Hit */
50 #define ST2_WC           0x10      /* Wrong Cylinder */
51 #define ST2_CRC          0x20      /* CRC error in data field */
52 #define ST2_CM           0x40      /* Control Mark = deleted */

53
54 /* Bits of FD_ST3 */
55 #define ST3_HA           0x04      /* Head (Address) */
56 #define ST3_TZ           0x10      /* Track Zero signal (1=track 0) */
57 #define ST3_WP           0x40      /* Write Protect */

58
59 /* Values for FD_COMMAND */
60 #define FD_RECALIBRATE   0x07      /* move to track 0 */ // recalibrate.
61 #define FD_SEEK          0x0F      /* seek track */
62 #define FD_READ          0xE6      /* read with MT, MFM, SKip deleted*/
63 #define FD_WRITE         0xC5      /* write with MT, MFM */
64 #define FD_SENSEI        0x08      /* Sense Interrupt Status */
65 // ドライブのパラメータ (ステップレート、ヘッドアンロード時間など) を設定します。
66 #define FD_SPECIFY       0x03      /* specify HUT etc */

67 /* DMA commands */
68 #define DMA_READ          0x46      // The mode word of DMA read disk (to port 12, 11).
69 #define DMA_WRITE         0x4A      // The mode word of DMA write disk.

70
71 #endif
72

```

## 14.23 fs.h

### 14.23.1 Functionality

The fs.h header file mainly defines some constants and structures about the file system, including the data structure of the buffer block in the buffer cache, the super block and i-node structure in the MINIX 1.0 file system, and the file table structure and some pipeline operation macros. .

### 14.23.2 Code annotation

プログラム 14-20 linux/include/linux/fs.h

```

1 /*
2 * This file has definitions for some important file table
3 * structures etc.
4 */

```

```

5
6 #ifndef FS_H
7 #define FS_H
8
9 #include <sys/types.h>      // type header file. The basic system data types are defined.
10
11 /* devices are as follows: (same as minix, so we can use the minix
12 * file system. These are major numbers.)
13 *
14 * 0 - unused (nodev)
15 * 1 - /dev/mem           // memory device.
16 * 2 - /dev/fd
17 * 3 - /dev/hd
18 * 4 - /dev/ttyx          // tty serial terminal device.
19 * 5 - /dev/tty
20 * 6 - /dev/lp            // printer device.
21 * 7 - unnamed pipes
22 */
23
24 #define IS_SEEKABLE(x) ((x)>=1 && (x)<=3)    // Determine if a device can find a location.
25
26 #define READ 0
27 #define WRITE 1
28 #define READA 2        /* read-ahead - don't pause */
29 #define WRITEA 3       /* "write-ahead" - silly, but somewhat useful */
30
31 void buffer_init(long buffer_end);      // buffer cache init function.
32
33 #define MAJOR(a) (((unsigned)(a))>>8) // get device major number.
34 #define MINOR(a) ((a)&0xff)           // get device minor number.
35
36 #define NAME_LEN 14                // name length is 14.
37 #define ROOT_INO 1                 // root i-node number.
38
39 #define I_MAP_SLOTS 8             // the number of i-node bitmap slots (blocks).
40 #define Z_MAP_SLOTS 8             // the number of logical block bitmap slots.
41 #define SUPER_MAGIC 0x137F        // File system magic number.
42
43 #define NR_OPEN 20               // The maximum number of files opened by process.
44 #define NR_INODE 32              // The maximum number of I-nodes used by system.
45 #define NR_FILE 64               // The maximum number of files in the system.
46 #define NR_SUPER 8               // The maximum number of superblocks in system.
47 #define NR_HASH 307              // Buffer hash-table array items.
48 #define NR_BUFFERS nr_buffers   // The number of buffer blocks in the system.
49 #define BLOCK_SIZE 1024          // Data block size (in bytes).
50 #define BLOCK_SIZE_BITS 10        // The number of bits used by the block size.
51 #ifndef NULL
52 #define NULL ((void *) 0)
53 #endif
54
// 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)))
// 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
61 // 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])
62 #define PIPE_TAIL(inode) ((inode).i_zone[1])
63 #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
66 #define NIL_FILP      ((struct file *)0)    // null file structure pointer.
67 #define SEL_IN        1
68 #define SEL_OUT       2
69 #define SEL_EX        4
70
71 typedef char buffer_block[BLOCK_SIZE];           // buffer block array (1024 items).
72
// バッファブロックヘッダのデータ構造。(とても重要!!)
73 // bhはコード上、buffer_headの略語を表すことが多いです。
74 struct buffer_head {
75     char * b_data;          /* pointer to data block (1024 bytes) */
76     unsigned long b_blocknr; /* block number */
77     unsigned short b_dev;   /* device (0 = free) */
77     unsigned char b_uptodate;
78     unsigned char b_dirt;   /* 0-clean, 1-dirty */ // Modified flag.
79     unsigned char b_count;  /* users using this block */
80     unsigned char b_lock;   /* 0 - ok, 1 -locked */
81     struct task_struct * b_wait; // task wait queue.
82     struct buffer_head * b_prev; // the previous block on the hash queue.
83     struct buffer_head * b_next; // the next block on the hash queue.
84     struct buffer_head * b_prev_free; // the previous block on the free list.
85     struct buffer_head * b_next_free; // the next block on the free list.
86 };
87
// ディスク上のI-nodeデータ構造(32バイト)。
88     unsigned short i_mode; // File type and attribute (rwx bit).
stru
ct
d_in
ode
t
89
90     unsigned short i_uid; // User id (file owner identifier).
91     unsigned long i_size; // File size (in bytes).
92     unsigned long i_time; // Modified time (from 1970.1.1:0, in seconds).
93     unsigned char i_gid; // Group id (the group in which file owner belong).
94     unsigned char i_nlinks; // Number of links (entries pointed to the i-node).
95     unsigned short i_zone[9]; // logical block number array.
96 }; // direct(0-6), indirect(7) or secondary indirect(8).
97
98 // 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。
99 struct m_inode {

```

```
100     unsigned short i_mode;
101    unsigned short i_uid;
102    unsigned long i_size;
103    unsigned long i_mtime;
104    unsigned char i_gid;
```

```

105     unsigned char i_nlinks;
106     unsigned short i_zone[9];
106 /* these are in memory also */
107     struct task_struct * i_wait; // task waiting queue for waiting for the i-node.
108     struct task_struct * i_wait2; /* for pipes */
109     unsigned long i_atime; // i-node access time.
110     unsigned long i_ctime; // i-node change time.
111     unsigned short i_dev; // the device where the i-node is located.
112     unsigned short i_num; // i-node number.
113     unsigned short i_count; // i-node used count, 0 indicates it's idle (free).
114     unsigned char i_lock; // lock flag.
115     unsigned char i_dirt; // modified flag
116     unsigned char i_pipe; // the i-node is used for pipe.
117     unsigned char i_mount; // mount flag.
118     unsigned char i_seek; // used for lseek method of the file.
119     unsigned char i_update; // updated flag.
120 };
121 /
121
/     unsigned short f_mode; // file mode (RW bits).

```

ファイル構造(ファイルハンドルとi-nodeとの関係を確立するために使用される)

```

2
2

s
t
r
u
c
t

f
i
l
e

{
.

123
124     unsigned short f_flags;      // file open and control flags.
125     unsigned short f_count;     // file reference count.
126     struct m_inode * f_inode; // file's i-node.
127     off_t f_pos;           // read and write position in the file.
128 };
129

130 // インメモリーディスクのスーパー ブロック構造。
131 struct super_block {
132     unsigned short s_ninodes;      // number of i-nodes in the file system.
133     unsigned short s_nzones;       // number of zones (logica blocks).
134     unsigned short s_imap_blocks; // number of data blocks occupied by i-node map.
135     unsigned short s_zmap_blocks; // number of blocks occupied by logical block map.
136     unsigned short s_firstdatazone; // the first block number in the data zone.
137     unsigned short s_log_zone_size; // Log2(number of data blocks / logical block).
138     unsigned long s_max_size;     // the maximum file size.
139     unsigned short s_magic;       // file system magic number.
139 /* These are only in memory */
140     struct buffer_head * s_imap[8]; // an array of i-node bitmap buffer blocks.
141     struct buffer_head * s_zmap[8]; // an array of logical block bitmap buffer blocks.
142     unsigned short s_dev;          // the device number of the super block.
143     struct m_inode * s_isup;     // The root i-node of the mounted file system.
144     struct m_inode * s_imount;   // The i-node to which the file system is installed.
145     unsigned long s_time;          // modified time.
146     struct task_struct * s_wait;  // the wait queue for processes waiting for it.
147     unsigned char s_lock;          // the lock flag.
148     unsigned char s_rd_only;      // read only flag.
149     unsigned char s_dirt;          // dirty flag.
150 };
151

152 // 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。
153 struct d_super_block {
154     unsigned short s_ninodes;

```

```

155     unsigned short s_nzones;
156     unsigned short s_imap_blocks;
157     unsigned short s_zmap_blocks;
158     unsigned short s_firstdatazone;
159     unsigned short s_log_zone_size;
160     unsigned long s_max_size;
161     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 dir_entry {
165     unsigned short inode;           // i-node number.
166     char name[NAME_LEN];          // File name, NAME_LEN = 14.
166 } ;
167
168 extern struct m_inode inode_table[NR_INODE];    // i-node table (32 entries).
169 extern struct file file_table[NR_FILE];          // file table (64 items).
170 extern struct super_block super_block[NR_SUPER]; // super block array (8 items).
171 extern struct buffer_head * start_buffer;        // starting location of the buffer cache.
172 extern int nr_buffers;                          // the number of buffers.
173
     //// ディスク操作機能の試作品を紹介します。
174 // 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。
175 extern void check_disk_change(int dev);
176 // 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。
177 extern int floppy_change(unsigned int nr);
178 // 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)。
179 extern int ticks_to_floppy_on(unsigned int dev);
180 // 指定したドライブを起動する。
181 extern void floppy_on(unsigned int dev);
182 // 指定されたフロッピードライブの電源を切る。
183 extern void floppy_off(unsigned int dev);

     //// ファイルシステムの運用管理に関する機能のプロトタイプを以下に示します。
184 // 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。
185 extern void truncate(struct m_inode * inode);
186 // i-nodeの情報を更新 (同期) します。
187 extern void sync_inodes(void);
188 // 指定されたi-nodeを待ちます。
189 extern void wait_on(struct m_inode * inode);
190 // 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。
191 extern int bmap(struct m_inode * inode, int block);
    // 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。
192 // デバイスのブロック番号。
193 extern int create_block(struct m_inode * inode, int block);
// 指定されたパス名のi-node番号を取得します。184 extern
struct m_inode * namei(const char * pathname);
185 // 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。

```

```
186 extern struct m_inode * lnamei(const char * pathname);  
187 // パス名に応じてファイルを開く準備をします。  
188 extern int open_namei(const char * pathname, int flag, int mode,  
189             struct m_inode ** res_inode);  
// i-nodeをリリース(戻す)する(デバイスを書き込む)。
```

```

190 extern void input(struct m_inode * inode);
191 // デバイスからi-nodeを読み込みます。
192 extern struct m_inode * iget(int dev, int nr);
193 // inodeテーブルからidle i-nodeエントリを取得します。
194 extern struct m_inode * get_empty_inode(void);
195 // 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）。
196 extern struct m_inode * get_pipe_inode(void);
197 // 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。
198 extern struct buffer_head * get_hash_table(int dev, int block);
199 // 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）。
200 extern struct buffer_head * getblk(int dev, int block);
201 // 低レベルのリード/ライトブロック機能。
202 extern void ll_rw_block(int rw, struct buffer_head * bh);
203 // 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。
204 extern void ll_rw_page(int rw, int dev, int nr, char * buffer);
205 // 指定されたバッファブロックを解放します。
206 extern void brelse(struct buffer_head * buf);
207 // 指定されたデータブロックを読み込む。
208 extern struct buffer_head * bread(int dev, int block);
209 // 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。
210 extern void bread_page(unsigned long addr, int dev, int b[4]);
211 // 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。
212 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);
211 // デバイスデータエリアのロジックブロックを解放します。ロジックブロックのビットマップビットを
リセットします。
202 extern void free_block(int dev, int block);
203 // 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。
204 extern struct m_inode * new_inode(int dev);
205 // i-nodeを解放（フリー）する（ファイルを削除する場合）。
206 extern void free_inode(struct m_inode * inode);
207 // 指定されたデバイスバッファを更新します。
208 extern int sync_dev(int dev);
209 // 指定されたデバイスのスーパーブロックを取得します。
210 extern struct super_block * get_super(int dev);
206 extern int ROOT_DEV; // root device number.
207
// ルートファイルシステムをマウン
トします。208 extern void
mount_root(void); 209
210 #endif
211

```

## 14.24 hdreg.h

### 14.24.1 Functionality

The hdreg.h file mainly defines some command constant symbols for programming the hard disk controller. This includes the controller port, the status of each bit of the hard disk status register, controller commands, and

## 4.24.2

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。

## 14.24.3 Code annotation

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

```

1 /*
2 * This file contains some defines for the AT-hd-controller.
3 * Various sources. Check out some definitions (see comments with
4 * a ques).
5 */
6 #ifndef HDREG_H
7 #define HDREG_H
8
9 /* Hd controller regs. Ref: IBM AT Bios-listing */
10#define HD_DATA      0x1f0 /* _CTL when writing */
11#define HD_ERROR     0x1f1 /* see err-bits */
12#define HD_NSECTOR   0x1f2 /* nr of sectors to read/write */
13#define HD_SECTOR    0x1f3 /* starting sector */
14#define HD_LCYL     0x1f4 /* starting cylinder */
15#define HD_HCYL     0x1f5 /* high byte of starting cyl */
16#define HD_CURRENT   0x1f6 /* 101dhhh , d=drive, hhh=head */
17#define HD_STATUS    0x1f7 /* see status-bits */
18#define HD_PRECOMP   HD_ERROR /* same io address, read=error, write=precomp */
19#define HD_COMMAND   HD_STATUS /* same io address, read=status, write=cmd */
20
21#define HD_CMD       0x3f6 // Control register port.
22
23/* Bits of HD_STATUS */
24#define ERR_STAT     0x01 // Command execution error.
25#define INDEX_STAT   0x02 // Received the index.
26#define ECC_STAT     0x04 /* Corrected error */ // ECC checksum error.
27#define DRQ_STAT     0x08 // Request service.
28#define SEEK_STAT    0x10 // シークの終了。29
#define WRERR_STAT   0x20 // ドライブエラー。30
#define READY_STAT   0x40 // ドライブの準備完了。
31#define BUSY_STAT   0x80 // コントローラがビジー状態。
32
33/* Values for HD_COMMAND */
34#define WIN_RESTORE  0x10 // Drive reset (recalibration).
35#define WIN_READ     0x20 // Read sector.
36#define WIN_WRITE    0x30 // Write sector.
37#define WIN_VERIFY   0x40 // Sector verify.
38#define WIN_FORMAT   0x50 // Format track.
39#define WIN_INIT     0x60 // Controller initialize.
40#define WIN_SEEK     0x70 // Seek track.
41#define WIN_DIAGNOSE 0x90 // Controller diagnose.
42#define WIN_SPECIFY  0x91 // Establish drive parameters.
43
44/* HD_ERROR用のビット */
// 診断コマンドを実行した場合、その意味は他のコマンドとは異なり、以下のようになります。

```

```
//  
======  
======  
// Diagnostic command Other command  
// .....
```

```

// 0x01      No error          Data mark lost
// 0x02      Controller error   Track 0 error.
// 0x03      Sector buffer error
// 0x04      ECC part error     Command abort
// 0x05      Control process error
// 0x10
// 0x40
// 0x80      Bad sector

//-
45 #define MARK_ERR        0x01 /* Bad address mark ? */
46 #define TRKO_ERR        0x02 /* couldn't find track 0 */
47 #define ABRT_ERR        0x04 /* ? */
48 #define ID_ERR          0x10 /* ? */
49 #define ECC_ERR          0x40 /* ? */
50 #define BBD_ERR          0x80 /* ? */

51
52 // 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。
53 struct partition {
54     unsigned char boot_ind;      /* 0x80 - active (unused) */
55     unsigned char head;          /* ? */
56     unsigned char sector;        /* ? */
57     unsigned char cyl;           /* ? */
58     unsigned char sys_ind;       /* ? */
59     unsigned char end_head;      /* ? */
60     unsigned char end_sector;    /* ? */
61     unsigned char end_cyl;       /* ? */
62     unsigned int start_sect;     /* starting sector counting from 0 */
63     unsigned int nr_sects;       /* nr of sectors in partition */
64
65 #endif
66

```

## 14.24.4 Information

### 14.24.3.1 Hard disk partition table

In order to facilitate management of data or to achieve shared hard disk resources by multiple operating systems, the hard disk can be logically divided into 1--4 partitions. The sector numbers between each partition are contiguous. The partition table consists of four entries, each of which consists of 16 bytes and corresponds to the information of one partition. Each entry has a partition size, a cylinder number, a track number, and a sector number, as shown in Table 14-2. The partition table is stored at the 0x1BE--0x1FD position of the first sector of the 0 cylinder 0 head of the hard disk.

表	Name	Size	Description
1			
4			
-			
2			
八			

一 ド デ イ ス ク ・ パ ー テ イ シ ョ ン ・ テ ー ブ ル の エ ン ト リ 構 造 Offset			
0x00	boot_ind	1 byte	Boot index. Only one partition of the 4 partitions can be booted at a time. 0x00 - Do not boot from this partition; 0x80 - Boot from this partition.
0x01	head	1 byte	Partition start head number. The head number ranges from 0 to 255.
0x02	sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the beginning of the partition.

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

## 14.25 head.h

### 14.25.1 Functionality

The head.h header file defines the simple structure of the descriptor in the Intel CPU and specifies the item number of the descriptor.

### 14.25.2 Code annotation

[1 プログラム 14-22 linux/include/linux/head.h](#)

```

2 #ifndef HEAD\_H
3 #define HEAD\_H
3
// 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
// ことだけが書かれています。
// 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エントリであることを示しています。4
typedef struct desc_struct {
5     unsigned long a, b;
6 } desc\_table[256];
7
// ページング管理機構が使用するメモリページのディレクトリテーブルを宣言します。それぞれの
// ディレクトリエントリは4バイトです。このカーネルでは、テーブルは物理アドレス0から始まります。8
extern unsigned long pg_dir[1024];
9 extern desc\_table idt, gdt;           // Interrupt descriptor table, global descriptor table.
10
11 #define GDT\_NUL 0          // The 0th item of the GDT, not used.
12 #define GDT\_CODE 1         // The first item is the kernel code segment descriptor.
13 #define GDT\_DATA 2         // The second item is the kernel data segment descriptor.
14 #define GDT\_TMP 3          // The third item is system segment descriptor, not used.
15
16 #define LDT\_NUL 0          // The 0th item of the LDT, not used.
17 #define LDT\_CODE 1         // The first item is the user code segment descriptor .
18 #define LDT\_DATA 2         // The second item is the user data segment descriptor.
19
20 #endif
21

```

## 14.26 kernel.h

### 14.26.1 Functionality

The kernel.h file defines some function prototypes commonly used by the kernel.

### 14.26.2 Code annotation

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

```

1 /*
2 * 'kernel.h' contains some often-used function prototypes etc.
3 */
4 void verify_area(void * addr,int count);
5 volatile void panic(const char * str);
6 volatile
void do_exit(long error_code);
7 int printf(const char * fmt, ...);
8 int printk(const char * fmt, ...);
9 void
console_print(const char * str);
10 // 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。
11 int tty_write(unsigned ch, char * buf, int count);
12 // 汎用的なカーネルのメモリ割り当て関数です。(lib/malloc.c, 117)。
13 void * malloc(unsigned int size);
14 // 指定されたオブジェクトが占有していたメモリを解放します。(lib/malloc.c, 182) を参照してください。
15 void free_s(void * obj, int size);
16 // 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。
17 extern void hd_times_out(void);
18 // ビープ音を止めます。(kernel/chr_drv/console.c, 944)。
19 extern void sysbeepstop(void);
20 // ブラックスクリーン処理。(kernel/chr_drv/console.c, 981)となっています。
21 extern void blank_screen(void);
22 // 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。
23 extern void unblank_screen(void);
24
25 /*
26 * This is defined as a macro, but at some point this might become a

```

28 \* real subroutine that sets a flag if it returns true (to do  
29 \* BSD-style accounting where the process is flagged if it uses root  
30 \* privs). The implication of this is that you should do normal

---

```

31 * permissions checks first, and check suser() last.
31 */
32 #define suser() (current->euid == 0)           // Check if it is a super user.
33

```

---

## 14.27 math\_emu.h

### 14.27.1 Functionality

The math\_emu.h file contains the constant definitions and structures involved in Chapter 11 (Mathematic Coprocessor), including some of the data structures used by the kernel code to simulate various types of data when simulating mathematical coprocessors.

### 14.27.2 Code annotation

プログラム 14-24 linux/include/linux/math\_emu.h

---

```

1 /*
2  * linux/include/linux/math_emu.h
3  *
4 * (C) 1991 Linus Torvalds
5 */
6 #ifndef _LINUX_MATH_EMU_H
7 #define _LINUX_MATH_EMU_H
8
// スケジューラのヘッダファイルは、タスク構造、初期タスク0、およびいくつかの組み込みの
9 #include <linux/sched.h> // 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
11 // 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。
12 struct info {
13     long__ math_ret;      // The return address of the math_emulate() caller (INT 7).
14     long__ orig_eip;      // A place to temporarily save the original EIP.
15     long__ edi;          // The registers that the exception handler pushed.
16     long__ esi;
17     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。
18
// 以下（18～30行目）は、system-callが呼び出されたときのスタック内の構造と同じです。
19     long__ sys_call_ret;
20     long__ eax;
21     long__ ebx;
22     long__ ecx;
23     long__ edx;
24     long__ orig_eax;    // If it's not a sys-call but other interrupts, it is -1.
25     long__ fs;
26     long__ es;

```

```
27     long__ ds;
```

```

28     long__ eip;           // Lines 26 -- 30 are pushed by the CPU automatically.
29     long cs;
30     long eflags;
31     long esp;
32     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-> esi) 38
#define EDI (info-> edi) 39
#define EBP (info-> ebp) 40
#define ESP (info-> esp) 41
#define EIP (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。キーワード volatile を関数名の前に置くことで、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
52 50 void math_abort(struct info *, unsigned int); 51
53 #define math_abort(x, y) \
54 (((volatile void (*)(struct info *, unsigned int)) math_abort)((x), (y)))
54
55 /*
57 * Gcc forces this stupid alignment problem: I want to use only two longs
58 * for the temporary real 64-bit mantissa, but then gcc aligns out the
59 * structure to 12 bytes which breaks things in math_emulate.c. Shit. I
60 * want some kind of "no-align" pragma or something.
60 */
61

// 一時的な本物の構造
// 合計64ビットのマンタを持っています。ここで、「a」は下位32ビット、「b」は上位32ビットを表す

```

```
62 // (1固定ビットを含む) で、「exponent」は指数値です。
63 typedef struct {
64     long a, b;
65     short exponent;
65 } temp_real;
66
```

```

// 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
67 // で、上記の temp_real 構造体と同様に動作します。
68 typedef struct {
69     short m0, m1, m2, m3;
70     short exponent;
71 } temp_real_unaligned;
72 // 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) ) 。
73 #define real_to_real(a, b) \
74 ((*(long long *) (b) = *(long long *) (a)), ((b)->exponent = (a)->exponent))
75
75 // 長い実数（倍精度）の構造体です。
76 typedef struct {
77     long a, b;           // 'a' is the lower 32 bits; 'b' is the upper 32 bits.
78 } long_real;
79
80
81 // 一時的な整数構造。
82 typedef struct {
83     long a, b;           // 'a' is the lower 32 bits; 'b' is the upper 32 bits.
84     short sign;
85 } temp_int;
85
// coprocessor (see Figure 11-6).

```

```

86 struct swd {
87     int ie:1;          // Invalid operation exception.
88     int de:1;          // Denormalized exception.
89     int ze:1;          // Divide by zero exception.
90     int oe:1;          // Overflow exception.
91     int ue:1;          // Underflow exception.
92     int pe:1;          // Precision exception.
93     int sf:1;          // Stack error flag, caused by overflow of the accumulator.
94     int ir:1;          // Ir, b: Set if any of the above 6 unmasked exceptions occur.
95     int c0:1;          // c0--c3: Condition code bits.
96     int c1:1;
97     int c2:1;
98     int top:3;         // Indicates the 80-bit register currently at the top of the stack.
99     int c3:1;
100    int b:1;
101 };
102
// 80387 internal register control mode constants.
103 #define I387 (current->tss.i387)           // 80387 status information of the process.
104 #define SWD (*(struct swd *)&I387.swd)      // Status control word in 80387.
105 #define ROUNDING (((I387.cwd >> 10) & 3)   // Get the rounding mode in the control word.
106 #define PRECISION (((I387.cwd >> 8) & 3)  // Get the precision mode in the control word.
107
// Constant that define the significant digits of a precision.
108 #define BITS24      0          // Precision Effective bits: 24 bits.
109 #define BITS53      2          // 53 bits.
110 #define BITS64      3          // 64 bits.

```

111

// 丸めモードの定数を定義します。

```

112 #define ROUND_NEAREST 0           // Round to the nearest or even.
113 #define ROUND_DOWN    1           // Trend to negative infinite.
114 #define ROUND_UP      2           // Trend to positive infinite.
115 #define ROUND_0       3           // Trend to cut to zero.
116
// 定数の定義。
117 #define CONSTZ   (temp_real_unaligned) {0x0000,0x0000,0x0000,0x0000,0x0000} // 0
118 #define CONST1   (temp_real_unaligned) {0x0000,0x0000,0x0000,0x8000,0x3FFF} // 1.0
119 #define CONSTPI  (temp_real_unaligned) {0xC235,0x2168,0xDA2,0xC90F,0x4000} // Pi
120 #define CONSTLN2 (temp_real_unaligned) {0x79AC,0xD1CF,0x17F7,0xB172,0x3FFE} // Loge(2)
// Loge(2) 121 #define CONSTLG2 (temp_real_unaligned) {0xF799,0xFBCF,0x9A84,0x9A20,0x3FFD}.
// Log10(2) 122 #define CONSTL2E (temp_real_unaligned)
{0xF0BC,0x5C17,0x3B29,0xB8AA,0x3FFF}. // Log2(e) 123 #define CONSTL2T (temp_real_unaligned)
{0x8AFE,0xCD1B,0x784B,0xD49A,0x4000}. // Log2(10) 124
125 // 80387の状態を設定します。
126 #define set_IE() (I387.swd |= 1)
127 #define set_DE() (I387.swd |= 2)
128 #define set_ZE() (I387.swd |= 4)
129 #define set_OE() (I387.swd |= 8)
130 #define set_UE() (I387.swd |= 16)
131 #define set_PE() (I387.swd |= 32)
132
// 80387の制御条件を設定

```

140

```

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); 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
172 171 /* mul.c */
173 // Simulate floating point multiply instructions.
174 void fmul(const temp_real *, const temp_real *, temp_real *);
174
176 175 /* div.c */
177 // Simulate floating point division instructions.
178 void fdiv(const temp_real *, const temp_real *, temp_real *);
178
179 /* compare.c */
180 // Simulate floating point comparison instructions.
181 void fcom(const temp_real *, const temp_real *); // FCOM, compare two numbers.
182 void fucom(const temp_real *, const temp_real *); // FUCOM, no order comparison.
183 void ftst(const temp_real *); // FTST, top stack accumulator compared to 0.
184
185 #endif
186

```

---

## 14.28 mm.h

### 14.28.1 Functionality

The mm.h file is the memory management header file. It mainly defines the size of the memory page and several page release function prototypes.

### 14.28.2 Code annotation

[1](#) プログラム 14-25 linux/include/linux/mm.h

```

2 #ifndef MM_H
3 #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。

// used functions of the kernel.

```

// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナルの
定義
// manipulation function prototypes.
6 #include <linux/kernel.h>
7 #include <signal.h> (英語)
8
9 extern int SWAP_DEV;           // Memory page swap device number (mm/memory.c, line 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。
ll_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
14 // 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。
15 extern unsigned long get_free_page(void);
// 内容が変更された物理的なメモリページを、指定された
16 // 線形アドレス空間。put_page()とほぼ同じです。
17 extern unsigned long put_dirty_page(unsigned long page,unsigned long address);
18 // アドレス「addr」で始まる物理メモリのページを解放する。
19 extern void free_page(unsigned long addr);
20 // 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) 。
21 void swap_free(int page_nr);
22 // 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).
23 void swap_in(unsigned long *table_ptr);
24
// //// アウトオブメモリー (oom) 処理機能。
25 extern inline volatile void oom(void) 21 {
    // do_exit()では、終了コードを使用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
    // SIGSEGV(11)は "Resource is temporarily unavailable" であり、これは同義である。
23     printf("out of memory\n|r");
24     do_exit(SIGSEGV);
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
// トランスレーション・ルックサイド・バッファーと呼ばれるチップの内部キャッシュにあるページテ
ーブルデータを使用する
// (TLB) を使用しています。ページテーブルの情報を変更した後は、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() ♪♪♪♪♪
27 asm ("movl %%eax,%%cr3": "a" (0))
28

```

---

```

29 /* these are not to be changed without changing head.s etc */
// Linux 0.12カーネルがデフォルトでサポートしている最大メモリ容量は16MBであり、これらの
30 // 定義を変更することで、より多くのメモリを搭載す
    ることができます。
31 #define LOW MEM 0x100000          // Low end of the physical memory (1MB)._
31 extern unsigned long HIGH MEMORY; // The highest address of physical memory.
32 #define PAGING MEMORY (15*1024*1024) // The size of the paged memory (15MB).
33 #define PAGING PAGES (PAGING MEMORY>>12) // The number of pages after paging (3840)._
34 #define MAP NR(addr) (((addr)-LOW MEM)>>12) // Map the memory address to the page number.
35 #define USED 100                // Page used flag, see memory.c, line 449.
36

```

---

```
// メモリマッピングのバイトマップで、1バイトが1ページを表します。各ページの対応するバイトは
// ページが現在参照されている（使用されている）回数をマークする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。
#define PAGE_DIRTY      0x40          // Bit 6, the page is dirty (modified).
```

39

40 `#define PAGE_ACCESSED 0x20` // Bit 5, the page was accessed.

41 `#define PAGE_USER 0x04` // Bit 2, the page belongs to: 1-user; 0-superuser.

42 `#define PAGE_RW 0x02` // Bit 1, read/write rights: 1 - write; 0 - read.

43 `#define PAGE_PRESENT 0x01` // Bit 0, page exists: 1 - present; 0 - not exist.

44

45 `#endif`

46

---

### 14.29.1 Functionality

Sched.h is the scheduler header file, which defines the task structure task\_struct, initial task 0 data, and some embedded assembly function macros for memory descriptor parameter settings and acquisition and task context switch macro switch\_to(). Below we describe in detail the execution process of the task switch macro.

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stack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なく、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)

Next, we exchange the new task pointer in the ECX register with the current task pointer in the global variable 'current', let 'current' contain the pointer value of the new task we are going to switch to, and ECX saves the current task. Then execute the instruction LJMP that indirectly jumps to\_tmp. The instruction that jumps to the new task TSS selector will ignore the undefined value part of\_tmp, and the CPU will automatically jump to the new task specified by the TSS segment to execute, and the task (current task) will be suspended. This is why we don't need to set other undefined parts of the structure variable\_tmp. See Figure 4-37 in Section 4.7 for a schematic diagram of the task switching operation.

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

## 4.29.2

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。

### 14.29.3 Code annotation

[1](#) プログラム 14-26 linux/include/linux/sched.h

```

2 #ifndef SCHED_H
3 #define SCHED_H
4
5
6 #define HZ 100           // Define system clock tick frequency (10ms per tick)
7
8 #define NR_TASKS       64           // The max number of tasks in the system.
9 #define TASK_SIZE      0x04000000  // The size of each task (64MB).
10 #define LIBRARY_SIZE   0x00400000  // The size of the loaded library (4MB).
11
12 #if (TASK_SIZE & 0x3fffff)
13 #error "TASK_SIZE must be multiple of 4M"
14#endif
15
16 #if (LIBRARY_SIZE & 0x3fffff)
17 #error "LIBRARY_SIZE must be a multiple of 4M"
18#endif
19
20 #if (LIBRARY_SIZE >= (TASK_SIZE/2))
21 #error "LIBRARY_SIZE too damn big!"
22#endif
23 #if (((TASK_SIZE>>16)*NR_TASKS) !=0x10000)
24 #error "TASK_SIZE*NR_TASKS must be 4GB"
25#endif
26
27 // 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）。
28#define LIBRARY_OFFSET (TASK_SIZE - LIBRARY_SIZE)
29
30 // 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
31 // に使用されます。
32 // into seconds and microseconds.
33#define CT_TO_SECS(x) ((x) / HZ)
34#define CT_TO_USECS(x) (((x) % HZ) * 1000000/HZ)
35
36#define FIRST_TASK task[0]           // Task 0 is special, so we define a symbol for it.
37#define LAST_TASK task[NR_TASKS-1]    // The last task in the task array.
38
39 // <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
40 // ています。
41 // along with several selector constants.

```

```
// <linux/fs.h> ファイルシステムのヘッダーファイル。ファイルテーブル構造を定義する
// (file,buffer_head,
//   m_inode, etc.).
// <linux/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
//   release function prototypes.
// <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。
//     resources used by processes.
// <signal.h> シグナルのヘッダーファイルです。シグナルシンボル定数、シグナル構造体、シグナルの定義
//     manipulation function prototypes.

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 the close-on-exec-flags and select masks are in one long, max 32 files/proc"
44 #endif
45

    // プロセスの動作状態を定義します。
46 #define TASK_RUNNING          0 // process is running or is ready to run.
47 #define TASK_INTERRUPTIBLE    1 // in an interruptible wait state.
48 #define TASK_UNINTERRUPTIBLE  2 // in an uninterruptible wait state (mainly I/O op).
49 #define TASK_ZOMBIE           3 // in a dead state and the father has not yet signaled.
50 #define TASK_STOPPED          4 // process has stopped.

51
52 #ifndef NULL
53 #define NULL ((void *) 0)
54 #endif
55

    // プロセスのページディレクトリテーブルをコピーします。 ( mm/memory.c, 118 )
    // 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、その後、無限ループに入れます。 (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)();           // Define a function pointer type.
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      long    cwd;           // Control word.
stru
ct
i387
_str
uct
{.
68
69      long    swd;        // Status word.
70      long    twd;         // Tag word.
```

```

71     long    fip;           // Coprocessor code ip pointer.
72     long    fcs;           // Coprocessor code segment register.
73     long    foo;           // The offset of the memory operand.
74     long    fos;           // The segment of the memory operand.
75     long    st_space[20];   /* 8*10 bytes for each FP-reg = 80 bytes */
76 };
77

// TSS (Task Status Segment) データ構造。
78     long    back_link;      /* 16 high bits zero */
79
struct
tss_st
ruct {
79
80     long    esp0;
81     long    ss0;           /* 16 high bits zero */
82     long    esp1;
83     long    ss1;           /* 16 high bits zero */
84     long    esp2;
85     long    ss2;           /* 16 high bits zero */
86     long    cr3;
87     long    eip;
88     long    eflags;
89     long    eax, ecx, edx, ebx;
90     long    esp;
91     long    ebp;
92     long    esi;
93     long    edi;
94     long    es;             /* 16 high bits zero */
95     long    cs;             /* 16 high bits zero */
96     long    ss;             /* 16 high bits zero */
97     long    ds;             /* 16 high bits zero */
98     long    fs;             /* 16 high bits zero */
99     long    gs;             /* 16 high bits zero */
100    long    ldt;            /* 16 high bits zero */
101    long    trace_bitmap;   /* bits: trace 0, bitmap 16-31 */
102    struct i387\_struct i387;
103};

// 以下は、タスク（プロセス）データ構造、またはプロセス制御ブロック、またはプロセス記述子です。
// 詳細な説明は5.7節を参照してください。
//構造体 task_struct {
//  long state;           // -1 unrunnable, 0 runnable (ready), > 0 stopped.
//  long counter;          // Task run time tick (decrement), run time slice.
//  long priority;         // Priority. When task starts running, counter=priority.
//  long signal;           // Signal bitmap, each bit is a signal( = bit offset + 1).
//  struct sigaction sigaction[32]; // Signal attribute struct. Signal operation and flags.
//  long blocked;          // Process signal mask (Bitmap of masked signal).
//  int exit_code;          // Exit code after task stops, its parent will get it.
//  unsigned long start_code; // Code start location in linear address space.
//  unsigned long end_code;  // Code length or size (bytes).
//  unsigned long end_data; // Code size + data size (bytes).
//  unsigned long brk;       // Total size (number of bytes).
//  unsigned long start_stack; // Stack bottom location.
//  long pid;               // Process identifier.

```

```
// long pgrp;           // Process group number.  
// long session;        // Process session number.
```

```

// long leader;           // Leader session number.
// int groups[NGROUPS];   // Group numbers. A process can belong to more groups.
// task_struct *p_pptr;   // Pointer to parent process.
// task_struct *p_cptr;   // Pointer to youngest child process.
// task_struct *p_ysptr;   // Pointer to younger sibling process created afterwards.
// task_struct *p_osptr;   // Pointer to older sibling process created earlier.
// unsigned short uid;    // User id.
// unsigned short euid;   // Effective user id.
// unsigned short suid;   // Saved user id.
// unsigned short gid;    // Group id.
// unsigned short egid;   // Effective group id.
// unsigned short sgid;   // Saved group id.
// long timeout;          // Kernel timing timeout value.
// long alarm;             // Alarm timing value (ticks).
// long utime;             // User state running time (ticks).
// long stime;             // System state runtime (ticks).
// long cutime;            // Child process user state runtime.
// long cstime;            // Child process system state runtime.
// long start_time;         // Time the process started running.
// struct rlimit rlim[RLIM_NLIMITS]; // Resource usage statistics array.
// unsigned int flags;      // per process flags.
// unsigned short used_math; // Flag: Whether a coprocessor is used.
// int tty;                 // The tty subdevice number used. -1 means no use.
// unsigned short umask;    // The mask bit of the file creation attribute.
// struct m_inode *pwd;     // Current working directory i-node structure pointer.
// struct m_inode *root;    // Root i-node structure pointer.
// struct m_inode *executable; // The pointer to i-node structure of the executable file.
// struct m_inode *library;  // The loaded library i-node structure pointer.
// unsigned long close_on_exec; // A bitmap flags that close file handles on execution.
// struct file *filp[NR_OPEN]; // File structure pointer table, up to 32 items.
//                                // The index is the value of file descriptor.
// struct desc_struct ldt[3]; // LDT. 0-empty, 1-code seg cs, 2-data & stack seg ds & ss.
// struct tss_struct tss;    // The task status segment structure TSS of the process.
//};

105 struct task_struct {
106 /* these are hardcoded - don't touch */
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;   /* bitmap of masked signals */
113 /* various fields */
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     * pointers to parent process, youngest child, younger sibling,
120     * older sibling, respectively. (p->father can be replaced with
121     * p->p_pptr->pid)
122     */
123     struct task_struct *p_pptr, *p_cptr, *p_ysptr, *p_osptr;

```

```

124     unsigned short uid, euid, suid;
125     unsigned short gid, egid, sgid;
126     unsigned long timeout, alarm;
127     long utime, stime, cutime, cstime, start_time;
128     struct rlimit rlim[RLIM_NLIMITS];
129     unsigned int flags; /* per process flags, defined below */
130     unsigned short used_math;
131 /* file system info */
132     int tty; /* -1 if no tty, so it must be signed */
133     unsigned short umask;
134     struct m_inode * pwd;
135     struct m_inode * root;
136     struct m_inode * executable;
137     struct m_inode * library;
138     unsigned long close_on_exec;
139     struct file * filp[NR_OPEN];
140 /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
141     struct desc_struct ldt[3];
142 /* tss for this task */
143     struct tss_struct tss;
144 };
145
146 /*
149 147 * プロセスごとのフラグ
148 */
150 #define PF_ALIGNWARN 0x00000001 /* Print alignment warning msgs */
151 /* Not implemented yet, only for 486 */
152
153 /*
154 * INIT_TASK is used to set up the first task table, touch at
155 * your own risk!. Base=0, limit=0x9ffff (=640kB)
155 */
156 // 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。
157 #define INIT_TASK \
158 /* state etc */ {0,15,15, \ // state, counter, priority \
159 /* signals */ 0,{{}},0, \ // signal, sigaction[32], blocked \
160 /* ec, brk... */ 0,0,0,0,0, \ // exit_code, start_code, end_code, end_data, brk, start_stack \
161 /* pid etc.. */ 0,0,0,0, \ // pid, pgrp, session, leader \
162 /* suppl grp*/{NOGROUP}, \ // groups[] \
163 /* proc links*/&init_task.task,0,0,0, \ // p_pptr, p_cptr, p_ysptr, p_osptr \
164 /* uid etc */ 0,0,0,0,0,0, \ // uid, euid, suid, gid, egid, sgid \
165 /* timeout */ 0,0,0,0,0,0, \ // alarm, utime, stime, cutime, cstime, start_time, used_math \
166 /* rlimits */ { {0xffffffff, 0xffffffff}, {0xffffffff, 0xffffffff}, \ \
167 {0xffffffff, 0xffffffff}, {0xffffffff, 0xffffffff}, {0xffffffff, 0xffffffff}, \ \
168 /* flags */ 0, \ // flags \
169 /* math */ 0, \ // used_math, tty, umask, pwd, root, executable, close_on_exec \
170 /* fs info */ -1,0022,NULL,NULL,NULL,NULL,0, \ \
171 /* filp */ {NULL,}, \ // filp[20] \
172 { \ // ldt[3] \
173 {0,0}, \ \
174 /* ldt */ {0x9f,0xc0fa00}, \ // code size 640K, base 0x0,G=1,D=1,DPL=3,P=1 TYPE=0xa \
175 {0x9f,0xc0f200}, \ // data size 640K, base 0x0,G=1,D=1,DPL=3,P=1 TYPE=0x2

```

```

176     }, \
177 /*tss*/ {0, PAGE_SIZE+(long)&init_task, 0x10, 0, 0, 0, 0, (long)&pg_dir, \  
178     0, 0, 0, 0, 0, 0, 0, \
179     0, 0, 0x17, 0x17, 0x17, 0x17, 0x17, 0x17, \
180     LDT(0), 0x80000000, \
181     {} \
182 }, \
183 }
184
185 extern struct task_struct *task[NR_TASKS]; // An array of task pointers.
186 extern struct task_struct *last_task_used_math;
187 extern struct task_struct *current; // The current process pointer.
188 extern unsigned long volatile jiffies; // Ticks (10ms/tick) from start of boot.
189 extern unsigned long startup_time; // Boot time. seconds since 1970:0:0:0.
190 extern int jiffies_offset; // The number of ticks that need to be adjusted.
191
192 #define CURRENT_TIME (startup_time+(jiffies+jiffies_offset)/HZ) // Current time(seconds).
193
// 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 ) 194
extern void add_timer(long jiffies, void (*fn)(void));
// 途切れることのないスリープ待ち。(kernel/sched.c ) 195
extern void sleep_on(struct task_struct ** p);
// スリープ待ちで中断。(kernel/sched.c )
196 extern void interruptible_sleep_on(struct task_struct ** p);
// スリープの処理を明確に起こす。(kernel/sched.c ) 197 extern
void wake_up(struct task_struct ** p);
// 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。
198 extern int in_group_p(gid_t grp); 199
200 /*
201 * 最初のTSSを探すためのgdtへの入力。0-nul, 1-cs, 2-ds, 3-syscall 202 * 4-TSS0,
202 * 5-LDT0, 6-TSS1など ...
203 */
// グローバルテーブルで最初のTSSのエントリを探します。0-nul (使用しない) 、1-コードセグメント
// / cs、2-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、5-ロ
// ーカルテーブル
// LTD0、6タスクのステートセグメントTSS1などです。
// 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
// 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
// GDTテーブルの4番目のディスクリプターアイテム（シスコールアイテム）をアイドル状態にしまし
// た。
204 // 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。
205 #define FIRST_TSS_ENTRY 4
206 #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

207       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。

208 #define TSS(n) (((unsigned long) n)<<4)+(FIRST\_TSS\_ENTRY<<3))  
207 #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

```
// task into the local descriptor table register LDTR.
```

```

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。
210 // 返却：n - 現在のタスク番号。
211 #define str(n) \
212 __asm_ ("str %%ax|n|t" \           // Save the TSS selector in the task register to EAX.
213     "subl %2, %%eax|n|t" \         // (EAX - FIRST_TSS_ENTRY * 8) => EAX
214     "shrl $4, %%eax" \           // (EAX / 16) => EAX = current task number.
215     :"=a" (n) \                  // Output operand
216     :"a" (0), "i" (FIRST_TSS_ENTRY<<3))
217 /*
218 *      switch_to(n) should switch tasks to task nr n, first
219 * checking that n isn't the current task, in which case it does nothing.
220 * This also clears the TS-flag if the task we switched to has used
221 * the math co-processor latest.
222 */
// 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
228 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バイトは使用しません。）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
// 228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-bit segment selector: 32-bit offset.
// 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。

```
224 222 #define switch_to(n) {  

225     struct {long a,b;} tmp; ●。  

226     __asm__ ("cmp l%%ecx, _current|n|t" \      // Is task n the current task? (current ==task[n]?)  

227         "je 1f|n|t" \                            // If so, do nothing and return.  

228         "movw %%dx, %l|n|t" \                  // The new task TSS selector is stored in_ tmp.b  

229         "xchgl %%ecx, _current|n|t" \           // Current = task[n]; ECX =the task switched out.  

230         "ljmp %0|n|t" \                          // Long jump to *& tmp, causing task switching.  

231     //次の文は、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  

232     "cmpl %%ecx, _last_task_used_math\n\t" \ // original task used math last time?  

233 }
```

```
229  

230     "jne 1f|n|t" \                            // If not, jump forward and exit.  

231     "clts |n" \                             // If yes, then clear the TS flag in CRO.
```

```
232      "l:" \                                // exit.  
233      :: "m" (*&_tmp.a), "m" (*&_tmp.b), \  
234      "d" ( TSS(n)), "c" ((long) task[n])); \  
235 }  
236 // Page alignment (nowhere in the kernel to reference this!)
```

---

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 \_\_asm\_\_( "movw %%dx, %0|n|t" \ // Lower 16 bits(15-0) of the base=>[addr+2].

#d

efi

ne

s

et

b

as

e(

ad

dr

,b

as

e)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

```

256      "movb %%dl, %1" \           // (19-16)of the limit into 1 byte, stored at [addr+6].
257      :: "m" (*(addr)), \
258      "m" (*((addr)+6)), \
259      "d" (limit) \
260      :"dx")
261
// ローカルディスクリプターテーブルLDTのディスクリプターのベースアドレスフィールドを設定します。
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 )
264
// アドレス addr のディスクリプターから base を取得します。これは、_set_base()の逆です。
// EDX - store base ( base ); %1 - addr offset 2; %2 - addr offset 4; %3 - addr offset 7.
267 265 #define _get_base(addr) ({\
266 unsigned long base; })
268 __asm_ ( "movb %3, %%dh|n|t" \           // Upper 8 bits(31-24) of upper 16 bits [addr+7] =>DH.
269      "movb %2, %%dl|n|t" \           // Lower 8 bits(23-16) of upper 16 bits [addr+4] =>DL.
270      "shll $16, %%edx|n|t" \         // Upper 16 bits are moved to the upper 16 bits of EDX.
271      "movw %1, %%dx" \             // Lower 16 (15-0) bits of the base at [addr+2] => DX.
272      : "=d" ( base ) \            // Thus EDX contains a 32-bit segment base address.
272      : "m" (((addr)+2)), \
273      "m" (((addr)+4)), \
274      "m" (((addr)+7)); \
275 base;})
276
// LDTの'ldt'が指すセグメントディスクリプターのベースアドレスを取る。
277 #define get_base(ldt) _get_base( ((char *)&(ldt)) ) 278
// セグメントセレクタ'segment'で指定されたディスクリプタのセグメント制限を取る。
// LSLという命令は、Load Segment Limitの略です。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 ("lsll %1,%0n\%tincl %0":=r" ( limit): "r" (segment)); 282 limit;})

283

284 #endif

285

## 14.30 sys.h

### 14.30.1 Functionality

The sys.h header file lists the prototypes of all system-call functions in the kernel, as well as the system-call function pointer table.

### 14.30.2 Code annotation

プログラム 14-27 linux/include/linux/sys.h

```

1 /*
2 * Why isn't this a .c file? Enquiring minds....
3 */
4
5 extern int sys_setup();           // 0 - System initializations.      (kernel/blk_drv/hd.c, 74)
6 extern int sys_exit();           // 1 - Program exit.              (kernel/exit.c, 365)
7 extern int sys_fork();           // 2 - Create a new process.       (kernel/sys_call.s, 222)
8 extern int sys_read();           // 3 - Read file.                (fs/read_write.c, 55)
9 extern int sys_write();          // 4 - Write file.               (fs/read_write.c, 83)
10 extern int sys_open();           // 5 - Open file.                (fs/open.c, 171)
11 extern int sys_close();          // 6 - Close file.               (fs/open.c, 219)
12 extern int sys_waitpid();        // 7 - Wait process to terminate. (kernel/exit.c, 370)
13 extern int sys_creat();          // 8 - Create file.               (fs/open.c, 214)
14 extern int sys_link();           // 9 - Ceate hard linke to a file. (fs/namei.c, 837)
15 extern int sys_unlink();         // 10 - Delete a filename(or file) (fs/namei.c, 709)
16 extern int sys_execve();         // 11 - Execute a program.        (kernel/sys_call.s, 214)
17 extern int sys_chdir();          // 12 - Change current directory. (fs/open.c, 76)
18 extern int sys_time();           // 13 - Get current time.         (kernel/sys.c, 134)
19 extern int sys_mknod();          // 14 - Create block/char special file. (fs/namei.c, 464)
20 extern int sys_chmod();          // 15 - Change file mode.         (fs/open.c, 106)
21 extern int sys_chown();          // 16 - Chane file owner or group. (fs/open.c, 122)
22 extern int sys_break();          // 17 -                           (kernel/sys.c, 33)*
23 extern int sys_stat();           // 18 - Get file status using path name. (fs/stat.c, 36)
24 extern int sys_lseek();          // 19 - Reposite r/w file offset. (fs/read_write.c, 25)
25 extern int sys_getpid();         // 20 - Get process id.           (kernel/sched.c, 380)
26 extern int sys_mount();          // 21 - Mount a file-system.       (fs/super.c, 199)
27 extern int sys_umount();         // 22 - Unmount a file-system.     (fs/super.c, 166)

```

---

```

28 extern int sys_setuid();           // 23 - Set process user id.          (kernel/sys. c, 196)
29 extern int sys_getuid();           // 24 - Get process user id.          (kernel/sched. c, 390)
30 extern int sys_stime();            // 25 - Set system time.              (kernel/sys. c, 207)
31 extern int sys_ptrace();           // 26 - Process tracing.              (kernel/sys. c, 38)*
32 extern int sys_alarm();            // 27 - Set alarm time.               (kernel/sched. c, 370)
33 extern int sys_fstat();            // 28 - Get file status by using handle. (fs/stat. c, 58)
34 extern int sys_pause();             // 29 - Pause the process running.   (kernel/sched. c, 164)
35 extern int sys_utime();            // 30 - Set file access and modified time. (fs/open. c, 25)
36 extern int sys_stty();              // 31 - Modify terminal settings.    (kernel/sys. c, 43)*
37 extern int sys_gtty();              // 32 - Get terminal settings.       (kernel/sys. c, 48)*
38 extern int sys_access();            // 33 - Check file access permission. (fs/open. c, 48)
39 extern int sys_nice();              // 34 - Set execution priority.      (kernel/sched. c, 410)
40 extern int sys_ftime();             // 35 - Get date and time.           (kernel/sys. c, 28)*
41 extern int sys_sync();              // 36 - Synchronous data with dev.   (fs/buffer. c, 44)
42 extern int sys_kill();              // 37 - Terminate a process.         (kernel/exit. c, 205)
43 extern int sys_rename();             // 38 - Change filename.             (kernel/sys. c, 53)*
44 extern int sys_mkdir();             // 39 - Make a directory.            (fs/namei. c, 515)
45 extern int sys_rmdir();              // 40 - Remove a directory.          (fs/namei. c, 635)
46 extern int sys_dup();                // 41 - Duplicate a file handle.     (fs/fcntl. c, 42)
47 extern int sys_pipe();               // 42 - Create a pipe.                (fs/pipe. c, 76)
48 extern int sys_times();              // 43 - Get running time.            (kernel/sys. c, 216)
49 extern int sys_prof();               // 44 - Execution time zone.        (kernel/sys. c, 58)*
50 extern int sys_brk();                // 45 - Change data segment len.     (kernel/sys. c, 228)
51 extern int sys_setgid();              // 46 - Set process group id.        (kernel/sys. c, 98)
52 extern int sys_getgid();              // 47 - Set process group id.        (kernel/sched. c, 400)
53 extern int sys_signal();              // 48 - Signal processing.           (kernel/signal. c, 85)
54 extern int sys_geteuid();              // 49 - Get efficient user id.       (kenrl/sched. c, 395)
55 extern int sys_getegid();              // 50 - Get efficient group id.      (kenrl/sched. c, 405)
56 extern int sys_acct();                // 51 - Process accounting.          (kernel/sys. c, 109)*
57 extern int sys_phys();                // 52 - Map phy mem to process space. (kernel/sys. c, 114)*
58 extern int sys_lock();                // 53 -                               (kernel/sys. c, 119)*
59 extern int sys_ioctl();               // 54 - Device i/o control.         (fs/ioctl. c, 31)
60 extern int sys_fcntl();               // 55 - File operation control.      (fs/fcntl. c, 47)
61 extern int sys_mpx();                 // 56 -                               (kernel/sys. c, 124)*
62 extern int sys_setpgid();              // 57 - Set process group id.        (kernel/sys. c, 245)
63 extern int sys_ulimit();               // 58 - Statistical resource usage. (kernel/sys. c, 129)
64 extern int sys_uname();                // 59 - Show system info.            (kernel/sys. c, 343)
65 extern int sys_umask();                // 60 - Get default file creation mode. (kernel/sys. c, 515)
66 extern int sys_chroot();               // 61 - Change root directory.       (fs/open. c, 91)
67 extern int sys_ustat();                // 62 - Get file system states.      (fs/open. c, 20)
68 extern int sys_dup2();                 // 63 - Duplicate file handle.       (fs/fcntl. c, 36)
69 extern int sys_getppid();              // 64 - Get parent process id.        (kernel/sched. c, 385)
70 extern int sys_getpgrp();              // 65 - Get pid (getpgid(0))         (kernel/sys. c, 271)
71 extern int sys_setsid();               // 66 - Set new session id.          (kernel/sys. c, 276)
72 extern int sys_sigaction();              // 67 - Set signal operation.        (kernel/signal. c, 100)
73 extern int sys_sgetmask();              // 68 - Get signal mask code.        (kernel/signal. c, 14)
74 extern int sys_ssetmask();              // 69 - Set signal mask code.        (kernel/signal. c, 19)
75 extern int sys_setreuid();              // 70 - Set real/efficient uid.       (kernel/sys. c, 159)
76 extern int sys_setregid();              // 71 - Set real/efficient pid.       (kernel/sys. c, 74)
77 extern int sys_sigpending();              // 73 - Check pending signals.        (kernel/signal. c, 27)
78 extern int sys_sigsuspend();              // 72 - Suspending a process.         (kernel/signal. c, 48)
79 extern int sys_sethostname();              // 74 - Set host name.                (kernel/sys. c, 357)
80 extern int sys_setrlimit();              // 75 - Set resource using limit.     (kernel/sys. c, 387)

```

```

81 extern int sys_getrlimit(); // 76 - Get resource using limit. (kernel/sys.c, 375)
82 extern int sys_getrusage(); // 77 - Get resource usage. (kernel/sys.c, 412)
83 extern int sys_gettimeofday(); // 78 - Get time of the day. (kernel/sys.c, 440)
84 extern int sys_settimeofday(); // 79 - Set time of the day. (kernel/sys.c, 466)
85 extern int sys_getgroups(); // 80 - Get process all group ids. (kernel/sys.c, 289)
86 extern int sys_setgroups(); // 81 - Set process group array. (kernel/sys.c, 307)
87 extern int sys_select(); // 82 - wait for file change state. (fs/select.c, 216)
88 extern int sys_symlink(); // 83 - Create file sysmbol link. (fs/namei.c, 767)
89 extern int sys_lstat(); // 84 - Get link file state. (fs/stat.c, 47)
90 extern int sys_readlink(); // 85 - Read link file contents. (fs/stat.c, 69)
91 extern int sys_uselib(); // 86 - Select shared lib. (fs/exec.c, 42)
92
// システムコールの割り込みで使用される関数ポインタテーブル
// 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_ftime, sys_stty, sys_gtty, sys_access,
99 sys_nice, sys_fsync, 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 /* So we don't have to do any more manual updating.... */
111 int NR_syscalls = sizeof(sys_call_table)/sizeof(fn_ptr);
112

```

## 14.31 tty.h

### 14.31.1 Functionality

The tty.h file defines the terminal data structure and some constants, as well as the macros used by the tty queue buffer operations.

### 14.31.2 Code annotation

プログラム 14-28 linux/include/linux/tty.h

```

1 /*
2 * 'tty.h' defines some structures used by tty_io.c and some defines.

```

3 \*

4 \* NOTE! Don't touch this without checking that nothing in rs\_io.s or

5 \* con\_io.s breaks. Some constants are hardwired into the system (mainly

```

6 * offsets into 'tty_queue'.
7 */
8
9 #ifndef _TTY_H_
10#define _TTY_H_
11
12#define MAX_CONSOLES 8           // The maximum number of virtual consoles.
13#define NR_SERIALS 2            // The number of serial terminals.
14#define NR_PTYS 4               // The number of psesudo terminals.
15
16 extern int NR_CONSOLES;      // The number of virtual consoles.
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        // The size of the tty queue buffer.
21
// Tty キャラクタバッファキューのデータ構造。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。
// 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
22 //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。
23 struct tty_queue {
24     unsigned long data;          // Char lines in the buffer or the serial port.
25     unsigned long head;          // The data header in the buffer.
26     unsigned long tail;          // The data tail in the buffer.
27     struct task_struct * proc_list; // process list waiting for this queue buffer.
28     char buf[TTY_BUF_SIZE];      // The buffer.
29 };
30
#define IS_A_CONSOLE(min) (((min) & 0xC0) == 0x00) // console.

```

```

30
31 #define IS_A_SERIAL(min)      (((min) & 0xC0) == 0x40)      // serial terminal.
32 #define IS_A_PTY(min)        ((min) & 0x80)                  // psesudo terminal.
33 #define IS_A_PTY_MASTER(min) (((min) & 0xC0) == 0x80)      // master pty.
34 #define IS_A_PTY_SLAVE(min)   (((min) & 0xC0) == 0xC0)      // slate pty.
35 #define PTY_OTHER(min)       ((min) ^ 0x40)                 // other type terminal.
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
37 // バッファの右/左で、ポインタは周期的に移動します。
38 #define INC(a) ((a) = ((a)+1) & (TTY_BUF_SIZE-1))
39 #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。
#define EMPTY(a) ((a)->head == (a)->tail)

```

```
39  
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)->head-(a)->tail)&(TTY_BUF_SIZE-1))  
// 'queue' バッファの 'tail' から文字を取得し、tail++とします。  
// '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 PUTC(c, queue) \
47 (void)({(queue)->buf[(queue)->head]=(c);INC((queue)->head);}) \
48
    // 端末のキーボードに入力されている文字の種類を確認します。
49 #define INTR_CHAR(tty) ((tty)->termios.c_cc[VINTR])           // Send signal SIGINT. \
50 #define QUIT_CHAR(tty) ((tty)->termios.c_cc[VQUIT])            // Send signal SIGQUIT. \
51 #define ERASE_CHAR(tty) ((tty)->termios.c_cc[VERASE])          // Erase a char. \
52 #define KILL_CHAR(tty) ((tty)->termios.c_cc[VKILL])           // Kill a line of chars. \
53 #define EOF_CHAR(tty) ((tty)->termios.c_cc[VEOF])             // End of file char. \
54 #define START_CHAR(tty) ((tty)->termios.c_cc[VSTART])          // Start output. \
55 #define STOP_CHAR(tty) ((tty)->termios.c_cc[VSTOP])           // Stop output. \
56 #define SUSPEND_CHAR(tty) ((tty)->termios.c_cc[VSUSP])          // Send signal SIGTSTP. \
57
58 // 端末のデータ構造。
59 struct tty_struct {
60     struct termios termios;                                // Terminal io mode & control structure.
61     int pgrp;                                            // The pgroup the terminal belongs to.
62     int session;                                         // The session.
63     int stopped;                                         // Terminal stopped flag.
64     void (*write)(struct tty_struct * tty);             // tty write function pointer.
65     struct tty_queue *read_q;                            // tty read queue.
66     struct tty_queue *write_q;                           // tty write queue.
67     struct tty_queue *secondary;                         // tty aux or canonical queue.
68 }
69 extern struct tty_struct tty_table[];
70 extern int fg_console;                                // Front console number.
71
    // 以下のマクロは、端末に対応するtty構造体へのポインタを取得します。
    // 端末の種類に応じてtty_table[]に番号「nr」を入れる。
    // 73行目の後半は、tty_table[]内の対応するtty構造を選択するために使用されます。
    // サブデバイス番号'dev'に基づいてテーブルを作成します。dev = 0の場合は、フォアグラウンドの端末が
    // が使用されているので、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。
    // 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。
    // (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
    // 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。
    // 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。
    // 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構造は次のようになります。
    // 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。
    // tty_io.cプログラムの70~73行目をご覧ください。
72 #define TTY_TABLE(nr) \(^o^)
73 (tty_table + ((nr) ? (((nr) < 64)? (nr)-1:(nr)) : fg_console) ) 74
    // ここでは、特殊文字の配列c_cc[]の初期値を変更できるようにしています。
    // 端末のtermios構造。POSIX.1では11種類の特殊文字が定義されていますが、Linuxシステムでは

```

```
// さらに、SVR4で使用される6つの特殊文字を定義しています。_POSIX_VDISABLE(0)が
// 定義されている場合、アイテムの値が _POSIX_VDISABLE と一致するとき、対応する特別な
/*      intr=^C          quit=^|          erase=del      kill=^U
```

75  
76  
77

```
eof=^D      vtime=\0      vmin=\1      sxtc=\0
start=^Q    stop=^S       susp=^Z      eol=\0
```

---

```

78      reprint=^R      discard=^U      werase=^W      lnext=^V
79      eol2=\0
80 */
81 #define INIT_C_CC "\003\034\177\025\004\0\1\0\021\023\032\0\022\017\027\026\0"
82
83 void rs_init(void);           // serial terminal init. (kernel/chr_drv/serial.c)
84 void con_init(void);         // Control terminal 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 Header files in the include/sys/ directory

The include/sys/ directory contains eight header files that are closely related to the system hardware resources and their settings, as shown in Listing 14-4.

リスト 14-4 linux/include/sys/ディレクトリ内のファイル

Filename	Size	Last Modified Time (GMT)	Description
param.h	196 bytes	1992-01-06 21:10:22	
resource.h	1809 bytes	1992-01-03 18:52:56	
stat.h	1376 bytes	1992-01-11 18:42:48	
time.h	1799 bytes	1992-01-09 03:51:28	
times.h	200 bytes	1991-09-17 15:03:06	
types.h	928 bytes	1992-01-14 13:50:35	
utsname.h	272 bytes	1992-01-04 15:05:42	
wait.h	593 bytes	1991-12-22 15:08:01	

## 14.33 param.h

### 14.33.1 Functionality

14.33.2 param.hファイルには、システムのハードウェアに関連するいくつかのパラメータ値が含まれ、定義されています。

### 14.33.3 Code annotation

プログラム 14-29 linux/include/sys/param.h

```

1 #ifndef _SYS_PARAM_H
2 #define _SYS_PARAM_H
3
4 #define HZ 100           // The system clock frequency, 100 times per second.
5 #define EXEC_PAGESIZE 4096 // Executable page size.
6
7 #define NGROUPS        32   /* Max number of groups per user */
8 #define NOGROUP        -1
9
10#define MAXHOSTNAMELEN 8    // The maximum length of the host name, 8 bytes.
11
12#endif
13

```

## 14.34 resource.h

### 14.34.1 Functionality

The resource.h header file contains information about the limits and utilization of the system resources used by the process. It defines the rusage structure and symbolic constants RUSAGE\_SELF, RUSAGE\_CHILDREN used by the system-call (or library function) getrusage(). It also defines the rlimit structure used by system-calls or functions getrlimit() and setrlimit() and the symbol constants used in the arguments.

getrlimit()やsetrlimit()でアクセスする情報は、プロセスタスク構造体のrlim[]配列にあります。この配列には合計でRLIM\_NLIMITS項目があり、各項目は図14-5に示すようにリソースの使用制限を定義するrlimit構造体である。図のように、Linux 0.12カーネルのプロセスには6つのリソース制限が定義されており、本ヘッダファイルの41～46行目で定義されている。

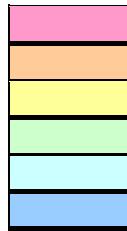


図14-5 プロセス記述子でのrlim[]配列の使い方

### 14.34.2 Code annotation

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

```

1 /*
2  * Resource control/accounting header file for linux
3  */
4
5 #ifndef _SYS_RESOURCE_H
6 #define _SYS_RESOURCE_H
7
// 以下のシンボル定数および構造体がgetrusage()で使用されます。の 412 行目を参照してください。
// kernel/sys.c ファイル。
8 /*
9  * Definition of struct rusage taken from BSD 4.3 Reno
10 *
11 * We don't support all of these yet, but we might as well have them...
12 * Otherwise, each time we add new items, programs which depend on this
13 * structure will lose. This reduces the chances of that happening.
14 */
// 以下は、getrusage()のパラメータ'who'で使用されるシンボル定数です。
// 現在のプロセスのリソース使用率情報を返します。

```

```

15 #define RUSAGE_SELF 0 // 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す。15 #define
RUSAGE_SELF 0 // 現在のプロセスのリソース使用率を返す。16 #define RUSAGE_CHILDREN -1 //
子プロセスのリソース使用率を返す。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 構造体である
    // (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。
    // ユーザーの状態；ru_stime フィールドはランニングタイムの統計を保存するために使用されます
    カー rusage {
        ネル
        状態
        のプ
        ロセ
        スの
        //。
18 struct
19     struct timeval ru_utime;           /* user time used */
20     struct timeval ru_stime;         /* system time used */
21     long    ru_maxrss;               /* maximum resident set size */
22     long    ru_ixrss;                /* integral shared memory size */
23     long    ru_idrss;                /* integral unshared data size */
24     long    ru_isrss;                /* integral unshared stack size */
25     long    ru_minflt;               /* page reclaims */
26     long    ru_majflt;               /* page faults */
27     long    ru_nswap;                /* swaps */
28     long    ru_inblock;              /* block input operations */
29     long    ru_oublock;              /* block output operations */
30     long    ru_msgsnd;              /* messages sent */
31     long    ru_msgrcv;              /* messages received */
32     long    ru_nsignals;             /* signals received */
33     long    ru_nvcs;                /* voluntary context switches */
34     long    ru_nivcs;               /* involuntary */
35 };
36
    // getrlimit() と setrlimit() で使用するシンボル定数と構造体を以下に示します。
37 /*
38 * リソースの制限 39
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、プロセススタック内の 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 time in ms */
42 #define RLIMIT_FSIZE     1           /* Maximum filesize */
43 #define RLIMIT_DATA      2           /* max data size */
44 #define RLIMIT_STACK     3           /* max stack size */
45 #define RLIMIT_CORE      4           /* max core file size */
46 #define RLIMIT_RSS       5           /* max resident set size */
47

```

```
48 #ifdef notdef
49 #define RLIMIT_MEMLOCK 6      /* max locked-in-memory address space*/
50 #define RLIMIT_NPROC    7      /* max number of processes */
51 #define RLIMIT_OFILE    8      /* max number of open files */
52 #endif
53 // このシンボリック定数は、Linuxで制限されているリソースの種類を定義します。この定数は
// 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 // The resource is unlimited or cannot be modified.
57
58 // リソースリミット構造
59 struct rlimit {
60     int    rlim_cur;           // Current resource limit, or soft limit.
61     int    rlim_max;          // Hard limit.
61 };
62
63#endif /* _SYS_RESOURCE_H */
64

```

## 14.35 stat.h

### 14.35.1 Functionality

The stat.h header file shows the data returned by the file function stat() and its structure type, as well as some property operation test macros and function prototypes.

### 14.35.2 Code annotation

1 プログラム 14-31 linux/include/sys/stat.h

```

2 ifndef SYS_STAT_H
3 define SYS_STAT_H
3
4 include <sys/types.h>
5
7 // 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。6
struct stat {
8     dev_t   st_dev;        // The device number that contains the file.
9     ino_t   st_ino;        // File i-node number.
10    umode_t st_mode;      // File type and modes (see below).
11    nlink_t st_nlink;     // The number of links to the file.
12    uid_t   st_uid;        // The user ID of the file.
13    gid_t   st_gid;        // The group ID of the file.
14    dev_t   st_rdev;       // Device number (if it is a special char or block file).
15    off_t   st_size;       // File size (in bytes).
16    time_t  st_atime;      // Last access time.
17    time_t  st_mtime;      // Last modify time.
18    time_t  st_ctime;      // The time the inode was last changed.
18 };
19
// 
// st_modeで使用する値に定義されているシンボル定数の一部をご紹介します。
// のフィールドになります。これらの値はすべて8進法で表されます。覚えやすくするために、これらの
記号の
// 名前は、いくつかの英単語の頭文字や略語を組み合わせたものです。例えば、以下のようになります。
// S_IFMTという名前の各大文字は、State、Inode、File、Mask、を表しています。

```

// また、S\_IFREGという名前は、State、Inode、Fileの頭文字を組み合わせたものです。  
// S\_IRWXUという名前は、State、Inode、Read、Write、eXecute、Userを表しています。その他の名称

```
// File types:
```

```

20 #define S_IFMT 00170000 // File type bit mask (in octal).
21 #define S_IFLNK 0120000 // Symbolic link.
22 #define S_IFREG 0100000 // Regular file.
23 #define S_IFBLK 0060000 // Block special device files, such as harddisk dev/hd0.
24 #define S_IFDIR 0040000 // Directory.
25 #define S_IFCHR 0020000 // Char device file.
26 #define S_IFIFO 0010000 // FIFO special file.

// ファイルモードのビットです。

// 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。
// ファイルが実行されます。S_ISGID は、グループIDに対しても同様の処理を行います。

27 #define S_ISUID 0004000 // Set the user ID (set-user-ID) at execution time.
28 #define S_ISGID 0002000 // Set the group ID (set-group-ID) at execution time.
29 #define S_ISVTX 0001000 // For directories, the restricted delete flag.
30
31 #define S_ISLNK(m) (((m) & S_IFMT) == S_IFLNK) // Test if it's a symbolic link file.
32 #define S_ISREG(m) (((m) & S_IFMT) == S_IFREG) // a regular file.
33 #define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR) // a directory.
34 #define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR) // a char device file.
35 #define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK) // a block device file.
36 #define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO) // a FIFO special file.
37
// File access permission:

38 #define S_IRWXU 00700 // The owner can read, write, execute/search (U for User).
39 #define S_IRUSR 00400 // The owner can read.
40 #define S_IWUSR 00200 // The owner can write.
41 #define S_IXUSR 00100 // The owner can execute/search.
42

```

```
43 #define S_IRWXG 00070      // Group members can read, write, execute/search (G for Group).
44 #define S_IRGRP 00040      // Group members can read.
45 #define S_IWGRP 00020      // Group members can write.
46 #define S_IXGRP 00010      // Group members can execute/search.
47
48 #define S_IRWXO 00007      // Others can read, write, execute/search (0 stands for Other).
49 #define S_IROTH 00004      // Others can read (the last 3 letters represent Other).
50 #define S_IWOTH 00002      // Others can write.
51 #define S_IXOTH 00001      // Others can execute/search.
52
53 extern int chmod(const char *_path, mode_t mode);           // Change file modes.
54 extern int fstat(int fildes, struct stat *stat_buf);        // Get file state info by fhandle.
55 extern int mkdir(const char *_path, mode_t mode);           // Make a directory.
56 extern int mkfifo(const char *_path, mode_t mode);          // Make a pipe file.
57 extern int stat(const char *filename, struct stat *stat_buf); // Get file state info.
58 extern mode_t umask(mode_t mask);                          // Set mode mask.
59
60 #endif
61
```

## 14.36 time.h

### 14.36.1 Functionality

The time.h header file defines the timeval structure and the internally used itimerval structure, as well as the time zone constants.

### 14.36.2 Code annotation

[1](#) プログラム 14-32 linux/include/sys/time.h

```

2 #ifndef _SYS_TIME_H_
3 #define _SYS_TIME_H_
3
4 /* gettimofday returns this */
5 struct timeval {
6     long    tv_sec;          /* seconds */
7     long    tv_usec;         /* microseconds */
8 };
9
// タイムゾーンの構造
10 // TZはTime Zoneの略で、DSTはDaylight Saving Timeの略です。
11 struct timezone {
12     int      tz_minuteswest; /* minutes west of Greenwich */
13     int      tz_dsttime;     /* type of dst correction */
13 };
14
15 #define DST_NONE          0      /* not on dst */
16 #define DST_USA           1      /* USA style dst */
17 #define DST_AUST          2      /* Australian style dst */
18 #define DST_WET           3      /* Western European dst */
19 #define DST_MET           4      /* Middle European dst */
20 #define DST_EET           5      /* Eastern European dst */
21 #define DST_CAN           6      /* Canada */
22 #define DST_GB            7      /* Great Britain and Eire */
23 #define DST_RUM           8      /* Rumania */
24 #define DST_TUR           9      /* Turkey */
25 #define DST_AUSTALT       10     /* Australian style with shift in 1986 */
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)))    // Set the fd in the fd set.
28 #define FD_CLR(fd, fdsetp)   (*(fdsetp) &= ~(1 << (fd)))   // Clear the fd in the set.
29 #define FD_ISSET(fd, fdsetp) ((*(fdsetp) >> fd) & 1)        // Is the fd in the set ?
30 #define FD_ZERO(fdsetp)      (*(fdsetp) = 0)                  // Clear all fds in the set.
31
32 /*
33 * タイムバルの操作。34*
35 * NB: timercmp は >= や <= では動作しません。
36 // timeval時間構造体の操作マクロです。
37 #define timerisset(tvp)      ((tvp)->tv_sec || (tvp)->tv_usec)

```

```

39 #define timercmp(tvp, uvp, cmp) \
40     ((tvp)->tv_sec cmp (uvp)->tv_sec || \
41      (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
44 43 /*
45  * Names of the interval timers, and structure
46  * defining a timer setting.
46 */
47 #define ITIMER_REAL    0           // Decrease in real time.
48 #define ITIMER_VIRTUAL 1           // Decrease in the virtual time of the process.
49 #define ITIMER_PROF    2           // Decrease in process virtual time or runtime.
50
51 // 内部の時間構造
52 struct itimerval {
53     struct timeval it_interval;   /* timer interval */
54     struct timeval it_value;     /* current value */
54 };
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

```

## 14.37 times.h

### 14.37.1 Functionality

The times.h header file mainly defines the file access and modification time structure tms. It will be returned by the times() function. Where time\_t is defined in sys/types.h. A function prototype times() is also defined.

### 14.37.2 Code annotation

[1](#) プログラム 14-33 linux/include/sys/times.h

```

2 ifndef TIMES_H
3 define TIMES_H
3
4 include <sys/types.h>      // Type header file. The basic system data types are defined.
5
6 struct tms {
7     time_t tms_utime; // CPU time used by the user.
8     time_t tms_stime; // CPU time used by the system (kernel).

```

---

```

9     time_t tms_cutime; // User CPU time used by the terminated child process.
10    time_t tms_cstime; // The system CPU time used by the terminated child.
11 };
12
13 extern time_t times(struct tms * tp); 14
15 #endif
16

```

---

## 14.38 types.h

### 14.38.1 Functionality

The types.h header file defines the basic data types. All types are defined as the appropriate mathematical type length. In addition, size\_t is an unsigned integer type; off\_t is an extended signed integer type; pid\_t is a signed integer type.

### 14.38.2 Code annotation

プログラム 14-34 linux/include/sys/types.h

---

```

1 #ifndef _SYS_TYPES_H
2 #define _SYS_TYPES_H
3
4 #ifndef _SIZE_T
5 #define _SIZE_T
6 typedef unsigned int size_t;           // Used for the size (length) of the object.
7 #endif
8
9 #ifndef _TIME_T
10 #define _TIME_T
11 typedef long time_t;                 // Used for time (in seconds).
12 #endif
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;                  // Used for process id and process group id.
24 typedef unsigned short uid_t;       // Used for the user id.
25 typedef unsigned char gid_t;        // Used for the group id.
26 typedef unsigned short dev_t;       // Used for the device number.
27 typedef unsigned short ino_t;        // Used for the inode number.
28 typedef unsigned short mode_t;      // Used for some file modes.

```

```

29 typedef unsigned short umode_t;      //  

30 typedef unsigned char nlink_t;    // Used for the file links counting.  

31 typedef int daddr_t;  

32 typedef long off_t;           // Used for the offset in a file.  

33 typedef unsigned char u_char;    // unsigned char.  

34 typedef unsigned short ushort;   // unsigned short.  

35  

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;  // File descriptor set. Each bit represents 1 descriptor.  

41  

42 typedef struct { int quot, rem; } div_t; // Used for DIV operation.  

43 typedef struct { long quot, rem; } ldiv_t; // Used for long DIV operation.  

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  

45 // そして常にNULLを返します。  

46 struct ustat {  

47     daddr_t f_tfree;          // Total free blocks in the system.  

48     ino_t f_tinode;          // Total free inodes.  

49     char f_fname[6];           // File system name.  

50     char f_fpack[6];          // The packed file system name.  

50 };  

51  

52 #endif  

53

```

## 14.39 utsname.h

### 14.39.1 Functionality

utsname.h is the system name structure header file. It defines the utsname structure and the function prototype uname(). This function uses the information in the utsname structure to give information such as the system identifier, version number, and hardware type. In POSIX, the size of the character array should be unspecified, but the data stored in it must be NULL terminated. Therefore, the kernel's utsname structure definition does not meet POSIX requirements (the string array size is defined as 9). In addition, the name utsname is an abbreviation for Unix Timesharing System name.

### 14.39.2 Code annotation

```

1 プログラム 14-35 linux/include/sys/utsname.h
2 #ifndef _SYS_UTSNAME_H
3 #define _SYS_UTSNAME_H
3
4 #include <sys/types.h>    // The basic system data types are defined.
5 #include <sys/param.h>    // Some hardware-related parameter values are given.
5

```

---

```

6 struct utsname {
7     char sysname[9];           // system name.
8     char nodename[_MAXHOSTNAMELEN+1]; // The node name in the network.
9     char release[9];         // release level.
10    char version[9];          // version.
11    char machine[9];          // hardware type.
12 };
13
14 extern int uname(struct utsname * utsbuf); 15
16 #endif
17

```

---

## 14.40 wait.h

### 14.40.1 Functionality

This header file describes the information when the process is waiting, including some symbol constants and wait(), waitpid() function prototype declarations.

### 14.40.2 Code annotation

1 プログラム 14-36 linux/include/sys/wait.h

```

2 ifndef SYS_WAIT_H
3 define SYS_WAIT_H
3
4 #include <sys/types.h>
5
6 define LOW(v)      ((v) & 0377)           // Get the low byte (in octal).
7 define HIGH(v)     (((v) >> 8) & 0377) // Get the high byte.
8
9 /* 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].
#define WNOHANG        1                  // Don't hang and return immediately.

```

```

10
11 #define WUNTRACED      2           // Reports the child status that was stopped.
12
13 // 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。
14 #define WIFEXITED(s) (!((s)&0xFF)) // 子供が正常に終了していれば真。
15 #define WIFSTOPPED(s) (((s)&0xFF)==0x7F) // 子供が停止している場合に真となります。
16 #define WEXITSTATUS(s) (((s)>>8)&0xFF) // The exit status.
17 #define WTERMSIG(s)     ((s)&0x7F)       // Signal that caused process to terminate.
18 #define WCOREDUMP(s)   ((s)&0x80)       // Check if a core dump has been performed.
19 #define WSTOPSIG(s)    (((s)>>8)&0xFF) // Signal that caused process to stop.
20 // 19 #define WIFSIGNALLED(s) (((unsigned int)(s)-1 & 0xFFFF) < 0xFF)
21 #define WIFSIGNALLED(s) (((unsigned int)(s)-1 & 0xFFFF) < 0xFF)
22
23 // 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。  
21 // 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。  
22 pid_t wait(int *stat_loc);  
23 pid_t waitpid(pid_t pid, int *stat_loc, int options);  
23  
24 #endif  
25
```

---

## 14.41 Summary

This chapter describes all the header files used by the kernel. From the next chapter we will introduce the library file code used by the kernel. The code for these library files will be linked into the kernel code when the kernel is compiled.

## 15 Library files (lib)

C言語ライブラリは、再利用可能なプログラムモジュールの集まりで、Linuxカーネルライブラリファイルは、カーネルで使用するためにコンパイルされた、よく使われるいくつかの関数の組み合わせです。リスト15-1のC言語ファイルは、カーネルライブラリファイルのモジュールを構成するプログラムです。主に、プロセスの実行・終了関数、ファイルアクセス操作関数、メモリ確保関数、文字列操作関数などが含まれています。

具体的には、終了関数\_exit()、ファイルクローズ関数close()、ファイルディスクリプターコピー関数dup()、ファイルオープン関数open()、ファイルライト関数write()、プログラム実行関数execve()、メモリ確保関数malloc()、子プロセスの状態待ち関数wait()、セッション作成システムコールsetsid()、include/string.hで実装されているすべての文字列操作関数が実装されている。タイツォさんが書いたmalloc.cのプログラムを除けば、プログラムのサイズは非常に短く、中には1～2行のコードしかないものもあります。それらは基本的にシステムコールを直接呼び出して機能を実現している。

リスト15-1 /linux/lib/ディレクトリのファイル  

Filename	Size	Last Modified Time(GMT)	Description
----------	------	-------------------------	-------------

<a href="#"><u>open.c</u></a>	389 bytes	1991-10-02 14:16:29	
<a href="#"><u>setsid.c</u></a>	128 bytes	1991-10-02 14:16:29	
<a href="#"><u>string.c</u></a>	177 bytes	1991-10-02 14:16:29	
<a href="#"><u>wait.c</u></a>	253 bytes	1991-10-02 14:16:29	

カーネルのコンパイル段階では、カーネルMakefileの関連命令によって、これらのプログラムが.oモジュールにコンパイルされた後、lib.aライブラリにビルドされ、カーネルモジュールにリンクされます。このライブラリは、通常のコンパイル環境で提供される様々なライブラリファイル（gccが提供するlibc.aやlibufc.aなど）とは異なり、主にカーネルの初期化段階のinit/main.cプログラムで、ユーザモードでinit()関数を実行するために使用される関数です。そのため、含まれている関数は少

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
```

このライブラリファイルに関数モジュールdup.oを追加するには、以下のコマンドを実行します。

---

```
ar -rs dup.o
```

---

## 15.1 \_exit.c

### 15.1.1 Functionality

15.1.2 \_exit.cファイルは、プログラムがカーネルのexitシステムコール関数を呼び出すために使用されます。

### 15.1.3 Code annotation

プログラム 15-1 linux/lib/\_exit.c

```

1 /*
2 * linux/lib/_exit.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <unistd.h> Linux標準のヘッダーファイルです。様々なシンボル定数や型が定義されており
// various functions are declared. If '_LIBRARY_' is defined, it also includes the
// system-call number and the inline assembly _syscall0().
7 #define _LIBRARY_
8 #include <unistd.h>
9
/// プログラムが終了（ターミネイト）する機能です。
// ライブラリ関数は、システム割り込みを直接呼び出す 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、コンパイラーの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、さらに重要なことには
// 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 {
    // %0 - eax( NR_exit ) ; %1 - ebx(exit_code) .
12     _asm_( "int $0x80": "a" _NR_exit, "b" (exit_code));
13 }
14

```

---

### 15.1.4 Information

For a description of the system-call interrupt number, see the description in the include/unistd.h file.

## 15.2 close.c

### 15.2.1 Functionality

15.2.2 ファイルクローズ関数close()は、close.cファイルで定義されています。

### 15.2.3 Code annotation

プログラム 15-2 linux/lib/close.c

```

1 /*
2 * linux/lib/close.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <unistd.h> Linux標準のヘッダーファイルです。様々なシンボル定数や型が定義されており
// various functions are declared. If '_LIBRARY_' is defined, it also includes the
// system-call number and the inline assembly _syscall0().
7 #define _LIBRARY_
8 #include <unistd.h>
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

```

## 15.3 ctype.c

### 15.3.1 Functionality

The ctype.c program is used to provide auxiliary array structure data for ctype.h for type determination of characters.

### 15.3.2 Code annotation

プログラム 15-3 linux/lib/ctype.c

```

1 /*
2 * linux/lib/ctype.c
3 *
4 * (C) 1991 Linus Torvalds
5 */
6
// <ctype.h> 文字型ファイルです。文字型変換のためのいくつかのマクロを定義しています。7
#include <ctype.h>
8
9 char _ctmp; // a tem variable for macros that convert characters in the ctype.h file.

```

```

// 以下は、対応する属性を定義する文字属性の配列です。
// を各文字に割り当てることができます。これらの属性タイプ (_Cなど) は、ctype.hで定義されています。
// 文字が制御文字(_C)、大文字(_U)であるかどうかを確認するために使用されます。
10 // 小文字(_L)など。
11 unsigned char ctype[] = {0x00, /* EOF */
12   C, C, C, C, C, C, C, C, /* 0-7 */
13   C, C|S, C|S, C|S, C|S, C|S, C|S, C, C, /* 8-15 */
14   C, C, C, C, C, C, C, C, /* 16-23 */
15   C, C, C, C, C, C, C, C, /* 24-31 */
16   S|SP, P, P, P, P, P, P, P, /* 32-39 */
17   P, P, P, P, P, P, P, P, /* 40-47 */
18   D, D, D, D, D, D, D, D, /* 48-55 */
19   D, D, P, P, P, P, P, P, /* 56-63 */
20   P, U|X, U|X, U|X, U|X, U|X, U|X, U|X, U, /* 64-71 */
21   U, U, U, U, U, U, U, U, /* 72-79 */
22   U, U, U, P, P, P, P, P, /* 80-87 */
23   U, U, U, L|X, L|X, L|X, L|X, L|X, L|X, L, /* 88-95 */
24   L, L, L, L, L, L, L, L, /* 96-103 */
25   L, L, L, L, L, L, L, L, /* 104-111 */
26   L, L, L, P, P, P, P, C, /* 112-119 */
27   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 120-127 */
28   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 128-143 */
29   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 144-159 */
30   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 160-175 */
31   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 176-191 */
32   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 192-207 */
33   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 208-223 */
34   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; /* 224-239 */
35
36

```

## 15.4 dup.c

### 15.4.1 Functionality

The dup.c program includes a function dup() that creates a copy of the file descriptor. After a successful return, the new and original descriptors can be used interchangeably. They share locks, file read and write pointers, and file flags. For example, if the file read/write position pointer is modified by one of the descriptors using lseek(), the file read/write pointer is also changed for the other descriptor. This function uses the smallest unused descriptor to create a new descriptor, but the two descriptors do not share the close-on-exec flag.

### 15.4.2 Code annotation

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
// various functions are declared. If '_LIBRARY_' is defined, it also includes the
// system-call number and the inline assembly _syscall0().
7 #define _LIBRARY_
8 #include <unistd.h>
9
//// ファイル記述子（ハンドル）機能の重複。
// 以下のマクロは、関数プロトタイプ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

```

---

## 15.5 errno.c

### 15.5.1 Functionality

The program only defines an variable errno to store the error number when the function call fails. Please refer to the description in the include/errno.h file.

### 15.5.2 Code annotation

プログラム 15-5 linux/lib/errno.c

```

1 /*
2  *  linux/lib/errno.c
3  *
4  *  (C) 1991 Linus Torvalds
5  */
6
7 int errno;
8

```

---

## 15.6 execve.c

### 15.6.1 Functionality

The execve.c program contains a system-call function that runs the executables.

### 15.6.2 Code annotation

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
// various functions are declared. If '_LIBRARY_' is defined, it also includes the
// system-call number and the inline assembly _syscall0().
7 #define _LIBRARY_
8 #include <unistd.h>
9
//// 子プロセス（他のプログラム）機能をロードして実行します。
// このマクロは、関数int execve(const char * file, char ** argv, char ** envp)に対応しています。
// パラメータ：file - 実行可能なファイル名 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

```

## 15.7 malloc.c

### 15.7.1 Functionality

The malloc.c program mainly includes the memory allocation function malloc(). In order not to be confused with the malloc() function used by the user program, it is called kmalloc() from the kernel version 0.98, and the free\_s() function is renamed to kfree\_s().

なお、アプリケーションが使用する同名のメモリ確保関数は、一般的にGCC環境のlibc.aライブラリのように、開発環境のライブラリファイルに実装されています。開発環境のライブラリ関数は、それ自体がユーザープログラムにリンクされているため、カーネルのget\_free\_page()などの関数を直接使用してメモリ確保関数を実装することはできません。もちろん、libc.aライブラリのメモリ割り当て機能は、プログラムの要求に応じてプロセスデータセグメントの末尾の設定値を動的に調整すればよく、エンド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。このプロセスデータセグメントの終了位置を調整する操作が、ライブラリのメモリ割り当て関数の主目的であり、カーネルのシステムコールであるbrk()が呼び出される。kernel/sys.cプログラムの228行目を参照。つまり、開発環境のライブラリ関数実装のソースコードを見ることができれば、malloc()やcalloc()などのメモリ割り当て関数は、動的なアプリケーションメモリ領域の管理に加えて、カーネルのシステムコールであるbrk()を呼び出しているだけであることがわかります。開発環境ライブラリのメモリ確保関数は、確保したメモリを動的に管理するという点では、ここで紹介した関数と同じ

です。その管理方法も基本的には同じです。

malloc()関数は、割り当てられたメモリを管理するために、バケットの原理を利用しています。基本的な考え方は、要求されたメモリブロックのサイズが異なる場合に、バケットディレクトリ（以下、ディレクトリ）を使用するというものです。例えば、要求されたメモリブロックのサイズが32バイト以下で16バイト以上の場合、バケットディレクトリの2番目の項目に対応するバケットディスクリ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。

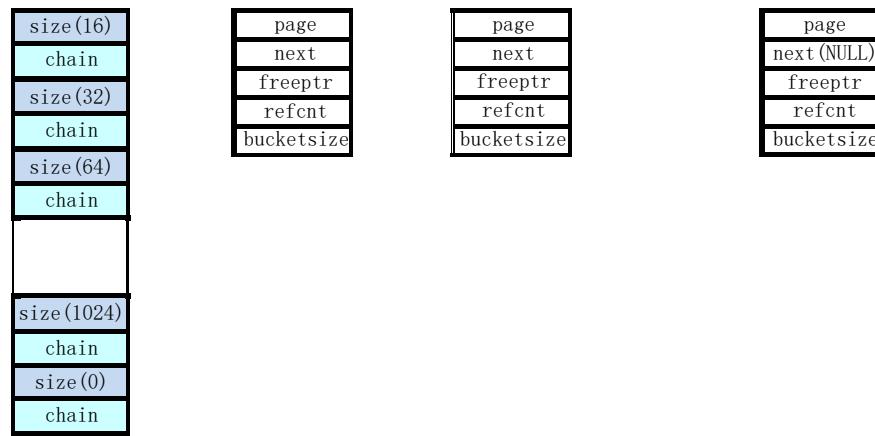


図15-1 バケットの原理を利用したメモリの割り当て管理

The first time you call the malloc() function, you first need to create a free bucket descriptor list for the page, which holds the descriptors that have not been used or have been reclaimed. The structure of the linked list is shown in Figure 15-2, where free\_bucket\_desc is the pointer to the linked list header. Extracting/putting a descriptor from/into the linked list starts from the beginning of the linked list. When a descriptor is fetched, the first descriptor pointed to by the header pointer is fetched; when an idle descriptor is released, it is also placed at the head of the list.

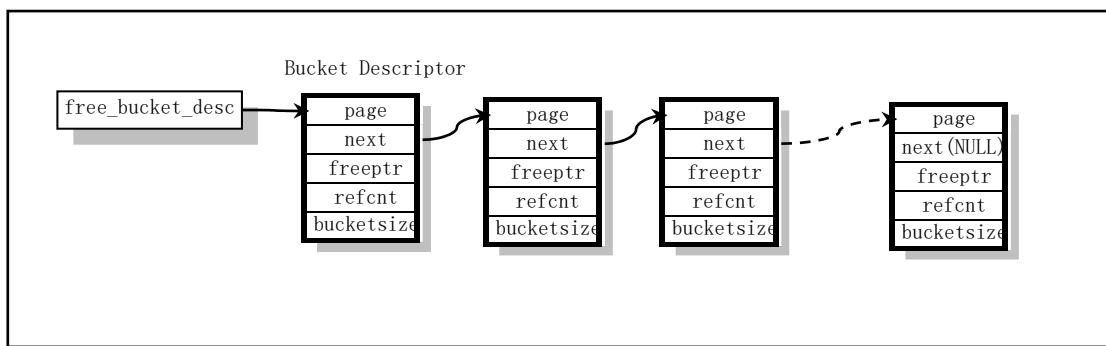


図15-2 フリーバケット記述子のチェーンテーブル構造

During the running of the system, if all bucket descriptors are occupied at a certain time, then free\_bucket\_desc will be NULL. Therefore, without the bucket descriptor being released, the next time you need to use the free bucket descriptor, the program will apply for a page again and create a new free bucket

図のように、その上に記述子のリストが表示されます。

1. malloc()関数の基本的な手順は以下の通りです。
2. First search the directory and look for the descriptor list corresponding to the directory entry that matches the size of the requested memory block. When the memory size of the directory entry is larger than the requested byte size, the corresponding directory entry is found. If the search for the entire directory does not find a suitable directory entry, then the memory block requested by the user is too large.
3. Find a descriptor with free space in the descriptor list corresponding to the directory entry. If the free memory pointer freeptr of a descriptor is not NULL, it means that the corresponding descriptor is found. If we don't find a descriptor with free space, then we need to create a new descriptor. The steps to create a new descriptor are as follows:
  - a. If the idle descriptor list header pointer is still NULL, it means that the malloc() function is called for the first time, or all empty bucket descriptors are used up. In this case, you need to use the function init\_bucket\_desc() to create a list of idle descriptors.
  - b. Then get a descriptor from the header of the idle descriptor chain, initialize the descriptor, make its object reference count 0, the object size is equal that of the directory entry, and apply for a memory page, let the descriptor page pointer points to the memory page, and the free memory pointer of the descriptor also points to the beginning of the page.
  - c. Initialize the page for the memory page according to the size of the object used in this directory entry, and establish a linked list of all objects. That is, each object's head stores a pointer to the next object, and the last object stores a NULL at the beginning.
  - d. Then insert the descriptor into the beginning of the descriptor list for the corresponding directory entry.
4. Copy the free memory pointer freeptr of the descriptor to the memory pointer returned to the user, and then adjust the freeptr to point to the next free object location in the memory page corresponding to the descriptor, and increment the descriptor reference count by one.

free\_s()関数は、ユーザーが解放したメモリブロックを再利用するために使用されます。基本的な方法は、まずメモリブロックのアドレスに応じて対応するページのアドレスを変換し、ディレクトリ内のすべてのディスクリプタを検索して、そのページに対応するディスクリプタを見つけるというものである。解放されたメモリブロックは、freeptrが指すフリーオブジェクトリストにチェーンされ、ディスクリプターのオブジェクト参照カウント値が1つデクリメントされる。このとき、参照カウント値が0になつていれば、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 Code annotation

プログラム 15-7 linux/lib/malloc.c

---

```

1  /*
2  * malloc.c --- a general purpose kernel memory allocator for Linux.
3  *
4  * Written by Theodore Ts'o (tytso@mit.edu), 11/29/91

```

5    \*  
6    *\* This routine is written to be as fast as possible, so that it*

```

7  * can be called from the interrupt level.
8  *
9  * Limitations: maximum size of memory we can allocate using this routine
10 *      is 4k, the size of a page in Linux.
11 *
12 * The general game plan is that each page (called a bucket) will only hold
13 * objects of a given size. When all of the object on a page are released,
14 * the page can be returned to the general free pool. When malloc() is
15 * called, it looks for the smallest bucket size which will fulfill its
16 * request, and allocate a piece of memory from that bucket pool.
17 *
18 * Each bucket has as its control block a bucket descriptor which keeps
19 * track of how many objects are in use on that page, and the free list
20 * for that page. Like the buckets themselves, bucket descriptors are
21 * stored on pages requested from get_free_page(). However, unlike buckets,
22 * pages devoted to bucket descriptor pages are never released back to the
23 * system. Fortunately, a system should probably only need 1 or 2 bucket
24 * descriptor pages, since a page can hold 256 bucket descriptors (which
25 * corresponds to 1 megabyte worth of bucket pages.) If the kernel is using
26 * that much allocated memory, it's probably doing something wrong. :-
27 *
28 * Note: malloc() and free() both call get_free_page() and free_page()
29 *      in sections of code where interrupts are turned off, to allow
30 *      malloc() and free() to be safely called from an interrupt routine.
31 *      (We will probably need this functionality when networking code,
32 *      particularly things like NFS, is added to Linux.) However, this
33 *      presumes that get_free_page() and free_page() are interrupt-level
34 *      safe, which they may not be once paging is added. If this is the
35 *      case, we will need to modify malloc() to keep a few unused pages
36 *      "pre-allocated" so that it can safely draw upon those pages if
37 *      it is called from an interrupt routine.
38 *
39 * Another concern is that get_free_page() should not sleep; if it
40 * does, the code is carefully ordered so as to avoid any race
41 * conditions. The catch is that if malloc() is called re-entrantly,
42 * there is a chance that unnecessary pages will be grabbed from the
43 * system. Except for the pages for the bucket descriptor page, the
44 * extra pages will eventually get released back to the system, though,
45 * so it isn't all that bad.
46 */
47

// <linux/kernel.h> カーネルのヘッダーファイルです。一般的に使用されているいくつかの製品のプロト
タイプ定義が含まれています。
//      used functions of the kernel.
// <linux/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
//      release function prototypes.
// <asm/system.h> システムのヘッダーファイルです。を定義または変更する埋め込みアセンブリマクロ
//      です。
//      descriptors/interrupt gates, etc. is defined.

48 #include <linux/kernel.h>
49 #include <linux/mm.h>
50 #include <asm/system.h>
51

```

```
// バケットディスクリプター構造  
52 struct bucket_desc { /* 16 bytes */
```

```

53     void             *page;          // Memory page pointer.
54     struct bucket_desc *next;        // The next descriptor pointer.
55     void             *freetop;       // A pointer to the free memory.
56     unsigned short    refcnt;        // Reference count.
57     unsigned short    bucket_size;   // The size of the bucket.
58 };
59
// Bucket descriptor directory structure.
60 struct bucket_dir { /* 8 bytes */
61     int              size;          // The size (in bytes) of this bucket.
62     struct bucket_desc *chain;      // Bucket descriptor list pointer.
63 };
64
65 /*
66 * The following is the where we store a pointer to the first bucket
67 * descriptor for a given size.
68 *
69 * If it turns out that the Linux kernel allocates a lot of objects of a
70 * specific size, then we may want to add that specific size to this list,
71 * since that will allow the memory to be allocated more efficiently.
72 * However, since an entire page must be dedicated to each specific size
73 * on this list, some amount of temperance must be exercised here.
74 *
75 * Note that this list *must* be kept in order.
76 */
// バケットのディレクトリリストです。
' { 16,    (struct bucket_desc *) 0},    // A 16-byte memory block.

```

```

78
79     { 32,    (struct bucket_desc *) 0},      // A 32-byte memory block.
80     { 64,    (struct bucket_desc *) 0},
81     { 128,   (struct bucket_desc *) 0},
82     { 256,   (struct bucket_desc *) 0},
83     { 512,   (struct bucket_desc *) 0},
84     { 1024,  (struct bucket_desc *) 0},
85     { 2048,  (struct bucket_desc *) 0},
86     { 4096,  (struct bucket_desc *) 0},      // A 4096-byte memory block.
87     { 0,     (struct bucket_desc *) 0} } ; /* End of list marker */
88
89 /*
90 * This contains a linked list of free bucket descriptor blocks
91 */
92 struct bucket_desc *free_bucket_desc = (struct bucket_desc *) 0; 93
93 /*
94 * This routine initializes a bucket description page.
95 */
96

    //// バケットディスクリプターを初期化します。
    // フリーバケットの記述子リストを作成し、'free_bucket_desc'に最初のフリーバケットを指定させる。
    // ディスクリプタ。
```

99

```

97 static inline void init_bucket_desc() 98 {
100     struct bucket_desc *bdesc, *first;
101     int      i;
```

102

// 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

103

// そのリンクポインタ

```

104     first = bdesc = (struct bucket_desc *) get_free_page();
105     if (!bdesc)
106         panic("Out of memory in init_bucket_desc());
107     for (i = PAGE_SIZE/sizeof(struct bucket_desc); i > 1; i--) {
108         bdesc->next = bdesc+1;
109         bdesc++;
110     }
111     /*
112      * This is done last, to avoid race conditions in case
113      * get_free_page() sleeps and this routine gets called again....
114      */

```

113

// 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。

```

114     bdesc->next = free_bucket_desc;
115     free_bucket_desc = first;
115 }
116
117     ///// メモリの割り当て機能。
118     // パラメータ : len - 要求されたメモリブロックのサイズ。
119     // 戻り値: 割り当てられたメモリへのポインタ。失敗した場合は NULL を返します。

```

19

117 void \*malloc(unsigned int len)

118 {

```

120     struct bucket_dir      *bdir;
121     struct bucket_desc    *bdesc;
122     void                  *retval;
122
123     /*
124      * First we search the bucket_dir to find the right bucket change
125      * for this request.
126      */

```

127

// 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。

// 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を探します。

```

128     for (bdir = bucket_dir; bdir->size; bdir++)
129         if (bdir->size >= len)
130             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

131

// 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ます。

```

132     if (!bdir->size) {
131         printk("malloc called with impossibly large argument (%d) |n",
132                 len);
133         panic("malloc: bad arg");
134     }
135     /*

```

```
136     * Now we search for a bucket descriptor which has free space
137     */
138     cli();      /* Avoid race conditions */
// 対応するバケットディレクトリエントリの記述子リストを検索して、バケットを見つける
// 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                  if (bdesc->freeptr)  
 140                      break;  
 141                  /\*  
 142                  \* If we didn't find a bucket with free space, then we'll  
 143                  \* allocate a new one.  
 144                  \*/  
 145                  if (!bdesc) {  
 146                      char                    \*cp;  
 147                      int                     i;  
 148                      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。  
 151                  if (!free\_bucket\_desc)  
 152                      init\_bucket\_desc();  
 // 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；フリーメモリポインタ  
 // この時点ではすべてがアイドル状態なので、//もページの最初を指しています。  
 154                  bdesc = free\_bucket\_desc;

```
155     free_bucket_desc = bdesc->next;
156     bdesc->refcnt = 0;
157     bdesc->bucket_size = bdir->size;
158     bdesc->page = bdesc->freetr = (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
// of the last object is set to 0 (NULL).
```

```

157         if (!cp)
158             panic("Out of memory in kernel malloc()");
159             /* Set up the chain of free objects */
160             for (i=PAGE_SIZE/bdir->size; i > 1; i--) {
161                 *((char **) cp) = cp + bdir->size;
162                 cp += bdir->size;
163             }
164             *((char **) cp) = 0;
// そして、バケットディスクリプタの次のディスクリプタポインタフィールドを指して、ディスクリプタの
// もともとバケットディレクトリのエントリポインタが指していたもので、バケットディレクトリのチ
// エイン
// バケットの記述子を指している、つまり記述子に挿入されている
// チェインヘッダーです。
166             bdesc->next = bdir->chain;      /* OK, link it in! */
167             bdir->chain = bdesc;
168         }
// 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
// 解放されたメモリオブジェクトへのポインタを返します。
169         retval = (void *) bdesc->freeptr;

```

```
170     bdesc->freeptr = *((void **) retval);
171     bdesc->refcnt++;
172     sti(); /* OK, we're safe again */
173     return(retval);
173 }
174
175 /*
176 * Here is the free routine. If you know the size of the object that you
177 * are freeing, then free_s() will use that information to speed up the
178 * search for the bucket descriptor.
179 */
180 * We will #define a macro so that "free(x)" is becomes "free_s(x, 0)"
181 */

/// バケットオブジェクトをリリースします。
void free_s(void *obj, int size)
```

```

182
183 {
184     void             *page;
185     struct bucket_dir    *bdir;
186     struct bucket_desc   *bdesc, *prev;
187
188     /* Calculate what page this object lives in */
189     page = (void *) ((unsigned long) obj & 0xfffff000);
190     /* Now search the buckets looking for that page */
191     for (bdir = bucket_dir; bdir->size; bdir++) {
192         prev = 0;
193         /* If size is zero then this conditional is always false */
194         if (bdir->size < size)
195             continue;
196
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、そのよ
198         // 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場合
201         // は
202         // そうすると、ディスクリプターのポインター'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ーが、そのディレクトリエン
204         // トを見つけられなかった場合
205         // 指定したページにアクセスすると、エラーメッセージが表示され、コンピュータがクラッシュします。
206
207         for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
208             if (bdesc->page == page)
209                 goto found;
210             prev = bdesc;
211         }
212     }
213     panic("Bad address passed to kernel free_s()");
214 found:
215     // 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
216     // メモリロックは、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ウ
217     // ントが
218     // そのディスクリプターの // を1つずつ減らしていきます。
219
220     cli(); /* To avoid race conditions */
221     *((void **)obj) = bdesc->freeptr;
222     bdesc->freeptr = obj;
223     bdesc->refcnt--;
224
225     // 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
226     // バケットの記述子。

```

```

208     if (bdesc->refcnt == 0) {
209         /*
210          * We need to make sure that prev is still accurate. It
211          * may not be, if someone rudely interrupted us....
212          */
213
214     // '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
215     // 現在のディスクリプターを再検索します。
216     if ((prev && (prev->next != bdesc)) ||
217         (!prev && (bdir->chain != bdesc)))
218         for (prev = bdir->chain; prev; prev = prev->next)
219             if (prev->next == bdesc)
220                 break;
221
222     // 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。
223     // 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し
224     // ます。
225     // 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
226     // 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。
227     // 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
228     // 次の記述子を指示する「チェーン」。
229     if (prev)
230         prev->next = bdesc->next;
231     else {
232         if (bdir->chain != bdesc)
233             panic("malloc bucket chains corrupted");
234         bdir->chain = bdesc->next;
235     }
236
237     // 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
238     // がアイドルディスクリプターリストの先頭に挿入されます。
239     free_page((unsigned long) bdesc->page);
240     bdesc->next = free_bucket_desc;
241     free_bucket_desc = bdesc;
242
243     }
244     sti();           // Enable interrupt and return.
245     return;
246 }
247
248
249
250
251
252
253

```

## 15.8 open.c

### 15.8.1 Functionality

The `open()` library function in the `open.c` file is used to open a file with the specified file name. When `open()` is called successfully, the file descriptor of the file is returned. This call creates a new open file and is not shared with any other process. When the `exec()` function is executed, the new file descriptor will remain open at all times. The file's read and write pointer is set at the beginning of the file.

関数のパラメータ'flag'には、O\_RDONLY, O\_WRONLY, O\_RDWRのいずれかを指定することができます。これらはそれぞれ、ファイルが読み取り専用オープン、書き込み専用オープン、読み書き可能オープンであることを意味し、他のいくつかのフラグと一緒に使用することができます。fs/open.cプログラムのsys\_open()関数の実装(171行目)も参照してください。

## 15.8.2 Code annotation

プログラム 15-8 linux/lib/open.c

```

1 /*
2 *  linux/lib/open.c
3 *
4 *  (C) 1991 Linus Torvalds
5 */
6
// <unistd.h> Linux標準のヘッダーファイルです。様々なシンボル定数や型が定義されており
// various functions are declared. If '_LIBRARY_' is defined, it also includes the
// system-call number and the inline assembly _syscall0().
// <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。
// of macros. It mainly describes one type (va_list) and three macros (va_start, va_arg and
// va_end) for the vsprintf, vprintf, and vfprintf functions.
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 {
14     register int res;
15     va_list arg;
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) 。
// %2 - ebx (ファイル名); %3 - ecx (ファイルオープンフラグ); %4 - edx (ファイルモードの追従)。
16     va_start(arg, flag);
17     __asm__ ("int $0x80"
18             : "=a" (res)
19             : "" (_NR_open), "b" (filename), "c" (flag),
20             "d" (va_arg(arg, int)));
21
// 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、戻り値が以下の
// 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を返します。
22     if (res>=0)
23         return res;
23     errno = -res;
24     return -1;
25 }
26

```

## 15.9 setsid.c

### 15.9.1 Functionality

The setsid.c program includes a setsid() system call function. This function is used to create a new session if the calling process is not a leader of a group. The calling process will become the leader of the new session, the group leader of the new process group, and there is no controlling terminal. The group ID and session ID of the calling process are set to the PID of the process. The calling process will become the only process in the new process group and the new session.

### 15.9.2 Code annotation

プログラム 15-9 linux/libsetsid.c

---

```

1 /*
2 * linux/libsetsid.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
// various functions are declared. If '_LIBRARY' is defined, it also includes the
// system-call number and the inline assembly _syscall0().
7 #define _LIBRARY
8 #include <unistd.h>
9
//// セッションを作成し、プロセスグループ番号を設定します。
// 以下のシステムコールマクロは、pid_t setsid()という関数に対応しています。
10 //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。
11 _syscall0(pid_t, setsid)
11

```

---

## 15.10 string.c

### 15.10.1 Functionality

All string manipulation functions already exist in the string.h header file, but appear as inline code. Here, we give the implementation code that contains the string function in string.c by first declaring the 'extern' and 'inline' prefixes to be empty, and then including the string.h header file. See the instructions before the include/string.h header file.

### 15.10.2 Code annotation

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
9#endif
10
12 #define extern
13 #define inline
14 #define _LIBRARY_
15 #include <string.h>
15

```

---

## 15.11 wait.c

### 15.11.1 Functionality

The wait.c program includes the functions `waitpid()` and `wait()`. These two functions allow the process to get state information for one of its child processes. Various options allow you to get child process status information that has been terminated or stopped. If there are status information for two or more child processes, the order of the reports is not specified.

`wait()`は、子プロセスの1つが終了（ターミネイト）するか、プロセスの終了を要求するシグナルを受け取るか、シグナルハンドラを呼び出す必要があるまで、現在のプロセスを中断します。

`waitpid()`は、`pid`で指定された子プロセスが終了（ターミネイト）するか、プロセスの終了を要求するシグナルを受け取るか、シグナルハンドラを呼び出す必要があるまで、現在のプロセスを一時停止します。

`pid=-1`、`options=0`の場合、`waitpid()`は`wait()`関数と同じ動作をしますが、そうでない場合は`pid`と`options`のパラメータに応じて動作が変わります（`kernel/exit.c`, 370参照）。

### 15.11.2 Code annotation

プログラム 15-11 linux/lib/wait.c

```

1 /*
2  *  linux/lib/wait.c
3  *
4  *  (C) 1991 Linus Torvalds
5  */
6
7 // <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
8 // various functions are declared. If '_LIBRARY_' is defined, it also includes the
9 // system-call number and the inline assembly _syscall0().
10 // <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
11 // constant symbols.

```

```
7 #define _LIBRARY
8 #include <unistd.h>
9 #include <sys/wait.h>
```

```

10     /// プロセスが終了するのを待ちます。
// pid_t waitpid(pid_t pid, int * wait_stat, int options)
// 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
// wait_stat - 状態情報の保存に使用されます。
// オプション - WNOHANG or WUNTRACED or 0.
11 _syscall3(pid_t,waitpid,pid_t,pid,int *,wait_stat,int,options)
12     /// wait() システムコールです。waitpid()関数を直接呼び出します。
13 pid_t wait(int * wait_stat) 14 {
15         return waitpid(-1,wait_stat,0);
16 }
17

```

---

## 15.12 write.c

### 15.12.1 Functionality

The write.c program includes a write function write() to the file descriptor. This function writes the data of the 'count' byte to the file 'buf' of the file specified by the file descriptor.

### 15.12.2 Code annotation

プログラム 15-12 linux/lib/write.c

```

1 /*
2  *  linux/lib/write.c
3  *
4  *  (C) 1991 Linus Torvalds
5 */
6
7 // <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
8 // various functions are declared. If '_LIBRARY' is defined, it also includes the
9 // system-call number and the inline assembly _syscall0().
7 #define _LIBRARY
8 #include <unistd.h>
9
10     /// ファイルの書き込みを行います。
11     // このマクロは、以下の関数に対応しています。
12     // 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 - 書き込みバイト
13     // 数。
14     // 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
15     // 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 Summary

This chapter describes several library function files used by several tasks that the kernel runs in user mode at initialization time. These library functions are implemented in exactly the same way as the generic library functions used in the development environment.

次の章では、カーネルコードツリーに含まれるtools/build.cというツールについて説明します。このツールは、すべてのカーネルモジュールを組み合わせてカーネルイメージファイルを生成するためのものです。



# 16 Building Kernel (tools)

Linuxカーネルソースコードの「tools」ディレクトリには、カーネルのディスクイメージファイルを生成するユーティリティプログラムbuild.cが含まれています。このプログラムは個別に実行ファイルにコンパイルされ、Makefileの中で呼び出されて、カーネルのコンパイル済みモジュールをすべて接続して作業用イメージファイルImageにマージするために使用されます。Makefileの内容によると、Makeプログラムはまず8086アセンブリを使って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、ビルトツールを使って、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。

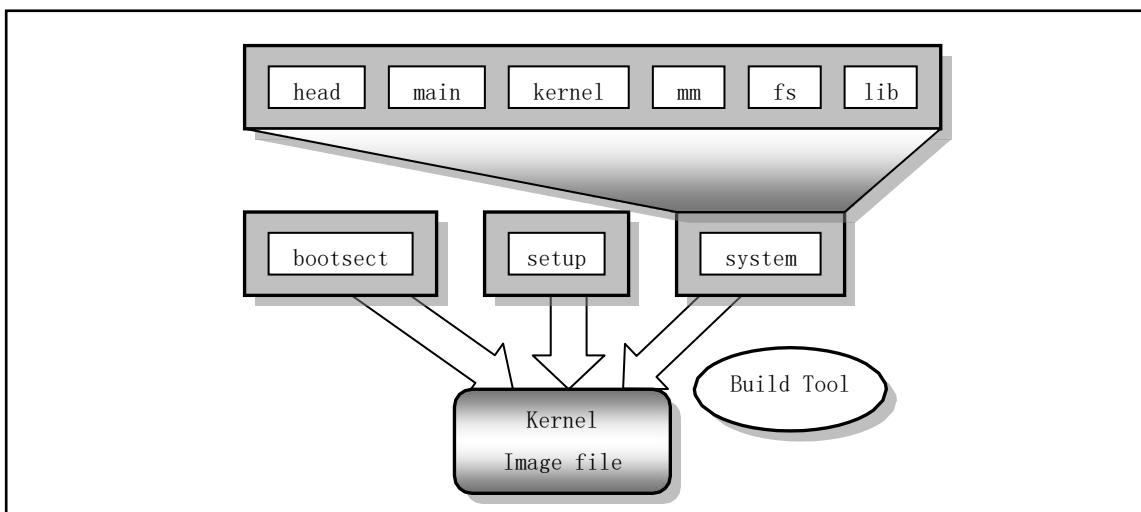


図16-1 カーネルのコンパイル・リンク・結合構造

## 16.1 build.c

### 16.1.1 Functionality

On the linux/Makefile line 42--44, the command line form for executing the build program is as follows:

---

```
tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) $(SWAP_DEV) > Image
```

---

The build program uses five parameters, namely the bootsect, setup, system, the optional root fs device name ROOT\_DEV, and the optional swap device SWAP\_DEV. The bootsect and setup modules are compiled by as86,

they have the MINIX executable file format (see the description of the program list), and the system module is linked by modules compiled from other source code, with GNU a.out executable file format. The main job of

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。

- The first sector stores the bootsect code, which is exactly 512 bytes long;
- The 4 sectors (2 - 5 sectors) starting from the 2nd sector store the setup code, and the size is no more than 4 sectors;
- The system module is stored starting from the sixth sector, and its length does not exceed the size (128KB) defined on line 37 of build.c.

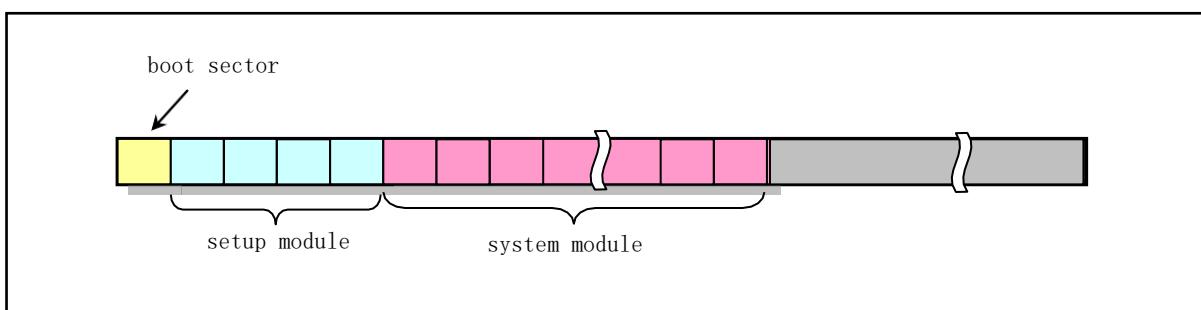


図 16-2 ディスク上の Linux カーネルのフォーマット

### 16.1.2 Code annotation

```

3 *
4 * (C) 1991 Linus Torvalds
5 */
6
7 */

```

```

1 /*
2 * linux/tools/build.c

```

## プログラム 16-1 linux/tools/build.c

```
8 * This file builds a disk-image from three different files:  
9 *  
10 * - bootsect: max 510 bytes of 8086 machine code, loads the rest  
11 * - setup: max 4 sectors of 8086 machine code, sets up system param
```

```

12 * - system: 80386 code for actual system
13 *
14 * It does some checking that all files are of the correct type, and
15 * just writes the result to stdout, removing headers and padding to
16 * the right amount. It also writes some system data to stderr.
17 */
18
19 /*
20 * Changes by tytso to allow root device specification
21 *
22 * Added swap-device specification: Linux 20.12.91
23 */
24
25 #include <stdio.h>      /* fprintf */
26 #include <string.h>
27 #include <stdlib.h>      /* contains exit */
28 #include <sys/types.h>   /* unistd.h needs this */
29 #include <sys/stat.h>    // file state information structure
30 #include <linux/fs.h>
31 #include <unistd.h>      /* contains read/write */
32 #include <fcntl.h>        // file operation mode constants
33
34 #define MINIX_HEADER 32 // MINIXオブジェクトファイルのヘッダサイズは32バイトで
す。35 #define GCC_HEADER 1024 // GCCのヘッダー情報のサイズは1024バイトです。36
37 #define SYS_SIZE 0x3000 // システムモジュールの最大サイズ(SYS_SIZE*16=128KB)です。38
// デフォルトでは、Linuxのルートfsデバイスは、2番目のハードディスクの第1パーティション（デバイ
ス
// 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の
39 // をMINIXのシステムディスクとして、2台目のハードディスクをLinuxのルートファイルシステムのデ
ィスクとして使用します。
40 #define DEFAULT_MAJOR_ROOT 3      // major device number - 3, hard disk.
41 #define DEFAULT_MINOR_ROOT 6     // minor device number - 6, 1st partition of the 2nd disk.
42
43 #define DEFAULT_MAJOR_SWAP 0    // swap device number.
44
45 /* max nr of sectors of setup: don't change unless you also change
46 * bootsect etc */
47 #define SETUP_SECTS 4           // The maximum size of setup is 4 sectors (2KB).
48
49 #define STRINGIFY(x) #x        // Convert x to string type for use in an error display.
50
/// エラーメッセージを表示して、プログラムを終了します。
51 void die(char * str) 52
{
54     fprintf(stderr, "%s\n", str);
55     exit(1);
55 }
56
/// プログラムの使用状況を表示して終了します。
57 void usage(void)

```

```
58 {  
59     die("Usage: build bootsect setup system [rootdev] [> image]");
```

```

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 int main(int argc, char ** argv)
68
69
70 ///////////////////////////////////////////////////////////////////
71 // (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
72 // 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
73 // 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
74 // プログラムの使用状況を表示して終了します。
75 // 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
76 // が "FLOPPY"でない場合は、デバイスファイルのステータス情報を取得し、メジャー、マイナー
77 // 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。
78 // 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を0
79 // に設定する
80 // ブートデバイス。
81 if ((argc < 4) || (argc > 6))
82     usage();
83 if (argc > 4) {
84     if (strcmp(argv[4], "FLOPPY")) {
85         if (stat(argv[4], &sb)) {
86             perror(argv[4]);
87         }
88     }
89 }
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
918
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1090
1091
1092
1093
1094
1095
1096
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1188
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1288
1289
1290
1291
1292
1293
1294
1295
1296
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1388
1389
1390
1391
1392
1393
1394
1395
1396
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1488
1489
1490
1491
1492
1493
1494
1495
1496
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1588
1589
1590
1591
1592
1593
1594
1595
1596
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1688
1689
1690
1691
1692
1693
1694
1695
1696
1696
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1790
1791
1792
1793
1794
1795
1796
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1890
1891
1892
1893
1894
1895
1896
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1990
1991
1992
1993
1994
1995
1996
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2090
2091
2092
2093
2094
2095
2096
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2190
2191
2192
2193
2194
2195
2196
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2218
2219
2220
2221
2222
2223
2224
2225
```

```

76             die( "Couldn't stat root device. " ); 77
77         }
78         major_root = MAJOR( sb.st_rdev ); // Get the device number.
79         minor_root = MINOR( sb.st_rdev );
80     } else {
81         major_root = 0;
82         minor_root = 0;
83     }
// 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
// デフォルトのルートデバイス番号。
84
85     } else {
86         major_root = DEFAULT\_MAJOR\_ROOT;
87         minor_root = DEFAULT\_MINOR\_ROOT; 87
}
// コマンドラインに6つのパラメータがある場合、最後のパラメータが示しているのが
// スイッチングデバイスが "NONE" ではない場合、デバイスファイルのステータス情報を取得して
// 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、最
// 後のパラメータ
// that the swap device is the current boot device.

```

```
88     if (argc == 6) {
89         if (strcmp(argv[5], "NONE")) {
90             if (stat(argv[5], &sb)) {
91                 perror(argv[5]);
```

```

92             die("Couldn't stat root device.");
93         }
94         major_swap = MAJOR(sb.st_rdev);
95         minor_swap = MINOR(sb.st_rdev);
96     } else {
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
オルトであるスワップ
// } else {
// デ
// バ
// イ
// ス
// 番
// 号
//
100
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
// カーネルイメージファイル。
104
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                 major_root);
111         die("Bad root device --- major #");
112     }
113     if (major_swap && major_swap != 3) {
114         fprintf(stderr, "Illegal swap device (major = %d)\n",
115                 major_swap);
116         die("Bad root device --- major #");
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
// 以下のリスト）をバッファbufに格納します。
118     for (i=0;i<sizeof buf; i++) buf[i]=0;
119     if ((id=open(argv[1], O_RDONLY, 0))<0)
120         die("Unable to open 'boot'");
121     if (read(id, buf, MINIX_HEADER) != MINIX_HEADER)

```

---

122                    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ヘッドマジック値 a\_magic; 0x10 - 実行可能フラグ a\_flag;  
// 0x04 - マシンタイプ a\_cpu, Intel 8086 のマシンコード。  
// 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。  
// 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
126     if (((long *) buf)[3] != 0)
127         die("Illegal data segment in 'boot'");
128     if (((long *) buf)[4] != 0)
129         die("Illegal bss in 'boot'");
130     if (((long *) buf)[5] != 0)
131         die("Non-Minix header of 'boot'");
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。ブートセクトファイルには
// flag 0xAA55.
```

```

134     i=read(id, buf, sizeof buf);
135     fprintf(stderr, "Boot sector %d bytes. |n", i);
136     if (i != 512)
137         die("Boot block must be exactly 512bytes");
138     if ((*(unsigned short *) (buf+510)) != 0xAA55)
139         die("Boot block hasn't got boot flag (0xAA55)");
140 // 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。
// 507、508、509のオフセットにルートデバイス番号を格納しています。
141     buf[506] = (char) minor_swap;
142     buf[507] = (char) major_swap;
143     buf[508] = (char) minor_root;
144     buf[509] = (char) major_root;
145 // 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
//      i=write(1, buf, 512);
I
m
ag
e
の
最
初
の
51
2
バ
イ
ト
。
144
145     if (i!=512)
146         die("Write call failed");
147     close (id);
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。

---

// 上記のような方法です。  
。

```

149
150     die( "Unable to open 'setup' ");
151     if (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)           // header size (32 bytes)
156         die( "Non-Minix header of 'setup' ");
157     if (((long *) buf)[3]!=0)                      // data size field a_data
158         die("Illegal data segment in 'setup'");
159     if (((long *) buf)[4]!=0)                      // a_bss field.

```

```

160         die("Illegal bss in 'setup'");
161     if (((long *) buf)[5] != 0)           // a_entry point.
162         die("Non-Minix header of 'setup'");
163     if (((long *) buf)[7] != 0)
164         die("Illegal symbol table in 'setup'");
// ファイル内の後続の実際のコードやデータは、上記の条件で読み込まれます。
// のチェックは正しく、端末の標準出力に書き込まれます。それと同時に、長さ
書き込みの//をカウントし、動作終了後に設定ファイルを閉じます。その後、チェック
// 書き込み操作のコードとデータのサイズが、(SETUP_SECTS
// /* 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。
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し
てください。すべてが問題なければ
165 // セットアップの実際の長さの値が表示されます。
166     for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
167         if (write(1,buf,c)!=c)
168             die("Write call failed");
169         close (id);                      // close setup file.
170         if (i > SETUP_SECTS*512)
171             die("Setup exceeds " STRINGIFY(SETUP_SECTS)
172                 " sectors - rewrite build/boot/setup");
173         fprintf(stderr, "Setup is %d bytes. |n",i);
// バッファbufをクリアした後、実際の書き込み設定長と
// setup with NULL characters to 4 *512 bytes.

```

```
173     for (c=0 ; c<sizeof(buf) ; c++)
174         buf[c] = '|';
175     while (i<SETUP_SECTS*512) {
176         c = SETUP_SECTS*512-i;
```

```

177         if (c > sizeof(buf))
178             c = sizeof(buf);
179         if (write(1,buf,c) != c)
180             die("Write call failed");
181         i += c;
182     }
183
// (4) Start processing the system module file below. This file is compiled with gas/gcc and
// therefore has the GNU a.out object file format.
// First open the system module file in read-only mode, and read the a.out format header structure
// information (1KB size). After confirming that system is a valid a.out format file, write all
// subsequent data of the file to the standard output (Image file) and close the file. Then show
// the size of the system. If the system code and data size exceed the SYS_SIZE section (128KB
// bytes), an error message is displayed and exits. If there is no error, it returns 0, indicating
// normal exit.
184     if ((id=open(argv[3],O_RDONLY,0))<0)
185         die("Unable to open 'system'");
186     if (read(id,buf,GCC_HEADER) != GCC_HEADER)
187         die("Unable to read header of 'system'");
188     if (((long *) buf)[5] != 0)           // entry location should be 0.
189         die("Non-GCC header of 'system'");
190     for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
191         if (write(1,buf,c)!=c)
192             die("Write call failed");
193     close(id);
194     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

```

---

## 16.1.3 Information

### 16.1.3.1 MINIX module and executable header data structure

The header structure of the modules and executables generated by MINIX's compiler and linker is as follows:

---

構造体exec {。

```

unsigned char a_magic[2];           // Magic number, should be 0x0301.
unsigned char a_flags;             // Flags (see below).
unsigned char a_cpu;               // Machine CPU identifier.
unsigned char a_hdrlen;            // Reserved header size, 32 or 48 bytes.
unsigned char a_unused;             // Reserved.
unsigned short a_version;          // Version information (not used currently).
long        a_text;                // Code section size (in bytes).
long        a_data;                // Data section size (in bytes).
long        a_bss;                 // Stack size (in bytes).
long        a_entry;                // Execute entry point.
long        a_total;                // Total amount of memory allocated.
long        a_syms;                // Symbol table size.

// ヘッダーサイズが32バイトの場合、構造体はここで終了します。
long        a_trsize;               // Code section relocation table size.
long        a_drsize;                // Data section relocation table size.
long        a_tbase;                 // Code section relocation base address.
long        a_dbase;                 // Data section relocation base address.
};


```

---

Among them, the flag field a\_flags in the MINIX execution file header is defined as:

---

A_UZP	0x01	// Unmapped 0 pages (pages).
A_PAL	0x02	// Adjusted at the page boundary.
A_NSYM	0x04	// New type symbol table.
A_EXEC	0x10	// Executable file.
A_SEP	0x20	// The code and data are separate (I and D are independent).

---

CPUの識別番号フィールドa\_cpuには

---

A_NONE	0x00	// Unknown.
A_I8086	0x04	// Intel i8086/8088.
A_M68K	0x0B	// Motorola m68000.
A_NS16K	0x0C	// National Semiconductor Co. 16032.
A_I80386	0x10	// Intel i80386.
A_SPARC	0x17	// Sun SPARC.

---

The above MINIX execution header structure exec is similar to the a.out format header structure used by the Linux 0.12 system. See the linux/include/a.out.h file for the header structure and related information of the Linux a.out format executable file.

## 16.2 Summary

この章では、カーネルソースツリーにあるビルドツール build.c について説明します。このプログラムは、主にカーネルモジュールを修正・結合して、起動可能なカーネルブートイメージファイルを生成するために使用します。これまでに、カーネル内のすべてのソースコードファイルについて、詳細な説明とコメントを終えています。

カーネルの動作メカニズムをより深く理解するために、次の章では、Bochsシミュレーションプログラムを使って、Linux 0.12オペレーティングシステムをセットアップして動作させるテスト方法を詳しく説明し、いくつかの実験について具体的なテスト手順を示します。

# 17 Experimental Environment Settings and 使用方法

本章では、Linux 0.1x カーネルの動作原理を学ぶために、PC シミュレーション・ソフトウェアを使用して、実際のコンピュータ上で Linux 0.1x システムを動作させる実験方法を紹介します。具体的には、カーネルのコンパイルプロセス、シミュレーション環境でのファイルアクセスとコピー、起動ディスクとルートファイルシステムの作成方法、Linux 0.1xシステムの使用方法などです。最後に、既存のRedHatシステム（gcc 3.x）でのコンパイルプロセスを成功させ、対応するカーネル・イメージ・ファイルを作成するために、カーネル・コードに少数の構文変更を加える方法も紹介しました。

- ♦ 実験を始める前に、まず便利なツールを用意する必要があります。Windowsプラットフォームで実験を行う場合は、以下のソフトウェアを準備する必要があります。

- ♦ Bochs 2.6.x open source PC simulation package (<https://sourceforge.net/projects-bochs/>);
- ♦ Notepad++ Editor. Used to edit binary files (<https://sourceforge.net/projects-notepad-plus/>);
- ♦ HxD hex editor. Used to edit binary or disk files, even in-memory data (<https://mh-nexus.de/en/hxd/>);
- ♦ WinImage DOS format floppy image file editing software (<http://www.winimage.com/>).

最新のLinuxシステム（Redhat、Ubuntuなど）で実験を行う場合は、通常、Bochsパッケージをインストールするだけでシミュレーションを行うことができます。実験におけるその他の操作は、Linuxシステムの一般的なツールを使って行うことができます。

Linux 0.1xのシステムを動かすには、PCのエミュレーションソフトを使うのが一番です。現在、世界で人気のあるPCシミュレーションソフトは4つあります。VMware社のソフトウェア「VMware Workstation」、Oracle社のオープンソースソフトウェア「VirtualBox」、Microsoft社の「Virtual PC」、そしてオープンソースソフトウェア「Bochs」（発音は「ボックス」と同じ）です。これらは、PCの操作を完全に仮想化し、シミュレートすることができる。これらのタイプのソフトウェアは、インテルx86ハードウェア環境を仮想化またはエミュレートすることができ、ソフトウェアが動作しているプラットフォーム上で、他のさまざまな「お客様」のオペレーティングシステムを実行することができます。

使用範囲や操作性の面で、4つのシミュレーションソフトウェアにはまだいくつかの違いがあります。Bochsは、x86CPU搭載PCのハードウェア環境（CPU命令）とその周辺機器をすべてソフトウェアでシミュレーションしているため、多くのOSやアーキテクチャの異なるプラットフォームへの移植が容易である。主にソフトウェアのシミュレーション技術を利用しているため、他のシミュレー

ションソフトウェアに比べて実行性能や速度が大幅に低下します。Virtual PCソフトウェアの性能は、BochsとVMware Workstation（またはVirtualBox）の間です。ほとんどのx86命令をエミュレートし、その他の部分は仮想技術を使って実装しています。VMware WorkstationやVirtualBoxは一部のI/O機能をシミュレートしているだけで、それ以外の部分はx86リアルタイムハードウェア上で直接実行されます。つまり、ゲストOSがある命令を実行する必要があるとき、VMwareやVirtualBoxはその命令をシミュレーションで実行するのではなく、その命令を実際のシステムのハードウェアに直接「渡す」だけなのです。ですから、VMwareやVirtualBoxは、これらのソフトウェアの中では速度と性能の面で最高のものです。もちろん、Qemuのような他のシミュレーションソフトでも、高いシミュレーション性能を発揮することができます。

アプリケーションの観点から、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トウェアを開発する必要がある場合（オペレーティングシステムの開発やデバッグ、コンパイラシステムの開発など）は

Bochs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 Simulation Software

Bochsは、Intel 80X86コンピュータを完全にエミュレートするプログラムです。インテルの80386、486、Pentium以上の新しいCPUプロセッサをエミュレートするように設定することができます。実行段階を通じて、Bochsは、標準的なPC周辺機器のすべてのデバイス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・ウィンドウでキーが押されると、キーストローク・イベントがキーボード・デバイス・プロセッシング・モジュールに送られます。シミュレーションされているマシンが、シミュレーションされているハードディスクからの読み取り操作を行う必要がある場合、Bochsはホスト上のハードディスク・イメージ・ファイルに対して読み取り操作を行います。

Bochsソフトウェアのインストールはとても便利です。<http://bochs.sourceforge.net> から直接Bochsのインストールパッケージをダウンロードすることができます。お使いのコンピュータのOSがWindowsの場合、インストール方法は通常のソフトウェアと全く同じです。Bochsソフトウェアがインストールされると、C:\Program Files\Bochs-2.6\というディレクトリが生成されます（バージョンによって番号が異なります）。お使いのシステムがRedHatなどのLinuxであれば、BochsのRPMパッケージをダウンロードして、以下のようにインストールすることができます。

```
user$ su  
パスワード  
root# rpm -i bochs-2.6.i386.rpm  
root# exit  
ユーザー$ _
```

You need root privileges when installing Bochs, otherwise you will have to recompile the Bochs system in your own directory. In addition, Bochs needs to run in the X11 environment, so you must have the X Window System installed on your Linux system to use Bochs. After installing Bochs, it is recommended to test and familiarize yourself with the Bochs system using the Linux dlx demo system included with the Bochs package. You can also download some of the other Linux image files from the Bochs website to do some experimentation. We recommend downloading the SLS Linux emulation system package (sls-0.99pl.tar.bz2) on the Bochs website as an auxiliary platform for creating Linux 0.1x emulation systems. When making new hard disk image files, we can use these systems to partition and format the hard disk image files. The image file for this SLS

Linux システムは、oldlinux.org 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>。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。

17.1.1 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。

### 17.1.2 Setting up the Bochs system

BochsでOSを動かすためには、以下のリソースや情報のうち、少なくともいくつかが必要です。

- 
- Bochs and bochsdbg executable files;
  - BIOS image files (commonly referred to as 'BIOS-bochs-latest');
  - VGABIOS image files (eg 'VGABIOS-lgpl-latest');
  - at least one boot image file (floppy, hard disk or CDROM image) file).
- 

17.1.3 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。

### 17.1.4 \*.bxrc Configuration File

1. Bochsは、設定ファイルの情報をもとに、使用したディスクイメージファイル、動作環境の周辺機器の設定など、仮想マシンの設定情報を見つけ出します。模擬システムごとに、対応する設定ファイルを設定する必要があります。インストールされているBochsシステムが2.1以降であれば、拡張子が「.bxrc」の設定ファイルを自動的に認識し、ファイルのアイコンをダブルクリックすると、Bochsシステムが自動的に起動します。例えば、設定ファイル名を「bochsrc-0.12.bxrc」とします。Bochsインストールのホームディレクトリ（通常はC:\Program Files\Bochs-2.6\）には、「bochsrc-sample.txt」というテンプレートの設定ファイルがあり、利用可能なすべてのパラメータが詳細な説明とともにリストアップされています。ここでは、私たちの実験でよく変更されるパラメータをいくつか紹介します。

#### 2. megs

模擬システムのメモリ容量を設定します。デフォルトでは32MBです。例えば、シミュレーション・マシンに128MBのシステムを設定したい場合は、設定ファイルに次の行を記述する必要があります。

---

メガ：128

---

3. floppya (floppyb)

floppyaは1台目のフロッピードライブ、floppyb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	# Access to 1.44MB A drive under Linux.
floppya: 1_44=b:, status=inserted	# Access to 1.44MB B drive under Win.
floppya: 1_44=bootimage.img, status=inserted	# Use imagefile bootimage.img. floppyb:
1_44=..\Linux\rootimage.img, status=inserted	# Use image .. \Linux\rootimage.img.

---

#### 4. ata0、ata1、ata2、ata3

これらの4つのパラメータ名は、シミュレーションされたシステムで最大4つのATAチャネルを起動するために使用されます。有効なチャンネルごとに、2つのIOベースアドレスと1つの割り込み要求番号を指定する必要があります。デフォルトでは、ata0のみが有効で、パラメータは以下の値に設定されています。

---

ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14	
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14	

---

#### 5. 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="hd-largeimg", mode=flat, cylinders=1187, heads=16,	
translation=echs	

---

表 17-1 デバ イス の構	Description	Available value

成才 プシ ヨン Options		
type	Connected device type	[disk   cdrom]
path	Image file path name	
mode	Image file type, valid only for disk	[flat   concat   external   dll   sparse   vmware3   undoable   growing   volatile ]

cylinders	Valid only for disk	
heads	Valid only for disk	
spt	Valid only for disk	
status	Valid only for disk	[inserted   ejected]
biosdetect	Bios detect type	[none   auto], Only valid for disk on ata0 [cmos]
translation	The type of bios conversion (int13), valid only for disk	[none   lba   large   rechs   auto]
mode	Confirm the string returned by the device ATA command	

When configuring an ATA device, you must specify the type of the connected device, which can be ‘disk’ or ‘cdrom’. You must also specify the pathname of the device. The “pathname” can be a hard disk image file, an iso file of the CDROM, or a CDROM drive pointing directly to the system. In Linux systems, system devices can be used as Bochs hard drives, but for security reasons, direct use of physical hard disks on the system is not recommended under Windows.

「disk」タイプのデバイスでは、「path」、「cylinders」、「head」、「spt」のオプションが必要です。 「cdrom」タイプのデバイスでは、「path」オプションが必要です。

ディスク変換方式（従来のint13 bios関数で実装され、DOSなどの古いOSで使用されていた）は次のように定義できます。

- ♦ none: No need to translate, suitable for hard disks with a capacity less than 528MB (1032192 sectors);
- ♦ large: Standard bit shift algorithm for hard disks with a capacity of less than 4.2 GB (8257536 sectors);
- ♦ rechs: The modified shift algorithm uses a pseudo-physical hard disk parameter of 15 heads for hard disks with a capacity less than 7.9 GB (15482880 sectors);
- ♦ lba: Standard lba-assisted algorithm. Suitable for hard drives with a capacity less than 8.4GB (16,450,560 sectors);
- ♦ auto: Automatically select the best conversion scheme (should be changed if system does not start).

モードオプションは、ハードディスクイメージファイルの使用方法を説明するために使用されます。以下のモードのいずれかになります。

- ♦ flat: a flat sequential file;
- ♦ concat: Multiple files;
- ♦ external: Dedicated by the developer, specified by the C++ class;
- ♦ dll: Dedicated by the developer, used by the DLL;
- ♦ sparse: Stackable, identifiable, retractable;
- ♦ vmware3: Support vmware3 hard disk format;
- ♦ undoable: a flat file with a confirmed redo log;
- ♦ growing: Capacity scalable image file;
- ♦ volatile: A flat file with a variable redo log.

上記オプションのデフォルト値は

---

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

---

## 6. boot

boot」は、エミュレートされたマシンのブート用ドライブを定義するために使用します。指定できるのは、フロッピーディスク、ハードディスク、CDROM、ドライブレター「c」「a」などです。例は以下の通りです。

---

```
boot:a.boot;
```

---

## 7. 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わる前に実行できる最大の命令数を指定するために使用します。また、このオプションはSMPをサポートするBochsプログラムでのみ利用可能です。

RESET\_ON\_TRIPLE\_FAULT」は、プロセッサにトリプルエラーが発生したときに、CPUが単なるパニックではなくリセット操作を行う必要があることを指定するために使用します。

IPSは、シミュレーションする1秒あたりの命令数を指定します。これは、Bochsがホストシステム上で実行するIPSの値です。この値は、シミュレーションシステムの時間に関する多くのイベントに影響します。例えば、IPS値を変更すると、VGAの更新速度やその他のシミュレーションシステムの評価に影響を与えます。そのため、使用するホスト性能に応じてこの値を設定する必要があります。設定方法は表17-2を参照してください。例えば

---

```
cpu: count=1, ips=50000000, reset on triple fault=1
```

---

表17-2 IPSの設 定例 <b>Bochs version</b>	<b>Speed</b>	<b>Machine / Compiler</b>	<b>Typical IPS</b>
2.4.6	3.4Ghz	Intel Core i7 2600 with Win7x64/g++ 4.5.2	85 to 95 MIPS
2.3.7	3.2Ghz	Intel Core 2 Q9770 with WinXP/g++ 3.4	50 to 55 MIPS
2.3.7	2.6Ghz	Intel Core 2 Due with WinXP/g++ 3.4	38 to 43 MIPS
2.2.6	2.6Ghz	Intel Core 2 Due with WinXP/g++ 3.4	21 – 25 MIPS

2.2.6	2.1Ghz	Athlon XP with Linux 2.6/g++ 3.4	12 – 15 MIPS
-------	--------	----------------------------------	--------------

8. log

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 Running Linux 0.1x system in Bochs

To run a Linux operating system, we need a root filesystem (root fs) in addition to the kernel. The root file system is usually an external device that stores the necessary files (such as system configuration files and device files) and data files for Linux system runtime. In modern Linux operating systems, the kernel image file (bootimage) is stored in the root file system. The system boot initiator loads the kernel execution code into memory from this root file system device for execution.

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>

---

The last 6 digits in the package name are date information. You should usually choose the latest package with the latest download date. After the download is complete, you can use a general decompression program such as unzip, 7-zip or rar to decompress it. Note that you need about 250MB of disk space to unzip this file.

### 17.2.1 Description of the files in the package

linux-0.12-080324.zip ファイルを解凍すると、linux-0.12-080324 という名前のディレクトリが生成されます。そのディレクトリに入ってみると、以下のように20個ほどのファイルがあることがわかります。

```
[ 1 3078642 Mar 24 10:49 bochs-2.3.6-1.i586.rpm
r
o
o
t
@
w
w
w

l
i
n
u
x
-
0
.
1
2
-
0
8
0
3
2
4
]
#
l
s
-
o
-
g
t
o
t
a
l

2
5
6
9
1
6
-rw-r--r--
-rw-r--r-- 1 3549736 Mar 24 10:48 Bochs-2.3.6.exe
-rw-r--r-- 1      15533 Mar 24 18:04 bochsout.txt
```

```
-rw-r--r-- 1      1774 Mar 24 20:13 bochsrc-0.12-fd.bxrc
-rw-r--r-- 1      5903 Mar 24 17:56 bochsrc-0.12-hd.bxrc
-rw-r--r-- 1      35732 Dec 24 20:15 bochsrc-sample.txt
-rw-r--r-- 1     150016 Mar  6 2004 bootimage-0.12-fd
-rw-r--r-- 1     154624 Aug 27 2006 bootimage-0.12-hd
-rw-r--r-- 1      68 Mar 24 12:21 debug.bat
-rw-r--r-- 1    1474560 Mar 24 15:27 diskA.img
-rw-r--r-- 1    1474432 Aug 27 2006 diskB.img
-rw-r--r-- 1      7917 Mar 24 11:32 linux-0.12-README
-rw-r--r-- 1    1474560 Mar 24 17:03 rootimage-0.12-fd
-rw-r--r-- 1   251338752 Mar 24 18:04 rootimage-0.12-hd
-rw-r--r-- 1     21253 Mar 13 2004 SYSTEM.MAP
```

[root@www linux-0.12-080324]#.

This package contains two Bochs installers, two Bochs .bxrc configuration files, two bootimage files containing kernel code, a floppy disk and a hard disk root file system (rootimage) file, and other files. 其中 The README file briefly describes the purpose of each file. Here we will explain in more detail the purpose of each file.

- Bochs-2.3.6-1.i586.rpm is the Bochs installer for the Linux operating system. You can re-download the latest program.
- Bochs-2.3.6.exe is the Bochs installer for the Windows operating system platform. The latest version of the Bochs software can be downloaded from <http://sourceforge.net/projects/bochs/>. As Bochs continues to improve, some newer versions may cause compatibility issues. This needs to be resolved by modifying the .bxrc configuration file, and some issues need to be resolved by modifying the Linux 0.1x kernel code.
- bochsout.txt is a log file that is automatically generated when the Bochs system is running. It contains various status information for the Bochs runtime. When running Bochs has problems, you can check the contents of this file to preliminarily determine the cause of the problem.
- bochsrc-0.12-fd.bxrc is the configuration file that allows the system to boot from a floppy disk. This configuration file is used to boot the Linux 0.12 system from the Bochs Virtual A drive (/dev/fd0), ie the kernel image file is set in virtual disk A and the subsequent root file system is required to be inserted into the current virtual boot drive. During the boot process it will ask us to "insert" the root filesystem disk (rootimage-0.12-fd) in the virtual A drive. The kernel image and boot file used by this configuration file is bootimage-0.12-fd. After Bochs is properly installed, double-click this configuration file to run the configured Linux 0.12 system.
- bochsrc-0.12-hd.bxrc is also a configuration file set to boot from drive A, but will use the root file system in the hard drive image file (rootimage-0.12-hd). This configuration file is booted using bootimage-0.12-hd. Similarly, after properly installing Bochs, double-click on this configuration file to run the configured Linux 0.12 system.
- bootimage-0.12-fd is an image file generated by the compiled kernel. It contains the code and data for the entire kernel, including the code for the floppy boot sector. You can run the configured Linux 0.12 system by double-clicking on the relevant configuration file.
- bootimage-0.12-hd is the kernel image file used to use the root file system on the virtual hard disk, that is, the root file system device number of the 509th and 510th bytes of the file has been set to the 1st partition of the C hard disk (/dev/hd1), the device number is 0x0301.
- debug.bat is a batch program that starts the Bochs debugging function on the Windows platform.

- なお、Bochsがインストールされているディレクトリによっては、パス名を変更する必要がありますのでご注意ください。また、LinuxシステムにインストールされているBochsシステムには、デフォルトではデバッグ機能が含まれていません。Linuxではgdbプログラムを使って直接デバッグすることができます。それでもBochsのデバッグ機能を利用したいのであれば、Bochsのソースコードをダウンロードして、自分でカスタマイズする必要があります。
- disk.a.img and diskb.img are two floppy image files in DOS format. It contains some utilities. In Linux 0.12 you can use the command mc当地 and other commands to access these two image files. Of course, you need to dynamically "insert" the corresponding "floppy disk" before accessing. When you double-click the bochsrc-0.12-fd.bxrc or bochsrc-0.12--hd.bxrc configuration file to run the Linux
- 0.12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 is the virtual hard disk image file mentioned above, which contains 3 partitions. The first partition is a MINIX file system type 1.0 root file system, and the other two partitions are also MINIX 1.0 file system types, and some source code files for testing are stored. You can load and use these spaces by using the mount command.
- rootimage-0.12-fd is the root file system on the floppy disk. This root file system disk is used when running the Linux 0.12 system using the bochsrc-0.12-fd configuration file.
- The SYSTEM.MAP file is the kernel memory storage location information file generated when the Linux 0.12 kernel is compiled. The contents of this file are very useful when debugging the kernel.

## 17.2.2 Installing the Bochs

パッケージに含まれるbochs-2.3.6-1.i586.rpmファイルは、Linuxで使用するBochsのインストーラです。Bochs-2.3.6.exeは、Windows オペレーティングシステムで使用するBochs インストーラです。Bochsソフトウェアの最新版は、以下のウェブサイトの場所で常に入手可能です。

---

<http://sourceforge.net/projects/bochs/>

---

If we are experimenting with a Linux system, you can install the Bochs software by running the rpm command on the command line or by double-clicking on the first file in the above package in the X window:

---

rpm -i bochs-2.3.6-1.i586.rpm

---

If you are on a Windows system, simply double-click the Bochs-2.3.6.exe file icon to install the Bochs system. After the installation, please modify the contents of the batch file debug.bat according to the specific directory of the installation. In addition, in the following experimental procedures and examples, we mainly introduce the use of Bochs on the Windows platform.

## 17.2.3 Running the Linux 0.1x System

BochsでLinux 0.1xシステムを動かすのはとても簡単です。Bochsソフトウェアが正しくインストールされたら、適切なBochs設定ファイル(\*.bxrc)をダブルクリックするだけで開始できます。ランタイムシミュレーションのためのPC環境は、各設定ファイルで設定されています。これらのファイルは、

任意のテキストエディタで変更することができます。Linux 0.12システムを実行するためには、対応する設定ファイルは通常、以下の行を含むだけで十分です。

```
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-
latest megs: 16
floppya: 1_44="bootimage-0.12-hd", status=inserted
ata0-master: type=disk, path="rootimage-0.12-hd", mode=flat, cylinders=487, heads=16, spt=63 boot: a
```

The first two lines indicate the ROM BIOS and VGA display card ROM program of the simulated PC, and generally do not need to be modified. Line 3 indicates the physical memory capacity of the PC, which is set to 16MB. Because the default Linux 0.12 kernel only supports up to 16MB of memory, the big settings don't work either. The parameter floppya specifies that the floppy disk drive A of the simulated PC uses the 1.44MB disk type, and has been set to use the bootimage-0.12-fd floppy image file, and is in the inserted state. The corresponding floppyb can be used to indicate the floppy image file used or inserted in the B drive. The parameter ata-master is used to specify the virtual hard disk capacity and hard disk parameters attached to the simulated PC. For the specific meaning of these hard disk parameters, please refer to the previous description. In addition, ata0-slave can be used to specify the image file and parameters used by the second virtual hard disk. The last 'boot' is used to specify the boot drive that can be set to boot from the A drive or from the C drive (hard drive). Here we set it to boot from the A drive(a).

### 1. Run the Linux 0.12 system using the bochsrc-0.12-fd.bxrc file.

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り、ブートイメージファイルがbootimage-0.12-fdに置き換えられているだけです。この設定ファイルをダブルクリックしてLinux 0.12システムを実行すると、図17-1に示すように、Bochsディスプレイのメインウィンドウにメッセージが表示されます。

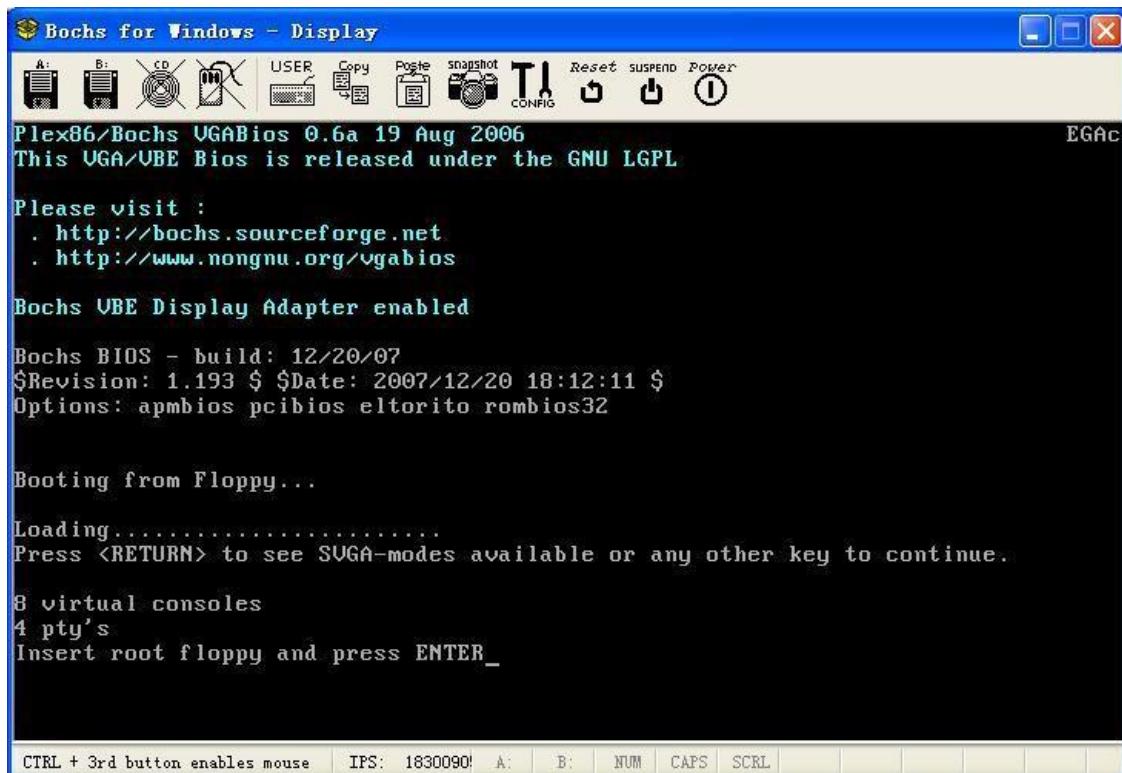


図17-1 フロッピーディスクからの起動とフロッピーディスク内のルートfsの使用

Since bochsrc-0.12-fd.bxrc configures the Linux 0.12 runtime to boot from drive A, and the kernel image file bootimage-0.12-fd will require the root file system to be in the drive currently being booted (disk A). , so the kernel will display a message asking us to "remove" the kernel boot image file bootiamge-0.12-fd and "insert in" the root file system. At this point we can use the A disk icon at the top left of the window to "replace" the A disk. Click this icon and change the original image file name (bootimage-0.12-fd) to rootimage-0.12-fd, so that we have completed the floppy disk replacement operation. After clicking the "OK" button to close the dialog window, press the Enter key to let the kernel load the root file system on the floppy disk, and finally the command prompt line appears, as shown in Figure 17-2.

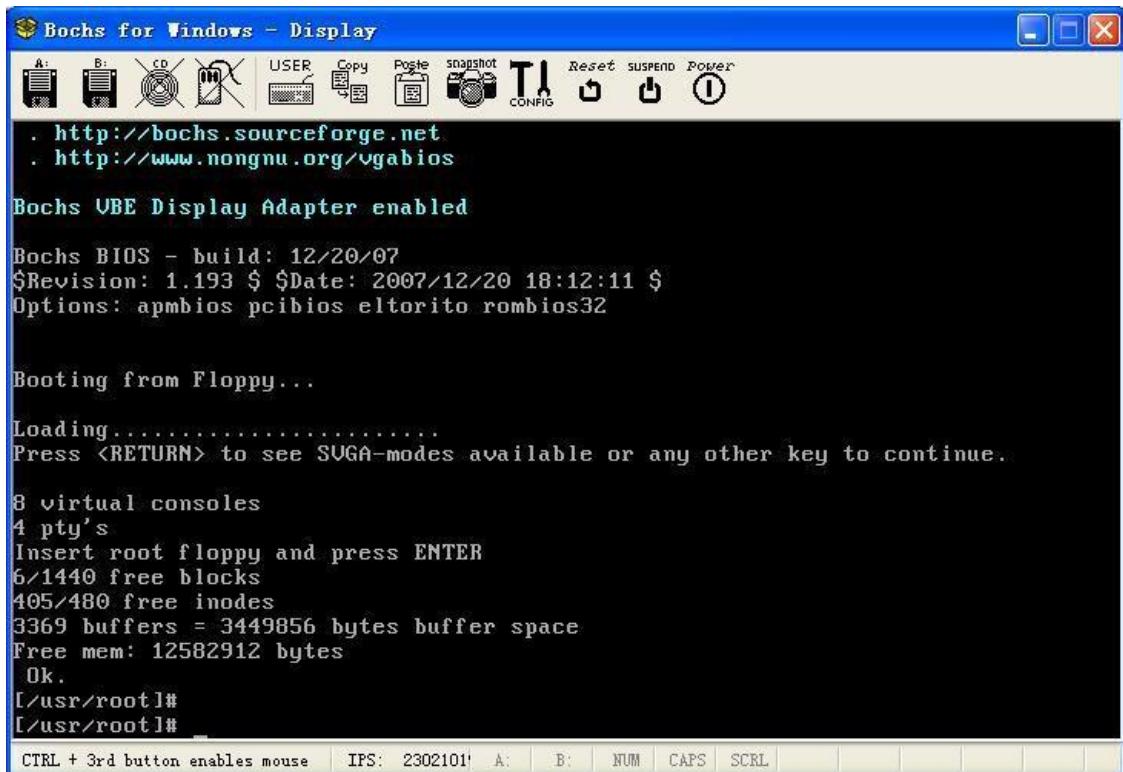


図17-2 フロッピー・ディスクを「交換」し、Enterを押して実行を続ける

## 2. 2. Run the Linux 0.12 system using the bochsrc-0.12-hd.bxrc file

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、カーネルの初期化時に自動的に仮想Cドライブの第1パーティションからルートファイルシステムの読み込みが開始されます。この時点で、bochsrc-0.12-hd.bxrcファイル名をダブルクリックして、Linux 0.12システムを直接実行すると、図17-2のような画面が表示されます。

## 17.3 Access Information in a Disk Image File

Bochsは、ディスク・イメージ・ファイルを使って、シミュレーション・システムの外部記憶装置をエミュレートしています。模擬オペレーティングシステムのすべてのファイルは、フロッピーディスクやハードディスク装置の形式でイメージファイルに保存されます。このため、BochsではホストOSと模擬システムの間で情報を交換するという問題が生じる。Bochsシステムは、ホストのフロッピーディスクドライブやCDROMドライブなどの物理的なデバイスを使って直接実行するように構成することができますが、そのような情報交換方法を利用するには面倒です。したがって、Imageファイルの情報を直接読み書きするのがベストです。模擬OSにファイルを追加したい場合は、Imageファイルに保存し、ファイルを取得したい場合は、Imageファイルから読み出します。しかし、Imageファイルに保存されている情報は、対応するフロッピーディスクやハードディスクのフォーマットに保存されているだけでなく、特定のファイルシステムのフォーマットにも保存されているため、Imageファ

イルにアクセスするプログラムは、そのファイルシステムのフォーマットに対応していなければなりません。そのため、Imageファイルにアクセスするプログラムは、そのファイルシステムを認識できなければ動作しません。本章の目的のためには、Imageファイルの中のMINIXおよび/またはDOSファイルシステム形式を識別するためのいくつかのツールが必要です。

一般的に、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。

- Use the disk image tool to access information (small files or split files) in the floppy image file;
- Use the loop device to access the hard disk image file in Linux. (a large amount exchange);
- Use iso format file for information exchange (a large amount exchange).

### 17.3.1 Using WinImage Software

フロッピー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
```

---

That is, the second 1.44MB floppy disk device is added to the simulated system, and the image file name used by the device is diskb.img.

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。

---

```
[/usr/root]# mc当地 hello.c b:  
HELLO.Cをコピーす  
る [/usr/root]# mdir b:  
ドライブBのボリュームにラベル  
がない B:/のディレクトリ  
  
HELLO C 74 4-30-104 4:47p  
1 File(s) 1457152 bytes free
```

---

Now exit the Bochs system and open the diskb.img file with WinImage. There will be a hello.c file in the WinImage main window. Use the mouse to select the file and drag it to the desktop, which completes the entire operation of fetching the file. If you need to put a file into the simulated system, the steps are exactly the opposite. Also note that WinImage can only access and manipulate disk files with DOS format, it can not access disk files in other formats such as MINIX file system.

### 17.3.2 Using an Existing Linux System

既存のLinuxシステム（Redhatなど）では、ループデバイスを使ってイメージファイルに格納されたファイルシステムにアクセスするなど、さまざまなファイルシステムに対応しています。フロッピーディスク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]#.
```

The "-t minix" option of the mount command indicates that the file system type being read is MINIX, and the "-o loop" option indicates that the file system is loaded by the loop device. If you need to access the DOS format floppy image file, just replace the file type option "minix" in the mount command with "msdos".

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にアクセスするためには、まず、イメージファイルのパーティション情報を理解して、イメージファイルでアクセスするパーティションの開始セクタ・オフセット位置を決定する必要があります。ハードディスクのイメージファイルのパーティション情報については、fdiskコマンドを使ってシステム上で確認する方法と、ここで説明する方法があります。ここでは、以下のパッケージに含まれるイメージ・ファイル rootimage-0.12-hd.img を例にとり、第1パーティションのファイル・システムにアクセスする方法を説明します。

---

<http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.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
```

```
Nr AF Hd Sec Cyl Hd Sec Cyl Start Size ID
```

```
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（ヘルプの場合

はm) : q [root@www linux-0.12-

080324]#.

As can be seen from the partition information given by fdisk above, the Image file contains 3 MINIX partitions (ID=81) and 1 swap partition (ID=82). If we need to access the contents of the first partition, write down the starting sector number of the partition (that is, the contents of the 'Start' column in the partition table). If you need to access the hard disk Image of another partition, then you just need to remember the starting sector number of the relevant partition.

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、第1パーティションをloop1に再関連づけした後、mount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 vmlinu
[root@www mnt]#
```

---

After the access to the file system in the partition is over, finally unmount the file system and disassociate.

---

```
[root@www mnt]# cd
[root@www ~]# umount /dev/loop1
[root@www ~]# losetup -d /dev/loop1
[root@www ~]#
```

## 17.4 Compiling and running the simple kernel

A simple multitasking kernel sample program is given in the previous chapter on the 80386 protection mode and its programming in Chapter 4, which we call the Linux 0.00 system. It contains two tasks running on privilege level 3, which will cycle through the characters A and B on the screen and perform task switching operations under clock timing control. The packages that have been configured to run in the Bochs simulation

environment are given on the book's website:

---

---

<http://oldlinux.org/Linux.old/bochs/linux-0.00/>

---

We can download any of the above files to experiment. The program given in the first package is the same as described here. The program in the second package is slightly different (the kernel head code runs directly at the 0x10000 address), but the principle is exactly the same. Here we will use the program in the first software package as an example to illustrate the experiment. The software of the second package is left to the reader for experimental analysis. After unpacking the linux-0.00-050613.zip package with a decompression software, a linux-0.00 subdirectory will be generated in the current directory. We can see that this package contains the following files:

- 
- |                          |   |
|--------------------------|---|
| 1. linux-0.00.tar.gz     | - Compressed source file;                                       |
| 2. linux-0.00-rh9.tar.gz | - Compressed source file;                                       |
| 3. Image                 | - Kernel boo image file;  |
| 4. bochsrc-0.00.bxrc     | - Bochs configuration file;                                     |
| 5. rawrite.exe           | - The program to write an Image to a floppy disk under Windows. |
| 6. README                | - Package documentation;  |
- 

The first file, linux-0.00.tar.gz, is a compressed source file for the kernel example. It can be compiled in the Linux 0.12 system to generate a kernel image file. The second is also a compressed source file for the kernel example, but the source program can be compiled under RedHat 9 Linux. The third file Image is a 1.44MB floppy image file of the runnable code compiled by the source program. The fourth file, bochsrc-0.00.bxrc, is the Bochs configuration file used when running in the Bochs environment. If you have installed the PC emulation software Bochs on your system, you can run the kernel code in the Image by double-clicking on the bochsrc-0.00.bxrc filename. The fifth is a utility program under DOS or Windows for writing image files to a floppy disk. We can run the RAWRITE.EXE program directly and write the kernel image file here to a 1.44MB floppy disk to run.

上記のカーネル例のソースコードは、linux-0.00-tar.gzファイルに含まれています。このファイルを解凍すると、boot.sとhead.sのプログラムに加えて、makefileを含むソースファイルを含むサブディレクトリが生成されます。as86/ld86のコンパイルとリンクによって生成された「boot」ファイルの先頭部分には32バイトのMINIX実行ファイルのヘッダ情報が、as/ldのコンパイルとリンクによって生成された「head」ファイルの先頭部分には1024バイトのa.outヘッダデータが含まれているので、カーネルの「Image」ファイルを作成する際には、これらのヘッダ情報を削除する必要があります。2つの「dd」コマンドを使ってヘッダー情報を除去し、それをカーネルイメージのImageファイルにまとめることができます。

このImageファイルは、ソースコードのディレクトリで直接makeコマンドを実行することで生成されます。すでにmakeコマンドを実行している場合は、先に「make clean」を実行してからmakeコマンドを実行してください。

---

l	1	root	root	487	Jun 12 19:25	Makefile
/						
u						
s						

---

```
r  
/  
r  
o  
o  
t  
/  
l  
i  
n  
u  
x  
-  
0  
.0  
]  
#  
  
1  
s  
  
-  
1  
  
t  
o  
t  
a  
l  
  
9  
-rw-----  
-rw----- 1 root      4096          1557 Jun 12 18:55 boot.s  
-rw----- 1 root      root          5243 Jun 12 19:01 head.s
```

[/usr/root/linux-0.0]# make

```
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]#.
```

To copy the Image file to the A disk image file, we can execute the command 'make disk' as follows. However, before executing this command, if you are compiling the Linux 0.12 system under Bochs, please copy and save your boot image disk file (for example, bootimage-0.12-hd), so that you can restore the Linux 0.12 system after the test. Image file.

---

```
[/usr/roo  System.map  boot.o      head.o      system
t/linux-
0.0]# ls
Image
Makefile   boot       boot.s      head.s
[/usr/root/linux-0.0]# make disk dd
bs=8192 if=Image of=/dev/fd0 1+1
records in
1+1レコードアウト
sync;sync;sync
[/usr/root/linux-0.0]#.
```

To run this simple kernel example, we can use the mouse to directly click the RESET icon on the Bochs window. Its operation is shown in the figure below. After that, if you want to resume running the Linux 0.12 system, please overwrite the startup file with the image file you just saved.

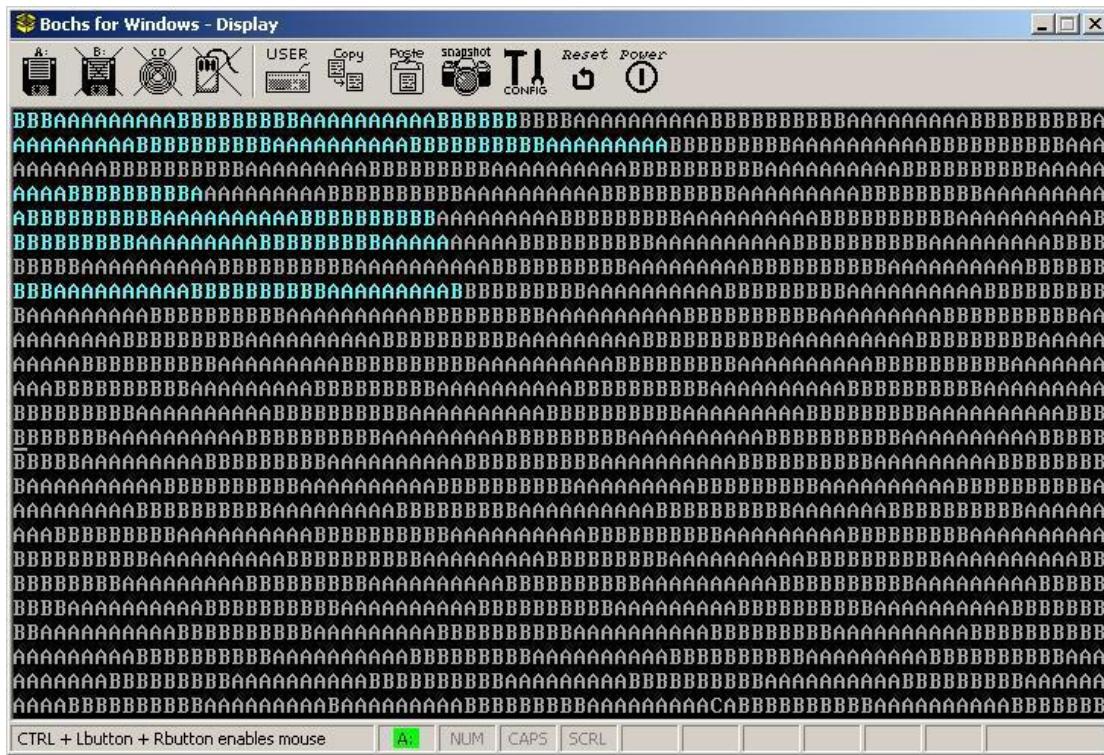


図17-3 シンプルなカーネルランタイムの画面表示

## 17.5 Using Bochs to Debug the Kernel

Bochs has very powerful operating system kernel debugging capabilities, which is one of the main reasons why Bochs was chosen as our preferred experimental environment. For a description of Bochs debugging features, see Section 17.2 above, which is based on the Linux 0.12 kernel to illustrate the basic methods of Bochs debugging operations in the Windows environment.

### 17.5.1 Running the Bochs Debugger

Bochsシステムが「C:¥¥Bochs-2.3.6¥」というディレクトリにインストールされ、Linux 0.12システム用のBochs設定ファイル名が「bochsrc-0.12-hd.bxrc」であるとします。次に、カーネルイメージファイルのあるディレクトリに、次のような簡単なバッチファイルrun.batを作成します。

```
"C:Program Files¥Bochs-2.3.6¥bochsdbg" -q -f bochsrc-0.12-hd.bxrc
```

Among them, bochsdbg is the debugging execution program of Bochs. The parameter "-q" means quick start (skip configuration interface), and the parameter "-f" means followed by a configuration file name. If the parameter is "-h", the program will display help information for all optional parameters. Run the batch command to enter the debugging environment. At this point, the main display window of Bochs is blank, and the control window will display the following similar content:

```
C:\Linux-0.12>"C:\Program Files\Bochs-2.3.6\bochsdbg" -q -f bochssrc-0.12-hd.bxrc
00000000000i[APIC?] local apic in initializing
=====
=====
Bochs x86エミュレータ 2.3.6
2007年12月24日のCVSスナップショットからの構築
=====
=====
00000000000i[      ] reading configuration from bochssrc-hd-new.bxrc
00000000000i[      ] installing win32 module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt

次はt=0で
(0) [0xfffffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b          ; ea5be000f0
<bochs:1>
```

At this point the Bochs debug system is ready to start running and the CPU execution pointer has been pointed to the instruction at address 0x000fffff0 in the ROM BIOS. Where "<bochs:1>" is the command line input prompt, where the number indicates the current command serial number. By typing the "help" command after the command prompt, we can list the basic commands for debugging the system. To find out how to use a command in detail, type the 'help' command followed by a specific command enclosed in single quotes, for example: " help 'vbreak' ". See below.

---

```
<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|stepp, p|n|next, modebp
-*- ブレークポイントの管理 -*-
    v|vbreak, lb|lbreak, pb|pbreak|b|break, sb, sba, blist, bpe, bpd,
    d|del|delete
-*- CPUとメモリの内容 -*-
    x, xp, u|disas|disassemble, r|reg|registers, setpmem, crc, info, dump_cpu, set_cpu,
    ptime, print-stack, watch, unwatch, ?|calc.
<bochs:2> help 'vbreak'
help vbreak
vbreak seg:off - 仮想アドレス命令のブレークポイントを設定します。
<bochs:3>
```

---

Some of the more commonly used commands are listed below. A complete list of all debug commands can be found in Bochs' own help file (`internal-debugger.html`) or refer to the online help information ("help" command).

1. Execute control commands. Control single or multi-step execution of instructions.

c                    Continuous execution

stepi [count]	Execute 'count' instructions, the default is 1.
si [count]	lbid.
step [count]	lbid.
s [count]	lbid.
p	Similar to 's', but with interrupt and function instructions as single-step execution, that is, single-step execution of the interrupt or functions.
n (or next)	lbid.
Ctrl-C	Stop execution and go back to the command line prompt.
Ctrl-D	Exit Bochs if you type the command at an empty command line prompt.
quit, q	Exit debugging.

---

2. Breakpoint setting commands. Where 'seg', 'off' and 'addr' can be hexadecimal numbers starting with '0x', or decimal numbers or octal numbers starting with '0'.

vbreak seg:off	Set the instruction breakpoint on the virtual address.
vb seg:off	lbid.
lbreak addr	Set the instruction breakpoint on the linear address.
lb addr	lbid.
pbreak [*] addr	Set the instruction breakpoint on the physical address. Where '*' is an option for compatibility with GDB.
pb [*] addr lbid. break [*]	
addr lbid. b [*] addr lbid.	
info break	Displays the status of all current breakpoints.
delete n	Delete a breakpoint.
del n	lbid.
d n	lbid.

---

### 3. Memory operation commands

x /nuf addr	Check the memory contents at the linear address 'addr'. If 'addr' is not specified, the default is the next address.
xp /nuf addr	Check the memory contents at the physical address 'addr'.
The optional parameters 'n', 'u' and 'f' can be:	
n	The number of memory units to display. The default value is 1.
u	Indicates the unit size, the default selection is the following 'w': 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	Display format, the default selection is 'x': 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。

dr1 addr2 addr1 から addr2 までの物理メモリの CRC チェックサムを表示します。

info dirty	Displays the physical memory page that has been modified since the last time this command was executed. Only the first 20 bytes of the page are displayed.
------------	--

---

### 4. Information display and CPU register operation commands

info program	Displays the execution status of the program.
--------------	---

---

info registers	Displays the CPU registers (no floating point registers).
info break	Displays the current breakpoint setting status.
	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。
	For example, set \$eax = 0x01234567; set \$edx = 25
dump_cpu	Displays all status information of the CPU.
set_cpu	Set all status information of the CPU.

---

The content format displayed by the "dump\_cpu" and "set\_cpu" commands is:

---

```

"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%0x, dl=0x%0x, dh=0x%0x, valid=%u\n"
"ss:s=0x%0x, dl=0x%0x, dh=0x%0x, valid=%u\n"
"ds: s=0x%0x, dl=0x%0x, dh=0x%0x, valid=%u\n"
"es:s=0x%0x, dl=0x%0x, dh=0x%0x, valid=%u★n"
"fs:s=0x%0x, dl=0x%0x, dh=0x%0x, valid=%u★n"
"gs: s=0x%0x, dl=0x%0x, dh=0x%0x, valid=%u\n"
"ldtr:s=0x%0x, dl=0x%0x, dh=0x%0x,
valid=%u★n" "tr:s=0x%0x, dl=0x%0x, dh=0x%0x,
valid=%u★n" "gdtr:base=0x%0x, limit=0x%0x"
"idtr:base=0x%0x, limit=0x%0x" "dr0:0x%0x★n"
"dr1:0x%0x\$"
"dr2:0x%0x\$"
"dr3:0x%0x%%"
"dr4:0x%0x%%"
"dr5:0x%0x%%"
"dr6:0x%0x%%"
"dr7:0x%0x%%"
"tr3:0x%0x%%"
"tr4:0x%0x%%" "tr5:
0x%0x\n_"
[tr6:0x%0x\n_]
[tr7:0x%0x\n_]
```

```
「cr0:0x%x\n」  
「cr1:0x%x\n」  
「cr2:0x%x\n」  
「cr3:0x%x\n」  
「cr4:0x%x\n」  
「inhibit_int:%u\n」  
「done\n」
```

among them:

- 's' means a selector;
- 'dl' is the low 4-byte of the segment descriptor in the selector shadow register;
- 'dh' is the high 4-byte of the segment descriptor in the selector shadow register;
- 'valid' indicates whether a valid shadow descriptor is being stored in the segment register;
- 'inhibit\_int' is an instruction delay interrupt flag. If set, it means that the instruction that has just been executed by the previous one is an instruction that delays the CPU from accepting the interrupt (for example, STI, MOV SS);

また、"set\_cpu"コマンドの実行時には、"Error: ... "というフォーマットでエラーが報告されます。これらのエラーメッセージは、入力行ごとに表示されたり、最後の「done」が表示された後に表示されたりします。set\_cpu "コマンドが正常に実行された場合は、"OK"を表示してコマンドを終了します。

## 5. Disassembly command

---

```
disassemble start end      Disassemble instructions within a given linear address range.  
disas  
u
```

---

The following are some of the new commands from Bochs, but the commands involving file names in the Windows environment may not work properly.

- record *filename* -- Write your input command sequence to the file '*filename*' during execution. The file will contain lines of the form "%s %d %x". The first parameter is the event type; the second is the timestamp; the third is the data of the related event.
- playback *filename* -- The execution command is played back using the contents of the file '*filename*'. You can also type other commands directly in the control window. Each event in the file will be played back, and the playback time will be counted relative to the time the command was executed.
- print-stack [num words] -- Display num 16-bit words at the top of the stack. The default value of num is 16. When the base address of the stack segment is 0, the command can be used normally only in protected mode.
- load-symbols [global] *filename* [offset] -- Load symbol information from the file '*filename*'. If the keyword global is given, then all symbols will be visible in the context before the symbol was loaded. The 'offset' (default is 0) is added to each symbol item. The symbol information is loaded into the context of the currently executing code. The format of each line in the symbol file *filename* is "%x %s". The first value is the address and the second is the symbol name.

BochsがLinuxブートローダの先頭までの実行を直接シミュレートするためには、まずbreakpointコマンドを使って0x7c00にブレークポイントを設定し、システムを0x7c00まで継続して実行させて停止させます。実行されるコマンドの順序は以下の通りです。

---

```
<bochs:3> vbreak 0x0000:0x7c00  
<bochs:4> c  
(0) ブレークポイント 1, 0x7c00 (0x0:0x7c00)  
次はt=4409138で
```

```
(0) [0x000007c00] 0000:7c00 (unk. ctxt): mov ax, 0x7c0 ; b8c007  
<bochs:5>
```

At this point, the CPU executes the first instruction at the beginning of the boot.s program, and the Bochs main window will display some information such as "Boot From floppy...". Now we can follow the debugger by stepping through the command 's' or 'n' (not tracking into the subroutine). Bochs' breakpoint setting commands, disassembly commands, information display commands, etc. can be used to assist in our debugging operations. Here are some examples of common commands:

```
<bochs:8> u /10                                # Disassemble 10 instructions.  
00007c00: (          ): mov ax, 0x7c00          ; b8c007  
00007c03: (          ): mov ds, ax              ; 8ed8  
00007c05: (          ): mov ax, 0x9000          ; b80090  
00007c08: (          ): mov es, ax              ; 8ec0  
00007c0a: (          ): mov cx, 0x100             ; b90001  
00007c0d: (          ): sub si, si              ; 29f6  
00007c0f: (          ): sub di, di              ; 29ff  
00007c11: (          ): rep movs word ptr [di], word ptr [si] ; f3a5  
00007c13: (          ): jmp 9000:0018           ; ea18000090  
00007c18: (          ): mov ax, cs              ; 8cc8  
  
<bochs:9> info r                                # View the contents of the current register  
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                            # Display the current stack  
0000ffffe [0000ffffe] 0000  
00010000 [00010000] 0000  
00010002 [00010002] 0000  
00010004 [00010004] 0000  
00010006 [00010006] 0000  
00010008 [00010008] 0000  
0001000a [0001000a] 0000  
  
...  
<bochs:11> dump_cpu                               # Displays all registers in the CPU.  
eax:0xaa55  
ebx:0x0  
ecx:0x110001  
edx:0x0  
ebp:0x0  
esi:0x0
```

```

edi:0xffe4
esp:0xffffe
eflags:0x282
eip:0x7c00
cs:s=0x0, dl=0xffff, dh=0x9b00, valid=1      # s=selector;dl,dh - low & high 4-byte of desc.
ss:s=0x0, dl=0xffff, dh=0x9300, valid=7
ds:s=0x0, dl=0xffff, dh=0x9300, valid=1
es:s=0x0, dl=0xffff, dh=0x9300, valid=1
fs:s=0x0, dl=0xffff, dh=0x9300, valid=1 gs:
s=0x0, dl=0xffff, dh=0x9300, valid=1
ldtr:s=0x0, dl=0x0, dh=0x0, valid=0
tr:s=0x0, dl=0x0, dh=0x0, valid=0
gdtr:base=0x0, limit=0x0
idtr:base=0x0, limit=0x3ff
dr0:0x0
dr1:0x0 dr2:0x0
dr3:0x0
dr6:0xffff0ff0
dr7:0x400
tr3:0x0 tr4:0x0
tr5:0x0 tr6:0x0
tr7:0x0
cr0:0x6000001
0
cr1:0x0 cr2:0x0
cr3:0x0 cr4:0x0
inhibit_mask:0
done
<bochs:12>

```

Since the 32-bit code of the Linux 0.1X kernel is stored from the absolute physical address 0, if you want to execute directly to the beginning of the 32-bit code (that is, at the beginning of the head.s program), we can set a breakpoint at linear address 0x0000, and run the command 'c' to execute to that location.

また、コマンドプロンプトで直接Enterキーを押すと、前のコマンドが繰り返し実行されます。上矢印を押すと前のコマンドが表示されます。その他のコマンドについては、「help」コマンドを参照してください。

## 17.5.2 Locating Variables or Data Structures in the Kernel

カーネルのコンパイル時に「system.map」ファイルが生成されます。このファイルには、カーネルイメージ（ブートイメージ）ファイル内のグローバル変数と、各モジュール内のローカル変数のオフセットアドレスの位置が記載されています。カーネルがコンパイルされた後、上述のファイルエクス

ポート方法を使って、「system.map」ファイルをホスト環境（Windows）に展開することができます。「system.map」ファイルの詳しい目的や役割については、第3章を参照してください。サンプルファイル「system.map」の内容の一部を以下に示します。このファイルを使うことで、Bochs debug systemの中で、変数の位置を特定したり、指定した関数コードにジャンプしたりすることができます。

...  
グローバルシンボル。

```
_dup: 0x16e2c
_nmi: 0x8e08
_bmapです。0xc364
_iput: 0xc3b4
_blk_dev_init: 0x10ed0
を開きます。0x16dbc
_do_execveです。0xe3d4
_con_init: 0x15ccc
_put_super: 0xd394
_sys_setgid: 0x9b54
_sys_umask: 0x9f54
_con_write: 0x14f64
_show_task: 0x6a54
_buffer_init: 0xd1ec
_sys_settimeofday: 0x9f4c
_sys_getgroups: 0x9edc
```

...

---

Similarly, since the 32-bit code of the Linux 0.1X kernel is stored from the absolute physical address 0, the offset position of the global variable in 'system.map' is the linear address position in the CPU. So we can set breakpoints directly at the variable or function name location of interest and let the program execute continuously to the specified location. For example, if we want to debug the function buffer\_init(), we can see that it is located at 0xd1ec from the 'system.map' file. At this point we can set a linear address breakpoint there and execute the command 'c' to let the CPU execute to the beginning of the specified function, as shown below.

---

```
<bochs:12> lb 0xd1ec          # Set a linear address breakpoint.
<bochs:13> c                  # Continuous execution.
(0) ブレークポイント 2, 0xd1ec in ? ()
次はt=16689666で
(0) [0x0000d1ec] 0008:0000d1ec (unk. ctxt): push ebx      ; 53
<bochs:14> n                  # Next instruction.
次はt=16689667で
(0) [0x0000d1ed] 0008:0000d1ed (unk. ctxt): mov eax, dword ptr ss:[esp+0x8] ; 8b442408
<bochs:15> n                  # Next instruction.
次はt=16689668で
(0) [0x0000d1f1] 0008:0000d1f1 (unk. ctxt): mov edx, dword ptr [ds:0x19958] ; 8b1558990100
<bochs:16>
```

Program debugging is a skill that requires more practice to make it happen. Some of the basic commands

described above need to be combined to give you the flexibility to observe the overall environment of kernel code execution.

## 17.6 Creating a Disk Image File

ディスクイメージファイルは、フロッピーディスクやハードディスク上のデータを完全にイメージ化し、ファイルとして保存したものです。ディスクイメージファイルに格納されている情報の形式は、実際のディスクに格納されている情報の形式と全く同じです。空のディスクイメージファイルとは、作成したディスクと同じ容量で、中身がすべて「0」のファイルである。この空のイメージファイルは、購入したばかりの新品のフロッピーディスクやハードディスクのようなもので、使用するためには、パーティションや/を切ってフォーマットする必要があります。

ディスクイメージファイルを作成する前に、まず、作成したイメージファイルの容量を決定する必要があります。フロッピーのイメージファイルの場合、各種仕様の容量（1.2MBや1.44MB）が決まっています。そこで、ここでは、必要なハードディスクのイメージファイルの容量を決める方法を紹介します。従来のハードディスクの構造は、金属製のディスクを積み重ねたものである。各ディスクの上面と下面にはデータが格納されており、表面全体が同心円状の「シリンダー」によってトラックに分割されている。ディスクにデータを読み書きするためには、各ディスクの両面にヘッドが必要である。ディスクが回転すると、ヘッドは半径方向に移動するだけでどのトラックにも移動できるため、ディスク表面のすべての有効な位置にアクセスできる。各トラックはセクタに分割され、セクタサイズは一般的に256～1024バイトで構成されている。多くのシステムでは、セクタサイズは通常512バイトである。典型的なハードディスクの構造を図17-4に示します。

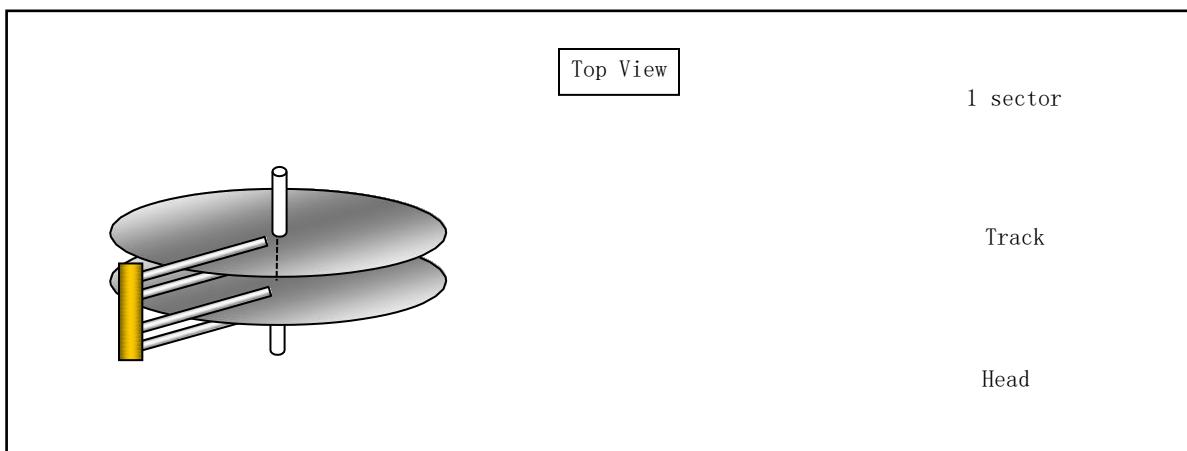


図17-4 典型的なハードディスクの内部構造

The figure shows a hard disk structure with two metal disks with four physical heads. The maximum number of cylinders contained is determined at the time of production. When the hard disk is partitioned and formatted, the magnetic media on the surface of the disk is initialized to data in a specified format such that each track (or cylinder) is divided into a specified number of sectors. Therefore the total number of sectors of this hard disk is:

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

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かったため、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、互換性と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、 $64 \times 4 = 256$ 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。

**17.6.1 USBフラッシュディスクの場合は、ハードディスクとして扱うことができます。**フロッピーディスクの場合は、トラック数（シリンドー数）、ヘッド数、トラックあたりのセクタ数が決まっている、分割されていない超小型ハードディスクと考えることができます。例えば、容量1.44MBのフロッピーディスクは、1トラック80本、1ヘッド2セクタ18個、1セクタ512バイトである。総セクタ数は2880で、総容量は $80 \times 2 \times 18 \times 512 = 1474560$ バイトとなります。したがって、以下に説明するハードディスクのイメージファイルの作成方法は、すべてUSBディスクやフロッピーのイメージファイルの作成にも利用できます。説明の便宜上、特に断りのない限り、すべてのディスクイメージファイルをイメージファイルと呼びます。

## 17.6.2 Using Bochs' own Image Creation Tool

Bochsシステムには、ディスクイメージ作成ツール「bximage.exe」が付属しており、これを使ってフロッピーディスクやハードディスク用の空のイメージファイルを作成することができます。bximage.exeを実行すると、イメージ作成画面が表示され、作成するイメージの種類（ハードディスクhdまたはフロッピーディスクfd）を選択する画面が表示されます。ハードディスクを作成する場合は、ハードディスクイメージのモードタイプを入力するよう促されますが、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

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に以下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してください。

If you already have a hard disk Image file with the required capacity, you can also copy the file directly to generate another Image file, and then you can process the file according to your own requirements. The process for creating a floppy Image file is similar to the above, except that you will be prompted to select a floppy disk type. Similarly, if you already have other floppy Image files, you can use the direct copy method.

### 17.6.3 Creating an Image File Using the dd Command on a Linux

ddコマンドは、Linuxシステムのコマンドラインツールで、主にファイルのコピーやファイルのデータ形式の変換に使用します。上記で説明したように、先ほど作成したImageファイルは、すべてのコンテンツが0の空のファイルですが、その容量は必要条件と一致しています。そこで、まず、容量を必要とするImageファイルのセクタ数を算出し、「dd」コマンドを使用して対応するImageファイルを生成します。

例えば、1トラックのシリンダー数が520、ヘッド数が16、セクタ数が63のハードディスクのイメージファイルを作成したいとします。総セクタ数は  $520 * 16 * 63 = 524160$  とすると、コマンドは次のようにになります。

---

```
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
```

### 17.6.4 Creating a floppy disk image file in DOS format using WinImage

WinImage is a DOS format Image file access and creation tool. After associating with the software, double-click the icon of the DOS floppy Image file and you can browse, delete or add files to it. In addition, it can also be used to browse the contents of the CDROM iso file. When you use WinImage to create a floppy disk

Image, you can generate an Image file with DOS format. Methods as below:

- a) Run WinImage. Select the "Options->Settings" menu and select the Image Settings page. Set Compression to "None" (that is, pull the indicator to the far left);
- b) Create an Image file. Select the menu "File->New" and a floppy disk selection box will pop up.

- c) 1.44MBの容量のフォーマットをお選びください。
  - d) Select the boot sector property menu item "Image->Boot Sector properties" and click the MS-DOS button in the dialog box;
  - e) Save the file.
- なお、「保存の種類」ダイアログボックスで「すべてのファイル (\*.\*)」を選択する必要があります。そうしないと、作成されたイメージファイルにWinImageの情報が含まれてしまい、Bochs上でイメージファイルが正常に動作しなくなります。作成されたイメージが要件を満たしているかどうかは、ファイルサイズを見ればわかります。標準的な1.44MBのフロッピーディスクの場合、1474560バイトの容量が必要です。新しいImageのファイルサイズがこの値より大きい場合は、再作成するか、Notepad++などのバイナリエディタ（Hex-Editorプラグインが必要）を使って余分なバイトを削除してください。削除の方法は以下の通りです。
- Open the Image file with Notepad++ and run the plugin Hex-Editor. According to the 511, 512 bytes of the disk image file are 55, AA two hexadecimal numbers, we push back 512 bytes, delete all the previous bytes. At this point, for a disk using MSDOS 5.0 as the boot, the first few bytes of the file should be similar to "EB 3C 90 4D ...".
  - Then pull down the right scroll bar and move to the end of the img file. Delete all data after "...F6 F6". Usually it is to delete all data starting from 0x168000. The last line when the operation is completed should be a complete line "F6 F6 F6...". Save and exit to use the Image file.

## 17.7 Making a Root File System

ここでは、ハードディスクにルートファイルシステムを作成することが目的です。フロッピーディスクとハードディスクのルートファイルシステムのImageファイルはoldlinux.orgのウェブサイトからダウンロードできますが、ここでは参考のために作成手順を詳しく説明します。作成の過程では、Linus氏が書いたインストール記事：INSTALL-0.11も参考にしてください。ルートファイルシステムのディスクを作成する前に、まず、rootimage-0.12とbootimage-0.12のイメージファイルをダウンロードします（最新のファイルをダウンロードしてください）。

---

<http://oldlinux.org/Linux.old/images/bootimage-0.12-20040306>

---

Modify these two file names into easy-to-remember names bootimage-0.12 and rootimage-0.12, and create a subdirectory named Linux-0.12 for them. During the making process, we need to copy some of the executables in the rootimage-0.12 floppy disk and boot the simulation system using the bootimage-0.12 boot disk. So before you start working on the root file system, you first need to confirm that you have been able to run the minimum Linux system consisting of these two floppy Image files.

### 17.7.1 Root File System and Root File Device

Linuxの起動時には、ルートディレクトリを含むデフォルトのファイルシステムがルートファイルシステムとなります。一般的にルートディレクトリには、以下のサブディレクトリやファイルが含まれています。

- ♦ etc/      This directory mainly contains some configuration files, such as 'rc' file;
- ♦ dev/      Contains device special files for operating the device with files;

- ◆ bin/ Store system execution programs. Such as sh, mkfs, fdisk, etc.;
- ◆ usr/ Store library functions, manuals, and other files;
- ◆ usr/bin/ Store commands commonly used by users;
- ◆ var/ Used to store data when the system is running or for information such as logs.

The device that holds the file system is the file system device. For example, for the Windows operating system, the hard disk C drive is the file system device, and the files stored on the hard disk according to certain rules constitute the file system. Windows can have file systems in formats such as NTFS or FAT32, while the file system supported by the Linux 0.1X kernel is the MINIX 1.0 file system.

Linuxの起動ディスクがルートファイルシステムをロードする際には、起動ディスクのブートセクタの509バイト目と510バイト目にあるワード (ROOT\_DEV) のルートファイルシステムのデバイス番号に従って、指定されたデバイスからルートファイルシステムがロードされます。デバイス番号が0の場合、ルートファイルシステムは、起動ディスクがある現在のドライブからロードされる必要があることを意味します。デバイス番号がハードディスク・パーティション・デバイス番号の場合、ルート・ファイル・システムは指定されたハードディスク・パーティションからロードされます。Linux 0.1X カーネルでサポートされているハードディスクのデバイス番号を表17-3に示します。

表 17- 3 ハ ード デ ィス クの 論 理 デ バ イ ス 番 号	Device file	Description
Device nr		
0x0300	/dev/hd0	Represents the entire first hard driv
0x0301	/dev/hd1	The first partition of the first disk
0x0302	/dev/hd2	The second partition of the first disk
0x0303	/dev/hd3	The third partition of the first disk
0x0304	/dev/hd4	The fourth partition of the first disk
0x0305	/dev/hd5	Represents the entire second hard driv
0x0306	/dev/hd6	The first partition of the second disk
0x0307	/dev/hd7	The second partition of the second disk
0x0308	/dev/hd8	The third partition of the second disk
0x0309	/dev/hd9	The fourth partition of the second disk

If the device number is a floppy device number, the kernel will load the root file system from the floppy drive specified by the device number. The floppy device numbers used in the Linux 0.1X kernel are shown in Table 17-4. For the calculation method of the floppy disk drive device number, please refer to the description after the floppy.c program in Chapter 9.

表	Device file	Description
---	-------------	-------------

17- 4 フ ロッ ピー ディ スク ドラ イブ のロ ジッ クデ バイ ス番 号  Device Nr		
0x0208	/dev/at0	1.2MB A drive
0x0209	/dev/at1	1.2MB B drive
0x021c	/dev/fd0	1.44MB A drive
0x021d	/dev/fd1	1.44MB B drive

### 17.7.2 Creating a File System

For the hard disk Image file created above, we must also partition and create a file system on it before it can be used. The usual practice is to attach the hard disk Image file to be processed to the existing simulation system (such as SLS Linux mentioned above) under Bochs , and then use the commands in the simulation system to process the new Image file. The following assumes that you have installed the SLS Linux emulation

1. 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。
2. Create a subdirectory named Linux-0.12 in the SLS-Linux directory and move the hdc.img file to it.
3. Go to the SLS-Linux directory and edit the Bochs configuration file 'bochsrc.bxrc' for the SLS Linux system. Add the configuration parameter line of our hard disk Image file under the option 'ata0-master':

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

4. Exit the editor. Double-click the icon for the 'bochsrc.bxrc' file to run the SLS Linux emulation system. Type 'root' at the Login prompt and press Enter. If Bochs does not work properly at this time, generally because the configuration file information is incorrect, please re-edit the configuration file.
5. Use fdisk to create 1 partition in the hdc.img file. Below is the sequence of commands to create the first partition. The process of creating another three partitions is similar. Since the partition type established by SLS Linux by default is 81 type (Linux/MINIX) that supports the MINIX2.0 file system, you need to use the fdisk t command to change the type to 80 (Old MINIX) type. Please note here that we have hooked hdc.img to the second hard drive under the SLS Linux system. According to the Linux 0.1X hard disk naming rules, the overall device name of the hard disk should be /dev/hd5. However, since the Linux kernel version 0.95, the naming rules for the hard disk have been changed to the currently used rules, so the device name of the second hard disk under SLS Linux is /dev/hdb.

---

```
[/]# fdisk /dev/hdb コマ  
ンド (m for help): n コマ  
ンドアクション  
e   extended  
p   primary partition (1-4)  
p  
パーティション番号 (1-4) : 1  
第1シリンドー (1~520) : 1  
最後のシリンドーまたは+サイズまたは+サイズMまたは+サイズK (1-520) 。 +63M  
  
コマンド(m for help): t パー  
ティション番号(1-4): 1  
ヘックスコード (Lを入力するとコードが一覧表示されます) 。 L
```

16進コード (Lを入力するとコードが表示されます) : 80 コマンド (mを入力するとヘルプが表示されます) : p  
ディスク /dev/hdb: 16 ヘッド、63 セクタ、520 シリンダ 単位 = 1008 \* 512 バイトのシリンド

Device	Boot	Begin	Start	End	Blocks	Id	System
--------	------	-------	-------	-----	--------	----	--------

```
/dev/hdb1      1      1    129   65015+ 80 Old MINIX
```

```
コマンド (mはヘルプ) :w  
パーティションテーブルが変更されています。  
再起動してから操作してください。[/]#
```

6. Remember the number of data blocks in this partition (here is 65015), which is used when creating the file system. When the partition is set up, it is necessary to restart the system as usual, so that the SLS Linux kernel can correctly identify the newly added partition.
7. After entering the SLS Linux emulation system again, we use the mkfs command to create the MINIX file system on the first partition we just created. The commands and information are as follows. This creates a partition with 64,000 data blocks (one block of data is 1 KB).

---

```
[/]# mkfs /dev/hdb1 64000  
21333 inodes  
64000 ブロック  
Firstdatazone=680 (680)  
Zonesize=1024  
Maxsize=268966912  
[/]#
```

---

At this point, we have completed the work of creating a file system in the first partition of the hdc.img file. Of course, creating a file system can also be established when running the root file system on a Linux 0.12 floppy disk. Now we can build this partition into a root file system.

### 17.7.3 Bochs Configuration File for Linux-0.12

BochsでLinux 0.12システムを運用する場合、通常は設定ファイルbochsrc.bxrcに以下の設定が必要となります。

---

```
romimage: file=$BXSHARE/BIOS-bochs-latest  
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-  
latest megs: 16  
floppya: 1_44="bootimage-0.12", status=inserted  
ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=520, heads=16, spt=63 boot:  
a  
log: bochsout.txt panic:  
action=ask #error:  
action=report #info:  
action=report #debug:  
action=ignore ips:  
1000000  
マウス: enabled=0
```

We can copy the Bochs configuration file bochsrc.bxrc from SLS Linux into the Linux-0.12 directory and

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。

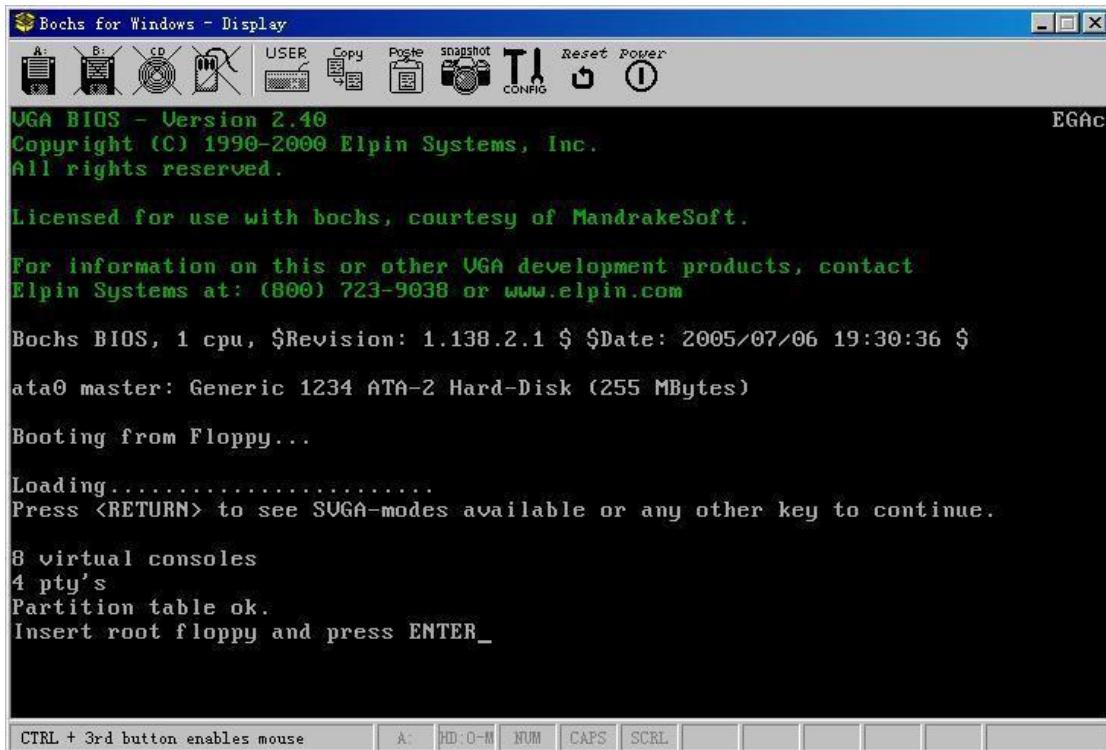


図 17-5 Bochs システムの実行ウィンドウ

At this point, you should click the A: floppy disk icon on the window menu bar, and configure the A disk as the rootimage-0.12 file in the dialog box. Or use the Bochs configuration window to set it up by clicking the 'CONFIG' icon on the menu bar to enter the Bochs settings window (you need to click the mouse to bring the window to the front). The contents of the setting window display are shown in Figure 17-6.

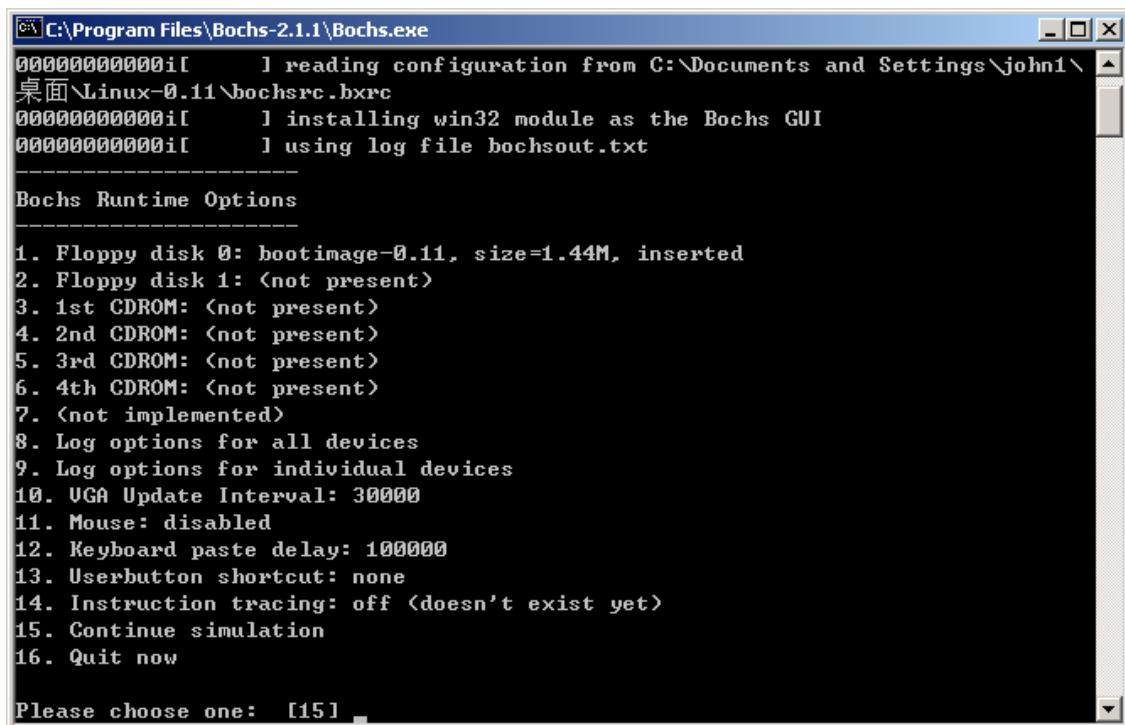


図 17-6 Bochs システム・コンフィギュレーション・ウィンドウ

Modify the floppy disk setting of item 1 to point to the rootimage-0.12 disk. Then press the Enter key continuously until the last line of the setup window displays 'Continuing simulation'. At this point, switch to the Bochs Run window and click Enter to formally enter the Linux 0.12 system, as shown in Figure 17-7.

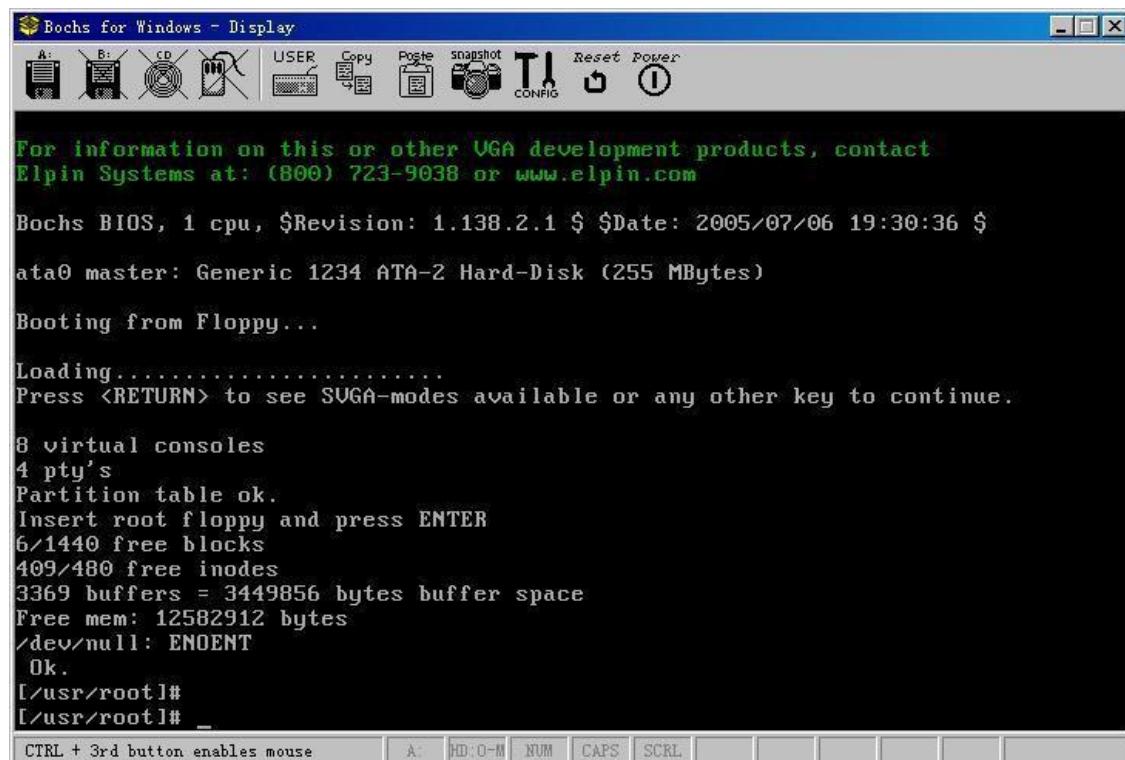


図17-7 Bochsで動作するLinux 0.12システム

### 17.7.4 Establishing a Root File System in hdc.img

Since the floppy disk capacity is too small, if you want the Linux 0.12 system to really do something, you need to create a root file system on the hard disk (here, the hard disk Image file). In the previous section, we have created a 256MB hard disk image file hdc.img, and it is already connected to the running Bochs environment, so a message about the hard disk appears in the figure above:

---

```
"ata0 マスター。汎用1234 ATA-2ハードディスク (255Mバイト) "
```

---

If you do not see this message, your Linux 0.12 configuration file is not set correctly. Please re-edit the bochssrc.bxrc file and re-run the Bochs system until the same screen as above appears. Now, we have previously created the MINIX filesystem on the first partition of hdc.img. If you haven't built it yet or want to try it again, then type the command to create a 64MB file system:

---

```
[/usr/root]# mkfs /dev/hd1 64000
```

---

Now we can start loading the file system on the hard disk. Execute the following command to load the new file system into the /mnt directory.

---

```
[/usr/root]# cd /
[/]# mount /dev/hd1 /mnt
```

---

After loading the file system on the hard disk partition, we can copy the root file system on the floppy disk to the hard disk. Please execute the following command:

---

```
[/]# cd /mnt
> [/mnt]# for i in bin dev etc usr tmp
>   do
>     cp -recursive +verbose /$i $i
>   done
```

---

この時点で、フロッピーのルートファイルシステムにあるすべてのファイルが、ハードディスクのファイルシステムにコピーされます。コピーの過程では、以下のような多くの情報が表示されます。

---

```
/usr/bin/mv -> usr/bin/mv
/usr/bin/rm -> usr/bin/rm
/usr/bin/rmdir -> usr/bin/rmdir
/usr/bin/tail -> usr/bin/tail
```

---

```
/usr/bin/more -> usr/bin/more  
/usr/local -> usr/local  
/usr/root -> usr/root  
/usr/root/.bash_history -> usr/root/.bash_history  
/usr/root/a.out -> usr/root/a.out  
/usr/root/hello.c -> usr/root/hello.c  
/tmp ->  
tmp  
[/mnt]#.
```

Now that you have built a basic root filesystem on your hard drive. You can view it anywhere in the new file system, then unmount the hard disk file system and type 'logout' or 'exit' to exit the Linux 0.12 system. The following messages will be displayed:

---

```
[/mnt]# cd /  
[/]# umount /dev/hd1  
[/]# logout  
Child 4 died with code 0000
```

---

## 17.7.5 Using the Root File System on the Hard Disk Image

Once you have created a filesystem on your hard disk image file, you can have Linux 0.12 launch it as the root filesystem. This can be done by modifying the contents of the 509th, 510th byte (0x1fc, 0x1fd) of the bootimage-0.12 file. Please follow the steps below:

1. First copy the two files bootimage-0.12 and bochssrc.bxrc to generate the bootimage-0.12-hd and bochssrc-hd.bxrc files.
2. Edit the bochssrc-hd.bxrc configuration file, change the file name on the 'floppya:' option to 'bootimage-0.12-hd', and save it;
3. Edit the bootimage-0.12-hd binary with Notepad++ or any other binary editor and modify the 509th and 510th bytes (ie 0x1fc, 0x1fd). The original value should be 00, 00, modified to 01, 03, indicating that the root file system device is on the first partition of the hard disk Image, and then save and exit. If you have the file system installed on another partition, you will need to modify the first byte to correspond to your partition.

---

```
000001f0h: 00 00 00 00 00 00 00 00 00 00 00 00 01 03 55 AA ;..... U?
```

---

Now you can double-click on the icon of the bochssrc-hd.bxrc file. The Bochs system should quickly enter the Linux 0.12 system and display the graphics in Figure 17-8.

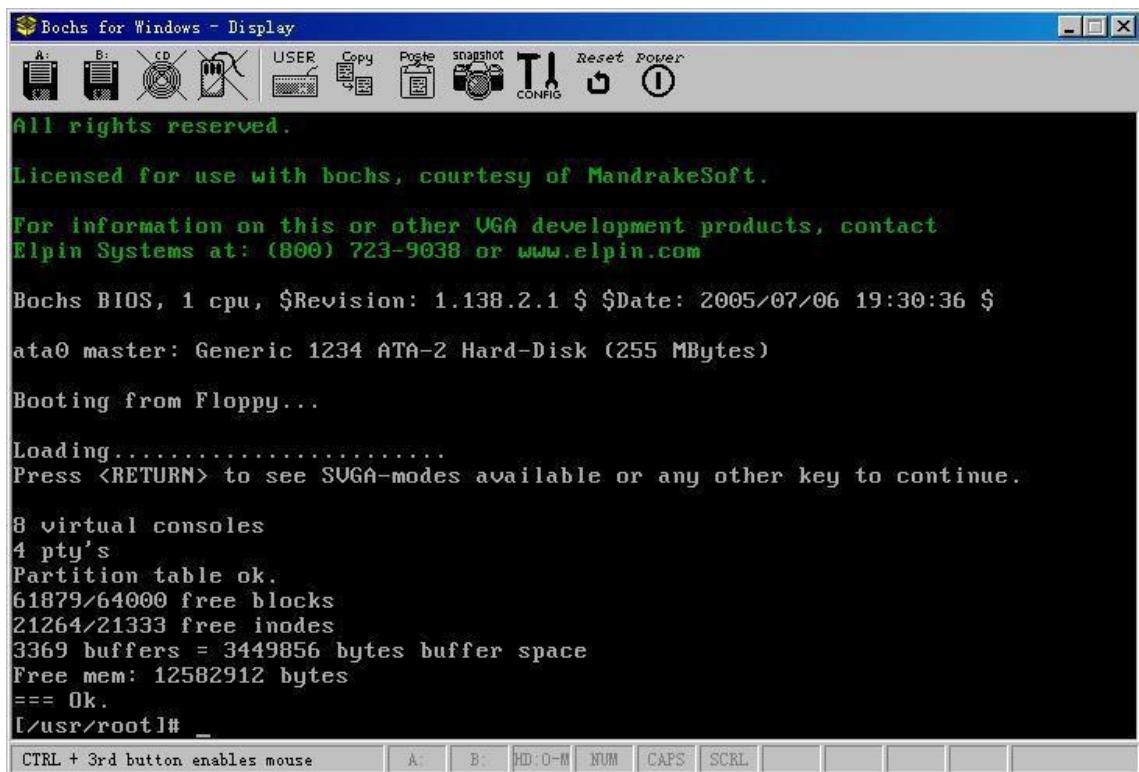


図17-8 ハードディスク・イメージ・ファイルのファイル・システムの使用

## 17.8 Compile Kernel on Linux 0.12 System

The author has reorganized a Linux 0.12 system package with the gcc 1.40 build environment. The system is set up to run under the Bochs simulation system and the corresponding bochs configuration file has been configured. This package is available from the following address:

<http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip>

The package contains a README file that explains the purpose and use of all the files in the package. If you have a bochs system installed on your system, you can run this Linux 0.12 by simply double-clicking the icon of the configuration file bochsrc-0.12-hd.bxrc. It uses the hard disk Image file as the root file system. After running the system, type the 'make' command in the /usr/src/linux directory to compile the Linux 0.12 kernel source code and generate the boot image file 'Image'. If you need to output this Image file, you can first backup the bootimage-0.12-hd file, and then use the following command to replace bootimage-0.12-hd with the new boot file. Now restart Bochs directly, you can use the bootimage-0.12-hd generated by the new compiler to boot the system.

---

```
[/usr/src/linux]# make
[~/usr/src/linux]# dd if=bootimage192 of=bootimage192
```

---

You can also use the mtools command to write the newly generated Image file to the second floppy image file 'diskb.img', and then use the tool software WinImage to extract the 'Image' file in 'diskb.img'.

---

```
[/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:
ドライブBの B:/ボリュームは
Bです。
Directory for
GCCLIB-1 TAZ      934577   3-29-104  7:49p
IMAGE          121344   4-29-104  11:46p
SYSTEM MAP       17162    4-29-104  11:47p
README           764     3-29-104  8:03p
4 File(s)      382976 bytes free
[/usr/src/linux]#
```

---

If you want to use the new boot image file with the root file system rootimage-0.12 on the floppy disk, first edit the Makefile before compiling, and comment out the 'ROOT\_DEV=' line with '#'.

通常、カーネルをコンパイルする際には非常にスムーズに行われます。考えられる問題は、gcc コンパイラが「-mstring-ins」というオプションを認識しないことです。このオプションは、Linus が gcc 1.40 コンパイラを単体でコンパイルする際に実装した拡張実験パラメータです。このオプションは、文字列命令を生成する際に gcc を最適化するために使用されます。この問題を解決するには、すべてのMakefileでこのオプションを直接削除し、カーネルを再コンパイルします。また、「gar」コマンドが見つからないという問題もあります。この場合、/usr/local/bin/ の下にある 'ar' を直接リンクするか、'gar' にコピー/リネームしてください。

## 17.9 Compile kernel under Redhat system

オリジナルのLinuxオペレーティングシステムのカーネルは、Minix 1.5.10オペレーティングシステムの拡張版であるMinix-i386上でクロスコンパイルされ開発された。Minix 1.5.10オペレーティングシステムは、A.S.Tanenbaumの「Design and Implementation of Minix」の初版とともにPrentice Hall Publishing Companyからリリースされました。このバージョンのMinixは、80386およびその互換性のあるマイクロコンピュータ上で動作可能ですが、80386の32ビット保護メカニズムを利用ていません。このシステムで32ビットのオペレーティングシステムを開発するために、リナスはブルース・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bootimageを生成する方法を簡単に紹介します。読者は、通常のPCでも、Bochsなどの仮想マシンソフトウェアでも実行できます。ここでは、主な修正点のみを示します。すべての修正点は、ツールdiffを使って、修正されたコードと修正されていないコードを比較して、次のことがわかります。

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
```

---

The file 'dif.out' contains all the modified places in the source code. The Linux 0.1X kernel source code that has been modified and can be compiled under RedHat 9 can be downloaded from the following address:

<http://oldlinux.org/Linux.old/kernel/linux-0.11-040327-rh9.tar.gz>

---

When booting with the compiled boot image file, the following information should appear on the screen:

---

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

Note that if there is no response after displaying "Loding system...", this means that the kernel does not recognize the hard disk controller subsystem in the computer. At this point, you can use VirtualBox, VMware, bochs and other virtual machine software to test. When you are asked to insert the root file system disk, if you press the Enter key directly, the following information that cannot be loaded the root file system will be displayed and it will crash. To run the Linux 0.1X operating system completely, you need a matching root file system, which can be downloaded from the oldlinux.org website.

### 17.9.1 Modifying the Makefile

- a. 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。
  - b. Rename 'gas' to 'as', 'gld' to 'ld'. Because now 'gas' and 'gld' have been directly renamed to 'as' and 'ld'.
  - c. 'as' (original gas) has no need to use the '-c' option, so the -c compilation option needs to be removed from the Makefile in the kernel home directory Linux.
  - d. Remove gcc's compile flag options: '-fcombine-reg', '-mstring-insns', and these two options in the Makefile in all subdirectories. The '-fcombine-reg' option was not found in the 1994 gcc manual, and '-mstring-insns' is an option that Linus added to gcc modifications, so this optimization option is definitely not included in your gcc.
  - e. In the gcc compile option, add the '-m386' option. In this way, the kernel image file compiled under RedHat 9 will not contain the instructions of the CPU of 80486 and above, so the kernel can run on the 80386 machine.

## 17.9.2 Modifying Comments in the Assembly Language Programs

17.9.3 as86コンパイラは、cコメント文を認識しないので、boot/bootsect.sファイルの中で、「!」を使ってCコメントをコメントアウトする必要があります。

## 17.9.4 Modifying the align value of the memory alignment statement

ブートディレクトリにある3つのアセンブリ言語プログラムでは、「align」ステートメントの使い方が変わっています。オリジナルの'align'の後の値は、メモリ位置の累乗値を参照していますが、現在では、整数アドレスの値を直接与える必要があります。そのため、オリジナルのステートメントである  
.align 3

---

Need to be modified to ( $2^3=8$ ):

.align 8

---

## 17.9.5 Modifying Inline Macro Assembly Language Programs

Due to the continuous improvement of the as assembler, it is now more and more automated, so there is no need to manually specify the CPU register to be used for a variable. Therefore all '\_asm\_("ax")' in the kernel code needs to be removed. For example, on lines 20 and 26 of the fs(bitmap.c file, on line 68 of the fs/namei.c file.

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");

---

All registers need to be removed, leaving only the colon and the right parenthesis: ":)";.

17.9.6 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ため、場所によっては、修正されるレジスタの内容を削除するとgccの最適化エラーが発生します。そのため、プログラムコードの中には、include/string.hファイルのmemcpy()定義の342行目のように、状況に応じてこれらの宣言の一部を残しておく必要がある場所があります。

## 17.9.7 Reference representation of C variables in assembly statements

Linux 0.1X カーネルの開発に使用されているアセンブリでは、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

---

Need to be changed to:

---

.globl idt,gdt,pg dir,tmp floppy area

---

The variable name "\_stack\_start" on the 31st line statement needs to be modified to "stack\_start".

### 17.9.8 Debug Display Function in Protected Mode

プロテクトモードに入る前は、ROM BIOSのint 0x10コールを使って画面に情報を表示することができます。しかし、プロテクトモードに入った後は、これらの割り込みコールは使用できません。プロテクトモード環境でのカーネルの内部データ構造や状態を把握するためには、以下の関数check\_data32()を使ってカーネルデータを表示することができます（以前、oldlinux.orgフォーラムの友人「notrump」が提供してくれました）。カーネルにはprintk()という表示関数がありますが、これはtty\_write()を呼び出す必要があり、カーネルが完全に機能していないときには利用できません。プロテクトモードに入った後、このcheck\_data32()関数は、興味のあるものを画面に表示することができます。ページ機能を有効にしてもしなくとも、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のエントリを使うだけで、ページテーブルディレクトリは物理アドレス0から始まり、さらにカーネルデータセグメントのベースアドレスは0なので、4Mの範囲では、仮想メモリ、リニアメモリ、物理メモリのアドレスは同じになるからだ。リーナス氏はそもそもこのことを考慮して、この設定の方が使い勝手が良いと感じているのだろう。

---

```
/*
 * Purpose: Display a 32-bit integer in hexadecimal on the screen.
 * Params: value -- the integer to display.
 *          pos -- The screen position, in units of 16 chars wide, for example, 2, which means
 *                  that the display starts at the width of 32 chars from the upper left corner.
 * Return: None.
 * If you want to use it in an assembly language program, make sure that the function is compiled
 * and linked into the kernel. The usage in gcc assembly is as follows:
 * pushl pos           // 'pos' should be replaced with your actual data, such as pushl $4
 * pushl value         // 'pos' and 'value' can be any legal addressing method.
 * call check_data32
 */
inline void check_data32(int value, int pos)
{
    __asm__ volatile__ (
        "shl    $4, %%ebx\n\t"           // %0 -the value to be displayed; EBX -screen position pos.
        "addl   $0xb8000, %%ebx\n\t"     // Multiply the pos by 16, plus VGA memory start address,
        "movl   $0xf0000000, %%eax\n\t"  // get the position from top left of the screen in EBX.
        "movb   $28, %%cl\n\t"          // Set a 4-bit mask.
        "1:\n\t"                      // Set the initial right shift bit value.
        "movl   %0, %%edx\n\t"          // Put the displayed value to EDX.
        "andl   %%eax, %%edx\n\t"        // Take 4 bits specified by EAX in EDX.
        "shr    %%cl, %%edx\n\t"         // Shift 28 bits to right, EDX is the value of 4 bits taken.
        "add    $0x30, %%dx\n\t"         // Convert this value to ASCII code.
        "cmp    $0x3a, %%dx\n\t"         // If less than 10, jumps forward to label 2.
    );
}
```

```
"jb2f\n\t"  
"add    $0x07, %%dx\n"           // Otherwise add 7 and convert the value to A-F.
```

```
"2:n%">
"add    $0x0c00, %%dx\n\t"          // Set the display attributes.
"movw  %%dx, (%ebx)\n\t"          // Put this value in the display memory.
"sub   $0x04, %%cl\n\t"          // Prepare the next hex number, the number of shifts minus 4.
"shr   $0x04, %%eax\n\t"          // The mask is shifted to the right by 4 bits.
"add   $0x02, %%ebx\n\t"          // Update the display memory position.
"cmpl  $0x0, %%eax\n\t"          // The mask has moved out of the right (8 hexs displayed)?
"jnz1b\n"                         // No, there still numbers to be displayed, jump to lable 1.
:: "m"(value)、"b"(pos)) .
}
```

---

## 17.10 Integrated Boot Disk and Root FS

This section explains how to make an integrated disk image file that is a combination of a kernel boot image file and a root file system. Its main purpose is to understand the working principle of the Linux 0.1X kernel memory virtual disk, and further understand the concept of the boot disk and the root file system disk, and deepen the understanding of the kernel/blk\_drv/ramdisk.c program running method. In fact, the boot module, kernel module, and file system module image structure stored in Flash in a typical embedded system is similar to the integrated disk here.

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/hochs/linux-0.11-devel-040923.zip>

'linux-0.11-devel' is a Linux 0.11 system with a development environment running under Bochs. The 'rootimage-0.11' is a Linux 0.11 root file system in a 1.44MB floppy image file. The suffix 'for-orig' refers to the kernel boot image file that is compiled for the unmodified Linux 0.11 kernel source code. Of course, the "unmodified" mentioned here means that there has not been any major changes to the kernel, because we still need to modify the compiled configuration file Makefile to compile the kernel code containing the memory virtual disk.

### 17.10.1 Integrated disk building principle

通常、Linux 0.1Xシステムをフロッピーディスク（ここでいうディスクとは、フロッピーディスクに対応するイメージファイルのこと）で起動する際には、カーネルブートディスクとルートファイルシステムディスクの2つのディスクが必要になります。このため、基本的なLinuxシステムを動作させるためには、システムを起動するために2つのディスクが必要となり、実行時にはルートファイルシステムディスクをフロッピーディスクドライブに残しておく必要がある。ここで紹介する統合ディスクは、カーネルブートディスクと基本的なルートファイルシステムディスクの内容を1枚のディスクにまとめたものです。このようにして、Linuxを起動することができます。

0.1Xシステムでは、1枚の統合ディスクを使ってコマンドプロンプトを表示します。この統合ディスクは、実際にはルートファイルシステムを持つカーネルブートディスクです。

### 7.10.1.1

統合ディスクシステムを動作させるためには、ディスク上のカーネルコードでメモリ仮想ディスク (RAMDISK) の機能をオンにする必要がある。統合ディスク上のルートファイルシステムをメモリ上の仮想ディスクにロードすることで、システム上の2つのフロッピードライブを他のファイルシステムディスクのマウントなどのために解放することができます。以下、1.44MBのディスクに統合ディスクを作成する原理と手順を詳しく紹介します。

#### 17.10.1.2 Principle of the boot process

Linux 0.1X カーネルは、コンパイル時の Makefile で設定された RAMDISK オプションに応じて、システムの物理メモリに仮想ディスク領域を確保するかどうかを判断します。RAMDISK が設定されていない（サイズが 0）場合、カーネルは ROOT\_DEV で設定されたルートファイルシステムのデバイス番号に従って、フロッピーディスクまたはハードディスクからルートファイルシステムをロードし、仮想ディスクがない場合の一般的な起動処理を行う。

Linux 0.1X のカーネルソースコードをコンパイルする際に、その linux/makefile に RAMDISK のサイズが定義されている場合、カーネルコードは起動後、RAMDISK のメモリ領域を初期化した後、まず 256 番目のディスクブロックから起動ディスクを検出しようとします。ルートファイルシステムがあるか？ 検出方法は、257 番目のディスクブロックに有効なファイルシステムのスーパーブロックがあるかどうかを確認します。もしあれば、そのファイルシステムが RAMDISK 領域にロードされ、ルートファイルシステムとして使用されます。そのため、ルートファイルシステムを統合した起動ディスクを使用して、システムをシェルコマンドプロンプトに起動することができます。有効なルートファイルシステムが起動ディスクの指定されたディスクブロックの位置（256 番目のディスクブロックから始まる）に格納されていない場合、カーネルはルートファイルシステムのディスクを挿入するよう促します。ユーザーが Enter キーを押して確認すると、カーネルは独立したディスク上のルートファイルシステムを読み取り、それをメモリの仮想ディスク領域に読み込む。この検出と読み込みのプロセスを図9-7に示す。

#### 17.10.1.3 Structure of the integrated disk

Linux 0.1X のカーネルでは、コード + データセグメントのサイズが非常に小さく、120KB～160KB程度となっています。Linux システム開発の初期段階では、カーネルの拡張を考慮しても、リーナス氏はカーネルのサイズが 256KB を超えることはないと考えているので、1.44MB の起動ディスクの 256 番目のディスクブロックの先頭に、基本的なルートファイルシステムを格納することができます。これにより、統合されたディスクとなります。基本的なルートファイルシステムを追加した起動ディスク（すなわち統合ディスク）の模式図を図17-9 に示す。ファイルシステムの詳細な構造については、「ファイルシステム」の章の記述を参照してください。

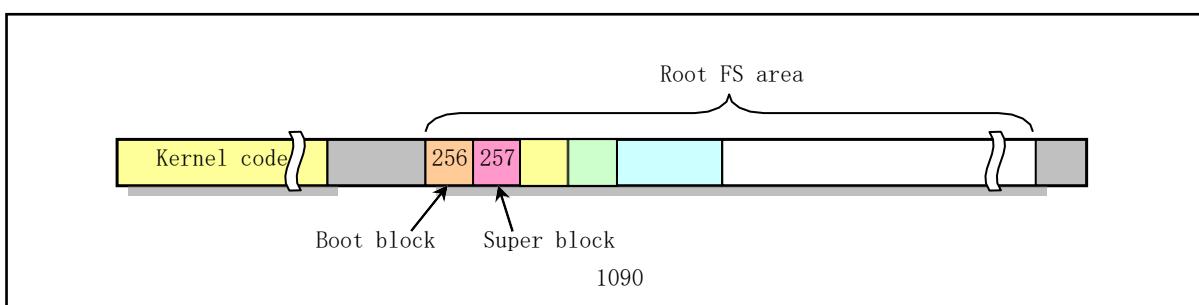


図17-9 統合ディスクのコード構造

As mentioned above, the location and size of the root file system placement on the integrated disk is primarily related to the length of the kernel and the size of the defined RAMDISK area. Mr. Linus defines in the ramdisk.c program that the root file system's starting placement position is the beginning of the 256th disk

## 7.10.2

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。

### 17.10.3 Integrated disk construction process

#### 7.10.2.1

カーネルプログラムramdisk.cのデフォルトのディスクブロックの位置を変えずに、統合ディスクのルートファイルシステムは1024KB（最大1184KB）必要だと仮定します。統合ディスクの作成の主旨は、まず1.44MBの空のイメージディスクファイルを作成し、RAMDISK機能で新たにコンパイルしたカーネルイメージファイルをディスクの先頭にコピーします。次に、1024KB以下のカスタマイズされたファイルシステムを、ディスクの256番目のディスクブロックの先頭にコピーする。具体的な構築手順は以下の通りです。

#### 17.10.2.2 Recompiling the kernel

RAMDISK領域が2048KBに設定されていると仮定して、RAMDISK定義を持つカーネルImageファイルを再コンパイルします。方法は、linux-0.1XシステムをBochsで実行します。usr/src/linux/Makefileファイルを編集し、以下の設定行を修正します。

---

```
ramdisk=-dramdisk=2048
```

---

Then recompile the kernel source code to generate a new kernel image file:

---

```
make clean; make
```

---

#### 17.10.2.3 Making a Temporary Root File System

Make a root file system Image file with a size of 1024KB, and now assume its file name is 'rootram.img'. The Bochs system is run during the building process using a configuration file (bochsrc-hd.bxrc) with a hard disk Image. The building method is as follows:

- (1) Make an empty Image file of 1024KB in size using the method described earlier in this chapter. The name of the file is specified as 'rootram.img'. You can use the following command to generate under the current Linux system:

---

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

---

- (2) Run the linux-0.1X system in Bochs. Then configure the driver disks in the main Bochs window: disk A is rootimage-0.1X (0.11 kernel is rootimage-0.11-origin); disk B is rootram.img.
- (3) Use the following command to create an empty file system of size 1024KB on the rootram.img disk. Then mount the A and B disks to the /mnt and /mnt1 directories respectively. If the directory /mnt1 does not exist, you can create one.

---

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

- (4) Use the 'cp' command to selectively copy /mnt files from rootimage-0.1X to the /mnt1 directory and create a root filesystem in /mnt1. If you encounter any error message, then the content is usually more than 1024KB. First reduce the files in /mnt/ to meet the capacity requirements of no more than 1024KB. We can remove some files under /bin and /usr/bin to achieve this. Regarding capacity, we can use the 'df' command to view it. For example, the files we can choose to keep are the following:

---

```
[/mnt/bin]# ll
total 495
-rwx--x--x 1 root root 29700 Apr 29 20:15 mkfs
-rwx--x--x 1 root root 21508 Apr 29 20:15 mknod
-rwx--x--x 1 root root 25564 Apr 29 20:07 mount
-rwxr-xr-x 1 root root 283652 Sep 28 10:11 sh
-rwx--x--x 1 root root 25646 Apr 29 20:08 umount
-rwxr-xr-x 1 root 4096 116479 Mar 3 2004 vi

[/mnt/bin]# cd /mnt/usr/bin
[/mnt/usr/bin]# ll
合計 364 1 root root 29700 Jan 15 1992 cat
-rwxr-xr-x 1 root root 29700 Mar 4 2004 chmod
-rwxr-xr-x 1 root root 33796 Mar 4 2004 chown
-rwxr-xr-x 1 root root 37892 Mar 4 2004 cp
-rwxr-xr-x 1 root root 29700 Mar 4 2004 dd
-rwx--x--x 1 root 4096 36125 Mar 4 2004 df
-rwx--x--x 1 root root 46084 Sep 28 10:39 ls
-rwxr-xr-x 1 root root 29700 Jan 15 1992 mkdir
-rwxr-xr-x 1 root root 33796 Jan 15 1992 mv
-rwxr-xr-x 1 root root 29700 Jan 15 1992 rm
-rwxr-xr-x 1 root root 25604 Jan 15 1992 rmdir

[/mnt/usr/bin]#.
```

- (5) Then use the following command to copy the file. In addition, you can modify the contents of /mnt/etc/fstab and /mnt/etc/rc as needed. At this point, we have created a file system with a size of 1024KB or less in fd1(/mnt1/).

---

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

---

- (6) Use the 'umount' command to unmount the filesystems on /dev/fd0 and /dev/fd1, then use the 'dd' command to copy the filesystem from /dev/fd1 to the Linux-0.1X system and create a name called rootram-0.1X root file system Image file:

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

---

At this time, in the Linux-0.1X system under Bochs, we have a newly compiled kernel image file /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。

#### 17.10.2.4 Creating an Integrated Disk

ここで、上記2つのイメージファイルを組み合わせて統合ディスクを作成します。BochsのメインウインドウでAディスクの設定を変更し、先に用意したbootroot-0.1Xという1.44MBのイメージファイルに設定します。そして、以下のコマンドを実行します。

```
dd bs=8192 if=/usr/src/linux/Image of=/dev/fd0 dd  
of=/dev/fd0 seek=256
```

---

The option 'bs=1024' means that the size of the definition buffer is 1KB; 'seek=256' means that the first 256 disk blocks are skipped when the output file is written. Then exit the Bochs system. At this point, we get a running integrated disk image file bootroot-0.1X in the current directory of the host.

#### 17.10.4 Running the Integrated Disk System

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

```
floppya: 1 44=bootroot-0.1X
```

---

Then double-click the configuration file with the mouse to run the Bochs system. At this point, the results should be as shown in Figure 17-10.

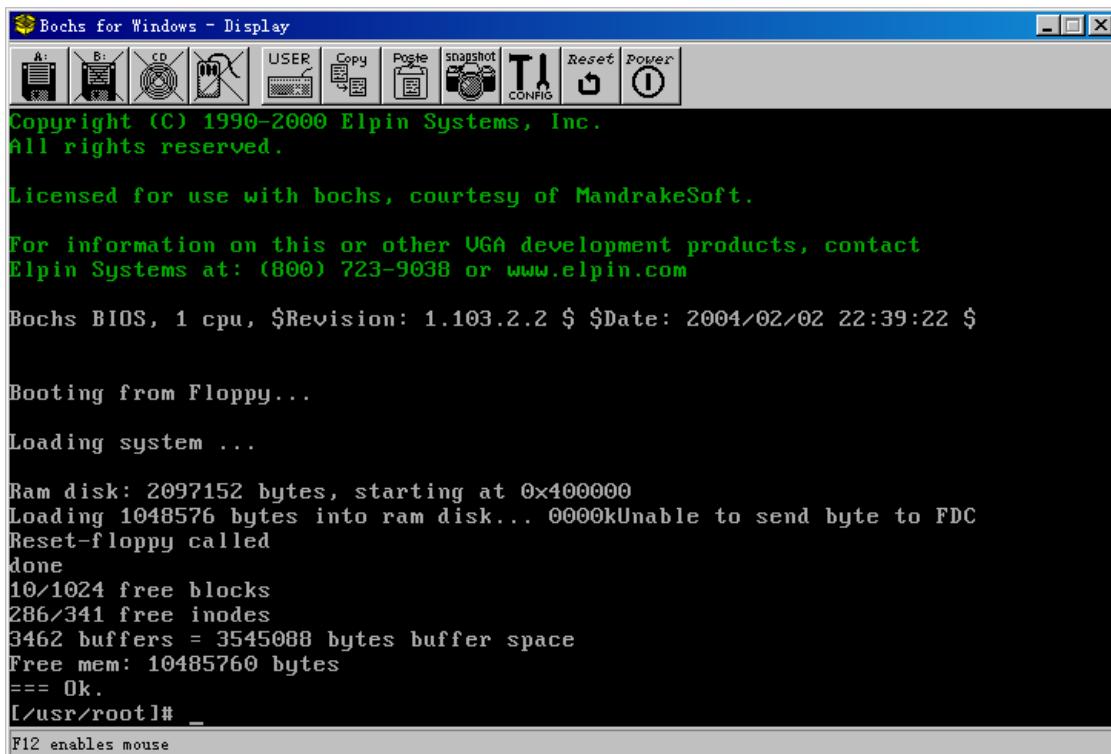


図17-10 統合ディスク実行インターフェース

In order to facilitate the experiment, you can also download the integrated disk software of 0.11 kernel that is ready and can run immediately from the following website:

<http://oldlinux.org/Linux.old/bochs/bootroot-0.11-040928.zip>

## 17.11 Debugging Kernel Code with GDB and Bochs

This section explains how to use the Bochs emulation environment and gdb tools to debug Linux 0.1X kernel source code on existing Linux systems such as RedHat or Fedora. Before using this method, the X window system should already be installed on the existing Linux system. Since the Bochs executable in the RPM installation package provided by the Bochs website does not have the 'gdbstub' module that communicates with the gdb debugger, we need to download the Bochs source code to compile the running program with this module.

### 7.11.1

「gdbstub」モジュールは、Bochsプログラムがローカルの1234ネットワークポート上でgdbからのコマンドを待ち受け、コマンド実行結果をgdbに送信することを可能にします。つまり、gdbを使ってLinux0.1XカーネルのC言語レベルのデバッグを行うことができるのです。もちろん、生成されたカーネルコードにデバッグ情報を持たせるためには、Linux0.1Xカーネルも「-g」オプションを付けて再コンパイルする必要があります。

#### 17.11.2 Compiling a Bochs System with gdbstub

Bochsユーザーマニュアルには、自分でBochsシステムをコンパイルする方法が書かれています。ここでは、gdbstubを使ってBochsシステムをコンパイルする方法と手順を紹介します。まず、最新の

Bochsシステムのソース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/>

---

Decompressing the package with 'tar' will generate a bochs-2.6 subdirectory in the current directory. After entering this subdirectory, run the configuration program 'configure' with the option "--enable-gdb-stub", then run 'make' and 'make install' as shown below:

---

```
[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
```

---

If we encounter some problems when running './configure' and cannot generate the Makefile used for compilation, this is usually caused by not installing the X window development environment software or related library files. At this point we must first install the necessary software and then recompile Bochs.

### 17.11.3 Compiling the Linux 0.1X Kernel with Debug Information

Bochs社のシミュレーション実行環境とシンボリック・デバッグ・ツールgdbをリンクすることで、Linux 0.1X系でコンパイルされたデバッグ情報付きカーネルモジュールを使ってデバッグすることも、RedHat環境でコンパイルされた0.1Xカーネルモジュールを使ってデバッグすることも可能になります。どちらの環境でも、0.1XカーネルのソースディレクトリにあるすべてのMakefileを修正し、コンパイルフラグラインに「-g」オプションを追加し、リンクフラグラインの「-s」オプションを削除する必要があります。

---

```
LDFLAGS = -M -x                                     // Remove '-s' flag.
CFLAGS  =-Wall -O -g -fomit-frame-pointer \          // Add '-g' flag.
```

---

After entering the kernel source directory, we can use the 'find' command to find all the following Makefiles that need to be modified:

---

```
[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]#.

In addition, since the kernel code module compiled at this time contains debugging information, the system module size may exceed the default maximum value of the write kernel code image file SYSSIZE = 0x3000 (defined in line 7 of the boot/bootsect.s file). At this point, we can modify the rules of the Image file generated in the Makefile in the root directory of the source code by removing the symbol information in the kernel module 'system' and then writing it to the Image file. The original 'system' module with the symbol information is reserved for use by the gdb debugger. Note that the implementation command for the target in the Makefile needs to start with a tab.

---

```
Image: boot/bootsect boot/setup tools/system  
tools/build cp -f tools/system system.tmp  
strip system.tmp  
tools/build boot/bootsect boot/setup system.tmp $(ROOT_DEV) $(SWAP_DEV) >  
Image rm -f system.tmp  
sync
```

---

Of course, we can also modify the SYSSIZE value in boot/bootsect.s and tools/build.c to 0x8000 to handle this situation.

#### 17.11.4 Debugging methods and steps

##### 7.11.4.1

以下では、最新のLinuxシステム（RedHatやFedoraなど）でコンパイルされたカーネルコードと、Bochsで動作するLinux 0.1Xシステムでコンパイルされたカーネルコードに応じた、デバッグ方法と手順を説明します。Linux 0.11のカーネルコードのデバッグ方法と手順を以下に示します。0.12カーネルのデバッグ方法と手順は全く同じです。

##### 17.11.4.2 Debugging the Linux 0.11 kernel compiled on modern Linux

Linux 0.11のカーネルソースのルートディレクトリがlinux-rh9-gdb/だとすると、まずこのディレクトリにあるすべてのMakefileを上記の方法で修正し、その中にBochs設定ファイルを作成して、カーネルをサポートするルートファイルシステムのイメージファイルをダウンロードします。また、設定されている以下のパッケージをWebサイトから直接ダウンロードして実験を行います。

---

<http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-rh9-050619.tar.gz>

---

After unpacking this package with the command "tar zxvf linux-gdb-rh9-050619.tar.gz", you can see that it contains the following files and directories:

---

```
[root@plinux linux-gdb-rh9]# ls -l  
total 1600  
-rw-r--r--    1 root      root        18055 Jun 18 15:07 bochssrc-fd1-gdb.bxrc  
drwxr-xr-x   10 root     root        4096 Jun 18 22:55 linux  
-rw-r--r--    1 root      root     1474560 Jun 18 20:21 rootimage-0.11-for-orig  
-rwxr-xr-x    1 root      root        35 Jun 18 16:54 run  
[root@plinux linux--gdb-rh9]#
```

---

The first file 'bochsrc-fd1-gdb.bxrc' is the Bochs configuration file, in which the file system image file 'rootimage-0.11-for-orig' has been set to be inserted in the second "floppy drive". The main difference between this Bochs configuration file and other Linux 0.1X configuration files is that the following line is added to the front of the file, indicating that when Bochs runs with this configuration file, it will listen for commands from the gdb debugger on the local network port 1234:

---

```
gdbstub: enabled=1, port=1234, text base=0, data base=0, bss base=0
```

---

The second item of linux/ is the Linux 0.11 source code directory, which contains the kernel source code files that have been modified for all Makefiles. The third file 'rootimage-0.11-for-orig' is the root file system image file that is associated with this kernel code. The fourth file 'run' is a simple script that contains a line of Bochs startup command. The basic steps to run this experiment are as follows:

1. Open two terminal windows under the X window system;
2. In one of the terminal windows, switch the working directory to the linux-gdb-rh9/ directory and run the program './run'. At this point, a message waiting for gdb to connect is displayed: "Wait for gdb connection On localhost:1234", and the system will create a Bochs main window (no content at this time);
3. In another terminal window, we switch the working directory to the kernel source directory linux-gdb-rh9/linux/ and run the command: "gdb tools/system";
4. Type the command "break main" and "target remote localhost:1234" in the window where gdb is run. At this time, gdb will display the information that has been connected to Bochs.
5. Execute the command "cont" in the gdb environment. After a while, gdb will show that the program stops at the main() function of init/main.c.

その後、gdbコマンドを使ってソースコードを観察し、カーネルをデバッグすることができます。例えば、ソースコードの観測には「list」コマンドを、オンラインヘルプ情報の取得には「help」コマンドを、他のブレークポイントの設定には「break」を、変数値の表示・設定には「print/set」を、シングルステップデバッグの実行には「Next/step」を、gdbの終了には「quit」コマンドを、といった具合です。gdbの具体的な使い方については、gdbのマニュアルを参照してください。以下に、gdbを起動し、その中で実行するコマンドの例を示します。

---

```
[root@plinux linux]# gdb tools/system // Start gdb to execute the system module.  
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) break main // Set a breakpoint at the main() function.  
Breakpoint 1 at 0x6621: file init/main.c, line 110.  
(gdb) target remote localhost:1234 // Connecting to Bochs.  
Remote debugging using localhost:1234
```

```
0x0000ffff in sys_mkdir (pathname=0x0, mode=0) at namei.c:481
```

```
481      namei.c: No such file or directory.
```

```
      in namei.c
(gdb) cont                                // Continue execution until the breakpoint.
```

継続中です。

```
Breakpoint 1, main () at init/main.c:110          // Stop running at the breakpoint.
110      ROOT_DEV = ORIG_ROOT_DEV;
(gdb) list                                         // View the source code.
105  {
106  /*
107   * Interrupts are still disabled. Do necessary setups, then
108   * enable them
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) next                                         // Single step execution.
111      drive_info = DRIVE_INFO;
(gdb) next                                         // Single step.
112      memory_end = (1<<20) + (EXT_MEM_K<<10);
(gdb) print /x ROOT_DEV                           // Print the variable ROOT_DEV.
$3 = 0x21d                                       // The second floppy device number.
(gdb) quit                                         // Exit gdb.
プログラムは動いています。とにかく終了しますか？(y or n)
y [root@plinux linux]#.
```

When debugging kernel source code in gdb, sometimes the problem that the source program does not find is displayed. For example, gdb sometimes displays "memory.c: No such file or directory". This is because when compiling mm/memory.c and other files, the Makefile indicates that the ld linker has linked the file module under mm/ to generate a relocatable module 'mm.o', and in the source code root directory linux / Down, it is again used as the input module for ld. Therefore, we can copy these files to the linux/ directory and re-execute the kernel debugging operation.

#### 17.11.4.3 Debug the 0.1X kernel compiled on Linux 0.1X system

1. 0.1Xシステムでコンパイルされたカーネルを、RedHatなどの最新のLinux OSでデバッグするには、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、mcopyコマンドを使ってBochsの2番目のフロッピーイメージファイルに書き込み、最後にWinImageソフトウェアまたはmountコマンドを使って圧縮ファイルを取り出すことができます。ファイルのコンパイルと抽出のプロセスの基本的な手順を以下に示す。
  2. Run Linux-0.1X under Bochs, enter the directory /usr/src/, and create the directory 'linux-gdb';
  3. Use the command to copy the entire 0.1X kernel source tree first: "cp -a linux linux-gdb/". Then enter the linux-gdb/linux/ directory, modify all Makefiles as described above, and compile the kernel;
  4. Go back to the /usr/src/ directory and use the 'tar' command to compress the linux-gdb/ directory to get the 'linux-gdb.tgz' file.
  5. Copy the compressed file to the second floppy (b drive) image file: "mcopy linux-gdb.tgz b:". If the b disk space is not enough, please use the delete file command "mdel b: file name" to make some space on the b disk.
  6. If the host environment is a Windows operating system, then use WinImage to extract the compressed

7. ホスト環境がRedhatやその他の最新のLinuxシステムの場合は、「mount」コマンドを使ってbディスクイメージファイルを読み込み、そこから圧縮カーネルファイルをコピーします。
8. Decompressing the copied compressed file on the modern Linux system will generate a linux-gdb/ directory containing the 0.1X kernel source tree. Go to the linux-gdb/ directory and create the bochs configuration file 'bochsrc-fd1-gdb.bxrc'. You can also take the contents of the 'bochsrc-fdb.bxrc' configuration file from the 'linux-0.11-devel' package and add the 'gdbstub' parameter line yourself. Then download the 'rootimage-0.11' root file system floppy image file from the oldlinux.org website, which is also saved in the linux-gdb/ directory.

その後は、前節の手順に従って、ソースコードのデバッグ実験を続けます。以下に、上記の手順の例を示します。ホスト環境をRedhat系とし、BochsのLinux 0.11系を実行するとします。

```
[/usr/root]# cd /usr/src                                // Enter the source code directory.  
[/usr/src]# mkdir linux-gdb                            // Create directory linux-gdb/  
[/usr/src]# cp -a linux linux-gdb/                      // Copy source code to linux-ddb/  
[/usr/src]# cd linux-gdb/linux                         // Modify the Makefiles.  
[/usr/src/linux-gdb/linux]# vi Makefile  
...  
[/usr/src/linux-gdb/linux]# make clean; make           // Compling the kernel.  
...  
[/usr/src/linux-gdb/linux]# cd ../../..  
[/usr/src]# tar zcvf linux-gdb.tgz linux-gdb          // Create compressed file.  
...  
[/usr/src]# mdir b:                                    // Check the contents of b disk.  
Volume in drive B is Bt  
B:/のディレクトリ  
LINUX-GD TGZ      827000   6-18-105 10:28p  
TPUT    TAR      184320   3-09-132  3:16p  
LILO    TAR      235520   3-09-132  6:00p  
SHOELA~1 Z      101767   9-19-104  1:24p  
SYSTEM  MAP      17771    10-05-104 11:22p  
      5 File(s)     90624 bytes free  
[/usr/src]# mdel b:linux-gd.tgz                     // Space not enough, delete some file.  
[/usr/src]# mcopy linux-gdb.tgz b:                  // Copy the linux-gdb.tgz to b disk.  
LINUX-GD.TGZをコ  
ピーする [/usr/src]#.
```

After closing the Bochs system, we get a compressed file named 'LINUX-GD.TGZ' in the b disk image file. The debug experiment directory can be established by using the following command sequence in the Redhat Linux host environment.

---

```
[root@plinux 0.11]# mount -t msdos diskb.img /mnt/d4 -o loop,r  // Mount b disk image file.  
[root@plinux 0.11]# ls -l /mnt/d4                           // Check contents.  
合計 1234  1 root    root      235520 Mar  9  2032 lilo.tar  
-rwxr-xr-x  
-rwxr-xr-x  1 root    root      723438 Jun 19  2005 linux-gd.tgz  
-rwxr-xr-x  1 root    root      101767 Sep 19  2004 shoela~1.z
```

```
-rwxr-xr-x    1 root      root      17771 Oct  5  2004 system.map
-rwxr-xr-x    1 root      root     184320 Mar  9 2032 tput.tar
[root@plinux 0.11]# cp /mnt/d4/linux-gd.tgz .                         // Copy the file.
[root@plinux 0.11]# umount /mnt/d4                                     // Unmount the b disk.
[root@plinux 0.11]# tar zxvf linux-gd.tgz                           // Untar the file.
...
[root@plinux 0.11]# cd linux-gdb
[root@plinux linux-gdb]# ls -l total
4
drwx--x--x  10 15806   root        4096 Jun 19 2005 linux
[root@plinux linux-gdb]#
```

---

After that, we also need to create the Bochs configuration file 'bochsrc-fd1-gdb.bxrc' in the linux-gdb/ directory and download the floppy root file system image file 'rootimage-0.11'. For convenience, we can also create a script file 'run' containing only one line of "bochs -q -f bochsrc-fd1-gdb.bxrc" and set the file attribute to executable. In addition, a package for direct debugging experiments has been created for everyone on oldlinux.org, which contains the same content as the package compiled directly under Redhat:

<http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-050619.tar.gz>

---

## 17.12 Summary

This is the last chapter of the book. This chapter describes the experimental operation of Linux 0.1X using the Bochs simulation environment. The basic usage of the Bochs system is given. The method of transferring files between the simulation system and the host system is described in detail. It also gives specific methods and steps for compiling and debugging the Linux 0.1X kernel.

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。皆さんのが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] Intel Co. INTEL 80386 Programmer's Reference Manual 1986, INTEL CORPORATION,1987.
- [2] Intel Co. IA-32 Intel Architecture Software Developer's Manual Volume.3:System Programming Guide.  
<http://www.intel.com/>, 2005.
- [3] James L. Turley. Advanced 80386 Programming Techniques. Osborne McGraw-Hill,1988.
- [4] Brian W. Kernighan, Dennis M. Ritchie. The C programming Language. Prentice-Hall 1988.
- [5] Leland L. Beck. System Software: An Introduction to Systems Programming,3<sup>nd</sup>. Addison-Wesley,1997.
- [6] Richard Stallman, Using and Porting the GNU Compiler Collection,the Free Software Foundation, 1998.
- [7] The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001, The IEEE and The Open Group.
- [8] David A Rusling, The Linux Kernel, 1999.<http://www.tldp.org/>
- [9] Linux Kernel Source Code, <http://www.kernel.org/>
- [10] Digital co.ltd. VT100 User Guide, <http://www.vt100.net/>
- [11] Clark L. Coleman. Using Inline Assembly with gcc. <http://oldlinux.org/Linux.old/>
- [12] John H. Crawford, Patrick P. Gelsinger. Programming the 80386. Sybex, 1988.
- [13] FreeBSD Online Manual, <http://www.freebsd.org/cgi/man.cgi>
- [14] Andrew S.Tanenbaum.Operating Systems: Design and Implementation. Prentice-Hall-International Editions. 1990.4
- [15] Maurice J. Bach. The Design of the UNIX Operating System. Prentice Hall. 1990
- [16] John Lions. Lions' Commentary on UNIX 6th Edition with Source Code. Peer-to-Peer Communications, Inc. 1996
- [17] Andrew S. Tanenbaum, Albert S. Woodhull. Operating Systems:Design and Implementation (Second Edition). Prentice Hall. 1997.
- [18] Alessandro Rubini, Jonathan. Linux Device Drivers. O'Reilly & Associates. Inc. 2001
- [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] 李凤华, 周利华, 赵丽松. MS-DOS 5.0 内核剖析. 西安电子科技大学出版社, 1992.
- [22] RedHat 9.0 Online manual. <http://www.plinux.org/cgi-bin/man.cgi>
- [23] W.Richard Stevens. Advanced Programming in the UNIX Environment. China Machine Press. 2000.2
- [24] Linux Weekly Edition News. <http://lwn.net/>
- [25] P.J. Plauger. The Standard C Library. Prentice Hall, 1992
- [26] Free Software Foundation. The GNU C Library. <http://www.gnu.org/> 2001
- [27] Chuck Allison. The Standard C Library. C/C++ Users Journal CD-ROM, Release 6. 2003
- [28] Bochs simulation system. <http://bochs.sourceforge.net/>
- [29] Brennan "Bas" Underwood. Brennan's Guide to Inline Assembly.<http://www.rt66.com/~brennan/>
- [30] John R. Levine. Linkers & Loaders. <http://www.iecc.com/linker/>
- [31] Randal E. Bryant, David R. O'Hallaron. Computer Systems A programmer's Perspective. Publishing House of Electronics Industry. 2004.3
- [32] Intel. Data Sheet: 8254 Programmable Interval Timer. 1993.9
- [33] Intel. Data Sheet: 8259A Programmable Interrupt Controller. 1988.12
- [34] Intel. Data Sheet: 82077A CHMOS Single-chip Floppy Disk Controller. 1994.5
- [35] Robert Love. Linux Kernel Development. China Machine Press. 2004
- [36] Adam Chapweske. The PS/2 Keyboard Interface. <http://www.computer-engineering.org/>

- [37] Dean Elsner, Jay Fenlason & friends. Using as: The GNU Assembler. <http://www.gnu.org/1998>
- [38] Steve Chamberlain. Using ld: The GNU linker. <http://www.gnu.org/1998>
- [39] Michael K. Johnson. The Linux Kernel Hackers' Guide. <http://www.tldp.org/1995>
- [40] Richard F. Ferraro. Programmer's Guide to the EGA, VGA, and Super VGA Cards. 3rd ed. Addison-Wesley, 1995.

# Appendix

## A1 ASCII Code Table

Decimal	Hex	Character	Decimal	Hex	Character	Decimal	Hex	Character
0	00	NUL	43	2B	+	86	56	V
1	01	SOH	44	2C	,	87	57	W
2	02	STX	45	2D	-	88	58	X
3	03	ETX	46	2E	.	89	59	Y
4	04	EOT	47	2F	/	90	5A	Z
5	05	ENQ	48	30	0	91	5B	[
6	06	ACK	49	31	1	92	5C	\
7	07	BEL	50	32	2	93	5D	]
8	08	BS	51	33	3	94	5E	^
9	09	TAB	52	34	4	95	5F	_
10	0A	LF	53	35	5	96	60	-
11	0B	VT	54	36	6	97	61	a
12	0C	FF	55	37	7	98	62	b
13	0D	CR	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DLE	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6A	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	(space)	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	\$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	
39	27	,	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

## A2 Common C0, C1 Control Characters

Common C0 control characters table

Mnemonic	Code	Actions taken
NUL	0x00	Null -- Ignored when received (not saved in the input buffer).
ENQ	0x05	Enquiry -- Sends a reply message.
BEL	0x07	Bell -- makes a sound.
BS	0x08	Backspace -- Moves the cursor one character position to the left. If the cursor is already on the left edge, there is no action.
HT	0x09	Horizontal Tabulation -- Moves the cursor to the next tab stop. If there is no tab stop on the right side, move to the right edge.
LF	0x0a	Linefeed -- This code causes a carriage return or line feed operation (see linefeed mode).
VT	0x0b	Vertical Tabulation -- acts like LF.
FF	0x0c	Form Feed -- acts like LF.
CR	0x0d	Carriage Return -- Moves the cursor to the left edge of the current line.
SO	0x0e	Shift Out -- Uses the G1 character set selected by the SCS control sequence. G1 can specify one of five character sets.
SI	0x0f	Shift In -- Uses the G0 character set selected by the SCS control sequence. G0 can specify one of five character sets.
DC1	0x11	Device Control 1 -- XON. Let the terminal resume transmission.
DC3	0x13	Device Control 3 -- XOFF. Stop sending all other codes except sending XOFF and XON.
CAN	0x18	Cancel -- If sent during a control sequence, the sequence will not execute and will terminate immediately. The error character is also displayed.
SUB	0x1a	Substitute -- works the same as CAN.
ESC	0x1b	Escape -- Generates an Escape Control Sequence.
DEL	0x7f	Delete -- Ignore when typing (not saved in the input buffer).

C 1 制 御 共 通 文 字 表  Mnemonic	Code	7B seq.	Actions taken
IND	0x84	ESC D	Index -- The cursor moves down one row in the same column. If the cursor is already on the bottom line, a scrolling operation is performed.
NEL	0x85	ESC H	Next Line -- The cursor moves to the first column of the next line. If the cursor is already on the bottom line, a scrolling operation is performed.

HTS	0x88	ESC E	Horizontal Tab Set -- Sets a horizontal tab stop at the cursor.
RI	0x8d	ESC M	Reverse Index -- The cursor moves one line up the same column. If the cursor is already on the top line, a scrolling operation is performed.
SS2	0x8e	ESC N	Single Shift G2 -- Temporarily uses the G2 character set in GL for the display of the next character. G2 is specified by the Selective Character Set (SCS) control sequence (see the escape sequence and control sequence table in Appendix 3).
SS3	0x8f	ESC O	Single Shift G3 -- Temporarily calls the G3 character set in GL for the display of the next character. G3 is specified by the Selective Character Set (SCS) control sequence (see the

			escape sequence and control sequence table in Appendix 3).
DCS	0x90	ESC P	Device Control String -- Used as the starting qualifier of the device control string.
CSI	0x9b	ESC [	Control Sequence Introducer -- Used as a control sequence leader code.
ST	0x9c	ESC \	String Terminator -- Used as the ending qualifier of the DCS string.

## A3 Escape and Control Sequences

Sequence and Name	Description
<b>ESC ( Ps or ESC ) Ps</b> Select Character Set	Select Character Set (SCS) -- The G0 and G1 character sets can each specify one of five character sets. 'ESC (Ps' specifies the character set used by G0, 'ESC) Ps' specifies the character set used by G1. Parameters Ps: A - UK character set; B - US character set; 0 - graphic character set; 1 - alternative ROM character set; 2 - optional ROM special character set.  A terminal can display up to 254 different characters, however the terminal only stores 127 display characters in its ROM. You must install additional character set ROM for the other 127 display characters. At some point, the terminal is able to select 94 characters (one character set). Therefore, the terminal can use one of five character sets, some of which appear in multiple character sets. At any one time, the terminal can use two active character sets. The computer can use the SCS sequence to specify any two character sets as G0 and G1. You can then switch between these two character sets using a single control character. The Shift In - SI (14) control character is used to select the G0 character set, and the Shift Out - SO (15) control character can be used to select the G1 character set. The specified character set will be used as the current character set until the terminal receives another SCS sequence.
<b>ESC [ Pn A</b> Cursor up (Terminal <--> Host)	Cursor Up (CUU) -- The CUU control sequence moves the cursor up but the column position is unchanged. The number of moving character positions is determined by parameters. If the parameter is 'Pn', the cursor moves up the 'Pn' line. The cursor is moved up to the top row at most. Note that 'Pn' is an ASCII numeric variable. If you do not select a parameter or the parameter value is 0, the terminal will assume a parameter value of 1.
<b>ESC [ Pn B or ESC [ Pn e</b> Cursor Down (Terminal <--> Host)	Cursor Down (CUD) -- The CUD control sequence moves the cursor down but the column position is unchanged. The number of moving character positions is determined by parameters. If the parameter is 1 or 0, the cursor moves down 1 line. If the parameter is 'Pn', the cursor moves down the 'Pn' line. The cursor moves down to the bottom line at most.
<b>ESC [ Pn C or ESC [ Pn a</b> Cursor Forward (Terminal <--> Host)	Cursor Forward (CUF) -- The CUF control sequence moves the current cursor to the right. The number of moving positions is determined by parameters. If the argument is 1 or 0, move 1 character position. If the parameter value is 'Pn', the cursor moves by 'Pn' character positions. The cursor moves up to the right border at most.
<b>ESC [ Pn D</b> Cursor Backward (Terminal <--> Host)	Cursor Backward (CUB) -- The CUB control sequence moves the current cursor to the left. The number of moving positions is determined by parameters. If the argument is 1 or 0, move 1 character position. If the parameter value is 'Pn', the cursor moves by 'Pn' character positions. The cursor moves up to the left border at most.
<b>ESC [ Pn E</b> Cursor moves down	Cursor Next Line (CNL) -- This control sequence moves the cursor to the first character of the 'Pn' line below.
<b>ESC [ Pn F</b> Cursor moves up	Cursor Last Line (CLL) -- This control sequence moves the cursor up to the first character of the 'Pn' line.
<b>ESC [ Pn G or ESC [ Pn `</b> Cursor moves in line	Cursor Horizon Absolute (CHA) -- This control sequence moves the cursor to the 'Pn' character position of the current line.
<b>ESC [ Pn ; Pn H or</b>	Cursor Position (CUP), Horizontal And Vertical Position (HVP) -- The CUP control sequence moves

<b>ESC [ Pn;Pn f</b>	the current cursor to the position specified by the parameter. The two parameters specify the row and column values, respectively. If the value is 0, it is the same as 1, indicating that one position is moved. In the default condition without parameters, it is equivalent to moving the cursor to the home position (ie <b>ESC [ H</b> ).
<b>ESC [ Pn d</b>	Vertical Line Position Absolute -- Moves the cursor to the 'Pn' line of the current column. If you try to move below the last line, the cursor will stay on the last line.
<b>ESC [ s</b>	Save Current Cursor Position -- This control sequence has the same effect as DECSC except that the page number displayed on the cursor is not saved.
<b>ESC [ u</b>	Restore Saved Cursor Position -- This control sequence has the same effect as DECRC except that the cursor is still on the same display page and not moved to the display page where the cursor is saved.
<b>ESC D</b>	Index (IND) -- This control sequence moves the cursor down one line, but the column number does not change. If the cursor is on the bottom line, it will cause the screen to scroll up one line.
<b>ESC M</b>	Reverse Index (RI) -- This control sequence moves the cursor up one line, but the column number does not change. If the cursor is on the top line, it will cause the screen to scroll down one line.
<b>ESC E</b>	Next Line (NEL) -- This control sequence will move the cursor to the beginning of the left side of the next line. If the cursor is on the bottom line, it will cause the screen to scroll up one line.
<b>ESC 7</b>	Save Cursor (DECSC) -- This control sequence will cause the cursor position, graphics to be reproduced, and the character set to be saved.
<b>ESC 8</b>	Restore Cursor (DECRC) -- This control sequence will cause the previously saved cursor position, graphics to be reproduced, and the character set to be restored.
<b>ESC [ Ps; Ps; ... ; Ps m</b>	Select Graphic Rendition (SGR) - Character re-display and attribute are characteristics that affect the display of a character without changing the character code. The control sequence sets the character display attributes according to the parameters. All characters sent to the terminal in the future will use the attributes specified here until the control sequence reset the attributes of the character again. Parameter 'Ps': 0 - no attribute (default attribute); 1 - bold and bright; 4 - underline; 5 - flashing; 7 - reverse; 22 - non-bold; 24 - no underline; 25 - no flicker; ;30--38 Set foreground color; 39 - Default foreground color (White); 40--48 - Set background color; 49 - Default background color (Black). 30--37 and 40-47 correspond to colors: Black, Red, Green, Yellow, Blue, Magenta, Cyan, White.
<b>ESC [ Pn L</b>	Insert Line (IL) -- This control sequence inserts one or more blank lines at the cursor. The cursor position does not change after the operation is completed. When a blank line is inserted, the line in the scroll area below the cursor moves down. The line scrolling out of the display page is lost.
<b>ESC [ Pn M</b>	Delete Line (DL) -- This control sequence deletes one or more lines from the line where the cursor is located in the scroll area. When the line is deleted, the line below the deleted line in the scroll area moves up, and 1 blank line is added to the bottom line. If 'Pn' is greater than the number of lines remaining on the display page, then this sequence only deletes these remaining lines and does not work outside the scroll area.
<b>ESC [ Pn @</b>	Insert Character (ICH) -- This control sequence inserts one or more space characters at the current cursor using the normal character attribute. 'Pn' is the number of characters inserted. The default is 1. The cursor will still be at the first inserted space character. The character at the cursor and right border will shift to the right. Characters that exceed the right border will be lost.
<b>ESC [ Pn P</b>	Delete Character (DCH) -- This control sequence deletes 'Pn' characters from the cursor. When a character is deleted, all characters to the right of the cursor are shifted to the left. This will produce a

	null character at the right border. Its properties are the same as the last left-shift character.																		
<b>ESC [ Ps J</b> Erase character	Erase In Display (ED) -- This control sequence erases some or all of the displayed characters, depending on the parameters. The erase operation removes characters from the screen without affecting other characters. The erased characters are discarded. The cursor position does not change when erasing characters or lines. While erasing characters, the attributes of the characters are also discarded. Any entire line erased by this control sequence will return the line to a single character width mode. Parameter 'Ps': 0 - Erase the cursor to all characters at the bottom of the screen; 1 - Erase the top of the screen to the cursor except all characters; 2 - Erase the entire screen.																		
<b>ESC [ Ps K</b> Erase in line	Erase In Line (EL) -- Erases some or all of the characters in the line of the cursor according to the parameters. The erase operation removes characters from the screen without affecting other characters. The erased characters are discarded. The cursor position does not change when erasing characters or lines. While erasing characters, the attributes of the characters are also discarded. Parameter 'Ps': 0 - Erases the cursor to all characters at the end of the line; 1 - Erases the left border to all characters at the cursor; 2 - Erases an entire line.																		
<b>ESC [ Pn ; Pn r</b> Set top &bottom margins	Set Top and Bottom Margins (DECSTBM) -- This control sequence sets the upper and lower areas of the scroll screen. The scrolling margin is an area on the screen where we can receive new characters by taking away the original characters from the screen. This area is defined by the top and bottom borders of the screen. The first parameter is the first line of the start of the scrolling area, and the second parameter is the last line of the scrolling area. By default it is the entire screen. The smallest scrolling area is 2 lines, ie the top border line must be smaller than the bottom border line. The cursor will be placed in the home position.																		
<b>ESC [ Pn c</b> or <b>ESC Z</b> Device attributes (Terminal <--> Host)	In response to a host request, the terminal can send a report message. These messages provide the identity (terminal type), cursor position, and terminal operational status. There are two types of reports: device attributes and device status reports. Device Attributes (DA) -- The host sends a device attribute (DA) control sequence (the same as ESC Z) with no parameters or parameter 0. The terminal sends one of the following sequences in response to the host's sequence. <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th style="text-align: center;">Terminal optional attribute</th> <th style="text-align: center;">Send sequence</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">None, VT101</td> <td style="text-align: center;"><b>ESC [?1;0c</b></td> </tr> <tr> <td style="text-align: center;">Processor option (STP)</td> <td style="text-align: center;"><b>ESC [?1;1c</b></td> </tr> <tr> <td style="text-align: center;">Advance Video (AVO) VT100</td> <td style="text-align: center;"><b>ESC [?1;2c</b></td> </tr> <tr> <td style="text-align: center;">AVO and STP</td> <td style="text-align: center;"><b>ESC [?1;3c</b></td> </tr> <tr> <td style="text-align: center;">Graphic property option (GPO)</td> <td style="text-align: center;"><b>ESC [?1;4c</b></td> </tr> <tr> <td style="text-align: center;">GPO and STP</td> <td style="text-align: center;"><b>ESC [?1;5c</b></td> </tr> <tr> <td style="text-align: center;">GPO and AVO, VT102</td> <td style="text-align: center;"><b>ESC [?1;6c</b></td> </tr> <tr> <td style="text-align: center;">GPO, STP and AVO</td> <td style="text-align: center;"><b>ESC [?1;7c</b></td> </tr> </tbody> </table>	Terminal optional attribute	Send sequence	None, VT101	<b>ESC [?1;0c</b>	Processor option (STP)	<b>ESC [?1;1c</b>	Advance Video (AVO) VT100	<b>ESC [?1;2c</b>	AVO and STP	<b>ESC [?1;3c</b>	Graphic property option (GPO)	<b>ESC [?1;4c</b>	GPO and STP	<b>ESC [?1;5c</b>	GPO and AVO, VT102	<b>ESC [?1;6c</b>	GPO, STP and AVO	<b>ESC [?1;7c</b>
Terminal optional attribute	Send sequence																		
None, VT101	<b>ESC [?1;0c</b>																		
Processor option (STP)	<b>ESC [?1;1c</b>																		
Advance Video (AVO) VT100	<b>ESC [?1;2c</b>																		
AVO and STP	<b>ESC [?1;3c</b>																		
Graphic property option (GPO)	<b>ESC [?1;4c</b>																		
GPO and STP	<b>ESC [?1;5c</b>																		
GPO and AVO, VT102	<b>ESC [?1;6c</b>																		
GPO, STP and AVO	<b>ESC [?1;7c</b>																		
<b>ESC c</b> Reset to initial state	Reset To Initial State (RIS) -- Lets the terminal reset to its initial state, that is, just turned on. All characters received during the reset phase will be lost. There are two ways to avoid this: 1. (Auto XON/XOFF) After the transmission, the host assumes that the terminal has sent XOFF. The host stops sending characters until it receives XON. 2. Delay at least 10 seconds and wait for the terminal reset operation to complete.																		

## A4 The First Set of Keyboard Scan Code

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	SPACE	39	B9	PG DN	E0, 51	E0, D1
H	23	A3	TAB	0F	8F	Up Arrow	E0, 48	E0, C8
I	17	97	CAPS	3A	BA	Left Arrow	E0, 4B	E0, CB
J	24	A4	Left SHFT	2A	AA	Down Arrow	E0, 50	E0, DO
K	25	A5	Left CTRL	1D	9D	Right Arrow	E0, 4D	E0, CD
L	26	A6	Left GUI	E0, 5B	E0, DB	NUM LOCK	45	C5
M	32	B2	Left ALT	38	B8	KP /	E0, 35	E0, B5
N	31	B1	Right SHFT	36	B6	KP *	37	B7
O	18	98	Right CTRL	E0, 1D	E0, 9D	KP -	4A	CA
P	19	99	Right GUI	E0, 5C	E0, DC	KP +	4E	CE
Q	10	90	Right 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, E0, 37	E0, B7, E0, AA	.	34	B4
7	08	88	SCROLL	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

のついた部分はすべて拡張キーです。