# 第六章　Linux 内存管理

存储器是一种必须仔细管理的重要资源。在理想的情况下，每个程序员都喜欢无穷大、快速并且内容不易变（即掉电后内容不会丢失）的存储器，同时又希望它是廉价的。但不幸的是，当前技术没有能够提供这样的存储器，因此大部分的计算机都有一个存储器层次结构，即少量、快速、昂贵、易变的高速缓存（cache）；若干兆字节的中等速度、中等价格、易变的主存储器（RAM）；数百兆或数千兆的低速、廉价、不易变的磁盘。如图 6.1 所示，这些资源的合理使用与否 ，直接关系着系统的效率。

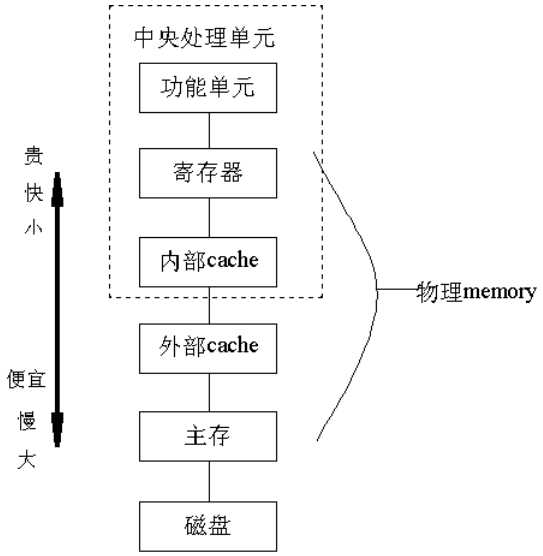


图 6.1　存储器的层次结构

Linux 内存管理是内核最复杂的任务之一。主要是因为它用到许多 CPU 提供的功能，而且和这些功能密切相关。正因为如此，我们在第二章较详细地介绍了 Intel 386 的段机制和页机制，可以说这是进行内存管理分析的物质基础 。这一章用大量的篇幅描述了 Linux 内存管理所涉及到的各种机制，如内存的初始化机制、地址映射机制、请页机制、交换机制、内存分配和回收机制、缓存和刷新机制以及内存共享机制，最后还分析了程序的创建和执行。

## 6.1　Linux 的内存管理概述

Linux 是为多用户多任务设计的操作系统，所以存储资源要被多个进程有效共享；且由

于程序规模的不断膨胀，要求的内存空间比从前大得多。 Linux 内存管理的设计充分利用了计算机系统所提供的虚拟存储技术，真正实现了虚拟存储器管理。

第二章介绍的 Intel 386 的段机制和页机制是 Linux 实现虚拟存储管理的一种硬件平台。实际上， Linux 2.0 以上的版本不仅仅可以运行在 Intel 系列个人计算机上，还可以运行在 Apple 系列、DEC Alpha 系列、MIPS 和 Motorola 68k 等系列上， 这些平台都支持虚拟存储器管理，我们之所以选择 Intel 386，是因为它具有代表性和普遍性。

Linux 的内存管理主要体现在对虚拟内存的管理。我们可以把 Linux 虚拟内存管理功能概括为以下几点：

- 大地址空间；
- 进程保护；
- 内存映射；
- 公平的物理内存分配；
- 共享虚拟内存。

关于这些功能的实现，我们将会陆续介绍。

### 6.1.1 Linux 虚拟内存的实现结构

我们先从整体结构上了解 Linux 对虚拟内存的实现结构，如图 6.2 所示。

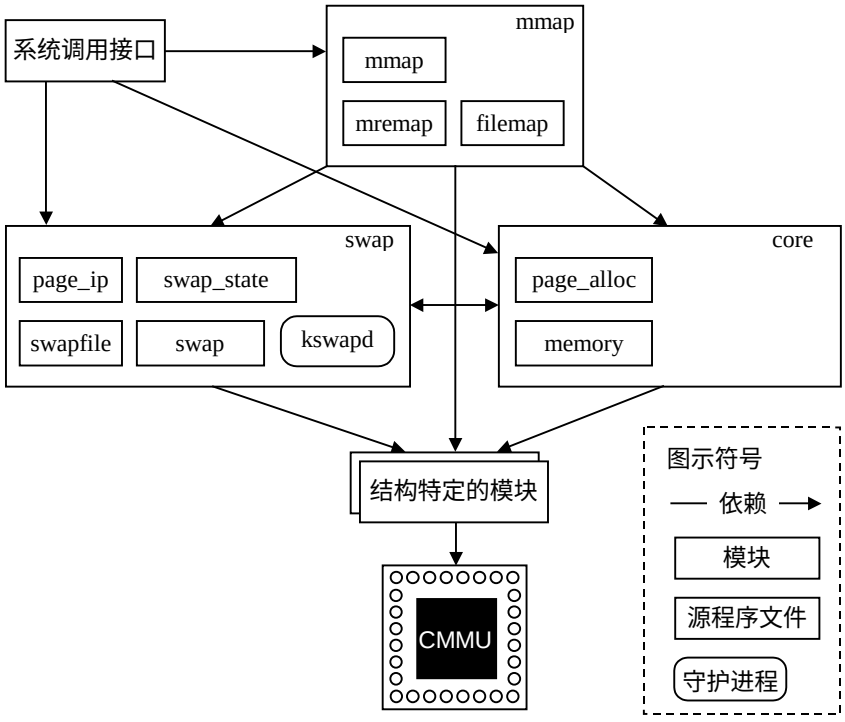


图 6.2 Linux 虚拟内存的实现结构

从图 6.2 中可看到实现虚拟内存的组成模块。其实现的源代码大部分放在/mm 目录下。

（1）内存映射模块(mmap)：负责把磁盘文件的逻辑地址映射到虚拟地址，以及把虚拟地址映射到物理地址。

（2）交换模块（swap)：负责控制内存内容的换入和换出，它通过交换机制，使得在物理内存的页面（RAM 页）中保留有效的页 ，即从主存中淘汰最近没被访问的页，保存近来访问过的页。

（3）核心内存管理模块（core)：负责核心内存管理功能，即对页的分配、回收、释放及请页处理等，这些功能将被别的内核子系统（如文件系统）使用。

（4）结构特定的模块：负责给各种硬件平台提供通用接口，这个模块通过执行命令来改变硬件 MMU 的虚拟地址映射，并在发生页错误时，提供了公用的方法来通知别的内核子系统。这个模块是实现虚拟内存的物理基础。

### 6.1.2　内核空间和用户空间

从第二章我们知道，Linux 简化了分段机制，使得虚拟地址与线性地址总是一致，因此，Linux 的虚拟地址空间也为 0～4G 字节。Linux 内核将这 4G 字节的空间分为两部分。将最高的 1G 字节（从虚拟地址 0xC0000000 到 0xFFFFFFFF)，供内核使用，称为“ 内核空间 ”。而将较低的 3G 字节（从虚拟地址 0x00000000 到 0xBFFFFFFF)，供各个进程使用，称为“ 用户空间 ”。因为每个进程可以通过系统调用进入内核，因此，Linux 内核由系统内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有 4G 字节的虚拟空间。图 6.3 给出了进程虚拟空间示意图。

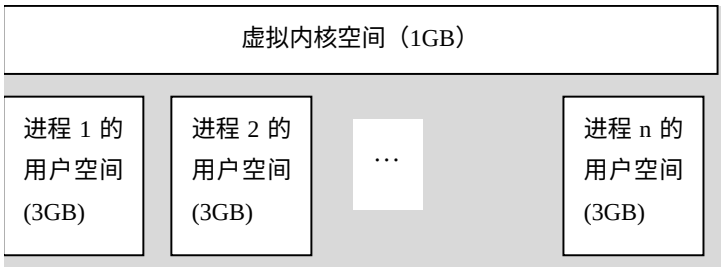


图 6.3　Linux 进程的虚拟空间

Linux 使用两级保护机制：0 级供内核使用，3 级供用户程序使用。从图 6.3 中可以看出，每个进程有各自的私有用户空间（0～3G)，这个空间对系统中的其他进程是不可见的。最高的 1G 字节虚拟内核空间则为所有进程以及内核所共享。

1．虚拟内核空间到物理空间的映射

内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中。读者会问，系统启动时，内核的代码和数据不是被装入到物理内存吗？它们为什么也处于虚拟内存中呢？这和编译程序有关，后面我们通过具体讨论就会明白这一点。

虽然内核空间占据了每个虚拟空间中的最高 1G 字节，但映射到物理内存却总是从最低

地址（0x00000000）开始。如图 6.4 所示，对内核空间来说，其地址映射是很简单的线性映射，0xC0000000 就是物理地址与线性地址之间的位移量，在 Linux 代码中就叫做 PAGE_OFFSET。

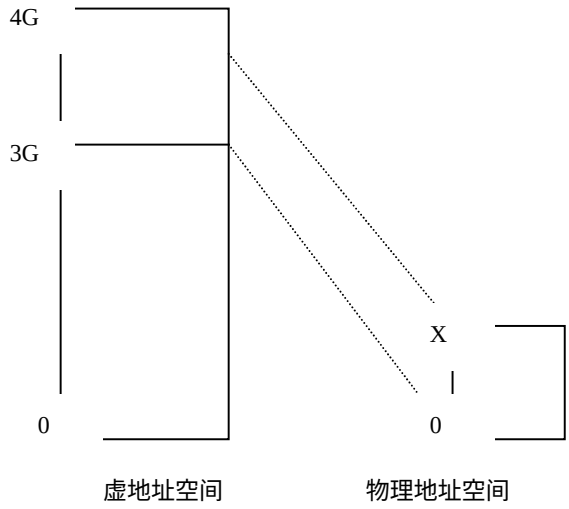


图 6.4　内核的虚拟地址空间到物理地址空间的映射

我们来看一下在 include/asm/i386/page.h 中对内核空间中地址映射的说明及定义：

```
/*
* This handles the memory map.. We could make this a config
* option, but too many people screw it up, and too few need
* it.
*
* A __PAGE_OFFSET of 0xC0000000 means that the kernel has
* a virtual address space of one gigabyte, which limits the
* amount of physical memory you can use to about 950MB.
*
* If you want more physical memory than this then see the CONFIG_HIGHMEM4G
* and CONFIG_HIGHMEM64G options in the kernel configuration.
*/

#define __PAGE_OFFSET           (0xC0000000)
......
#define PAGE_OFFSET             ( (unsigned long) __PAGE_OFFSET)
#define __pa (x)                  ( (unsigned long)  (x) -PAGE_OFFSET)
#define __va (x)                  ( (void *)  ( (unsigned long)  (x) +PAGE_OFFSET) )
```

源代码的注释中说明，如果你的物理内存大于 950MB，那么在编译内核时就需要加 CONFIG_HIGHMEM4G 和 CONFIG_HIGHMEM64G 选项，这种情况我们暂不考虑。如果物理内存小于 950MB，则对于内核空间而言，给定一个虚地址 x，其物理地址为“x- PAGE_OFFSET”，给定一个物理地址 x，其虚地址为“x+ PAGE_OFFSET”。

这里再次说明，宏__pa()仅仅把一个内核空间的虚地址映射到物理地址，而决不适用于用户空间，用户空间的地址映射要复杂得多。

2．内核映像

在下面的描述中，我们把内核的代码和数据就叫内核映像（Kernel Image)。当系统启动时，Linux 内核映像被安装在物理地址 0x00100000 开始的地方，即 1MB 开始的区间(第 1M 留作它用)。然而，在正常运行时， 整个内核映像应该在虚拟内核空间中，因此，连接程序在连接内核映像时，在所有的符号地址上加一个偏移量 PAGE_OFFSET，这样，内核映像在内核空间的起始地址就为 0xC0100000。

例如，进程的页目录 PGD（属于内核数据结构）就处于内核空间中。在进程切换时，要将寄存器 CR3 设置成指向新进程的页目录 PGD，而该目录的起始地址在内核空间中是虚地址，但 CR3 所需要的是物理地址，这时候就要用__pa()进行地址转换。在 mm_context.h 中就有这么一行语句：

```
asm volatile（“movl %0,%%cr3”: :”r” （__pa（next->pgd））;
```

这是一行嵌入式汇编代码，其含义是将下一个进程的页目录起始地址 next_pgd，通过__pa()转换成物理地址，存放在某个寄存器中，然后用 mov 指令将其写入 CR3 寄存器中。经过这行语句的处理，CR3 就指向新进程 next 的页目录表 PGD 了。

### 6.1.3　虚拟内存实现机制间的关系

Linux 虚拟内存的实现需要各种机制的支持，因此，本章我们将对内存的初始化进行描述以后，围绕以下几种实现机制进行介绍：

- 内存分配和回收机制；
- 地址映射机制；
- 缓存和刷新机制；
- 请页机制；
- 交换机制；
- 内存共享机制。

这几种机制的关系如图 6.5 所示。

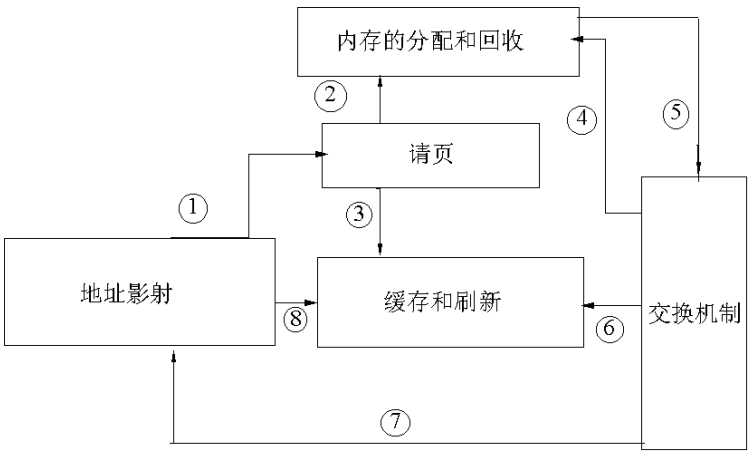


图 6.5　虚拟内存实现机制间的关系

首先内存管理程序通过映射机制把用户程序的逻辑地址映射到物理地址，在用户程序运行时如果发现程序中要用的虚地址没有对应的物理内存时，就发出了请页要求①；如果有空闲的内存可供分配，就请求分配内存②（于是用到了内存的分配和回收），并把正在使用的物理页记录在页缓存中③（使用了缓存机制）。如果没有足够的内存可供分配，那么就调用交换机制，腾出一部分内存④⑤。另外在地址映射中要通过 TLB（翻译后援存储器）来寻找物理页⑧；交换机制中也要用到交换缓存⑥，并且把物理页内容交换到交换文件中后也要修改页表来映射文件地址⑦。

## 6.2 Linux 内存管理的初始化

在对内存管理（MM）的各种机制介绍之前，首先需要对 MM 的初始化过程有所了解，以下介绍的内容是以第二章分段和分页机制为基础的，因此，读者应该先温习一下相关内容。因为在嵌入式操作系统的开发中，内存的初始化是重点关注的内容之一，因此本节将对内存的初始化给予详细描述。

### 6.2.1 启用分页机制

当 Linux 启动时，首先运行在实模式下，随后就要转到保护模式下运行。因为在第二章段机制中，我们已经介绍了 Linux 对段的设置，在此我们主要讨论与分页机制相关的问题。Linux 内核代码的入口点就是/arch/i386/kernel/head.S 中的 startup_32。

1．页表的初步初始化

```
/*
  * The page tables are initialized to only 8MB here - the final page
  * tables are set up later depending on memory size.
  */
.org 0x2000
ENTRY (pg0)

.org 0x3000
ENTRY (pg1)

/*
  * empty_zero_page must immediately follow the page tables ! (The
  * initialization loop counts until empty_zero_page)
  */

.org 0x4000
ENTRY (empty_zero_page)

 /*
  * Initialize page tables
  */
```

```
        movl $pg0-__PAGE_OFFSET,%edi /* initialize page tables */
        movl $007,%eax          /* "007" doesn't mean with right to kill, but
                                   PRESENT+RW+USER */
2:      stosl
        add $0x1000,%eax
        cmp $empty_zero_page-__PAGE_OFFSET,%edi
        jne 2b
```

内核的这段代码执行时，因为页机制还没有启用，还没有进入保护模式，因此指令寄存器 EIP 中的地址还是物理地址，但因为 pg0 中存放的是虚拟地址（gcc 编译内核以后形成的符号地址都是虚拟地址），因此，“ $pg0-__PAGE_OFFSET ”获得 pg0 的物理地址，可见 pg0 存放在相对于内核代码起点为 0x2000 的地方,即物理地址为 0x00102000，而 pg1 的物理地址则为 0x00103000。Pg0 和 pg1 这个两个页表中的表项则依次被设置为 0x007、0x1007、0x2007 等。其中最低的 3 位均为 1，表示这两个页为用户页，可写，且页的内容在内存中（参见图 2.24）。所映射的物理页的基地址则为 0x0、0x1000、0x2000 等，也就是物理内存中的页面 0、1、2、3 等等，共映射 2K 个页面，即 8MB 的存储空间。由此可以看出，Linux 内核对物理内存的最低要求为 8MB。紧接着存放的是 empty_zero_page 页(即零页)，零页存放的是系统启动参数和命令行参数，具体内容参见第十三章。

2．启用分页机制

```
/*
  * This is initialized to create an identity-mapping at 0-8M  (for bootup
  * purposes)  and another mapping of the 0-8M area at virtual address
  * PAGE_OFFSET.
  */
.org 0x1000
ENTRY (swapper_pg_dir)
        .long 0x00102007
        .long 0x00103007
        .fill BOOT_USER_PGD_PTRS-2,4,0
        /* default: 766 entries */
        .long 0x00102007
        .long 0x00103007
        /* default: 254 entries */
       .fill BOOT_KERNEL_PGD_PTRS-2,4,0
/*

 * Enable paging
 */
3:
     movl $swapper_pg_dir-__PAGE_OFFSET,%eax
     movl %eax,%cr3     /* set the page table pointer.. */
     movl %cr0,%eax
     orl $0x80000000,%eax
     movl %eax,%cr0     /* ..and set paging  (PG)  bit */
     jmp 1f             /* flush the prefetch-queue */
1:
     movl $1f,%eax
```

```
        jmp *%eax      /* make sure eip is relocated */
    1:
```

我们先来看这段代码的功能。这段代码就是把页目录 swapper_pg_dir 的物理地址装入控制寄存器 cr3，并把 cr0 中的最高位置成 1，这就开启了分页机制。

但是，启用了分页机制，并不说明 Linux 内核真正进入了保护模式，因为此时，指令寄存器 EIP 中的地址还是物理地址，而不是虚地址。“ jmp 1f ”指令从逻辑上说不起什么作用，但是，从功能上说它起到丢弃指令流水线中内容的作用（这是 Intel 在 i386 技术资料中所建议的），因为这是一个短跳转，EIP 中还是物理地址。紧接着的 mov 和 jmp 指令把第 2 个标号为 1 的地址装入 EAX 寄存器并跳转到那儿。在这两条指令执行的过程中，EIP 还是指向物理地址“ 1MB＋某处 ”。因为编译程序使所有的符号地址都在虚拟内存空间中，因此，第 2 个标号 1 的地址就在虚拟内存空间的某处（PAGE_OFFSET+某处），于是，jmp 指令执行以后，EIP 就指向虚拟内核空间的某个地址，这就使 CPU 转入了内核空间，从而完成了从实模式到保护模式的平稳过渡。

然后再看页目录 swapper_pg_dir 中的内容。从前面的讨论我们知道 pg0 和 pg1 这两个页表的起始物理地址分别为 0x00102000 和 0x00103000。从图 2.22 可知，页目录项的最低 12 位用来描述页表的属性。因此，在 swapper_pg_dir 中的第 0 和第 1 个目录项 0x00102007、0x00103007，就表示 pg0 和 pg1 这两个页表是用户页表、可写且页表的内容在内存。

接着，把 swapper_pg_dir 中的第 2～767 共 766 个目录项全部置为 0。因为一个页表的大小为 4KB，每个表项占 4 个字节，即每个页表含有 1024 个表项，每个页的大小也为 4KB，因此这 768 个目录项所映射的虚拟空间为 768×1024×4K＝3G，也就是 swapper_pg_dir 表中的前 768 个目录项映射的是用户空间。

最后，在第 768 和 769 个目录项中又存放 pg0 和 pg1 这两个页表的地址和属性，而把第 770～1023 共 254 个目录项置 0。这 256 个目录项所映射的虚拟地址空间为 256×1024×4K＝1G，也就是 swapper_pg_dir 表中的后 256 个目录项映射的是内核空间。

由此可以看出，在初始的页目录 swapper_pg_dir 中，用户空间和内核空间都只映射了开头的两个目录项，即 8MB 的空间，而且有着相同的映射，如图 6.6 所示。

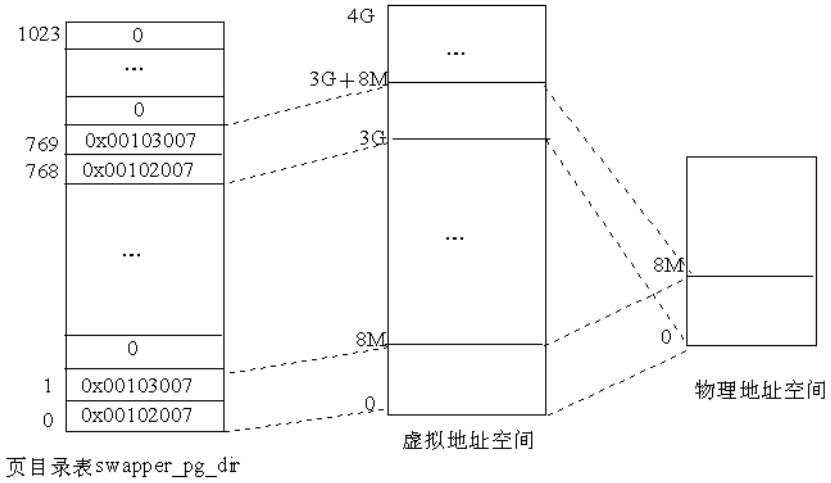

图 6.6　初始页目录 swapper_pg_dir 的映射图

读者会问，内核开始运行后运行在内核空间，那么，为什么把用户空间的低区（8M）也进行映射，而且与内核空间低区的映射相同？简而言之，是为了从实模式到保护模式的平稳过渡。具体地说，当 CPU 进入内核代码的起点 startup_32 后，是以物理地址来取指令的。在这种情况下，如果页目录只映射内核空间，而不映射用户空间的低区，则一旦开启页映射机制以后就不能继续执行了，这是因为，此时 CPU 中的指令寄存器 EIP 仍指向低区，仍会以物理地址取指令，直到以某个符号地址为目标作绝对转移或调用子程序为止。所以，Linux 内核就采取了上述的解决办法。

但是，在 CPU 转入内核空间以后，应该把用户空间低区的映射清除掉。后面读者将会看到，页目录 swapper_pg_dir 经扩充后就成为所有内核线程的页目录。在内核线程的正常运行中，处于内核态的 CPU 是不应该通过用户空间的虚拟地址访问内存的。清除了低区的映射以后，如果发生 CPU 在内核中通过用户空间的虚拟地址访问内存，就可以因为产生页面异常而捕获这个错误。

3．物理内存的初始分布

经过这个阶段的初始化，初始化阶段页目录及几个页表在物理空间中的位置如图 6.7 所示。

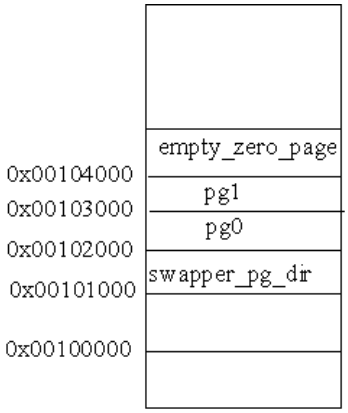


图 6.7　初始化阶段页目录及几个页表在物理空间中的位置

其中 empty_zero_page 中存放的是在操作系统的引导过程中所收集的一些数据，叫做引导参数。因为这个页面开始的内容全为 0，所以叫做“零页”，代码中常常通过宏定义 ZERO_PAGE 来引用这个页面。不过，这个页面要到初始化完成，系统转入正常运行时才会用到。为了后面内容介绍的方便，我们看一下复制到这个页面中的命令行参数和引导参数。这里假定这些参数已被复制到“零页”，在 setup.c 中定义了引用这些参数的宏：

```
/*
 * This is set up by the setup-routine at boot-time
 */
#define PARAM    ( (unsigned char *) empty_zero_page)
#define SCREEN_INFO  (* (struct screen_info *)   (PARAM+0) )
```

```
#define EXT_MEM_K  (* (unsigned short *)   (PARAM+2) )
#define ALT_MEM_K  (* (unsigned long *)   (PARAM+0x1e0) )
#define E820_MAP_NR  (* (char*)   (PARAM+E820NR) )
#define E820_MAP     ( (struct e820entry *)   (PARAM+E820MAP) )
#define APM_BIOS_INFO  (* (struct apm_bios_info *)   (PARAM+0x40) )
#define DRIVE_INFO  (* (struct drive_info_struct *)   (PARAM+0x80) )
#define SYS_DESC_TABLE  (* (struct sys_desc_table_struct*)  (PARAM+0xa0) )
#define MOUNT_ROOT_RDONLY  (* (unsigned short *)   (PARAM+0x1F2) )
#define RAMDISK_FLAGS  (* (unsigned short *)   (PARAM+0x1F8) )
#define ORIG_ROOT_DEV  (* (unsigned short *)   (PARAM+0x1FC) )
#define AUX_DEVICE_INFO  (* (unsigned char *)   (PARAM+0x1FF) )
#define LOADER_TYPE  (* (unsigned char *)   (PARAM+0x210) )
#define KERNEL_START  (* (unsigned long *)   (PARAM+0x214) )
#define INITRD_START  (* (unsigned long *)   (PARAM+0x218) )
#define INITRD_SIZE  (* (unsigned long *)   (PARAM+0x21c) )
#define COMMAND_LINE  ( (char *)   (PARAM+2048) )
#define COMMAND_LINE_SIZE 256
```

其中宏 PARAM 就是 empty_zero_page 的起始位置，随着代码的阅读，读者会逐渐理解这些参数的用途。这里要特别对宏 E820_MAP 进行说明。E820_MAP 是个 struct e820entry 数据结构的指针，存放在参数块中位移为 0x2d0 的地方。这个数据结构定义在 include/i386/e820.h 中：

```
struct e820map {
     int nr_map;
     struct e820entry {
         unsigned long long addr;        /* start of memory segment */
         unsigned long long size;        /* size of memory segment */
         unsigned long type;             /* type of memory segment */
     } map[E820MAX];
};

extern struct e820map e820;
```

其中，E820MAX 被定义为 32。从这个数据结构的定义可以看出，每个 e820entry 都是对一个物理区间的描述，并且一个物理区间必须是同一类型。如果有一片地址连续的物理内存空间，其一部分是 RAM，而另一部分是 ROM，那就要分成两个区间。即使同属 RAM，如果其中一部分要保留用于特殊目的,那也属于不同的分区。在 e820.h 文件中定义了 4 种不同的类型：

```
#define E820_RAM        1
#define E820_RESERVED   2
#define E820_ACPI       3 /* usable as RAM once ACPI tables have been read */
#define E820_NVS        4

#define HIGH_MEMORY      (1024*1024)
```

其中 E820_NVS 表示“ Non-Volatile Storage ”，即“ 不挥发 ”存储器，包括 ROM、EPROM、Flash 存储器等。

在 PC 中，对于最初 1MB 存储空间的使用是特殊的。开头 640KB（0x0~0x9FFFF）为 RAM，从 0xA0000 开始的空间则用于 CGA、EGA、VGA 等图形卡。现在已经很少使用这些图形卡，但是不管是什么图形卡，开机时总是工作于 EGA 或 VGA 模式。从 0xF0000 开始到 0xFFFFF，即最高的 4KB，就是在 EPROM 或 Flash 存储器中的 BIOS。所以，只要有 BIOS 存在，就至少有两

个区间，如果 nr_map 小于 2，那就一定出错了。由于 BIOS 的存在，本来连续的 RAM 空间就不连续了。当然，现在已经不存在这样的存储结构了。1MB 的边界早已被突破，但因为历史的原因，把 1MB 以上的空间定义为“ HIGH_MEMORY ”，这个称呼一直沿用到现在，于是代码中的常数 HIGH_MEMORY 就定义为“ 1024×1024 ”。现在，配备了 128MB 的内存已经是很普遍了。但是，为了保持兼容，就得留出最初 1MB 的空间。

这个阶段初始化后，物理内存中内核映像的分布如图 6.8 所示。

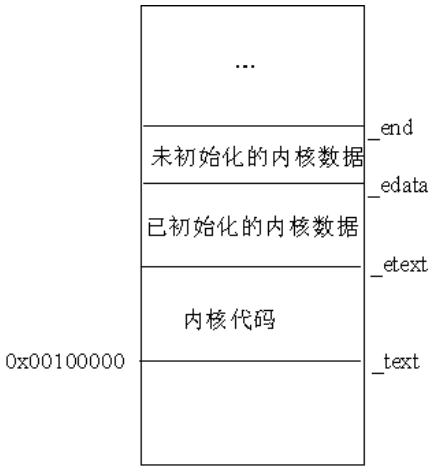


图 6.8　内核映像在物理内存中的分布

符号_text 对应物理地址 0x00100000，表示内核代码的第一个字节的地址。内核代码的结束位置用另一个类似的符号_etext 表示。内核数据被分为两组：初始化过的数据和未初始化过的数据。初始化过的数据在_etext 后开始，在_edata 处结束，紧接着是未初始化过的数据，其结束符号为_end，这也是整个内核映像的结束符号。

图中出现的符号是由编译程序在编译内核时产生的。你可以在 System.map 文件中找到这些符号的线性地址（或叫虚拟地址），System.map 是编译内核以后所创建的。

### 6.2.2　物理内存的探测

我们知道，BIOS 不仅能引导操作系统，还担负着加电自检和对资源的扫描探测，其中就包括了对物理内存的自检和扫描（你刚开机时所看到的信息就是此阶段 BIOS 显示的信息）。对于这个阶段中获得的内存信息可以通过 BIOS 调用“ int 0x15 ”加以检查。由于 Linux 内核不能作 BIOS 调用，因此内核本身就得代为检查，并根据获得的信息生成一幅物理内存构成图，这就是上面所介绍的 e820 图，然后通过上面提到的参数块传给内核。使得内核能知道系统中内存资源的配置。之所以称为 e820 图，是因为在通过 ”int 0x15 ”查询内存的构成时要把调用参数之一设置成 0xe820。

分页机制启用以后，与内存管理相关的操作就是调用 init/main.c 中的 start_kernel () 函数，start_kernel () 函数要调用一个叫 setup_arch()的函数，setup_arch () 位于 arch/i386/kernel/setup.c 文件中，我们所关注的与物理内存探测相关的内容就在这个函数

中。

1. setup_arch()函数

这个函数比较繁琐和冗长，下面我们只对 setup_arch () 中与内存相关的内容给予描述。

首先调用 setup_memory_region()函数，这个函数处理内存构成图（map）,并把内存的分布信息存放在全局变量 e820 中，后面会对此函数进行具体描述。

调用 parse_mem_cmdline(cmdline_p)函数。在特殊的情况下，有的系统可能有特殊的 RAM 空间结构，此时可以通过引导命令行中的选择项来改变存储空间的逻辑结构，使其正确反映内存的物理结构。此函数的作用就是分析命令行中的选择项，并据此对数据结构 e820 中的内容作出修正，其代码也在 setup.c 中。

宏定义：

```
#define PFN_UP (x)          ( ( (x)  + PAGE_SIZE-1)  >> PAGE_SHIFT)
#define PFN_DOWN (x)        ( (x)  >> PAGE_SHIFT)
#define PFN_PHYS (x)        ( (x)  << PAGE_SHIFT)
```

PFN_UP () 和 PFN_DOWN () 都是将地址 x 转换为页面号（PFN 即 Page Frame Number 的缩写），二者之间的区别为：PFN_UP () 返回大于 x 的第 1 个页面号，而 PFN_DOWN () 返回小于 x 的第 1 个页面号。宏 PFN_PHYS () 返回页面号 x 的物理地址。

宏定义

```
/*
* 128MB for vmalloc and initrd
*/
#define VMALLOC_RESERVE  (unsigned long)  (128 << 20)
#define MAXMEM  (unsigned long)  (-PAGE_OFFSET-VMALLOC_RESERVE)
#define MAXMEM_PFN PFN_DOWN (MAXMEM)
#define MAX_NONPAE_PFN  (1 << 20)
```

对这几个宏描述如下：

- VMALLOC_RESERVE：为 vmalloc () 函数访问内核空间所保留的内存区，大小为 128MB。
- MAXMEM：内核能够直接映射的最大 RAM 容量，为 1GB－128MB＝896MB（-PAGE_OFFSET 就等于 1GB）
- MAXMEM_PFN：返回由内核能直接映射的最大物理页面数。
- MAX_NONPAE_PFN：给出在 4GB 之上第 1 个页面的页面号。当页面扩充（PAE）功能启用时，才能访问 4GB 以上的内存。

获得内核映像之后的起始页面号：

```
/*
* partially used pages are not usable - thus
* we are rounding upwards:
*/
start_pfn = PFN_UP (__pa (&_end) ) ;
```

在上一节已说明，宏__pa () 返回给定虚拟地址的物理地址。其中标识符_end 表示内核映像在内核空间的结束位置。因此，存放在变量 start_pfn 中的值就是紧接着内核映像之后的页面号。

找出可用的最高页面号：

```
/*
* Find the highest page frame number we have available
*/
max_pfn = 0;
for  (i = 0; i < e820.nr_map; i++)  {
     unsigned long start, end;
     /* RAM? */
     if  (e820.map[i].type != E820_RAM)
          continue;
     start = PFN_UP (e820.map[i].addr) ;
     end = PFN_DOWN (e820.map[i].addr + e820.map[i].size) ;
     if  (start >= end)
          continue;
     if  (end > max_pfn)
          max_pfn = end;
}
```

上面这段代码循环查找类型为 E820_RAM（可用 RAM）的内存区，并把最后一个页面的页面号存放在 max_pfn 中。

确定最高和最低内存范围：

```
/*
* Determine low and high memory ranges:
*/
max_low_pfn = max_pfn;
if  (max_low_pfn > MAXMEM_PFN)  {
max_low_pfn = MAXMEM_PFN;
#ifndef CONFIG_HIGHMEM
/* Maximum memory usable is what is directly addressable */
printk (KERN_WARNING "Warning only %ldMB will be used.\n",
                                              MAXMEM>>20) ;
     if  (max_pfn > MAX_NONPAE_PFN)
          printk (KERN_WARNING "Use a PAE enabled kernel.\n") ;
     else
          printk (KERN_WARNING "Use a HIGHMEM enabled kernel.\n") ;
#else /* !CONFIG_HIGHMEM */
#ifndef CONFIG_X86_PAE
     if  (max_pfn > MAX_NONPAE_PFN)  {
          max_pfn = MAX_NONPAE_PFN;
          printk (KERN_WARNING "Warning only 4GB will be used.\n") ;
           printk (KERN_WARNING "Use a PAE enabled kernel.\n") ;
}
#endif /* !CONFIG_X86_PAE */
#endif /* !CONFIG_HIGHMEM */
}
```

有两种情况：

- 如果物理内存 RAM 大于 896MB,而小于 4GB,则选用 CONFIG_HIGHMEM 选项来进行访问；
- 如果物理内存 RAM 大于 4GB，则选用 CONFIG_X86_PAE（启用 PAE 模式）来进行访问。

上面这段代码检查了这两种情况，并显示适当的警告信息。

```
#ifdef CONFIG_HIGHMEM
highstart_pfn = highend_pfn = max_pfn;
```

```
if (max_pfn > MAXMEM_PFN) {
    highstart_pfn = MAXMEM_PFN;
    printk (KERN_NOTICE "%ldMB HIGHMEM available.\n",
        pages_to_mb (highend_pfn - highstart_pfn) ) ;
}
#endif
```

如果使用了 CONFIG_HIGHMEM 选项，上面这段代码仅仅打印出大于 896MB 的可用物理内存数量。

初始化引导时的分配器

```
* Initialize the boot-time allocator (with low memory only) :
*/
bootmap_size = init_bootmem (start_pfn, max_low_pfn) ;
```

通过调用 init_bootmem () 函数，为物理内存页面管理机制的建立做初步准备，为整个物理内存建立起一个页面位图。这个位图建立在从 start_pfn 开始的地方，也就是说，把内核映像终点_end 上方的若干页面用作物理页面位图。在前面的代码中已经搞清楚了物理内存顶点所在的页面号为 max_low_pfn，所以物理内存的页面号一定在 0～max_low_pfn 之间。可是，在这个范围内可能有空洞 (hole)，另一方面，并不是所有的物理内存页面都可以动态分配。建立这个位图的目的就是要搞清楚哪一些物理内存页面可以动态分配的。后面会具体描述 bootmem 分配器。

用 bootmem 分配器，登记全部低区（0～896MB）的可用 RAM 页面

```
/*
* Register fully available low RAM pages with the
* bootmem allocator.
*/
for (i = 0; i < e820.nr_map; i++) {
    unsigned long curr_pfn, last_pfn, size;
/*
* Reserve usable low memory
*/
 if (e820.map[i].type != E820_RAM)
    continue;
/*
* We are rounding up the start address of usable memory:
*/
    curr_pfn = PFN_UP (e820.map[i].addr) ;
    if (curr_pfn >= max_low_pfn)
        continue;
/*
* ... and at the end of the usable range downwards:
*/
last_pfn = PFN_DOWN (e820.map[i].addr + e820.map[i].size) ;
    if (last_pfn > max_low_pfn)
        last_pfn = max_low_pfn;
/*
* .. finally, did all the rounding and playing
* around just make the area go away?
*/
    if (last_pfn <= curr_pfn)
```

```
                continue;
            size = last_pfn - curr_pfn;
            free_bootmem (PFN_PHYS (curr_pfn) , PFN_PHYS (size) ) ;
        }
```

这个循环仔细检查所有可以使用的 RAM，并调用 free_bootmem()函数把这些可用 RAM 标记为可用。这个函数调用以后，只有类型为 1（可用 RAM）的内存被标记为可用的，参看后面对这个函数的具体描述。

保留内存：

```
        /*
        * Reserve the bootmem bitmap itself as well. We do this in two
        * steps  (first step was init_bootmem () )  because this catches
        * the  (very unlikely)  case of us accidentally initializing the
        * bootmem allocator with an invalid RAM area.
        */
        reserve_bootmem (HIGH_MEMORY,  (PFN_PHYS (start_pfn)  +
                bootmap_size + PAGE_SIZE-1)  -  (HIGH_MEMORY) ) ;
```

这个函数把内核和 bootmem 位图所占的内存标记为“保留”。 HIGH_MEMORY 为 1MB，即内核开始的地方，后面还要对这个函数进行具体描述。

分页机制的初始化

```
        paging_init () ;
```

这个函数初始化分页内存管理所需要的数据结构，参见后面的详细描述。

2．setup_memory_region() 函数

这个函数用来处理 BIOS 的内存构成图，并把这个构成图拷贝到全局变量 e820 中。如果操作失败，就创建一个伪内存构成图。这个函数的主要操作如下所述。

- 调用 sanitize_e820_map()函数，以删除内存构成图中任何重叠的部分，因为 BIOS 所报告的内存构成图可能有重叠。
- 调用 copy_e820_map()进行实际的拷贝。
- 如果操作失败，创建一个伪内存构成图，这个伪构成图有两部分：0 到 640K 及 1M 到最大物理内存。
- 打印最终的内存构成图。

3．copy_e820_map() 函数

函数原型为：

```
    static int __init sanitize_e820_map (struct e820entry * biosmap, char * pnr_map)
```

其主要操作如下概述。

(1) 如果物理内存区间小于 2，那肯定出错。因为 BIOS 至少和 RAM 属于不同的物理区间。

```
if  (nr_map < 2)
        return -1;
```

(2) 从 BIOS 构成图中读出一项。

```
    do {
        unsigned long long start = biosmap->addr;
        unsigned long long size = biosmap->size;
        unsigned long long end = start + size;
```

```
        unsigned long type = biosmap->type;
```

(3) 进行检查。

```
    /* Overflow in 64 bits? Ignore the memory map. */
        if (start > end)
            return -1;
```

(4) 一些 BIOS 把 640KB～1MB 之间的区间作为 RAM 来用，这是不符合常规的。因为从 0xA0000 开始的空间用于图形卡，因此，在内存构成图中要进行修正。如果一个区的起点在 0xA0000 以下，而终点在 1MB 之上，就要将这个区间拆开成两个区间，中间跳过从 0xA0000 到 1MB 边界之间的那一部分。

```
    /*
             * Some BIOSes claim RAM in the 640k - 1M region.
             * Not right. Fix it up.
             */
                 if (type == E820_RAM) {
                     if (start < 0x100000ULL && end > 0xA0000ULL) {
                         if (start < 0xA0000ULL)
                             add_memory_region (start, 0xA0000ULL-start, type)
                          if (end <= 0x100000ULL)
                             continue;
                         start = 0x100000ULL;
                          size = end - start;
                     }
                 }

                 add_memory_region (start, size, type) ;
             } while (biosmap++,--nr_map) ;
             return 0;
```

4. add_memory_region() 函数

这个函数的功能就是在 e820 中增加一项，其主要操作如下所述。

(1) 获得已追加在 e820 中的内存区数。

```
    int x = e820.nr_map;
```

(2) 如果数目已达到最大（32），则显示一个警告信息并返回。

```
    if (x == E820MAX) {
        printk (KERN_ERR "Oops! Too many entries in
                                            the memory map!\n") ;
        return;
    }
```

(3) 在 e820 中增加一项，并给 nr_map 加 1。

```
    e820.map[x].addr = start;
    e820.map[x].size = size;
    e820.map[x].type = type;
    e820.nr_map++;
```

5. print_memory_map() 函数

这个函数把内存构成图在控制台上输出，函数本身比较简单，在此给出一个运行实例。例如函数的输出为（BIOS 所提供的物理 RAM 区间）：

```
BIOS-e820: 0000000000000000 - 00000000000a0000 (usable)
BIOS-e820: 00000000000f0000 - 0000000000100000 (reserved)
BIOS-e820: 0000000000100000 - 000000000c000000 (usable)
BIOS-e820: 00000000fffff0000 - 0000000100000000 (reserved)
```

### 6.2.3　物理内存的描述

为了对内存的初始化内容进行进一步的讨论，我们首先要了解 Linux 对物理内存的描述机制。

1．一致存储结构（UMA）和非一致存储结构（NUMA）

在传统的计算机结构中，整个物理内存都是均匀一致的，CPU 访问这个空间中的任何一个地址所需要的时间都相同，所以把这种内存称为“一致存储结构（Uniform Memory Architecture）”，简称 UMA。可是，在一些新的系统结构中，特别是多 CPU 结构的系统中，物理存储空间在这方面的一致性却成了问题。这是因为，在多 CPU 结构中，系统中只有一条总线（例如，PCI 总线），有多个 CPU 模块连接在系统总线上，每个 CPU 模块都有本地的物理内存，但是也可以通过系统总线访问其他 CPU 模块上的内存。另外，系统总线上还连接着一个公用的存储模块，所有的 CPU 模块都可以通过系统总线来访问它。因此，所有这些物理内存的地址可以互相连续而形成一个连续的物理地址空间。

显然，就某个特定的 CPU 而言，访问其本地的存储器速度是最快的，而穿过系统总线访问公用存储模块或其他 CPU 模块上的存储器就比较慢，而且还面临因可能的竞争而引起的不确定性。也就是说，在这样的系统中，其物理存储空间虽然地址连续，但因为所处“位置”不同而导致的存取速度不一致，所以称为“非一致存储结构（Non-Uniform Memory Architecture），简称 NUMA。

事实上，严格意义上的 UMA 结构几乎不存在。就拿配置最简单的单 CPU 来说，其物理存储空间就包括了 RAM、ROM（用于 BIOS），还有图形卡上的静态 RAM。但是，在 UMA 中，除主存 RAM 之外的存储器空间都很小，因此可以把它们放在特殊的地址上，在编程时加以特别注意就行，那么，可以认为以 RAM 为主体的主存是 UMA 结构。

由于 NUMA 的引入，就需要存储管理机制的支持，因此，Linux 内核从 2.4 版本开始就提供了对 NUMA 的支持（作为一个编译可选项）。为了对 NUMA 进行描述，引入一个新的概念——“存储节点(或叫节点)”，把访问时间相同的存储空间就叫做一个“存储节点”。一般来说，连续的物理页面应该分配在相同的存储节点上。例如，如果 CPU 模块 1 要求分配 5 个页面，但是由于本模块上的存储空间已经不够，只能分配 3 个页面，那么此时，是把另外两个页面分配在其他 CPU 模块上呢，还是把 5 个页面干脆分配在一个模块上？显然，合理的分配方式因该是将这 5 个页面都分配在公用模块上。

Linux 把物理内存划分为 3 个层次来管理：存储节点(Node)、管理区(Zone)和页面(Page)，并用 3 个相应的数据结构来描述。

2．页面（Page）数据结构

对一个物理页面的描述在／include/linux/mm.h 中：

```
/*
  * Each physical page in the system has a struct page associated with
  * it to keep track of whatever it is we are using the page for at the
  * moment. Note that we have no way to track which tasks are using
  * a page.
  *
  * Try to keep the most commonly accessed fields in single cache lines
  * here  (16 bytes or greater) .  This ordering should be particularly
  * beneficial on 32-bit processors.
  *
  * The first line is data used in page cache lookup, the second line
  * is used for linear searches  (eg. clock algorithm scans) .
  *
  * TODO: make this structure smaller, it could be as small as 32 bytes.
  */

typedef struct page {
        struct list_head list;          /* ->mapping has some page lists. */
         struct address_space *mapping;  /* The inode  (or ...)  we belong to. */
         unsigned long index;            /* Our offset within mapping. */
         struct page *next_hash;         /* Next page sharing our hash bucket in
                                            the pagecache hash table. */
         atomic_t count;                 /* Usage count, see below. */
         unsigned long flags;            /* atomic flags, some possibly
                                            updated asynchronously */
         struct list_head lru;           /* Pageout list, eg. active_list;
                                            protected by pagemap_lru_lock !! */
         wait_queue_head_t wait;         /* Page locked?  Stand in line... */
         struct page **pprev_hash;       /* Complement to *next_hash. */
         struct buffer_head * buffers;   /* Buffer maps us to a disk block. */
         void *virtual;                  /* Kernel virtual address  (NULL if
                                            not kmapped, ie. highmem)  */
         struct zone_struct *zone;       /* Memory zone we are in. */
} mem_map_t;
extern mem_map_t * mem_map ;
```

源代码的注释中对这个数据结构给出了一定的说明，从中我们可以对此结构有一定的理解，后面还要对此结构中的每个域给出具体的解释。

内核中用来表示这个数据结构的变量常常是 page 或 map。

当页面的数据来自一个文件时，index 代表着该页面中的数据在文件中的偏移量；当页面的内容被换出到交换设备上，则 index 指明了页面的去向。结构中各个成分的次序是有讲究的，尽量使得联系紧密的若干域存放在一起，这样当这个数据结构被装入到高速缓存中时，联系紧密的域就可以存放在同一缓冲行（Cache Line）中。因为同一缓冲行（其大小为 16 字节）中的内容几乎可以同时存取，因此，代码注释中希望这个数据结构尽量地小到用 32 个字节可以描述。

系统中的每个物理页面都有一个 Page（或 mem_map_t）结构。系统在初始化阶段根据内

存的大小建立起一个 Page 结构的数组 mem_map，数组的下标就是内存中物理页面的序号。

3．管理区 Zone

为了对物理页面进行有效的管理，Linux 又把物理页面划分为 3 个区：

- 专供 DMA 使用的 ZONE_DMA 区（小于 16MB）；
- 常规的 ZONE_NORMAL 区（大于 16MB 小于 896MB）；
- 内核不能直接映射的区 ZONE_HIGME 区（大于 896MB)。

这里进一步说明为什么对 DMA 要单独设置管理区。首先，DMA 使用的页面是磁盘 I／O 所需的，如果在页面的分配过程中，所有的页面全被分配完，那么页面及盘区的交换就无法进行了，这是操作系统决不允许出现的现象。另外，在 i386 CPU 中，页式存储管理的硬件支持是在 CPU 内部实现的，而不像有些 CPU 那样由一个单独的 MMU 来提供，所以 DMA 对内存的访问不经过 MMU 提供的地址映射。这样，外部设备就要直接访问物理页面的地址。可是，有些外设（特别是插在 ISA 总线上的外设接口卡）在这方面往往有些限制，要求用于 DMA 的物理地址不能过高。另一方面，当 DMA 所需的缓冲区超过一个物理页面的大小时，就要求两个物理页面在物理上是连续的，但因为此时 DMA 控制器不能依靠 CPU 内部的 MMU 将连续的虚存页面映射到物理上也连续的页面上，因此，用于 DMA 的物理页面必须加以单独管理。

关于管理区的数据结构 zone_struct(或 zone_t)将在后面进行描述。

4．存储节点（Node）的数据结构

存储节点的数据结构为 pglist_data，定义于 Include/linux/mmzone.h 中：

```
typedef struct pglist_data {
        zone_t node_zones[MAX_NR_ZONES];
        zonelist_t node_zonelists[GFP_ZONEMASK+1];
        int nr_zones;
        struct page *node_mem_map;
        unsigned long *valid_addr_bitmap;
        struct bootmem_data *bdata;
        unsigned long node_start_paddr;
        unsigned long node_start_mapnr;
        unsigned long node_size;
        int node_id;
        struct pglist_data *node_next;
    } pg_data_t;
```

显然,若干存储节点的 pglist_data 数据结构可以通过 node_next 形成一个单链表队列。每个结构中的 node_mem_map 指向具体节点的 page 结构数组，而数组 node_zone[]就是该节点的最多 3 个页面管理区。

在 pglist_data 结构里设置了一个 node_zonelists 数组，其类型定义也在同一文件中：

```
typedef struct zonelist_struct {
     zone_t *zone[MAX_NR_ZONE+1];  //NULL delimited
     Int gfp_mast;
   } zonelist_t
```

这里的 zone[]是个指针数组，各个元素按特定的次序指向具体的页面管理区，表示分配页面时先试 zone[0]所指向的管理区,如果不能满足要求就试 zone[1]所指向的管理区,等等。

这些管理区可以属于不同的存储节点。关于管理区的分配可以有很多种策略，例如，CPU 模块 1 需要分配 5 个用于 DMA 的页面，可是它的 ZONE_DMA 只有 3 个页面，于是就从公用模块的 ZONE_DMA 中分配 5 个页面。就是说，每个 zonelist_t 规定了一种分配策略。然而，每个存储节点不应该只有一种分配策略，所以在 pglist_data 中提供的是一个 zonelist_t 数组，数组的大小 NR_GFPINDEX 为 100。

### 6.2.4 页面管理机制的初步建立

为了对页面管理机制作出初步准备，Linux 使用了一种叫 bootmem 分配器(Bootmem Allocator)的机制，这种机制仅仅用在系统引导时，它为整个物理内存建立起一个页面位图。这个位图建立在从 start_pfn 开始的地方，也就是说，内核映像终点_end 上方的地方。这个位图用来管理低区（例如小于 896MB），因为在 0 到 896MB 的范围内，有些页面可能保留，有些页面可能有空洞，因此，建立这个位图的目的就是要搞清楚哪一些物理页面是可以动态分配的。用来存放位图的数据结构为 bootmem_data（在 mm/numa.c 中） ：

```
typedef struct bootmem_data {
    unsigned long node_boot_start;
    unsigned long node_low_pfn;
    void *node_bootmem_map;
    unsigned long last_offset;
    unsigned long last_pos;
} bootmem_data_t;
```

node_boot_start 表示存放 bootmem 位图的第一个页面（即内核映像结束处的第一个页面)。

node_low_pfn 表示物理内存的顶点，最高不超过 896MB。

node_bootmem_map 指向 bootmem 位图

last_offset 用来存放在前一次分配中所分配的最后一个字节相对于 last_pos 的位移量。

last_pos 用来存放前一次分配的最后一个页面的页面号。这个域用在__alloc_bootmem_core()函数中，通过合并相邻的内存来减少内部碎片。

下面介绍与 bootmem 相关的几个函数，这些函数位于 mm/bootmeme.c 中。

1. init_bootmem()函数

```
unsigned long __init init_bootmem  (unsigned long start, unsigned long pages)
{
        max_low_pfn = pages;
        min_low_pfn = start;
        return (init_bootmem_core (&contig_page_data, start, 0, pages) ) ;
}
```

这个函数仅在初始化时用来建立 bootmem 分配器。这个函数实际上是 init_bootmem_core()函数的封装函数。init_bootmem () 函数的参数 start 表示内核映像结束处的页面号，而 pages 表示物理内存顶点所在的页面号。而函数 init_bootmem_core()就是对 contig_page_data 变量进行初始化。下面我们来看一下对该变量的定义：

```
int numnodes = 1;       /* Initialized for UMA platforms */

static bootmem_data_t contig_bootmem_data;
pg_data_t contig_page_data = { bdata: &contig_bootmem_data };
```

变量 contig_page_data 的类型就是前面介绍过的 pg_data_t 数据结构。每个 pg_data_t 数据结构代表着一片均匀的、连续的内存空间。在连续空间 UMA 结构中，只有一个节点 contig_page_data，而在 NUMA 结构或不连续空间 UMA 结构中，有多个这样的数据结构。系统中各个节点的 pg_data_t 数据结构通过 node_next 连接在一起成为一个链。有一个全局量 pgdat_list 则指向这个链。从上面的定义可以看出，contig_page_data 是链中的第一个节点。这里假定整个物理空间为均匀的、连续的，以后若发现这个假定不能成立，则将新的 pg_data_t 结构加入到链中。

pg_data_t 结构中有个指针 bdata，contig_page_data 被初始化为指向 bootmem_data_t 数据结构。下面我们来看 init_bootmem_core（）函数的具体代码：

```
/*
  * Called once to set up the allocator itself.
    */
static unsigned long __init init_bootmem_core  (pg_data_t *pgdat,
         unsigned long mapstart, unsigned long start, unsigned long end)
{
         bootmem_data_t *bdata = pgdat->bdata;
         unsigned long mapsize =  ( (end - start) +7) /8;

         pgdat->node_next = pgdat_list;
         pgdat_list = pgdat;

         mapsize =  (mapsize +  (sizeof (long)  - 1UL) )  & ~ (sizeof (long)  - 1UL) ;
         bdata->node_bootmem_map = phys_to_virt (mapstart << PAGE_SHIFT) ;
         bdata->node_boot_start =  (start << PAGE_SHIFT) ;
         bdata->node_low_pfn = end;

        /*
          * Initially all pages are reserved - setup_arch ()  has to
          * register free RAM areas explicitly.
         */
         memset (bdata->node_bootmem_map, 0xff, mapsize) ;

         return mapsize;
}
```

下面对这一函数给予说明。

- 变量 mapsize 存放位图的大小。(end - start)给出现有的页面数，再加个 7 是为了向上取整，除以 8 就获得了所需的字节数（因为每个字节映射 8 个页面）。
- 变量 pgdat_list 用来指向节点所形成的循环链表首部，因为只有一个节点，因此使 pgdat_list 指向自己。
- 接下来的一句使 memsize 成为下一个 4 的倍数（4 为 CPU 的字长）。例如，假设有 40 个物理页面，因此，我们可以得出 memsize 为 5 个字节。所以，上面的操作就变为（5＋（4－1））&~(4-1)即(00001000&11111100)，最低的两位变为 0，其结果为 8。这就有效地使

memsize 变为 4 的倍数。

- phys_to_virt(mapstart << PAGE_SHIFT)把给定的物理地址转换为虚地址。
- 用节点的起始物理地址初始化 node_boot_start（这里为 0x00000000）。
- 用物理内存节点的页面号初始化 node_low_pfn。
- 初始化所有被保留的页面，即通过把页面中的所有位都置为 1 来标记保留的页面。
- 返回位图的大小。

2．free_bootmem( )函数

这个函数把给定范围的页面标记为空闲（即可用），也就是，把位图中某些位清 0，表示相应的物理内存可以投入分配。

原函数为：

```
void __init free_bootmem  (unsigned long addr, unsigned long size)
{

    return (free_bootmem_core (contig_page_data.bdata, addr, size) ) ;
 }
```

从上面可以看出，free_bootmem()是个封装函数，实际的工作是由 free_bootmem_core()函数完成的：

```
static void __init free_bootmem_core (bootmem_data_t *bdata, unsigned long addr, unsigned
long size)
{
         unsigned long i;
         unsigned long start;
         /*
          * round down end of usable mem, partially free pages are
         * considered reserved.
          */
        unsigned long sidx;
        unsigned long eidx =  (addr + size - bdata->node_boot_start) /PAGE_SIZE;
        unsigned long end =  (addr + size) /PAGE_SIZE;

        if  (!size)  BUG () ;
          if  (end > bdata->node_low_pfn)
                BUG () ;

           /*
          * Round up the beginning of the address.
          */
        start =  (addr + PAGE_SIZE-1)  / PAGE_SIZE;
        sidx = start -  (bdata->node_boot_start/PAGE_SIZE) ;

        for  (i = sidx; i < eidx; i++)  {
                if  (!test_and_clear_bit (i, bdata->node_bootmem_map) )
                        BUG () ;
         }
}
```

对此函数的解释如下。

- 变量 eidx 被初始化为页面总数。
- 变量 end 被初始化为最后一个页面的页面号。
- 进行两个可能的条件检查。
- start 初始化为第一个页面的页面号（向上取整），而 sidx(start index)初始化为相对于 node_boot_start 的页面号。
- 清位图中从 sidx 到 eidx 的所有位，即把这些页面标记为可用。

3．reserve_bootmem()函数

这个函数用来保留页面。为了保留一个页面，只需要在 bootmem 位图中把相应的位置为 1 即可。

原函数为：

```
  void __init reserve_bootmem  (unsigned long addr, unsigned long size)
   {
         reserve_bootmem_core (contig_page_data.bdata, addr, size) ;
   }
```

reserve_bootmem () 为封装函数，实际调用的是 reserve_bootmem_core () 函数：

```
  static void __init reserve_bootmem_core (bootmem_data_t *bdata, unsigned long addr,
unsigned long size)
{
         unsigned long i;
         /*
          * round up, partially reserved pages are considered
          * fully reserved.
         */
         unsigned long sidx =  (addr - bdata->node_boot_start) /PAGE_SIZE;
         unsigned long eidx =  (addr + size - bdata->node_boot_start +
                                                    PAGE_SIZE-1) /PAGE_SIZE;
         unsigned long end =  (addr + size + PAGE_SIZE-1) /PAGE_SIZE;

         if  (!size)  BUG () ;

         if  (sidx < 0)
                BUG () ;
         if  (eidx < 0)
                BUG () ;
        if  (sidx >= eidx)
                BUG () ;
         if  ( (addr >> PAGE_SHIFT)  >= bdata->node_low_pfn)
                BUG () ;
         if  (end > bdata->node_low_pfn)
                BUG () ;
         for  (i = sidx; i < eidx; i++)
                if  (test_and_set_bit (i, bdata->node_bootmem_map) )
                       printk ("hm, page %08lx reserved twice.\n", i*PAGE_SIZE) ;
}
```

对此函数的解释如下。

- sidx (start index)初始化为相对于 node_boot_start 的页面号。

- 变量 eidx 初始化为页面总数（向上取整）。
- 变量 end 初始化为最后一个页面的页面号（向上取整）。
- 进行各种可能的条件检查。
- 把位图中从 sidx 到 eidx 的所有位置 1。

4．__alloc_bootmem()函数

这个函数以循环轮转的方式从不同节点分配页面。因为在 i386 上只有一个节点，因此只循环一次。

函数原型为：

```
void * __alloc_bootmem  (unsigned long size,
                         unsigned long align,
                         unsigned long goal) ;
void * __alloc_bootmem_core  (bootmem_data_t *bdata,
                              unsigned long size,
                              unsigned long align,
                              unsigned long goal) ;
```

其中__alloc_bootmem()为封装函数，实际调用的函数为__alloc_bootmem_core （），因为__alloc_bootmem_core （）函数比较长，下面分片断来进行仔细分析。

```
unsigned long i, start = 0;
void *ret;
unsigned long offset, remaining_size;
unsigned long areasize, preferred, incr;
unsigned long eidx = bdata->node_low_pfn -
                (bdata->node_boot_start >> PAGE_SHIFT) ;
```

把 eidx 初始化为本节点中现有页面的总数。

```
if  (!size)  BUG () ;
if  (align &  (align-1) )
    BUG () ;
```

进行条件检查。

```
/*
* We try to allocate bootmem pages above 'goal'
* first, then we try to allocate lower pages.
*/
if  (goal &&  (goal >= bdata->node_boot_start)  &&
          ( (goal >> PAGE_SHIFT)  < bdata->node_low_pfn) )  {
    preferred = goal - bdata->node_boot_start;
} else
preferred = 0;
preferred =  ( (preferred + align - 1)  & ~ (align - 1) )  >> PAGE_SHIFT;
```

开始分配后首选页的计算分为两步：

（1）如果 goal 为非 0 且有效，则给 preferred 赋初值，否则，其初值为 0。

（2）根据参数 align 来对齐 preferred 的物理地址。

```
areasize =  (size+PAGE_SIZE-1) /PAGE_SIZE;
```

获得所需页面的总数（向上取整）

```
incr = align >> PAGE_SHIFT ? : 1;
```

根据对齐的大小来选择增加值。除非大于 4KB（很少见），否则增加值为 1。

```
restart_scan:
     for  (i = preferred; i < eidx; i += incr)  {
     unsigned long j;
        if  (test_bit (i, bdata->node_bootmem_map) )
          continue;
```

这个循环用来从首选页面号开始，找到空闲的页面号。test_bit()宏用来测试给定的位，如果给定位为 1，则返回 1。

```
for  (j = i + 1; j < i + areasize; ++j)  {
    if  (j >= eidx)
        goto fail_block;
    if  (test_bit  (j, bdata->node_bootmem_map) )
       goto fail_block;
}
```

这个循环用来查看在首次满足内存需求以后，是否还有足够的空闲页面。如果没有空闲页，就跳到 *fail_block*。

```
start = i;
    goto found;
```

如果一直到了这里，则说明从 *i* 开始找到了足够的页面，跳过 fail_block 并继续。

```
fail_block:;
   }
 if  (preferred)  {
    preferred = 0;
    goto restart_scan;
       }
return NULL;
```

如果到了这里，从首选页面中没有找到满足需要的连续页面，就忽略 preferred 的值，并从 0 开始扫描。如果 preferred 为 1，但没有找到满足需要的足够页面，则返回 NULL。

```
found:
```

已经找到足够的内存，继续处理请求。

```
 if  (start >= eidx)
     BUG () ;
```

进行条件检查。

```
/*
* Is the next page of the previous allocation-end the start
* of this allocation's buffer? If yes then we can 'merge'
* the previous partial page with this allocation.
*/
if  (align <= PAGE_SIZE && bdata->last_offset
                      && bdata->last_pos+1 == start)  {
    offset =  (bdata->last_offset+align-1)  & ~ (align-1) ;
    if  (offset > PAGE_SIZE)
        BUG () ;
    remaining_size = PAGE_SIZE-offset;
```

if 语句检查下列条件：

(1) 所请求对齐的值小于页的大小（4KB）。

(2) 变量 last_offset 为非 0。如果为 0，则说明前一次分配达到了一个非常好的页面边界，没有内部碎片。

(3) 检查这次请求的内存是否与前一次请求的内存是相临的，如果是，则把两次分配合在一起进行。

如果以上 3 个条件都满足，则用前一次分配中最后一页剩余的空间初始化 remaining_size。

```
            if (size < remaining_size) {
                areasize = 0;
                // last_pos unchanged
                bdata->last_offset = offset+size;
                ret = phys_to_virt (bdata->last_pos*PAGE_SIZE
                            + offset + bdata->node_boot_start) ;
```

如果请求内存的大小小于前一次分配中最后一页中的可用空间，则没必要分配任何新的页。变量 last_offset 增加到新的偏移量，而 last_pos 保持不变，因为没有增加新的页。把这次新分配的起始地址存放在变量 ret 中。宏 phys_to_virt()返回给定物理地址的虚地址。

```
            } else {
                remaining_size = size - remaining_size;
                areasize = (remaining_size+PAGE_SIZE-1) /PAGE_SIZE;
                ret = phys_to_virt (bdata->last_pos*PAGE_SIZE
                             + offset + bdata->node_boot_start) ;
                bdata->last_pos = start+areasize-1;
                bdata->last_offset = remaining_size;
```

所请求的大小大于剩余的大小。首先求出所需的页面数，然后更新变量 last_pos 和 last_offset。

例如，在前一次分配中，如果分配了 9KB，则占用 3 个页面，内部碎片为 12KB-9KB=3KB。因此，page_offset 为 1KB，且剩余大小为 3KB。如果新的请求为 1KB，则第 3 个页面本身就能满足要求，但是，如果请求的大小为 10KB，则需要新分配((10KB- 3KB) + PAGE_SIZE-1)/PAGE_SIZE，即 2 个页面，因此，page_offset 为 3KB。

```
            }
            bdata->last_offset &= ~PAGE_MASK;
        } else {
            bdata->last_pos = start + areasize - 1;
            bdata->last_offset = size & ~PAGE_MASK;
            ret = phys_to_virt (start * PAGE_SIZE +
                                    bdata->node_boot_start) ;
           }
```

如果因为某些条件未满足而导致不能进行合并，则执行这段代码，我们刚刚把 last_pos 和 last_offset 直接设置为新的值，而未考虑它们原先的值。last_pos 的值还要加上所请求的页面数，而新 page_offset 值的计算就是屏蔽掉除了获得页偏移量位的所有位，即“size & PAGE_MASK”，PAGE_MASK 为 0x00000FFF，用 PAGE_MASK 的求反正好得到页的偏移量。

```
            /*
            * Reserve the area now:
            */

            for (i = start; i < start+areasize; i++)
                if (test_and_set_bit (i, bdata->node_bootmem_map) )
                    BUG () ;
            memset (ret, 0, size) ;
```

```
        return ret;
```

现在，我们有了内存，就需要保留它。宏 test_and_set_bit()用来测试并置位，如果某位原先的值为 0，则它返回 0；如果为 1，则返回 1。还有一个条件判断语句，进行条件判断(这种条件出现的可能性非常小，除非 RAM 坏)。然后，把这块内存初始化为 0，并返回给调用它的函数。

5. free_all_bootmem()函数

这个函数用来在引导时释放页面，并清除 bootmem 分配器。

函数原型为：

```
    void free_all_bootmem (void) ;
    void free_all_bootmem_core (pg_data_t *pgdat) ;
```

同前面的函数调用形式类似，free_all_bootmem () 为封装函数，实际调用 free_all_bootmem_core () 函数。下面，我们对 free_all_bootmem_core () 函数分片断来介绍。

```
        struct page *page = pgdat->node_mem_map;
        bootmem_data_t *bdata = pgdat->bdata;
        unsigned long i, count, total = 0;
        unsigned long idx;

        if (!bdata->node_bootmem_map) BUG () ;
        count = 0;
        idx = bdata->node_low_pfn - (bdata->node_boot_start
                                        >> PAGE_SHIFT) ;
```

把 idx 初始化为从内核映像结束处到内存顶点处的页面数。

```
    for (i = 0; i < idx; i++, page++) {
        if (!test_bit (i, bdata->node_bootmem_map) ) {
            count++;
            ClearPageReserved (page) ;
            set_page_count (page, 1) ;
            __free_page (page) ;
        }
    }
```

搜索 bootmem 位图,找到空闲页,并把 mem_map 中对应的项标记为空闲。set_page_count()函数把 page 结构的 count 域置 1，而__free_page()真正的释放页面，并修改伙伴（buddy）系统的位图。

```
        total += count;

        /*
        * Now free the allocator bitmap itself, it's not
        * needed anymore:
        */
        page = virt_to_page (bdata->node_bootmem_map) ;
        count = 0;
        for (i = 0; i < ( (bdata->node_low_pfn- (bdata->node_boot_start
                    >> PAGE_SHIFT) ) /8 + PAGE_SIZE-1) /PAGE_SIZE;
                                                i++,page++) {
            count++;
```

```
            ClearPageReserved (page) ;
            set_page_count (page, 1) ;
            __free_page (page) ;
        }
```

获得 bootmem 位图的地址，并释放它所在的页面。

```
        total += count;
        bdata->node_bootmem_map = NULL;
        return total;
```

把该存储节点的 bootmem_map 域置为 NULL，并返回空闲页面的总数。

### 6.2.5 页表的建立

前面已经建立了为内存页面管理所需的数据结构，现在是进一步完善页面映射机制，并且建立起内存页面映射管理机制的时候了，与此相关的主要函数有：

paging_init() 函数

pagetable_init() 函数

1. paging_init() 函数

这个函数仅被调用一次，即由 setup_arch()调用以建立页表，对此函数的具体描述如下：

```
        pagetable_init () ;
```

这个函数实际上才真正地建立页表，后面会给出详细描述。

```
__asm__ ( "movl %%ecx,%%cr3\n" ::"c" (__pa (swapper_pg_dir) ) ) ;
```

因为 pagetable_init()已经建立起页表，因此把 swapper_pg_dir（页目录）的地址装入 CR3 寄存器。

```
        #if CONFIG_X86_PAE
        /*
        * We will bail out later - printk doesnt work right now so
        * the user would just see a hanging kernel.
        */
        if  (cpu_has_pae)
            set_in_cr4 (X86_CR4_PAE) ;
        #endif

        __flush_tlb_all () ;
```

上面这一句是个宏，它使得转换旁路缓冲区（TLB）无效。TLB 总是要维持几个最新的虚地址到物理地址的转换。每当页目录改变时，TLB 就需要被刷新。

```
        #ifdef CONFIG_HIGHMEM
        kmap_init () ;
        #endif
```

如果使用了 CONFIG_HIGHMEM 选项，就要对大于 896MB 的内存进行初始化，我们不准备对这部分内容进行详细讨论。

```
        {
            unsigned long zones_size[MAX_NR_ZONES] = {0, 0, 0};
            unsigned int max_dma, high, low;
```

```
            max_dma = virt_to_phys ( (char *) MAX_DMA_ADDRESS)
                                    >> PAGE_SHIFT;
```

低于 16MB 的内存只能用于 DMA，因此，上面这条语句用于存放 16MB 的页面。

```
            low = max_low_pfn;
            high = highend_pfn;

        if  (low < max_dma)
            zones_size[ZONE_DMA] = low;
        else {
            zones_size[ZONE_DMA] = max_dma;
            zones_size[ZONE_NORMAL] = low - max_dma;
        #ifdef CONFIG_HIGHMEM
            zones_size[ZONE_HIGHMEM] = high - low;
        #endif
           }
```

计算 3 个管理区的大小，并存放在 zones_size 数组中。3 个管理区如下所述。

- ZONE_DMA：从 0～16MB 分配给这个区。
- ZONE_NORMAL：从 16MB～896MB 分配给这个区。
- ZONE_DMA ：896MB 以上分配给这个区。

```
            free_area_init (zones_size) ;
            }

            return;
```

这个函数用来初始化内存管理区并创建内存映射表，详细介绍参见后面内容。

2. pagetable_init()函数

这个函数真正地在页目录 swapper_pg_dir 中建立页表，描述如下：

```
        unsigned long vaddr, end;
        pgd_t *pgd, *pgd_base;
        int i, j, k;
        pmd_t *pmd;
        pte_t *pte, *pte_base;

        /*
        * This can be zero as well - no problem, in that case we exit
        * the loops anyway due to the PTRS_PER_* conditions.
        */
        end =  (unsigned long) __va (max_low_pfn*PAGE_SIZE) ;
```

计算 max_low_pfn 的虚拟地址，并把它存放在 end 中。

```
        pgd_base = swapper_pg_dir;
```

让 pgd_base（页目录基地址）指向 swapper_pg_dir。

```
        #if CONFIG_X86_PAE
        for  (i = 0; i < PTRS_PER_PGD; i++)
            set_pgd (pgd_base + i, __pgd (1 + __pa (empty_zero_page) ) ) ;
        #endif
```

如果 PAE 被激活，PTRS_PER_PGD 就为 4，且变量 swapper_pg_dir 用作页目录指针表，宏 set_pgd() 定义于 include/asm-i386/pgtable-3level.h 中。

```
i = __pgd_offset (PAGE_OFFSET) ;
pgd = pgd_base + i;
```

宏__pgd_offset()在给定地址的页目录中检索相应的下标。因此__pgd_offset (PAGE_OFFSET) 返回 0x300（或 十进制 768)，即内核地址空间开始处的下标。因此，pgd 现在指向页目录表的第 768 项。

```
for  (; i < PTRS_PER_PGD; pgd++, i++)  {
     vaddr = i*PGDIR_SIZE;
     if  (end &&  (vaddr >= end) )
        break;
```

如果使用了 CONFIG_X86_PAE 选项， PTRS_PER_PGD 就为 4，否则，一般情况下它都为 1024，即页目录的项数。PGDIR_SIZE 给出一个单独的页目录项所能映射的 RAM 总量，在两级页目录中它为 4MB，当使用 CONFIG_X86_PAE 选项时，它为 1GB。计算虚地址 vaddr，并检查它是否到了虚拟空间的顶部。

```
#if CONFIG_X86_PAE
     pmd =  (pmd_t *)  alloc_bootmem_low_pages (PAGE_SIZE) ;
     set_pgd (pgd, __pgd (__pa (pmd)  + 0x1) ) ;
#else
      pmd =  (pmd_t *) pgd;
#endif
```

如果使用了 CONFIG_X86_PAE 选项，则分配一页（4KB）的内存给 bootmem 分配器用，以保存中间页目录，并在总目录中设置它的地址。否则，没有中间页目录，就把中间页目录直接映射到总目录。

```
if  (pmd != pmd_offset (pgd, 0) )
     BUG () ;
     for  (j = 0; j < PTRS_PER_PMD; pmd++, j++)  {
     vaddr = i*PGDIR_SIZE + j*PMD_SIZE;
     if  (end &&  (vaddr >= end) )
          break;
      if  (cpu_has_pse)  {
          unsigned long __pe;
          set_in_cr4 (X86_CR4_PSE) ;
          boot_cpu_data.wp_works_ok = 1;
          __pe = _KERNPG_TABLE + _PAGE_PSE + __pa (vaddr) ;
          /* Make it "global" too if supported */
          if  (cpu_has_pge)  {
               set_in_cr4 (X86_CR4_PGE) ;
               __pe += _PAGE_GLOBAL;
          }
          set_pmd (pmd, __pmd (__pe) ) ;
           continue;
     }
```

现在，开始填充页目录（如果有 PAE，就是填充中间页目录)。计算表项所映射的虚地址，如果没有激活 PAE， PMD_SIZE 大小就为 0，因此，vaddr = i * 4MB。例如，表项 0x300 所映射的虚地址为 0x300 * 4MB = 3GB。接下来，我们检查 PSE (Page Size Extension)是否可用，如果是，就要避免使用页表而直接使用 4MB 的页。宏 cpu_has_pse () 用来检查处理器是否具有扩展页，如果有，则宏 set_in_cr4()就启用它。

从 Pentium II 处理器开始，就可以有附加属性 PGE (Page Global Enable)。当一个页被标记为全局的，且设置了 PGE，那么，在任务切换发生或 CR3 被装入时，就不能使该页所在的页表（或页目录项）无效。这将提高系统性能，也是让内核处于 3GB 以上的原因之一。选择了所有属性后，设置中间页目录项。

```
                pte_base = pte =  (pte_t *)
                            alloc_bootmem_low_pages (PAGE_SIZE) ;
```

如果 PSE 不可用，就执行这一句，它为一个页表（4KB）分配空间。

```
                for  (k = 0; k < PTRS_PER_PTE; pte++, k++)  {
                    vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
                    if  (end &&  (vaddr >= end) )
                        break;
```

在一个页表中有 1024 个表项（如果启用 PAE，就是 512 个），每个表项映射 4KB（1 页）。

```
                    *pte = mk_pte_phys (__pa (vaddr) , PAGE_KERNEL) ;
                }
```

宏 mk_pte_phys()创建一个页表项，这个页表项的物理地址为__pa(vaddr)。属性 PAGE_KERNEL 表示只有在内核态才能访问这一页表项。

```
                set_pmd (pmd, __pmd (_KERNPG_TABLE + __pa (pte_base) ) ) ;
                if  (pte_base != pte_offset (pmd, 0) )
                    BUG () ;
            }
        }
```

通过调用 set_pmd()把该页表追加到中间页目录中。这个过程一直继续，直到把所有的物理内存都映射到从 PAGE_OFFSET 开始的虚拟地址空间。

```
        /*
             * Fixed mappings, only the page table structure has to be
             * created - mappings will be set by set_fixmap () :
             */
             vaddr = __fix_to_virt (__end_of_fixed_addresses - 1)  & PMD_MASK;
             fixrange_init (vaddr, 0, pgd_base) ;
```

在内存的最高端(4GB～128MB)，有些虚地址直接用在内核资源的某些部分中，这些地址的映射定义在/include/asm/fixmap.h 中，枚举类型__end_of_fixed_addresses 用作索引，宏__fix_to_virt()返回给定索引的虚地址。函数 fixrange_init()为这些虚地址创建合适的页表项。注意，这里仅仅创建了页表项，而没有进行映射。这些地址的映射是由 set_fixmap()函数完成的。

```
             #if CONFIG_HIGHMEM
             /*
             * Permanent kmaps:
             */
             vaddr = PKMAP_BASE;
             fixrange_init (vaddr, vaddr + PAGE_SIZE*LAST_PKMAP, pgd_base) ;
             pgd = swapper_pg_dir + __pgd_offset (vaddr) ;
             pmd = pmd_offset (pgd, vaddr) ;
             pte = pte_offset (pmd, vaddr) ;
             pkmap_page_table = pte;
             #endif
```

如果使用了 CONFIG_HIGHMEM 选项，我们就可以访问 896MB 以上的物理内存，这些内存

的地址被暂时映射到为此目的而保留的虚地址上。PKMAP_BASE 的值为 0xFE000000（即 4064MB），LAST_PKMAP 的值为 1024。因此，从 4064MB 开始，由 fixrange_init()在页表中创建的表项能覆盖 4MB 的空间。接下来，把覆盖 4MB 内存的页表项赋给 pkmap_page_table。

```
#if CONFIG_X86_PAE
/*
* Add low memory identity-mappings - SMP needs it when
* starting up on an AP from real-mode. In the non-PAE
* case we already have these mappings through head.S.
* All user-space mappings are explicitly cleared after
* SMP startup.
*/
pgd_base[0] = pgd_base[USER_PTRS_PER_PGD];
#endif
```

### 6.2.6 内存管理区

前面已经提到，物理内存被划分为 3 个区来管理，它们是 ZONE_DMA、ZONE_NORMAL 和 ZONE_HIGHMEM。每个区都用 struct zone_struct 结构来表示，定义于 include/linux/mmzone.h：

```
typedef struct zone_struct {
/*
* Commonly accessed fields:
*/
    spinlock_t lock;
    unsigned long    free_pages;
    unsigned long    pages_min, pages_low, pages_high;
    int              need_balance;

/*
* free areas of different sizes
*/
    free_area_t free_area[MAX_ORDER];

/*
* Discontig memory support fields.
*/
    struct pglist_data *zone_pgdat;
    struct page *zone_mem_map;
    unsigned long          zone_start_paddr;
    unsigned long          zone_start_mapnr;

/*
* rarely used fields:
*/
    char                   *name;
    unsigned long          size;
} zone_t;

#define ZONE_DMA               0
```

```
#define ZONE_NORMAL            1
#define ZONE_HIGHMEM           2
#define MAX_NR_ZONES           3
```

对 struct zone_struct 结构中每个域的描述如下。

- lock ：用来保证对该结构中其他域的串行访问。
- free_pages ：在这个区中现有空闲页的个数。
- pages_min、pages_low 及 pages_high 是对这个区最少、次少及最多页面个数的描述。
- need_balance：与 kswapd 合在一起使用。
- free_area：在伙伴分配系统中的位图数组和页面链表。
- zone_pgdat：本管理区所在的存储节点。
- zone_mem_map：该管理区的内存映射表。
- zone_start_paddr：该管理区的起始物理地址。
- zone_start_mapnr：在 mem_map 中的索引（或下标）。
- name：该管理区的名字。
- size：该管理区物理内存总的大小。

其中，free_area_t 定义为：

```
#difine   MAX_ORDER  10
    type struct free_area_struct {
           struct list_head   free_list
            unsigned  int    *map
     } free_area_t
```

因此，zone_struct 结构中的 free_area[MAX_ORDER]是一组“空闲区间”链表。为什么要定义一组而不是一个空闲队列呢？这是因为常常需要成块地在物理空间分配连续的多个页面，所以要按块的大小分别加以管理。因此，在管理区数据结构中既要有一个队列来保持一些离散（连续长度为 1）的物理页面，还要有一个队列来保持一些连续长度为 2 的页面块以及连续长度为 4、8、16、……、直至 $2^{MAX\_ORDER}$（即 4M 字节）的队列。

如前所述，内存中每个物理页面都有一个 struct page 结构，位于 include/linux/mm.h，该结构包含了对物理页面进行管理的所有信息，下面给出具体描述：

```
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct list_head lru;
    wait_queue_head_t wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
    void *virtual;
    struct zone_struct *zone;
} mem_map_t;
```

对每个域的描述如下。

- list ：指向链表中的下一页。

- mapping ：用来指定我们正在映射的索引节点（inode)。
- index：在映射表中的偏移。
- next_hash：指向页高速缓存哈希表中下一个共享的页。
- count：引用这个页的个数。
- flags：页面各种不同的属性。
- lru：用在 active_list 中。
- wait：等待这一页的页队列。
- pprev_hash：与 next_hash 相对应。
- buffers：把缓冲区映射到一个磁盘块。
- zone：页所在的内存管理区。

与内存管理区相关的 3 个主要函数为：

- free_area_init() 函数；
- build_zonelists() 函数；
- mem_init() 函数。

1. free_area_init() 函数

这个函数用来初始化内存管理区并创建内存映射表，定义于 mm/page_alloc.c 中。

函数原型为：

```
void free_area_init (unsigned long *zones_size) ;
void free_area_init_core (int nid, pg_data_t *pgdat,
                          struct page **gmap,
                          unsigned long *zones_size,
                          unsigned long zone_start_paddr,
                          unsigned long *zholes_size,
                          struct page *lmem_map) ;
```

free_area_init () 为封装函数，而 free_area_init_core () 为真正实现的函数，对该函数详细描述如下：

```
          struct page *p;
          unsigned long i, j;
     unsigned long map_size;
          unsigned long totalpages, offset, realtotalpages;
     const unsigned long zone_required_alignment = 1UL <<  (MAX_ORDER-1) ;

          if  (zone_start_paddr & ~PAGE_MASK)
         BUG () ;
```

检查该管理区的起始地址是否是一个页的边界。

```
          totalpages = 0;
          for  (i = 0; i < MAX_NR_ZONES; i++)  {
               unsigned long size = zones_size[i];
               totalpages += size;
          }
```

计算本存储节点中页面的个数。

```
     realtotalpages = totalpages;
          if  (zholes_size)
```

```
            for (i = 0; i < MAX_NR_ZONES; i++)
                realtotalpages -= zholes_size[i];
        printk ("On node %d totalpages: %lu\n", nid, realtotalpages) ;
```

打印除空洞以外的实际页面数。

```
        INIT_LIST_HEAD (&active_list) ;
        INIT_LIST_HEAD (&inactive_list) ;
```

初始化循环链表。

```
        /*
        * Some architectures (with lots of mem and discontinous memory
        * maps) have to search for a good mem_map area:
        * For discontigmem, the conceptual mem map array starts from
        * PAGE_OFFSET, we need to align the actual array onto a mem map
        * boundary, so that MAP_NR works.
        */
        map_size = (totalpages + 1) *sizeof (struct page) ;
        if (lmem_map == (struct page *) 0) {
            lmem_map = (struct page *)
                       alloc_bootmem_node (pgdat, map_size) ;
            lmem_map = (struct page *) (PAGE_OFFSET +
            MAP_ALIGN ( (unsigned long) lmem_map - PAGE_OFFSET) ) ;
        }
```

给局部内存（即本节点中的内存）映射分配空间，并在 sizeof(mem_map_t)边界上对齐它。

```
    *gmap = pgdat->node_mem_map = lmem_map;
        pgdat->node_size = totalpages;
    pgdat->node_start_paddr = zone_start_paddr;
        pgdat->node_start_mapnr = (lmem_map - mem_map) ;
    pgdat->nr_zones = 0;
```

初始化本节点中的域。

```
            /*
        * Initially all pages are reserved - free ones are freed
        * up by free_all_bootmem () once the early boot process is
        * done.
        */
        for (p = lmem_map; p < lmem_map + totalpages; p++) {
            set_page_count (p, 0) ;
            SetPageReserved (p) ;
            init_waitqueue_head (&p->wait) ;
            memlist_init (&p->list) ;
        }
```

仔细检查所有的页，并进行如下操作。

(1) 把页的使用计数（count 域）置为 0。

(2) 把页标记为保留。

(3) 初始化该页的等待队列。

(4) 初始化链表指针。

```
        offset = lmem_map - mem_map;
```

变量 mem_map 是类型为 struct pages 的全局稀疏矩阵。mem_map 下标的起始值取决于第一个节点的第一个管理区。如果第一个管理区的起始地址为 0，则下标就从 0 开始，并且与

物理页面号相对应，也就是说，页面号就是 mem_map 的下标。每一个管理区都有自己的映射表，存放在 zone_mem_map 中，每个管理区又被映射到它所在的节点 node_mem_map 中，而每个节点又被映射到管理全局内存的 mem_map 中。

在上面的这行代码中,offset 表示该节点放的内存映射表在全局 mem_map 中的入口点(下标)。在这里，offset 为 0，因为在 i386 上，只有一个节点。

```
        for (j = 0; j < MAX_NR_ZONES; j++) {
```

这个循环对 zone 的域进行初始化。

```
        zone_t *zone = pgdat->node_zones + j;
        unsigned long mask;
        unsigned long size, realsize;
        realsize = size = zones_size[j];
```

管理区的实际数据是存放在节点中的，因此，让指针指向正确的管理区，并获得该管理区的大小。

```
        if (zholes_size)
            realsize -= zholes_size[j];

        printk ("zone (%lu) : %lu pages.\n", j, size) ;
```

计算各个区的实际大小，并进行打印。例如，在具有 256MB 的内存上，上面的输出为：

```
zone (0) : 4096 pages.
zone (1) : 61440 pages.
zone (2) : 0 pages.
```

这里，管理区 2 为 0，因为只有 256MB 的 RAM.。

```
        zone->size = size;
        zone->name = zone_names[j];
        zone->lock = SPIN_LOCK_UNLOCKED;
        zone->zone_pgdat = pgdat;
        zone->free_pages = 0;
        zone->need_balance = 0;
```

初始化管理区中的各个域。

```
        if (!size)
            continue;
```

如果一个管理区的大小为 0，就没必要进一步的初始化。

```
        pgdat->nr_zones = j+1;
        mask = (realsize / zone_balance_ratio[j]) ;
        if (mask < zone_balance_min[j])
            mask = zone_balance_min[j];
        else if (mask > zone_balance_max[j])
            mask = zone_balance_max[j];
```

计算合适的平衡比率。

```
        zone->pages_min = mask;
        zone->pages_low = mask*2;
        zone->pages_high = mask*3;
        zone->zone_mem_map = mem_map + offset;
        zone->zone_start_mapnr = offset;
        zone->zone_start_paddr = zone_start_paddr;
```

设置该管理区中页面数量的几个界限，并把在全局变量 mem_map 中的入口点作为

zone_mem_map 的初值。用全局变量 mem_map 的下标初始化变量 zone_start_mapnr。

```
        if  ( (zone_start_paddr >> PAGE_SHIFT)  &
                                    (zone_required_alignment-1) )
            printk ("BUG: wrong zone alignment, it will crash\n") ;

        for  (i = 0; i < size; i++)  {
            struct page *page = mem_map + offset + i;
            page->zone = zone;
            if  (j != ZONE_HIGHMEM)
                page->virtual = __va (zone_start_paddr) ;
            zone_start_paddr += PAGE_SIZE;
        }
```

对该管理区中的每一页进行处理。首先，把 struct page 结构中的 zone 域初始化为指向该管理区（zone）,如果这个管理区不是 ZONE_HIGHMEM，则设置这一页的虚地址（即物理地址 + PAGE_OFFSET)。也就是说，建立起每一页物理地址到虚地址的映射。

```
        offset += size;
```

把 offset 增加 size，使它指向 mem_map 中下一个管理区的起始位置。

```
        for  (i = 0; ; i++)  {
            unsigned long bitmap_size;
            memlist_init (&zone->free_area[i].free_list) ;
            if  (i == MAX_ORDER-1)  {
                zone->free_area[i].map = NULL;
                break;
            }
```

初始化 free_area[]链表，把 free_area[]中最后一个序号的位图置为 NULL。

```
            /*
            * Page buddy system uses "index >>  (i+1) ",
            * where "index" is at most "size-1".
            *
            * The extra "+3" is to round down to byte
            * size  (8 bits per byte assumption) . Thus
            * we get " (size-1)  >>  (i+4) " as the last byte
            * we can access.
            *
            * The "+1" is because we want to round the
            * byte allocation up rather than down. So
            * we should have had a "+7" before we shifted
            * down by three. Also, we have to add one as
            * we actually _use_ the last bit  (it's [0,n]
            * inclusive, not [0,n[) .
            *
            * So we actually had +7+1 before we shift
            * down by 3. But  (n+8)  >> 3 ==  (n >> 3)  + 1
            *  (modulo overflows, which we do not have) .
            *
            * Finally, we LONG_ALIGN because all bitmap
            * operations are on longs.
            */
                    bitmap_size =  (size-1)  >>  (i+4) ;
                    bitmap_size = LONG_ALIGN (bitmap_size+1) ;
```

```
                zone->free_area[i].map =  (unsigned long *)
                        alloc_bootmem_node (pgdat, bitmap_size) ;
            }
```

计算位图的大小，然后调用 alloc_bootmem_node 给位图分配空间。

```
        }
        build_zonelists (pgdat) ;
```

在节点中为不同的管理区创建链表。

2. build_zonelists () 函数

函数原型：

```
  static inline void build_zonelists (pg_data_t *pgdat)
```

代码如下：

```
    int i, j, k;

    for  (i = 0; i <= GFP_ZONEMASK; i++)  {
        zonelist_t *zonelist;
        zone_t *zone;
        zonelist = pgdat->node_zonelists + i;
        memset (zonelist, 0, sizeof (*zonelist) ) ;
```

获得节点中指向管理区链表的域，并把它初始化为空。

```
        j = 0;
        k = ZONE_NORMAL;
        if  (i & __GFP_HIGHMEM)
            k = ZONE_HIGHMEM;
        if  (i & __GFP_DMA)
            k = ZONE_DMA;
```

把当前管理区掩码与 3 个可用管理区掩码相“与”，获得一个管理区标识，把它用在下面的 switch 语句中。

```
        switch  (k)  {
            default:
                BUG () ;
        /*
        * fallthrough:
        */
            case ZONE_HIGHMEM:
                zone = pgdat->node_zones + ZONE_HIGHMEM;
                if  (zone->size)  {
                #ifndef CONFIG_HIGHMEM
                    BUG () ;
                #endif
                    zonelist->zones[j++] = zone;
                }
            case ZONE_NORMAL:
                zone = pgdat->node_zones + ZONE_NORMAL;
                if  (zone->size)
                    zonelist->zones[j++] = zone;
            case ZONE_DMA:
                zone = pgdat->node_zones + ZONE_DMA;
```

```
                if  (zone->size)
                    zonelist->zones[j++] = zone;
        }
```

给定的管理区掩码指定了优先顺序，我们可以用它找到在 switch 语句中的入口点。如果掩码为__GFP_DMA，管理区链表 zonelist 将仅仅包含 DMA 管理区，如果为__GFP_HIGHMEM，则管理区链表中就会依次有 ZONE_HIGHMEM、 ZONE_NORMAL 和 ZONE_DMA。

```
        zonelist->zones[j++] = NULL;
    }
```

用 NULL 结束链表。

3. mem_init() 函数

这个函数由 start_kernel()调用，以对管理区的分配算法进行进一步的初始化，定义于 arch/i386/mm/init.c 中，具体解释如下：

```
        int codesize, reservedpages, datasize, initsize;
        int tmp;
        int bad_ppro;

        if  (!mem_map)
            BUG () ;

        #ifdef CONFIG_HIGHMEM
        highmem_start_page = mem_map + highstart_pfn;
        max_mapnr = num_physpages = highend_pfn;
```

如果 HIGHMEM 被激活，就要获得 HIGHMEM 的起始地址和总的页面数。

```
        #else
        max_mapnr = num_physpages = max_low_pfn;
        #endif
```

否则，页面数就是常规内存的页面数。

```
        high_memory =  (void *)  __va (max_low_pfn * PAGE_SIZE) ;
```

获得低区内存中最后一个页面的虚地址。

```
        /* clear the zero-page */
        memset (empty_zero_page, 0, PAGE_SIZE) ;

        /* this will put all low memory onto the freelists */
        totalram_pages += free_all_bootmem () ;
        reservedpages = 0;
```

free_all_bootmem()函数本质上释放所有的低区内存，从此以后，bootmem 不再使用。

```
        /*
        * Only count reserved RAM pages
        */
            for  (tmp = 0; tmp < max_low_pfn; tmp++)
            if  (page_is_ram (tmp)  && PageReserved (mem_map+tmp) )
                reservedpages++;
```

对 mem_map 查找一遍，并统计所保留的页面数。

```
            #ifdef CONFIG_HIGHMEM
            for  (tmp = highstart_pfn; tmp < highend_pfn; tmp++)  {
                struct page *page = mem_map + tmp;
```

```
                if  (!page_is_ram (tmp) )  {
                    SetPageReserved (page) ;
                    continue;
                }
            if  (bad_ppro && page_kills_ppro (tmp) )  {
                SetPageReserved (page) ;
                continue;
            }
            ClearPageReserved (page) ;
            set_bit (PG_highmem, &page->flags) ;
            atomic_set (&page->count, 1) ;
            __free_page (page) ;
             totalhigh_pages++;
        }

        totalram_pages += totalhigh_pages;
        #endif
```

把高区内存查找一遍，并把保留但不能使用的页面标记为 PG_highmem，并调用__free_page()释放它，还要修改伙伴系统的位图。

```
        codesize =   (unsigned long)  &_etext -  (unsigned long)  &_text;
        datasize =   (unsigned long)  &_edata -  (unsigned long)  &_etext;
        initsize =   (unsigned long)  &__init_end -
                                      (unsigned long)  &__init_begin;
        printk ("Memory: %luk/%luk available  (%dk kernel code,
                     %dk reserved,
                     %dk data, %dk init, %ldk highmem) \n",
                      (unsigned long)  nr_free_pages ()  <<
                                                (PAGE_SHIFT-10) ,
                     max_mapnr <<  (PAGE_SHIFT-10) ,
                     codesize >> 10,
                     reservedpages <<  (PAGE_SHIFT-10) ,
                     datasize >> 10,
                     initsize >> 10,
                      (unsigned long)   (totalhigh_pages
                                        <<  (PAGE_SHIFT-10) ) ) ;
```

计算内核各个部分的大小，并打印统计信息。

从以上的介绍可以看出，在初始化阶段，对内存的初始化要做许多工作。但这里要说明的是，尽管在这个阶段建立起了初步的虚拟内存管理机制，但仅仅考虑了内核虚拟空间（3GB以上），还根本没有涉及用户空间的管理。因此，在这个阶段，虚拟存储空间到物理存储空间的映射非常简单，仅仅通过一种简单的线性关系就可以达到虚地址到物理地址之间的相互转换。但是，了解这个初始化阶段又非常重要，它是后面进一步进行内存管理分析的基础。

## 6.3 内存的分配和回收

在内存初始化完成以后，内存中就常驻有内核映像（内核代码和数据）。以后，随着用户程序的执行和结束，就需要不断地分配和释放物理页面。内核应该为分配一组连续的页面

而建立一种稳定、高效的分配策略。为此，必须解决一个比较重要的内存管理问题，即外碎片问题。频繁地请求和释放不同大小的一组连续页面，必然导致在已分配的内存块中分散许多小块的空闲页面。由此带来的问题是，即使这些小块的空闲页面加起来足以满足所请求的页面，但是要分配一个大块的连续页面可能就根本无法满足。Linux 采用著名的伙伴（Buddy）系统算法来解决外碎片问题。

但是请注意，在 Linux 中，CPU 不能按物理地址来访问存储空间，而必须使用虚拟地址；因此，对于内存页面的管理，通常是先在虚存空间中分配一个虚存区间，然后才根据需要为此区间分配相应的物理页面并建立起映射，也就是说，虚存区间的分配在前，而物理页面的分配在后，但是为了承接上一节的内容，我们先介绍内存的分配和回收，然后再介绍用户进程虚存区间的建立。

### 6.3.1　伙伴算法

#### 1．原理

Linux 的伙伴算法把所有的空闲页面分为 10 个块组，每组中块的大小是 2 的幂次方个页面，例如，第 0 组中块的大小都为 $2^0$（1 个页面），第 1 组中块的大小都为 $2^1$（2 个页面），第 9 组中块的大小都为 $2^9$（512 个页面）。也就是说，每一组中块的大小是相同的，且这同样大小的块形成一个链表。

我们通过一个简单的例子来说明该算法的工作原理。

假设要求分配的块的大小为 128 个页面（由多个页面组成的块我们就叫做页面块）。该算法先在块大小为 128 个页面的链表中查找，看是否有这样一个空闲块。如果有，就直接分配；如果没有，该算法会查找下一个更大的块，具体地说，就是在块大小 256 个页面的链表中查找一个空闲块。如果存在这样的空闲块，内核就把这 256 个页面分为两等份，一份分配出去，另一份插入到块大小为 128 个页面的链表中。如果在块大小为 256 个页面的链表中也没有找到空闲页块，就继续找更大的块，即 512 个页面的块。如果存在这样的块，内核就从 512 个页面的块中分出 128 个页面满足请求，然后从 384 个页面中取出 256 个页面插入到块大小为 256 个页面的链表中。然后把剩余的 128 个页面插入到块大小为 128 个页面的链表中。如果 512 个页面的链表中还没有空闲块，该算法就放弃分配，并发出出错信号。

以上过程的逆过程就是块的释放过程，这也是该算法名字的来由。满足以下条件的两个块称为伙伴：

（1）两个块的大小相同；

（2）两个块的物理地址连续。

伙伴算法把满足以上条件的两个块合并为一个块，该算法是迭代算法，如果合并后的块还可以跟相邻的块进行合并，那么该算法就继续合并。

#### 2．数据结构

在 6.2.6 节中所介绍的管理区数据结构 struct zone_struct 中，涉及到空闲区数据结构：

```
free_area_t free_area[MAX_ORDER];
```

我们再次对 free_area_t 给予较详细的描述。

```
#difine   MAX_ORDER  10
    type struct free_area_struct {
           struct list_head   free_list
            unsigned  int    *map
     } free_area_t
```

其中 list_head 域是一个通用的双向链表结构，链表中元素的类型将为 mem_map_t（即 struct page 结构）。Map 域指向一个位图，其大小取决于现有的页面数。free_area 第 $k$ 项位图的每一位，描述的就是大小为 $2^k$ 个页面的两个伙伴块的状态。如果位图的某位为 0，表示一对兄弟块中或者两个都空闲，或者两个都被分配，如果为 1，肯定有一块已被分配。当兄弟块都空闲时，内核把它们当作一个大小为 $2^{k+1}$ 的单独快来处理。如图 6.9 给出该数据结构的示意图。

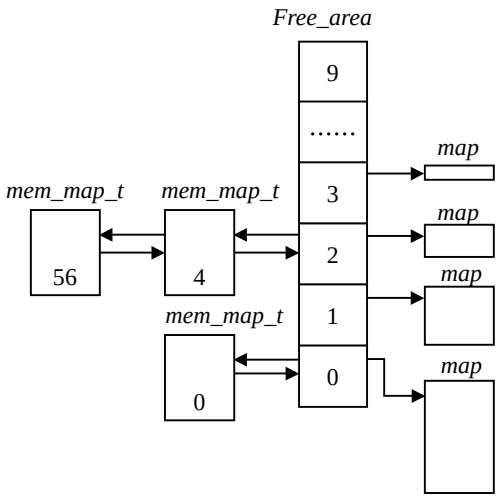


图 6.9　伙伴系统使用的数据结构

图 6.9 中，free_aea 数组的元素 0 包含了一个空闲页（页面编号为 0）；而元素 2 则包含了两个以 4 个页面为大小的空闲页面块，第一个页面块的起始编号为 4，而第二个页面块的起始编号为 56。

我们曾提到，当需要分配若干个内存页面时，用于 DMA 的内存页面必须是连续的。其实为了便于管理，从伙伴算法可以看出，只要请求分配的块大小不超过 512 个页面（2KB），内核就尽量分配连续的页面。

### 6.3.2　物理页面的分配和释放

当一个进程请求分配连续的物理页面时，可以通过调用 alloc_pages()来完成。Linux 2.4 版本中有两个 alloc_pages()，一个在 mm/numa.c 中，另一个在 mm/page_alloc,c 中，编译时根据所定义的条件选项 CONFIG_DISCONTIGMEM 来进行取舍。

1．非一致存储结构（NUMA）中页面的分配

CONFIG_DISCONTIGMEM 条件编译的含义是“不连续的存储空间”，Linux 把不连续的存储空间也归类为非一致存储结构（NUMA)。这是因为，不连续的存储空间本质上是一种广义的 NUMA，因为那说明在最低物理地址和最高物理地址之间存在着空洞，而有空洞的空间当然是“不一致”的。所以，在地址不连续的物理空间也要像结构不一样的物理空间那样划分出若干连续且均匀的“节点”。因此，在存储结构不连续的系统中，每个模块都有若干个节点，因而都有个 pg_data_t 数据结构队列。我们先来看 mm/numa.c 中的 alloc_page()函数：

```
/*
  * This can be refined. Currently, tries to do round robin, instead
  * should do concentratic circle search, starting from current node.
  */
struct page * _alloc_pages (unsigned int gfp_mask, unsigned int order)
{
         struct page *ret = 0;
         pg_data_t *start, *temp;
#ifndef CONFIG_NUMA
         unsigned long flags;
         static pg_data_t *next = 0;
#endif

         if  (order >= MAX_ORDER)
                 return NULL;
#ifdef CONFIG_NUMA
         temp = NODE_DATA (numa_node_id () ) ;
#else
         spin_lock_irqsave (&node_lock, flags) ;
         if  (!next)   next = pgdat_list;
         temp = next;
         next = next->node_next;
         spin_unlock_irqrestore (&node_lock, flags) ;
#endif
         start = temp;
         while  (temp)  {
                 if  ( (ret = alloc_pages_pgdat (temp, gfp_mask, order) ) )
                         return (ret) ;
                 temp = temp->node_next;
         }
         temp = pgdat_list;
         while  (temp != start)  {
                 if  ( (ret = alloc_pages_pgdat (temp, gfp_mask, order) ) )
                         return (ret) ;
                 temp = temp->node_next;
         }
         return (0) ;
  }
```

对该函数的说明如下。

该函数有两个参数。gfp_mask 表示采用哪种分配策略。参数 order 表示所需物理块的大小，可以是 1、2、3 直到 $2^{MAX\_ORDER-1}$。

如果定义了 CONFIG_NUMA，也就是在 NUMA 结构的系统中，可以通过 NUMA_DATA（）宏找到 CPU 所在节点的 pg_data_t 数据结构队列，并存放在临时变量 temp 中。

如果在不连续的 UMA 结构中,则有个 pg_data_t 数据结构的队列 pgdat_list,pgdat_list 就是该队列的首部。因为队列一般都是临界资源，因此，在对该队列进行两个以上的操作时要加锁。

分配时轮流从各个节点开始，以求各节点负荷的平衡。函数中有两个循环，其形式基本相同，也就是，对节点队列基本进行两遍扫描，直至在某个节点内分配成功，则跳出循环，否则，则彻底失败，从而返回 0。对于每个节点，调用 alloc_pages_pgdat（）函数试图分配所需的页面。

2．一致存储结构(UMA)中页面的分配

连续空间 UMA 结构的 alloc_page()是在 include/linux/mm.h 中定义的：

```
   #ifndef CONFIG_DISCONTIGMEM
     static inline struct page * alloc_pages (unsigned int gfp_mask, unsigned int order)
{
          /*
           * Gets optimized away by the compiler.
           */
          if  (order >= MAX_ORDER)
                  return NULL;
          return __alloc_pages (gfp_mask, order,
                  contig_page_data.node_zonelists+ (gfp_mask & GFP_ZONEMASK) ) ;
     }
     #endif
```

从这个函数的定义可以看出，alloc_page()是_alloc_pages（）的封装函数，而_alloc_pages（）才是伙伴算法的核心。这个函数定义于 mm/page_alloc.c 中，我们先对此函数给予概要描述。

_alloc_pages（）在管理区链表 zonelist 中依次查找每个区，从中找到满足要求的区，然后用伙伴算法从这个区中分配给定大小（$2^{order}$ 个）的页面块。如果所有的区都没有足够的空闲页面，则调用 swapper 或 bdflush 内核线程，把脏页写到磁盘以释放一些页面。

在__alloc_pages()和虚拟内存（简称 VM）的代码之间有一些复杂的接口（后面会详细描述）。每个区都要对刚刚被映射到某个进程 VM 的页面进行跟踪，被映射的页面也许仅仅做了标记，而并没有真正地分配出去。因为根据虚拟存储的分配原理，对物理页面的分配要尽量推迟到不能再推迟为止，也就是说，当进程的代码或数据必须装入到内存时，才给它真正分配物理页面。

搞清楚页面分配的基本原则后，我们对其代码具体分析如下：

```
/*
     * This is the 'heart' of the zoned buddy allocator:
     */
struct page * __alloc_pages (unsigned int gfp_mask, unsigned int order, zonelist_t *zonelist)
{
          unsigned long min;
          zone_t **zone, * classzone;
          struct page * page;
```

```
int freed;

  zone = zonelist->zones;
 classzone = *zone;
  min = 1UL << order;
 for  (;;)  {
          zone_t *z = * (zone++) ;
          if  (!z)
                  break;

        min += z->pages_low;
        if  (z->free_pages > min)  {
                page = rmqueue (z, order) ;
                if  (page)
                        return page;
        }
}
```

这是对一个分配策略中所规定的所有页面管理区的循环。循环中依次考察各个区中空闲页面的总量，如果总量尚大于“最低水位线”与所请求页面数之和，就调用 rmqueue（）试图从该区中进行分配。如果分配成功，则返回一个 page 结构指针，指向页面块中第一个页面的起始地址。

```
classzone->need_balance = 1;
mb () ;
if  (waitqueue_active (&kswapd_wait) )
        wake_up_interruptible (&kswapd_wait) ;
```

如果发现管理区中的空闲页面总量已经降到最低点，则把 zone_t 结构中需要重新平衡的标志（need_balance）置 1，而且如果内核线程 kswapd 在一个等待队列中睡眠，就唤醒它，让它收回一些页面以备使用（可以看出，need_balance 是和 kswapd 配合使用的）。

```
zone = zonelist->zones;
min = 1UL << order;
for  (;;)  {
        unsigned long local_min;
        zone_t *z = * (zone++) ;
        if  (!z)
                break;

        local_min = z->pages_min;
        if  (! (gfp_mask & __GFP_WAIT) )
                local_min >>= 2;
        min += local_min;
        if  (z->free_pages > min)  {
                page = rmqueue (z, order) ;
                if  (page)
                        return page;
        }
}
```

如果给定分配策略中所有的页面管理区都分配失败，那只好把原来的“最低水位”再向下调(除以 4),然后看是否满足要求(z->free_pages > min),如果能满足要求,则调用 rmqueue

() 进行分配。

```
        /* here we're in the low on memory slow path */

rebalance:
        if (current->flags & (PF_MEMALLOC | PF_MEMDIE)) {
            zone = zonelist->zones;
            for (;;) {
                zone_t *z = *(zone++);
                if (!z)
                    break;

                page = rmqueue(z, order);
                if (page)
                    return page;
            }
            return NULL;
        }
```

如果分配还不成功，这时候就要看是哪类进程在请求分配内存页面。其中 PF_MEMALLOC 和 PF_MEMDIE 是进程的 task_struct 结构中 flags 域的值，对于正在分配页面的进程（如 kswapd 内核线程），则其 PF_MEMALLOC 的值为 1（一般进程的这个标志为 0），而对于使内存溢出而被杀死的进程，则其 PF_MEMDIE 为 1。不管哪种情况，都说明必须给该进程分配页面（想想为什么）。因此，继续进行分配。

```
        /* Atomic allocations - we can't balance anything */
        if (!(gfp_mask & __GFP_WAIT))
            return NULL;
```

如果请求分配页面的进程不能等待，也不能被重新调度，只好在没有分配到页面的情况下“空手”返回。

```
        page = balance_classzone(classzone, gfp_mask, order, &freed);
        if (page)
            return page;
```

如果经过几番努力，必须得到页面的进程（如 kswapd）还没有分配到页面，就要调用 balance_classzone() 函数把当前进程所占有的局部页面释放出来。如果释放成功，则返回一个 page 结构指针，指向页面块中第一个页面的起始地址。

```
        zone = zonelist->zones;
        min = 1UL << order;
        for (;;) {
            zone_t *z = *(zone++);
            if (!z)
                break;

            min += z->pages_min;
            if (z->free_pages > min) {
                page = rmqueue(z, order);
                if (page)
                    return page;
            }
        }
```

继续进行分配。

```
            /* Don't let big-order allocations loop *
            if  (order > 3)
                  return NULL;

           /* Yield for kswapd, and try again */
            current->policy |= SCHED_YIELD;
            __set_current_state (TASK_RUNNING) ;
           schedule () ;
            goto rebalance;
          }
```

在这个函数中，频繁调用了 rmqueue () 函数，下面我们具体来看一下这个函数内容。

(1) rmqueue () 函数

该函数试图从一个页面管理区分配若干连续的内存页面。这是最基本的分配操作，其具体代码如下：

```
static struct page * rmqueue (zone_t *zone, unsigned int order)
{
         free_area_t * area = zone->free_area + order;
         unsigned int curr_order = order;
         struct list_head *head, *curr;
         unsigned long flags;
         struct page *page;

        spin_lock_irqsave (&zone->lock, flags) ;
         do {
                 head = &area->free_list;
                 curr = memlist_next (head) ;

                 if  (curr != head)  {
                         unsigned int index;

                         page = memlist_entry (curr, struct page, list) ;
                         if  (BAD_RANGE (zone,page) )
                                 BUG () ;
                         memlist_del (curr) ;
                         index = page - zone->zone_mem_map;
                         if  (curr_order != MAX_ORDER-1)
                                 MARK_USED (index, curr_order, area) ;
                         zone->free_pages -= 1UL << order;

                         page = expand (zone, page, index, order, curr_order, area) ;
                         spin_unlock_irqrestore (&zone->lock, flags) ;

                         set_page_count (page, 1) ;
                         if  (BAD_RANGE (zone,page) )
                                 BUG () ;
                         if  (PageLRU (page) )
                                 BUG () ;
                         if  (PageActive (page) )
                                 BUG () ;
                         return page;
```

```
                }
                curr_order++;
                area++;
        } while (curr_order < MAX_ORDER) ;
        spin_unlock_irqrestore (&zone->lock, flags) ;

        return NULL;
 }
```

对该函数的解释如下。

参数 zone 指向要分配页面的管理区，order 表示要求分配的页面数为 $2^{order}$。

do 循环从 free_area 数组的第 order 个元素开始，扫描每个元素中由 page 结构组成的双向循环空闲队列。如果找到合适的页块，就把它从队列中删除，删除的过程是不允许其他进程、其他处理器来打扰的。所以要用 spin_lock_irqsave () 将这个循环加上锁。

首先在恰好满足大小要求的队列里进行分配。其中 memlist_entry(curr, struct page, list)获得空闲块的第 1 个页面的地址，如果这个地址是个无效的地址，就陷入 BUG()。如果有效，memlist_del(curr)从队列中摘除分配出去的页面块。如果某个页面块被分配出去，就要在 frea_area 的位图中进行标记，这是通过调用 MARK_USED () 宏来完成的。

如果分配出去后还有剩余块，就通过 expand () 获得所分配的页块，而把剩余块链入适当的空闲队列中。

如果当前空闲队列没有空闲块，就从更大的空闲块队列中找。

(2) expand () 函数

该函数源代码如下。

```
    static inline struct page * expand (zone_t *zone, struct page *page,
         unsigned long index, int low, int high, free_area_t * area)
 {
        unsigned long size = 1 << high;

        while (high > low) {
                if (BAD_RANGE (zone,page) )
                        BUG () ;
                area--;
                high--;
                size >>= 1;
                memlist_add_head (& (page) ->list, & (area) ->free_list) ;
                MARK_USED (index, high, area) ;
                index += size;
                page += size;
        }
        if (BAD_RANGE (zone,page) )
                BUG () ;
        return page;
 }
```

对该函数解释如下。

参数 zone 指向已分配页块所在的管理区；page 指向已分配的页块；index 是已分配的页面在 mem_map 中的下标； low 表示所需页面块大小为 $2^{low}$，而 high 表示从空闲队列中实际进行分配的页面块大小为 $2^{high}$；area 是 free_area_struct 结构，指向实际要分配的页块。

通过上面介绍可以知道，返回给请求者的块大小为 $2^{low}$ 个页面，并把剩余的页面放入合适的空闲队列，且对伙伴系统的位图进行相应的修改。例如，假定我们需要一个 2 页面的块，但是，我们不得不从 order 为 3（8 个页面）的空闲队列中进行分配，又假定我们碰巧选择物理页面 800 作为该页面块的底部。在我们这个例子中，这几个参数值为：

```
page == mem_map+800
index == 800
   low == 1
   high == 3
      area == zone->free_area+high （ 也就是 frea_area 数组中下标为 3 的元素）
```

首先把 size 初始化为分配块的页面数(例如，size = 1<<3 == 8)

while 循环进行循环查找。每次循环都把 size 减半。如果我们从空闲队列中分配的一个块与所要求的大小匹配，那么 low = high，就彻底从循环中跳出，返回所分配的页块。

如果分配到的物理块所在的空闲块大于所需块的大小（即 $2^{high}>2^{low}$），那就将该空闲块分为两半（即 area--;high--; size >>= 1），然后调用 memlist_add_head（）把刚分配出去的页面块又加入到低一档（物理块减半）的空闲队列中，准备从剩下的一半空闲块中重新进行分配，并调用 MARK_USED（）设置位图。

在上面的例子中，第 1 次循环，我们从页面 800 开始，把页面大小为 4（即 $2^{high--}$）的块其首地址插入到 frea_area[2]中的空闲队列；因为 low<high，又开始第 2 次循环，这次从页面 804 开始，把页面大小为 2 的块插入到 frea_area[1]中的空闲队列，此时，page＝806，high=low＝1，退出循环，我们给调用者返回从 806 页面开始的一个 2 页面块。

从这个例子可以看出，这是一种巧妙的分配算法。

3．释放页面

从上面的介绍可以看出，页面块的分配必然导致内存的碎片化，而页面块的释放则可以将页面块重新组合成大的页面块。页面的释放函数为__free_pages(page struct *page, unsigned long order），该函数从给定的页面开始，释放的页面块大小为 $2^{order}$。原函数为:

```
void __free_pages (page struct *page, unsigned long order)
{
            if (!PageReserved (page) && put_page_testzero (page) )
            __free_pages_ok (page, order) ;
}
```

其中比较巧妙的部分就是调用 put_page_testzero()宏，该函数把页面的引用计数减 1，如果减 1 后引用计数为 0，则该函数返回 1。因此，如果调用者不是该页面的最后一个用户，那么，这个页面实际上就不会被释放。另外要说明的是不可释放保留页 PageReserved，这是通过 PageReserved（）宏进行检查的。

如果调用者是该页面的最后一个用户，则__free_pages() 再调用 __free_pages_ok()。__free_pages_ok（）才是对页面块进行释放的实际函数，该函数把释放的页面块链入空闲链表，并对伙伴系统的位图进行管理，必要时合并伙伴块。这实际上是 expand()函数的反操作，我们对此不再进行详细的讨论。

### 6.3.3 Slab 分配机制

采用伙伴算法分配内存时，每次至少分配一个页面。但当请求分配的内存大小为几十个字节或几百个字节时应该如何处理？如何在一个页面中分配小的内存区，小内存区的分配所产生的内碎片又如何解决？

Linux 2.0 采用的解决办法是建立了 13 个空闲区链表，它们的大小从 32 字节到 132056 字节。从 Linux 2.2 开始，MM 的开发者采用了一种叫做 Slab 的分配模式，该模式早在 1994 年就被开发出来，用于 Sun Microsystem Solaris 2.4 操作系统中。Slab 的提出主要是基于以下考虑。

内核对内存区的分配取决于所存放数据的类型。例如，当给用户态进程分配页面时，内核调用 get_free_page()函数，并用 0 填充这个页面。而给内核的数据结构分配页面时，事情没有这么简单，例如，要对数据结构所在的内存进行初始化、在不用时要收回它们所占用的内存。因此，Slab 中引入了对象这个概念，所谓对象就是存放一组数据结构的内存区，其方法就是构造或析构函数，构造函数用于初始化数据结构所在的内存区，而析构函数收回相应的内存区。但为了便于理解，你也可以把对象直接看作内核的数据结构。为了避免重复初始化对象，Slab 分配模式并不丢弃已分配的对象，而是释放但把它们依然保留在内存中。当以后又要请求分配同一对象时，就可以从内存获取而不用进行初始化，这是在 Solaris 中引入 Slab 的基本思想。

实际上，Linux 中对 Slab 分配模式有所改进，它对内存区的处理并不需要进行初始化或回收。出于效率的考虑，Linux 并不调用对象的构造或析构函数，而是把指向这两个函数的指针都置为空。Linux 中引入 Slab 的主要目的是为了减少对伙伴算法的调用次数。

实际上，内核经常反复使用某一内存区。例如，只要内核创建一个新的进程，就要为该进程相关的数据结构（task_struct、打开文件对象等）分配内存区。当进程结束时，收回这些内存区。因为进程的创建和撤销非常频繁，因此，Linux 的早期版本把大量的时间花费在反复分配或回收这些内存区上。从 Linux 2.2 开始，把那些频繁使用的页面保存在高速缓存中并重新使用。

可以根据对内存区的使用频率来对它分类。对于预期频繁使用的内存区，可以创建一组特定大小的专用缓冲区进行处理，以避免内碎片的产生。对于较少使用的内存区，可以创建一组通用缓冲区（如 Linux 2.0 中所使用的 2 的幂次方）来处理，即使这种处理模式产生碎片，也对整个系统的性能影响不大。

硬件高速缓存的使用，又为尽量减少对伙伴算法的调用提供了另一个理由，因为对伙伴算法的每次调用都会“弄脏”硬件高速缓存，因此，这就增加了对内存的平均访问次数。

Slab 分配模式把对象分组放进缓冲区（尽管英文中使用了 Cache 这个词，但实际上指的是内存中的区域，而不是指硬件高速缓存）。因为缓冲区的组织和管理与硬件高速缓存的命中率密切相关，因此，Slab 缓冲区并非由各个对象直接构成，而是由一连串的“大块（Slab）”构成，而每个大块中则包含了若干个同种类型的对象，这些对象或已被分配，或空闲，如图 6.10 所示。一般而言，对象分两种，一种是大对象，一种是小对象。所谓小对象，是指在一

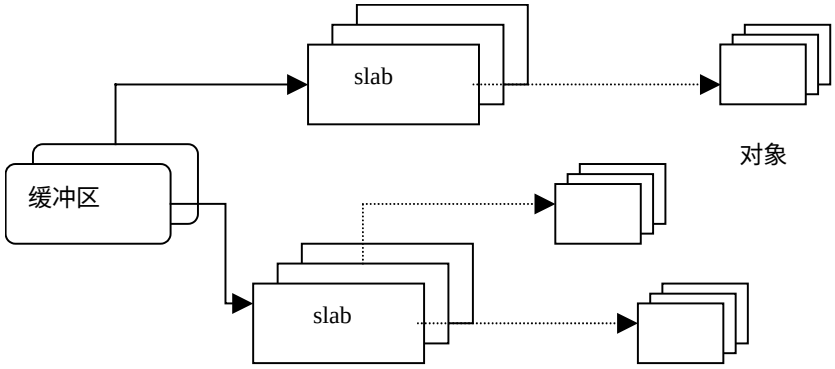


图 6.10　Slab 的组成

个页面中可以容纳下好几个对象的那种。例如，一个 inode 结构大约占 300 多个字节，因此，一个页面中可以容纳 8 个以上的 inode 结构，因此，inode 结构就为小对象。Linux 内核中把小于 512 字节的对象叫做小对象。

实际上，缓冲区就是主存中的一片区域，把这片区域划分为多个块，每块就是一个 Slab，每个 Slab 由一个或多个页面组成，每个 Slab 中存放的就是对象。

因为 Slab 分配模式的实现比较复杂，我们不准备对其进行详细的分析，只对主要内容给予描述。

1．Slab 的数据结构

Slab 分配模式有两个主要的数据结构，一个是描述缓冲区的结构 kmem_cache_t，一个是描述 Slab 的结构 kmem_slab_t，下面对这两个结构给予简要讨论。

(1) Slab

Slab 是 Slab 管理模式中最基本的结构。它由一组连续的物理页面组成，对象就被顺序放在这些页面中。其数据结构在 mm/slab.c 中定义如下：

```
/*
* slab_t
*
* Manages the objs in a slab. Placed either at the beginning of mem allocated
* for a slab, or allocated from an general cache.
* Slabs are chained into three list: fully used, partial, fully free slabs.
*/
 typedef struct slab_s {
       struct list_head        list;
       unsigned long           colouroff;
       void                   *s_mem;         /* including colour offset */
       unsigned int            inuse;          /* num of objs active in slab */
       kmem_bufctl_t           free;
} slab_t;
```

这里的链表用来将前一个 Slab 和后一个 Slab 链接起来形成一个双向链表，colouroff 为该 Slab 上着色区的大小，指针 s_mem 指向对象区的起点，inuse 是 Slab 中所分配对象的

个数。最后，free 的值指明了空闲对象链中的第一个对象，kmem_bufctl_t 其实是一个整数。Slab 结构的示意图如图 6.11 所示。

对于小对象，就把 Slab 的描述结构 slab_t 放在该 Slab 中；对于大对象，则把 Slab 结构游离出来，集中存放。关于 Slab 中的着色区再给予具体描述。

每个 Slab 的首部都有一个小小的区域是不用的，称为“着色区（Coloring Area）”。着色区的大小使 Slab 中的每个对象的起始地址都按高速缓存中的“缓存行（Cache Line）”大小进行对齐（80386 的一级高速缓存行大小为 16 字节，Pentium 为 32 字节）。因为 Slab 是由 1 个页面或多个页面（最多为 32）组成，因此，每个 Slab 都是从一个页面边界开始的，它自然按高速缓存的缓冲行对齐。但是，Slab 中的对象大小不确定，设置着色区的目的就是将 Slab 中第一个对象的起始地址往后推到与缓冲行对齐的位置。因为一个缓冲区中有多个 Slab，因此，应该把每个缓冲区中的各个 Slab 着色区的大小尽量安排成不同的大小，这样可以使得在不同的 Slab 中，处于同一相对位置的对象，让它们在高速缓存中的起始地址相互错开，这样就可以改善高速缓存的存取效率。

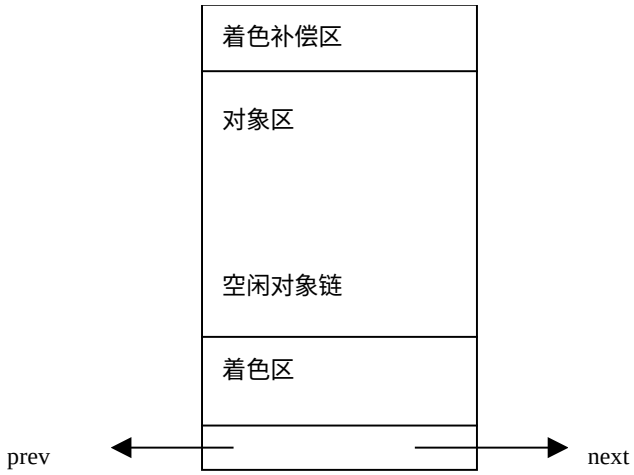


图 6.11　Slab 结构示意图

每个 Slab 上最后一个对象以后也有个小小的废料区是不用的，这是对着色区大小的补偿，其大小取决于着色区的大小，以及 Slab 与其每个对象的相对大小。但该区域与着色区的总和对于同一种对象的各个 Slab 是个常数。

每个对象的大小基本上是所需数据结构的大小。只有当数据结构的大小不与高速缓存中的缓冲行对齐时，才增加若干字节使其对齐。所以，一个 Slab 上的所有对象的起始地址都必然是按高速缓存中的缓冲行对齐的。

（2）缓冲区

每个缓冲区管理着一个 Slab 链表，Slab 按序分为 3 组。第 1 组是全满的 Slab（没有空闲的对象），第 2 组 Slab 中只有部分对象被分配，部分对象还空闲，最后一组 Slab 中的对象全部空闲。只所以这样分组，是为了对 Slab 进行有效的管理。每个缓冲区还有一个轮转锁（Spinlock），在对链表进行修改时用这个轮转锁进行同步。类型 kmem_cache_s 在 mm/slab.c

中定义如下：

```
 struct kmem_cache_s {
/* 1)  each alloc & free */
        /* full, partial first, then free */
       struct list_head        slabs_full;
        struct list_head        slabs_partial;
        struct list_head        slabs_free;
        unsigned int            objsize;
        unsigned int            flags;  /* constant flags */
        unsigned int            num;    /* # of objs per slab */
        spinlock_t              spinlock;
#ifdef CONFIG_SMP
        unsigned int            batchcount;
#endif

/* 2)  slab additions /removals */
        /* order of pgs per slab  (2^n)  */
        unsigned int            gfporder;

        /* force GFP flags, e.g. GFP_DMA */
        unsigned int            gfpflags;

        size_t                  colour;         /* cache colouring range */
        unsigned int            colour_off;     /* colour offset */
        unsigned int            colour_next;    /* cache colouring */
        kmem_cache_t            *slabp_cache;
        unsigned int            growing;
        unsigned int            dflags;         /* dynamic flags */

        /* constructor func */
        void  (*ctor)  (void *, kmem_cache_t *, unsigned long) ;

        /* de-constructor func */
        void  (*dtor)  (void *, kmem_cache_t *, unsigned long) ;

        unsigned long           failures;

/* 3)  cache creation/removal */
        char                    name[CACHE_NAMELEN];
        struct list_head        next;
#ifdef CONFIG_SMP
/* 4)  per-cpu data */
        cpucache_t              *cpudata[NR_CPUS];
#endif
…..
};
```

然后定义了 kmem_cache_t，并给部分域赋予了初值：

```
static kmem_cache_t cache_cache = {
        slabs_full:     LIST_HEAD_INIT (cache_cache.slabs_full) ,
        slabs_partial:  LIST_HEAD_INIT (cache_cache.slabs_partial) ,
        slabs_free:     LIST_HEAD_INIT (cache_cache.slabs_free) ,
```

```
        objsize:        sizeof（kmem_cache_t），
        flags:          SLAB_NO_REAP,
        spinlock:       SPIN_LOCK_UNLOCKED,
        colour_off:     L1_CACHE_BYTES,
        name:           "kmem_cache",
};
```

对该结构说明如下。

该结构中有 3 个队列 slabs_full、slabs_partial 以及 slabs_free，分别指向满 Slab、半满 Slab 和空闲 Slab，另一个队列 next 则把所有的专用缓冲区链成一个链表。

除了这些队列和指针外，该结构中还有一些重要的域：objsize 是原始的数据结构的大小，这里初始化为 kmem_cache_t 的大小；num 表示每个 Slab 上有几个缓冲区；gfporder 则表示每个 Slab 大小的对数，即每个 Slab 由 $2^{gfporder}$ 个页面构成。

如前所述，着色区的使用是为了使同一缓冲区中不同 Slab 上的对象区的起始地址相互错开，这样有利于改善高速缓存的效率。colour_off 表示颜色的偏移量，colour 表示颜色的数量；一个缓冲区中颜色的数量取决于 Slab 中对象的个数、剩余空间以及高速缓存行的大小。所以，对每个缓冲区都要计算它的颜色数量，这个数量就保存在 colour 中，而下一个 Slab 将要使用的颜色则保存在 colour_next 中。当 colour_next 达到最大值时，就又从 0 开始。着色区的大小可以根据（colour_off×colour）算得。例如，如果 colour 为 5，colour_off 为 8，则第一个 Slab 的颜色将为 0，Slab 中第一个对象区的起始地址（相对）为 0，下一个 Slab 中第一个对象区的起始地址为 8，再下一个为 16，24，32，0……等。

cache_cache 变量实际上就是缓冲区结构的头指针。

由此可以看出，缓冲区结构 kmem_cache_t 相当于 Slab 的总控结构，缓冲区结构与 Slab 结构之间的关系如图 6.12 所示。

在图 6.12 中，深灰色表示全满的 Slab，浅灰色表示含有空闲对象的 Slab，而无色表示空的 Slab。缓冲区结构之间形成一个单向链表，Slab 结构之间形成一个双向链表。另外，缓冲区结构还有分别指向满、半满、空闲 Slab 结构的指针。

2．专用缓冲区的建立和撤销

专用缓冲区是通过 kmem_cache_create（）函数建立的，函数原型为：

```
kmem_cache_t *kmem_cache_create (const char *name, size_t size, size_t offset,
        unsigned long c_flags,
        void  (*ctor)   (void *objp, kmem_cache_t *cachep, unsigned long flags),
        void  (*dtor)   (void *objp, kmem_cache_t *cachep, unsigned long flags))
```

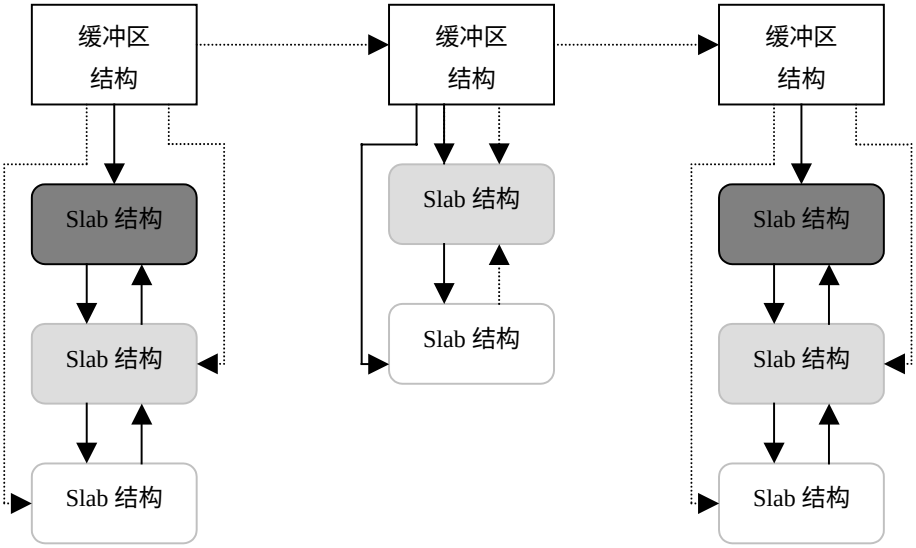


图 6.12　缓冲区结构 kmem_cache_t 与 Slab 结构 slab_t 之间的关系

对其参数说明如下。

(1) name： 缓冲区名 ( 19 个字符)。

(2) size： 对象大小。

(3) offset ：所请求的着色偏移量。

(4) c_flags ：对缓冲区的设置标志。

- SLAB_HWCACHE_ALIGN：表示与第一个高速缓存中的缓冲行边界 (16 或 32 字节) 对齐。
- SLAB_NO_REAP：不允许系统回收内存。
- SLAB_CACHE_DMA：表示 Slab 使用的是 DMA 内存。

(5) ctor ：构造函数 (一般都为 NULL)。

(6) dtor ：析构函数 (一般都为 NULL)。

(7) objp ：指向对象的指针。

(8) cachep ：指向缓冲区。

对专用缓冲区的创建过程简述如下。

kmem_cache_create () 函数要进行一系列的计算，以确定最佳的 Slab 构成。包括：每个 Slab 由几个页面组成，划分为多少个对象；Slab 的描述结构 slab_t 应该放在 Slab 的外面还是放在 Slab 的尾部；还有“ 颜色 ”的数量等等。并根据调用参数和计算结果设置 kmem_cache_t 结构中的各个域，包括两个函数指针 ctor 和 dtor。最后，将 kmem_cache_t 结构插入到 cache_cache 的 next 队列中。

但请注意，函数 kmem_cache_create () 所创建的缓冲区中还没有包含任何 Slab，因此，也没有空闲的对象。只有以下两个条件都为真时，才给缓冲区分配 Slab：

(1) 已发出一个分配新对象的请求；

(2) 缓冲区不包含任何空闲对象。

当这两个条件都成立时，Slab 分配模式就调用 kmem_cache_grow () 函数给缓冲区分配一个新的 Slab。其中，该函数调用 kmem_gatepages()从伙伴系统获得一组页面；然后又调用

kmem_cache_slabgmt()获得一个新的Slab结构 还要调用kmem_cache_init_objs()为新Slab中的所有对象申请构造方法（如果定义的话）;最后，调用 kmem_slab_link_end()把这个 Slab 结构插入到缓冲区中 Slab 链表的末尾。

Slab 分配模式的最大好处就是给频繁使用的数据结构建立专用缓冲区。但到目前的版本为止，Linux 内核中多数专用缓冲区的建立都用 NULL 作为构造函数的指针，例如，为虚存区间结构 vm_area_struct 建立的专用缓冲区 vm_area_cachep：

```
vm_area_cachep = kmem_cache_create ("vm_area_struct",
                        sizeof (struct vm_area_struct) , 0,
                        SLAB_HWCACHE_ALIGN, NULL, NULL) ;
```

就把构造和析构函数的指针置为 NULL，也就是说，内核并没有充分利用 Slab 管理机制所提供的好处。为了说明如何利用专用缓冲区，我们从内核代码中选取一个构造函数不为空的简单例子，这个例子与网络子系统有关，在 net/core/buff.c 中定义：

```
void __init  skb_init (void)
{
   int  i;
          skbuff_head_cache = kmem_cache_create ("skbuff_head_cache",
                                                 sizeof (struct sk_buff) ,
                                                  0,
                                                  SLAB_HWCACHE_ALIGN,
                                                  skb_headerinit, NULL) ;
        if  (!skbuff_head_cache)
                panic ("cannot create skbuff cache") ;

        for  (i=0; i<NR_CPUS; i++)
                skb_queue_head_init (&skb_head_pool[i].list) ;
     }
```

从代码中可以看出，skb_init () 调用 kmem_cache_create () 为网络子系统建立一个 sk_buff 数据结构的专用缓冲区，其名称为“ skbuff_head_cache ”( 你可以通过读取／proc/slabinfo/文件得到所有缓冲区的名字)。调用参数 offset 为 0,表示第一个对象在 Slab 中的位移并无特殊要求。但是参数 flags 为 SLAB_HWCACHE_ALIGN，表示 Slab 中的对象要与高速缓存中的缓冲行边界对齐。对象的构造函数为 skb_headerinit ()，而析构函数为空，也就是说，在释放一个 Slab 时无需对各个缓冲区进行特殊的处理。

当从内核卸载一个模块时，同时应当撤销为这个模块中的数据结构所建立的缓冲区，这是通过调用 kmem_cache_destroy()函数来完成的。从 Linux 2.4.16 内核代码中进行查找可知，对这个函数的调用非常少。

3．通用缓冲区

在内核中初始化开销不大的数据结构可以合用一个通用的缓冲区。通用缓冲区非常类似于物理页面分配中的大小分区，最小的为 32，然后依次为 64、128、……直至 128KB（即 32 个页面），但是，对通用缓冲区的管理又采用的是 Slab 方式。从通用缓冲区中分配和释放缓冲区的函数为：

```
void *kmalloc (size_t size, int flags) ;
Void  kree (const void *objp) ;
```

因此，当一个数据结构的使用根本不频繁时，或其大小不足一个页面时，就没有必要给其分配专用缓冲区，而应该调用 kmallo()进行分配。如果数据结构的大小接近一个页面，则干脆通过 alloc_page()为之分配一个页面。

事实上，在内核中，尤其是驱动程序中，有大量的数据结构仅仅是一次性使用，而且所占内存只有几十个字节，因此，一般情况下调用 kmallo()给内核数据结构分配内存就足够了。另外，因为，在 Linux 2.0 以前的版本一般都调用 kmallo()给内核数据结构分配内存，因此，调用该函数的一个优点是（让你开发的驱动程序）能保持向后兼容。

### 6.3.4　内核空间非连续内存区的管理

我们说，任何时候，CPU 访问的都是虚拟内存，那么，在你编写驱动程序，或者编写模块时，Linux 给你分配什么样的内存？它处于 4GB 空间的什么位置？这就是我们要讨论的非连续内存。

首先，非连续内存处于 3GB 到 4GB 之间，也就是处于内核空间，如图 6.13 所示。

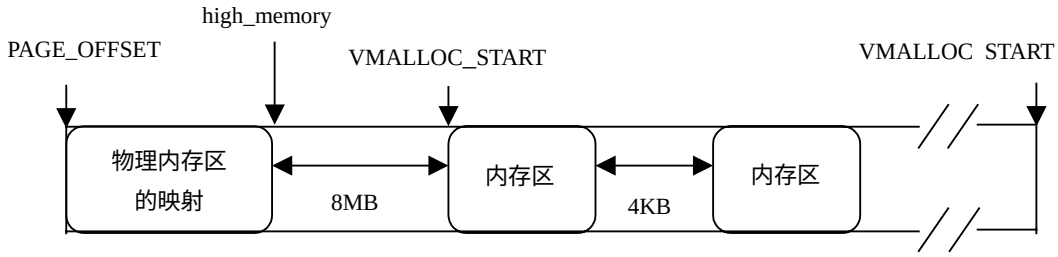


图 6.13　从 PAGE_OFFSET 开始的内核地址区间

图 6.13 中，PAGE_OFFSET 为 3GB，high_memory 为保存物理地址最高值的变量，VMALLOC_START 为非连续区的的起始地址，定义于 include／i386/pgtable.h 中：

```
    #define VMALLOC_OFFSET   (8*1024*1024)
    #define VMALLOC_START    ( ( (unsigned long)  high_memory + 2*VMALLOC_OFFSET-1)  & \~
(VMALLOC_OFFSET-1) )
```

在物理地址的末尾与第一个内存区之间插入了一个 8MB（VMALLOC_OFFSET）的区间，这是一个安全区，目的是为了“捕获”对非连续区的非法访问。出于同样的理由，在其他非连续的内存区之间也插入了 4KB 大小的安全区。每个非连续内存区的大小都是 4096 的倍数。

1．非连续区的数据结构

描述非连续区的数据结构为 struct vm_struct，定义于 include/linux/vmalloc.h 中：

```
struct vm_struct {
        unsigned long flags;
        void * addr;
        unsigned long size;
        struct vm_struct * next;
};
  struct vm_struct * vmlist;
```

非连续区组成一个单链表，链表第一个元素的地址存放在变量 vmlist 中。Addr 域是内存区的起始地址；size 是内存区的大小加 4096(安全区的大小)。

2．创建一个非连续区的结构

函数 get_vm_area（）创建一个新的非连续区结构，其代码在 mm/vmalloc.c 中：

```
   struct vm_struct * get_vm_area (unsigned long size, unsigned long flags)
  {
        unsigned long addr;
        struct vm_struct **p, *tmp, *area;

         area =  (struct vm_struct *)  kmalloc (sizeof (*area) , GFP_KERNEL) ;
         if  (!area)
                 return NULL;
         size += PAGE_SIZE;
         addr = VMALLOC_START;
         write_lock (&vmlist_lock) ;
        for  (p = &vmlist;  (tmp = *p)  ; p = &tmp->next)  {
                 if  ( (size + addr)  < addr)
                        goto out;
                 if  (size + addr <=  (unsigned long)  tmp->addr)
                         break;
                addr = tmp->size +  (unsigned long)  tmp->addr;
                 if  (addr > VMALLOC_END-size)
                        goto out;
         }
         area->flags = flags;
         area->addr =  (void *) addr;
         area->size = size;
         area->next = *p;
         *p = area;
         write_unlock (&vmlist_lock) ;
         return area;

out:
        write_unlock (&vmlist_lock) ;
         kfree (area) ;
         return NULL;
 }
```

这个函数比较简单，就是在单链表中插入一个元素。其中调用了 kmalloc()和 kfree()函数,分别用来为 vm_struct 结构分配内存和释放所分配的内存。

3．分配非连续内存区

vmalloc（）函数给内核分配一个非连续的内存区，在/include/linux/vmalloc.h 中定义如下：

```
    static inline void * vmalloc  (unsigned long size)
  {
         return __vmalloc (size, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL) ;
  }
```

vmalloc（）最终调用的是__vmalloc（）函数，该函数的代码在 mm/vmalloc.c 中：

```
 void * __vmalloc  (unsigned long size, int gfp_mask, pgprot_t prot)
{
       void * addr;
       struct vm_struct *area;

        size = PAGE_ALIGN (size) ;
        if  (!size ||  (size >> PAGE_SHIFT)  > num_physpages)  {
               BUG () ;
              return NULL;
        }
        area = get_vm_area (size, VM_ALLOC) ;
        if  (!area)
               return NULL;
        addr = area->addr;
        if  (vmalloc_area_pages (VMALLOC_VMADDR (addr) , size, gfp_mask, prot) )  {
               vfree (addr) ;
               return NULL;
        }
       return addr;
 }
```

函数首先把 size 参数取整为页面大小（4096）的一个倍数，也就是按页的大小进行对齐，然后进行有效性检查，如果有大小合适的可用内存，就调用 get_vm_area（）获得一个内存区的结构。但真正的内存区还没有获得，函数 vmalloc_area_pages（）真正进行非连续内存区的分配：

```
    inline int vmalloc_area_pages  (unsigned long address, unsigned long size,
                               int gfp_mask, pgprot_t prot)
 {
        pgd_t * dir;
        unsigned long end = address + size;
        int ret;

        dir = pgd_offset_k (address) ;
        spin_lock (&init_mm.page_table_lock) ;
do {
                  pmd_t *pmd;

                pmd = pmd_alloc (&init_mm, dir, address) ;
                ret = -ENOMEM;
                if  (!pmd)
                       break;

                ret = -ENOMEM;
                if  (alloc_area_pmd (pmd, address, end - address, gfp_mask, prot) )
                        break;

                address =  (address + PGDIR_SIZE)  & PGDIR_MASK;
                dir++;

                ret = 0;
        } while  (address &&  (address < end) ) ;
```

```
        spin_unlock (&init_mm.page_table_lock) ;
        return ret;
    }
```

该函数有两个主要的参数，address 表示内存区的起始地址，size 表示内存区的大小。内存区的末尾地址赋给了局部变量 end。其中还调用了几个主要的函数或宏。

(1) pgd_offset_k () 宏导出这个内存区起始地址在页目录中的目录项。

(2) pmd_alloc () 为新的内存区创建一个中间页目录。

(3) alloc_area_pmd () 为新的中间页目录分配所有相关的页表，并更新页的总目录；该函数调用 pte_alloc_kernel()函数来分配一个新的页表，之后再调用 alloc_area_pte () 为页表项分配具体的物理页面。

(4) 从 vmalloc_area_pages () 函数可以看出，该函数实际建立起了非连续内存区到物理页面的映射。

4. kmalloc()与 vmalloc()的区别

kmalloc()与 vmalloc() 都是在内核代码中提供给其他子系统用来分配内存的函数，但二者有何区别？

从前面的介绍已经看出，这两个函数所分配的内存都处于内核空间，即从 3GB～4GB；但位置不同，kmalloc()分配的内存处于 3GB～high_memory 之间，而 vmalloc()分配的内存在 VMALLOC_START～4GB 之间，也就是非连续内存区。一般情况下在驱动程序中都是调用 kmalloc() 来给数据结构分配内存，而 vmalloc()用在为活动的交换区分配数据结构，为某些 I/O 驱动程序分配缓冲区，或为模块分配空间，例如在 include/asm-i386/module.h 中定义了如下语句：

```
#define module_map (x)          vmalloc (x)
```

其含义就是把模块映射到非连续的内存区。

与 kmalloc()和 vmalloc()相对应，两个释放内存的函数为 kfree()和 vfree()。

## 6.4 地址映射机制

顾名思义地址映射就是建立几种存储媒介（内存，辅存，虚存）间的关联，完成地址间的相互转换，它既包括磁盘文件到虚拟内存的映射，也包括虚拟内存到物理内存的映射，如图 6.14 所示。本节主要讨论磁盘文件到虚拟内存的映射，虚拟内存到物理内存的映射实际上是请页机制完成的（请看 6.5 节）。

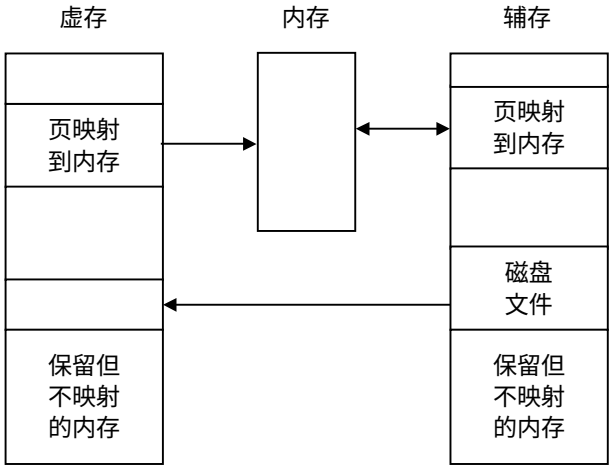


图 6.14　存储介质间的映射关系

### 6.4.1　描述虚拟空间的数据结构

前几节介绍的数据结构如存储节点（Node）、管理区（Zone）、页面（Page）及空闲区（free_area）都用于物理空间的管理。这一节主要关注虚拟空间的管理。虚拟空间的管理是以进程为基础的，每个进程都有各自的虚存空间（或叫用户空间，地址空间），除此之外，每个进程的“内核空间”是为所有的进程所共享的。

一个进程的虚拟地址空间主要由两个数据结构来描述。一个是最高层次的：mm_struct，一个是较高层次的：vm_area_structs。最高层次的 mm_struct 结构描述了一个进程的整个虚拟地址空间。较高层次的结构 vm_area_truct 描述了虚拟地址空间的一个区间(简称虚拟区)。

1．MM_STRUCT 结构

mm_strcut 用来描述一个进程的虚拟地址空间，在/include/linux/sched.h 中描述如下：

```
struct mm_struct {
        struct vm_area_struct * mmap;          /* 指向虚拟区间（VMA）链表 */
        rb_root_t mm_rb;              ／*指向 red_black 树*/
        struct vm_area_struct * mmap_cache;     /* 指向最近找到的虚拟区间*/
        pgd_t * pgd;                  ／*指向进程的页目录*/
             atomic_t mm_users;                  /* 用户空间中的有多少用户*/
             atomic_t mm_count;               /* 对"struct mm_struct"有多少引用*/
        int map_count;                        /* 虚拟区间的个数*/
        struct rw_semaphore mmap_sem;
      spinlock_t page_table_lock;        /* 保护任务页表和 mm->rss */
        struct list_head mmlist;            /*所有活动（active）mm 的链表 */
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
```

```
        unsigned long def_flags;
        unsigned long cpu_vm_mask;
        unsigned long swap_address;

        unsigned dumpable:1;

        /* Architecture-specific MM context */
        mm_context_t context;
};
```

对该结构进一步说明如下。

在内核代码中，指向这个数据结构的变量常常是 mm。

每个进程只有一个 mm_struct 结构，在每个进程的 task_struct 结构中，有一个指向该进程的结构。可以说，mm_struct 结构是对整个用户空间的描述。

一个进程的虚拟空间中可能有多个虚拟区间（参见下面对 vm_area_struct 描述），对这些虚拟区间的组织方式有两种,当虚拟区间较少时采用单链表,由 mmap 指针指向这个链表，当虚拟区间多时采用“红黑树（red_black tree）”结构，由 mm_rb 指向这颗树。在 2.4.10 以前的版本中，采用的是 AVL 树，因为与 AVL 树相比，对红黑树进行操作的效率更高。

因为程序中用到的地址常常具有局部性，因此，最近一次用到的虚拟区间很可能下一次还要用到，因此，把最近用到的虚拟区间结构应当放入高速缓存，这个虚拟区间就由 mmap_cache 指向。

指针 pgd 指向该进程的页目录(每个进程都有自己的页目录,注意同内核页目录的区别)，当调度程序调度一个程序运行时，就将这个地址转成物理地址，并写入控制寄存器（CR3）。

由于进程的虚拟空间及其下属的虚拟区间有可能在不同的上下文中受到访问，而这些访问又必须互斥，所以在该结构中设置了用于 P、V 操作的信号量 mmap_sem。此外，page_table_lock 也是为类似的目的而设置的。

虽然每个进程只有一个虚拟地址空间，但这个地址空间可以被别的进程来共享，如，子进程共享父进程的地址空间（也即共享 mm_struct 结构）。所以，用 mm_user 和 mm_count 进行计数。类型 atomic_t 实际上就是整数，但对这种整数的操作必须是“原子”的。

另外，还描述了代码段、数据段、堆栈段、参数段以及环境段的起始地址和结束地址。这里的段是对程序的逻辑划分，与我们前面所描述的段机制是不同的。

mm_context_t 是与平台相关的一个结构，对 i386 用处不大。

在后面对代码的分析中将对有些域给予进一步说明。

2. VM_AREA_STRUCT 结构

vm_area_struct 描述进程的一个虚拟地址区间，在/include/linux/mm.h 中描述如下：

```
struct vm_area_struct
        struct mm_struct * vm_mm;       /* 虚拟区间所在的地址空间*/
        unsigned long vm_start;         /* 在 vm_mm 中的起始地址*/
        unsigned long vm_end;           /*在 vm_mm 中的结束地址 */

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next;
```

```
        pgprot_t vm_page_prot;          /* 对这个虚拟区间的存取权限 */
        unsigned long vm_flags;         /* 虚拟区间的标志 */

        rb_node_t vm_rb;

        /*
         * For areas with an address space and backing store,
         * one of the address_space->i_mmap{,shared} lists,
         * for shm areas, the list of attaches, otherwise unused.
         */
        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;

        /*对这个区间进行操作的函数 */
        struct vm_operations_struct * vm_ops;

        /* Information about our backing store: */
        unsigned long vm_pgoff;         /* Offset (within vm_file) in PAGE_SIZE
                                           units, *not* PAGE_CACHE_SIZE */
        struct file * vm_file;          /* File we map to (can be NULL). */
        unsigned long vm_raend;         /* XXX: put full readahead info here. */
        void * vm_private_data;         /* was vm_pte (shared mem) */
};
```

vm_flag 是描述对虚拟区间的操作的标志，其定义和描述如表 6.1 所示。

表 6.1　　虚拟区间的标志

| 标 志 名 | 描　　述 |
|---|---|
| VM_DENYWRITE | 在这个区间映射一个打开后不能用来写的文件 |
| VM_EXEC | 页可以被执行 |
| VM_EXECUTABLE | 页含有可执行代码 |
| VM_GROWSDOWN | 这个区间可以向低地址扩展 |
| VM_GROWSUP | 这个区间可以向高地址扩展 |
| VM_IO | 这个区间映射一个设备的 I/O 地址空间 |
| VM_LOCKED | 页被锁住不能被交换出去 |
| VM_MAYEXEC | VM_EXEC 标志可以被设置 |
| VM_MAYREAD | VM_READ 标志可以被设置 |
| VM_MAYSHARE | VM_SHARE 标志可以被设置 |
| VM_MAYWRITE | VM_WRITE 标志可以被设置 |
| VM_READ | 页是可读的 |
| VM_SHARED | 页可以被多个进程共享 |
| VM_SHM | 页用于 IPC 共享内存 |
| VM_WRITE | 页是可写的 |

较高层次的结构 vm_area_struct 是由双向链表连接起来的，它们是按虚地址的降顺序来排列的，每个这样的结构都对应描述一个相邻的地址空间范围。之所以这样分割，是因为

每个虚拟区间可能来源不同，有的可能来自可执行映像，有的可能来自共享库，而有的则可能是动态分配的内存区,所以对每一个由 vm_area_struct 结构所描述的区间的处理操作和它前后范围的处理操作不同。因此 Linux 把虚拟内存分割管理，并利用了虚拟内存处理例程（vm_ops）来抽象对不同来源虚拟内存的处理方法。不同的虚拟区间其处理操作可能不同，Linux 在这里利用了面向对象的思想，即把一个虚拟区间看成一个对象，用 vm_area_struct 描述了这个对象的属性，其中的 vm_operation s-stract 结构描述了在这个对象上的操作，其定义在／include／linux／mm.h 中：

```
/*
 * These are the virtual MM functions - opening of an area, closing and
 * unmapping it (needed to keep files on disk up-to-date etc), pointer
 * to the functions called when a no-page or a wp-page exception occurs.
 */
struct vm_operations_struct {
        void (*open) (struct vm_area_struct * area);
        void (*close) (struct vm_area_struct * area);
struct page * (*nopage) (struct vm_area_struct * area, unsigned long address, int unused);
};
```

vm_operations 结构中包含的是函数指针；其中，open、close 分别用于虚拟区间的打开、关闭，而 nopage 用于当虚存页面不在物理内存而引起的“缺页异常”时所应该调用的函数。图 6.15 给出了虚拟区间的操作集。

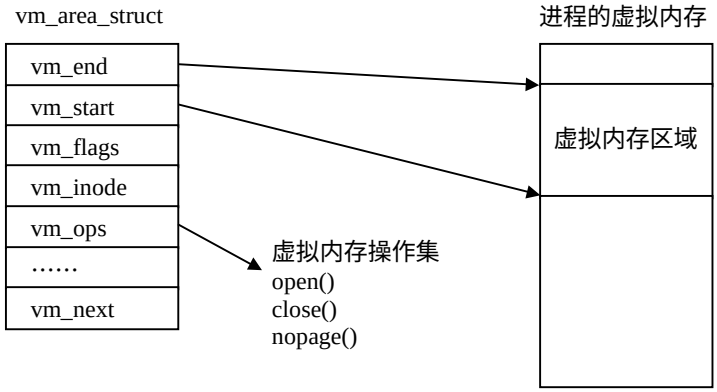


图 6.15　虚拟地址区间的操作集

3．红黑树结构

Linux 内核从 2.4.10 开始，对虚拟区间的组织不再采用 AVL 树，而是采用红黑树，这也是出于效率的考虑，虽然 AVL 树和红黑树很类似，但在插入和删除节点方面，采用红黑树的性能更好一些，下面对红黑树给予简单介绍。

一颗红黑树是具有以下特点的二叉树：

- 每个节点着有颜色，或者为红，或者为黑；
- 根节点为黑色；
- 如果一个节点为红色，那么它的子节点必须为黑色；

- 从一个节点到叶子节点上的所有路径都包含有相同的黑色节点数；

图 6.16 就是一颗红黑树。

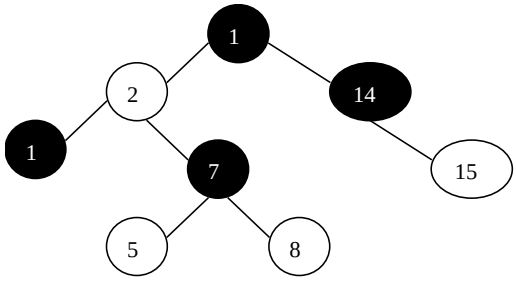


图 6.16　一颗红黑树

红黑树的结构在 include/linux/rbtree.h 中定义如下：

```
typedef struct rb_node_s
{
         struct rb_node_s * rb_parent;
         int rb_color;
#define RB_RED          0
#define RB_BLACK        1
        struct rb_node_s * rb_right;
        struct rb_node_s * rb_left;
} rb_node_t;
```

### 6.4.2　进程的虚拟空间

如前所述，每个进程拥有 3G 字节的用户虚存空间。但是，这并不意味着用户进程在这 3G 的范围内可以任意使用，因为虚存空间最终得映射到某个物理存储空间（内存或磁盘空间），才真正可以使用。

那么，内核怎样管理每个进程 3GB 的虚存空间呢？概括地说，用户进程经过编译、链接后形成的映象文件有一个代码段和数据段（包括 data 段和 bss 段），其中代码段在下，数据段在上。数据段中包括了所有静态分配的数据空间，即全局变量和所有申明为 static 的局部变量，这些空间是进程所必需的基本要求，这些空间是在建立一个进程的运行映像时就分配好的。除此之外，堆栈使用的空间也属于基本要求，所以也是在建立进程时就分配好的，如图 6.17 所示。

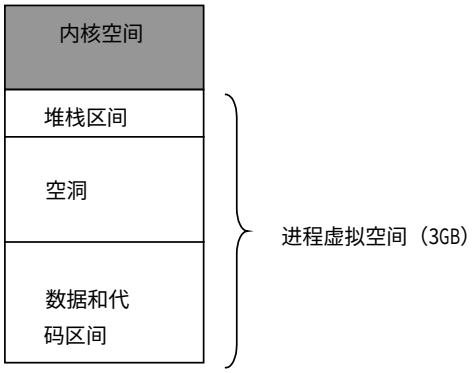


图 6.17　进程虚拟空间的划分

由图 6.17 可以看出，堆栈空间安排在虚存空间的顶部，运行时由顶向下延伸；代码段和数据段则在低部，运行时并不向上延伸。从数据段的顶部到堆栈段地址的下沿这个区间是一个巨大的空洞，这就是进程在运行时可以动态分配的空间（也叫动态内存）。

进程在运行过程中，可能会通过系统调用 mmap 动态申请虚拟内存或释放已分配的内存，新分配的虚拟内存必须和进程已有的虚拟地址链接起来才能使用；Linux 进程可以使用共享的程序库代码或数据，这样，共享库的代码和数据也需要链接到进程已有的虚拟地址中。在后面我们还会看到，系统利用了请页机制来避免对物理内存的过分使用。因为进程可能会访问当前不在物理内存中的虚拟内存，这时，操作系统通过请页机制把数据从磁盘装入到物理内存。为此，系统需要修改进程的页表，以便标志虚拟页已经装入到物理内存中，同时，Linux 还需要知道进程虚拟空间中任何一个虚拟地址区间的来源和当前所在位置，以便能够装入物理内存。

由于上面这些原因，Linux 采用了比较复杂的数据结构跟踪进程的虚拟地址。在进程的 task_struct 结构中包含一个指向 mm_struct 结构的指针。进程的 mm_struct 则包含装入的可执行映像信息以及进程的页目录指针 pgd。该结构还包含有指向 vm_area_struct 结构的几个指针，每个 vm_area_struct 代表进程的一个虚拟地址区间。

图 6.18 是某个进程的虚拟内存简化布局以及相应的几个数据结构之间的关系。从图中可

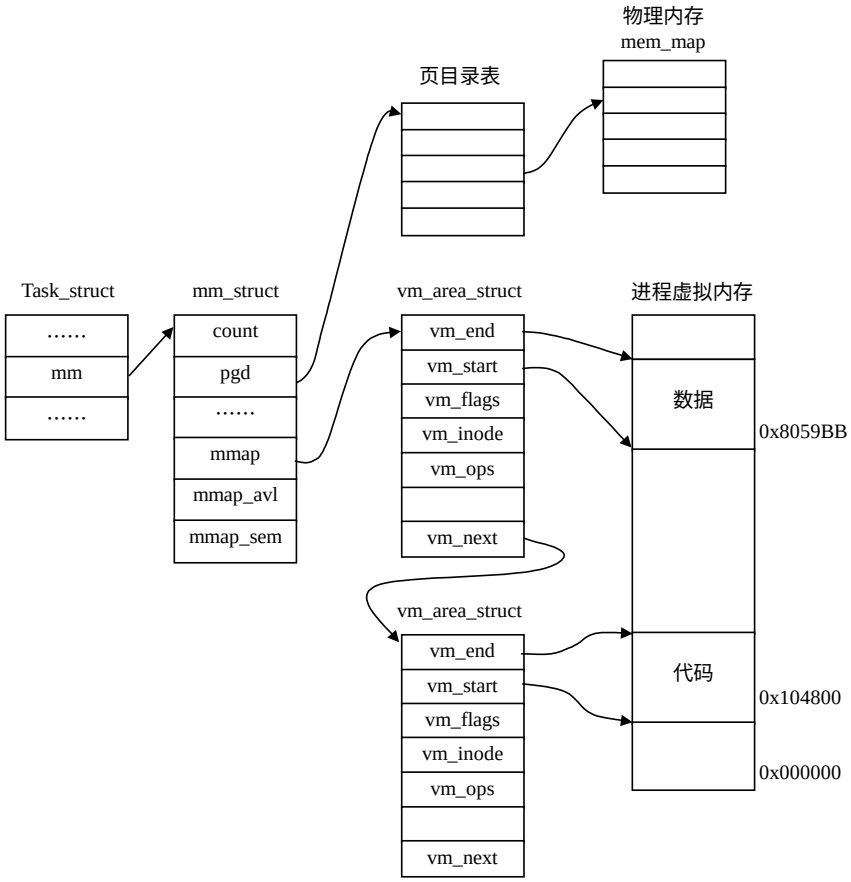


图 6.18　进程虚拟地址示意图

以看出，系统以虚拟内存地址的降序排列 vm_area_struct。在进程的运行过程中，Linux 要经常为进程分配虚拟地址区间，或者因为从交换文件中装入内存而修改虚拟地址信息，因此，vm_area_struct 结构的访问时间就成了性能的关键因素。为此，除链表结构外，Linux 还利用红黑（Red_black）树来组织 vm_area_struct。通过这种树结构，Linux 可以快速定位某个虚拟内存地址。

当进程利用系统调用动态分配内存时，Linux 首先分配一个 vm_area_struct 结构，并链接到进程的虚拟内存链表中，当后续的指令访问这一内存区间时，因为 Linux 尚未分配相应的物理内存,因此处理器在进行虚拟地址到物理地址的映射时会产生缺页异常(请看请页机制)，当 Linux 处理这一缺页异常时，就可以为新的虚拟内存区分配实际的物理内存。

在内核中，经常会用到这样的操作：给定一个属于某个进程的虚拟地址，要求找到其所属的区间以及 vma_area_struct 结构,这是由 find_vma()来实现的,其实现代码在 mm/mmap.c 中：

```
    * Look up the first VMA which satisfies  addr < vm_end,  NULL if none. */
struct vm_area_struct * find_vma (struct mm_struct * mm, unsigned long addr)
{
        struct vm_area_struct *vma = NULL;
```

```
        if (mm) {
                /* Check the cache first. */
                /* (Cache hit rate is typically around 35%.) */
                vma = mm->mmap_cache;
                if (! (vma && vma->vm_end > addr && vma->vm_start <= addr) ) {
                        rb_node_t * rb_node;

                        rb_node = mm->mm_rb.rb_node;
                        vma = NULL;

                        while (rb_node) {
                                struct vm_area_struct * vma_tmp;

                                vma_tmp = rb_entry (rb_node, struct vm_area_struct, vm_rb) ;

                                if (vma_tmp->vm_end > addr) {
                                        vma = vma_tmp;
                                        if (vma_tmp->vm_start <= addr)
                                                break;
                                        rb_node = rb_node->rb_left;
                                } else
                                        rb_node = rb_node->rb_right;
                        }
                        if (vma)
                                mm->mmap_cache = vma;
                }
        }
        return vma;
}
```

这个函数比较简单，我们对其主要点给予解释。

参数的含义：函数有两个参数，一个是指向 mm_struct 结构的指针，这表示一个进程的虚拟地址空间；一个是地址，表示该进程虚拟地址空间中的一个地址。

条件检查：首先检查这个地址是否恰好落在上一次（最近一次）所访问的区间中。根据代码作者的注释，命中率一般达到 35%，这也是 mm_struct 结构中设置 mmap_cache 指针的原因。如果没有命中，那就要在红黑树中进行搜索，红黑树与 AVL 树类似。

查找节点：如果已经建立了红黑树结构（rb_rode 不为空），就在红黑树中搜索。

如果找到指定地址所在的区间，就把 mmap_cache 指针设置成指向所找到的 vm_area_struct 结构。

如果没有找到，说明该地址所在的区间还没有建立，此时，就得建立一个新的虚拟区间，再调用 insert_vm_struct()函数将新建立的区间插入到 vm_struct 中的线性队列或红黑树中。

### 6.4.3 内存映射

当某个程序的映像开始执行时，可执行映像必须装入到进程的虚拟地址空间。如果该进程用到了任何一个共享库，则共享库也必须装入到进程的虚拟地址空间。由此可看出，Linux

并不将映像装入到物理内存，相反，可执行文件只是被连接到进程的虚拟地址空间中。随着程序的运行，被引用的程序部分会由操作系统装入到物理内存，这种将映像链接到进程地址空间的方法被称为“内存映射”。

当可执行映像映射到进程的虚拟地址空间时，将产生一组 vm_area_struct 结构来描述虚拟内存区间的起始点和终止点，每个 vm_area_struct 结构代表可执行映像的一部分，可能是可执行代码，也可能是初始化的变量或未初始化的数据，这些都是在函数 do_mmap()中来实现的。随着 vm_area_struct 结构的生成，这些结构所描述的虚拟内存区间上的标准操作函数也由 Linux 初始化。但要明确在这一步还没有建立从虚拟内存到物理内存的影射，也就是说还没有建立页表页目录。

为了对上面的原理进行具体的说明，我们来看一下 do_mmap()的实现机制。

函数 do_mmap()为当前进程创建并初始化一个新的虚拟区，如果分配成功，就把这个新的虚拟区与进程已有的其他虚拟区进行合并,do_mmap()在 include/linux/mm.h 中定义如下：

```
static inline unsigned long do_mmap (struct file *file, unsigned long addr,
        unsigned long len, unsigned long prot,
        unsigned long flag, unsigned long offset)
{
        unsigned long ret = -EINVAL;
        if  ( (offset + PAGE_ALIGN (len) )  < offset)
            goto out;
        if  (! (offset & ~PAGE_MASK) )
             ret = do_mmap_pgoff (file, addr, len, prot, flag, offset >> PAGE_SHIFT) ;
out:
        return ret;
}
```

函数中参数的含义如下。

file：表示要映射的文件，file 结构将在第八章文件系统中进行介绍。

offset\：文件内的偏移量，因为我们并不是一下子全部映射一个文件，可能只是映射文件的一部分，off 就表示那部分的起始位置。

len：要映射的文件部分的长度。

addr：虚拟空间中的一个地址，表示从这个地址开始查找一个空闲的虚拟区。

prot：这个参数指定对这个虚拟区所包含页的存取权限。可能的标志有 PROT_READ、PROT_WRITE、PROT_EXEC 和 PROT_NONE。前 3 个标志与标志 VM_READ、VM_WRITE 及 VM_EXEC 的意义一样。PROT_NONE 表示进程没有以上 3 个存取权限中的任意一个。

flag：这个参数指定虚拟区的其他标志：

MAP_GROWSDOWN，MAP_LOCKED，MAP_DENYWRITE 和 MAP_EXECUTABLE ：

它们的含义与表 6.1 中所列出标志的含义相同。

MAP_SHARED 和 MAP_PRIVATE ：

前一个标志指定虚拟区中的页可以被许多进程共享；后一个标志作用相反。这两个标志都涉及 vm_area_struct 中的 VM_SHARED 标志。

MAP_ANONYMOUS

表示这个虚拟区是匿名的，与任何文件无关。

MAP_FIXED

这个区间的起始地址必须是由参数 addr 所指定的。

```
MAP_NORESERVE
```

函数不必预先检查空闲页面的数目。

do_mmap()函数对参数 offset 的合法性检查后，就调用 do_mmap_pgoff（）函数，该函数才是内存映射的主要函数，do_mmap_pgoff（）的代码在 mm/mmap.c 中，代码比较长，我们分段来介绍：

```
unsigned long do_mmap_pgoff (struct file * file, unsigned long addr, unsigned long len,
         unsigned long prot, unsigned long flags, unsigned long pgoff)
{
         struct mm_struct * mm = current->mm;
         struct vm_area_struct * vma, * prev;
        unsigned int vm_flags;
        int correct_wcount = 0;
        int error;
         rb_node_t ** rb_link, * rb_parent;

        if  (file &&  (!file->f_op || !file->f_op->mmap) )
                return -ENODEV;

         if  ( (len = PAGE_ALIGN (len) )  == 0)
               return addr;

         if  (len > TASK_SIZE)
                return -EINVAL;

        /* offset overflow? */
         if  ( (pgoff +  (len >> PAGE_SHIFT) )  < pgoff)
                return -EINVAL;

        /* Too many mappings? */
         if  (mm->map_count > MAX_MAP_COUNT)
                return -ENOMEM;
```

函数首先检查参数的值是否正确，所提的请求是否能够被满足，如果发生以上情况中的任何一种，do_mmap()函数都终止并返回一个负值。

```
      /* Obtain the address to map to. we verify  (or select)  it and ensure
        * that it represents a valid section of the address space.
       */
            addr = get_unmapped_area (file, addr, len, pgoff, flags) ;
             if  (addr & ~PAGE_MASK)
                    return addr;
```

调用 get_unmapped_area（）函数在当前进程的用户空间中获得一个未映射区间的起始地址。PAGE_MASK 的值为 0xFFFFF000，因此，如果“addr & ～PAGE_MASK”为非 0，说明 addr 最低 12 位非 0，addr 就不是一个有效的地址，就以这个地址作为返回值；否则，addr 就是一个有效的地址（最低 12 位为 0），继续向下看：

```
/* Do simple checking here so the lower-level routines won't have
* to. we assume access permissions have been handled by the open
* of the memory object, so we don't do any here.
*/
vm_flags = calc_vm_flags(prot,flags) | mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
```

```
    /* mlock MCL_FUTURE? */
        if (vm_flags & VM_LOCKED) {
            unsigned long locked = mm->locked_vm << PAGE_SHIFT;
            locked += len;
            if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
                return -EAGAIN;
        }
```

如果 flag 参数指定的新虚拟区中的页必须锁在内存，且进程加锁页的总数超过了保存在进程的 task_struct 结构 rlim[RLIMIT_MEMLOCK].rlim_cur 域中的上限值，则返回一个负值。继续：

```
    if (file) {
        switch (flags & MAP_TYPE) {
        case MAP_SHARED:
            if ( (prot & PROT_WRITE) && ! (file->f_mode & FMODE_WRITE) )
                return -EACCES;

            /* Make sure we don't allow writing to an append-only file.. */
if (IS_APPEND (file->f_dentry->d_inode) && (file->f_mode & FMODE_WRITE) )
                return -EACCES;

            /* make sure there are no mandatory locks on the file. */
            if (locks_verify_locked (file->f_dentry->d_inode) )
                return -EAGAIN;

            vm_flags |= VM_SHARED | VM_MAYSHARE;
            if (! (file->f_mode & FMODE_WRITE) )
                vm_flags &= ~ (VM_MAYWRITE | VM_SHARED) ;

            /* fall through */
        case MAP_PRIVATE:
            if (! (file->f_mode & FMODE_READ) )
                return -EACCES;
            break;

        default:
            return -EINVAL;
        }
    } else {
        vm_flags |= VM_SHARED | VM_MAYSHARE;
        switch (flags & MAP_TYPE) {
        default:
            return -EINVAL;
        case MAP_PRIVATE:
            vm_flags &= ~ (VM_SHARED | VM_MAYSHARE) ;
            /* fall through */
        case MAP_SHARED:
            break;
        }
    }
```

如果 file 结构指针为 0，则目的仅在于创建虚拟区间，或者说，并没有真正的映射发生；如果 file 结构指针不为 0，则目的在于建立从文件到虚拟区间的映射，那就要根据标志指定的映射种类，把为文件设置的访问权考虑进去。

如果所请求的内存映射是共享可写的，就要检查要映射的文件是为写入而打开的，而不是以追加模式打开的，还要检查文件上没有上强制锁。

对于任何种类的内存映射，都要检查文件是否为读操作而打开的。

如果以上条件都不满足，就返回一个错误码。

```
/* Clear old maps */
        error = -ENOMEM;
munmap_back:
        vma = find_vma_prepare (mm, addr, &prev, &rb_link, &rb_parent) ;
        if  (vma && vma->vm_start < addr + len)  {
                if  (do_munmap (mm, addr, len) )
                        return -ENOMEM;
                goto munmap_back;
        }
```

函数 find_vma_prepare () 与 find_vma () 基本相同，它扫描当前进程地址空间的 vm_area_struct 结构所形成的红黑树，试图找到结束地址高于 addr 的第 1 个区间；如果找到了一个虚拟区，说明 addr 所在的虚拟区已经在使用，也就是已经有映射存在，因此要调用 do_munmap () 把这个老的虚拟区从进程地址空间中撤销，如果撤销不成功，就返回一个负数；如果撤销成功，就继续查找，直到在红黑树中找不到 addr 所在的虚拟区，并继续下面的检查：

```
        /* Check against address space limit. */
           if  ( (mm->total_vm << PAGE_SHIFT)  + len
               > current->rlim[RLIMIT_AS].rlim_cur)
                   return -ENOMEM;
```

total_vm 是表示进程地址空间的页面数，如果把文件映射到进程地址空间后，其长度超过了保存在当前进程 rlim[RLIMIT_AS].rlim_cur 中的上限值，则返回一个负数。

```
       /* Private writable mapping? Check memory availability.. */
             if  ( (vm_flags &  (VM_SHARED | VM_WRITE) )  == VM_WRITE &&
                 ! (flags & MAP_NORESERVE)  &&!vm_enough_memory (len >> PAGE_SHIFT) )
                     return -ENOMEM;
```

如果 flags 参数中没有设置 MAP_NORESERVE 标志，新的虚拟区含有私有的可写页，空闲页面数小于要映射的虚拟区的大小；则函数终止并返回一个负数；其中函数 vm_enough_memory () 用来检查一个进程的地址空间中是否有足够的内存来进行一个新的映射。

```
                /* Can we just expand an old anonymous mapping? */
             if  (!file && ! (vm_flags & VM_SHARED)  && rb_parent)
                     if  (vma_merge (mm, prev, rb_parent, addr, addr + len, vm_flags) )
                             goto out;
```

如果是匿名映射（file 为空），并且这个虚拟区是非共享的，则可以把这个虚拟区和与它紧挨的前一个虚拟区进行合并；虚拟区的合并是由 vma_merge()函数实现的。如果合并成功，则转 out 处，请看后面 out 处的代码。

```
           /* Determine the object being mapped and call the appropriate
              * specific mapper. the address has already been validated, but
              * not unmapped, but the maps are removed from the list.
```

```
        */
        vma = kmem_cache_alloc (vm_area_cachep, SLAB_KERNEL) ;
        if  (!vma)
                return -ENOMEM;
        vma->vm_mm = mm;
        vma->vm_start = addr;
        vma->vm_end = addr + len;
        vma->vm_flags = vm_flags;
        vma->vm_page_prot = protection_map[vm_flags & 0x0f];
       vma->vm_ops = NULL;
       vma->vm_pgoff = pgoff;
       vma->vm_file = NULL;
       vma->vm_private_data = NULL;
       vma->vm_raend = 0;
```

经过以上各种检查后，现在必须为新的虚拟区分配一个 vm_area_struct 结构。这是通过调用 Slab 分配函数 kmem_cache_alloc（）来实现的，然后就对这个结构的各个域进行了初始化。

```
            if  (file)  {
                error = -EINVAL;
                if  (vm_flags &  (VM_GROWSDOWN|VM_GROWSUP) )
                        goto free_vma;
                if  (vm_flags & VM_DENYWRITE)  {
                       error = deny_write_access (file) ;
                        if  (error)
                               goto free_vma;
                        correct_wcount = 1;
                }
                vma->vm_file = file;
                get_file (file) ;
                error = file->f_op->mmap (file, vma) ;
                if  (error)
                        goto unmap_and_free_vma;
       } else if  (flags & MAP_SHARED)  {
                error = shmem_zero_setup (vma) ;
                if  (error)
                        goto free_vma;
        }
free_vma:
       kmem_cache_free (vm_area_cachep, vma) ;
       return error;
}
```

如果建立的是从文件到虚存区间的映射，则情况下。

当参数 flags 中的 VM_GROWSDOWN 或 VM_GROWSUP 标志位为 1 时，说明这个区间可以向低地址或高地址扩展，但从文件映射的区间不能进行扩展，因此转到 free_vma，释放给 vm_area_struct 分配的 Slab，并返回一个错误。

当 flags 中的 VM_DENYWRITE 标志位为 1 时，就表示不允许通过常规的文件操作访问该文件，所以要调用 deny_write_access（）排斥常规的文件操作（参见第八章）。

get_file（）函数的主要作用是递增 file 结构中的共享计数。

每个文件系统都有个 fiel_operation 数据结构，其中的函数指针 mmap 提供了用来建立从该类文件到虚存区间进行映射的操作，这是最具有实质意义的函数；对于大部分文件系统，这个函数为 generic_file_mmap( )函数实现的，该函数执行以下操作。

初始化 vm_area_struct 结构中的 vm_ops 域。如果 VM_SHARED 标志为 1，就把该域设置成 file_shared_mmap，否则就把该域设置成 file_private_mmap。从某种意义上说，这个步骤所做的事情类似于打开一个文件并初始化文件对象的方法。

从索引节点的 i_mode 域（参见第八章）检查要映射的文件是否是一个常规文件。如果是其他类型的文件（例如目录或套接字），就返回一个错误代码。

从索引节点的 i_op 域中检查是否定义了 readpage( )的索引节点操作。如果没有定义，就返回一个错误代码。

调用 update_atime( )函数把当前时间存放在该文件索引节点的 i_atime 域中，并将这个索引节点标记成脏。

如果 flags 参数中的 MAP_SHARED 标志位为 1，则调用 shmem_zero_setup（）进行共享内存的映射。

继续看 do_mmap()中的代码。

```
        /* Can addr have changed??
         *
         * Answer: Yes, several device drivers can do it in their
        *         f_op->mmap method. -DaveM
         */
        addr = vma->vm_start;
```

源码作者给出了解释，意思是说，addr 有可能已被驱动程序改变，因此，把新虚拟区的起始地址赋给 addr。

```
        vma_link (mm, vma, prev, rb_link, rb_parent) ;
        if (correct_wcount)
                atomic_inc (&file->f_dentry->d_inode->i_writecount) ;
```

此时，应该把新建的虚拟区插入到进程的地址空间，这是由函数 vma_link（）完成的，该函数具有 3 方面的功能：

（1）把 vma 插入到虚拟区链表中；

（2）把 vma 插入到虚拟区形成的红黑树中；

（3）把 vam 插入到索引节点（inode）共享链表中。

函数 atomic_inc（x）给*x 加 1，这是一个原子操作。在内核代码中，有很多地方调用了以 atomic 为前缀的函数。所谓原子操作，就是在操作过程中不会被中断。

```
out:
        mm->total_vm += len >> PAGE_SHIFT;
        if (vm_flags & VM_LOCKED) {
                mm->locked_vm += len >> PAGE_SHIFT;
                make_pages_present (addr, addr + len) ;
        }
        return addr;
```

do_mmap()函数准备从这里退出，首先增加进程地址空间的长度，然后看一下对这个区间是否加锁，如果加锁，说明准备访问这个区间，就要调用 make_pages_present（）函数，建立虚拟页面到物理页面的映射，也就是完成文件到物理内存的真正调入。返回一个正数，

说明这次映射成功。

```
unmap_and_free_vma:
        if (correct_wcount)
                atomic_inc(&file->f_dentry->d_inode->i_writecount);
        vma->vm_file = NULL;
        fput(file);

       /* Undo any partial mapping done by a device driver. */
        zap_page_range(mm, vma->vm_start, vma->vm_end - vma->vm_start);
```

如果对文件的操作不成功，则解除对该虚拟区间的页面映射，这是由 zap_page_range() 函数完成的。

当你读到这里时可能感到困惑，页面的映射到底在何时建立？实际上，generic_file_mmap( )就是真正进行映射的函数。因为这个函数的实现涉及很多文件系统的内容，我们在此不进行深入的分析，当读者了解了文件系统的有关内容后，可自己进行分析。

这里要说明的是，文件到虚存的映射仅仅是建立了一种映射关系，也就是说，虚存页面到物理页面之间的映射还没有建立。当某个可执行映象映射到进程虚拟内存中并开始执行时，因为只有很少一部分虚拟内存区间装入到了物理内存，可能会遇到所访问的数据不在物理内存。这时，处理器将向 Linux 报告一个页故障及其对应的故障原因，于是就用到了请页机制。

# 6.5　请页机制

Linux 采用请页机制来节约内存，它仅仅把当前正在执行的程序要使用的虚拟页（少量一部分）装入内存。当需要访问尚未装入物理内存的虚拟内存区域时，处理器将向 Linux 报告一个页故障及其对应的故障原因。本节将主要介绍 arch／i386／mm／fault.c 中的页故障处理函数 do_page_fault，为了突出主题，我们将分析代码中的主要部分。

## 6.5.1　页故障的产生

页故障的产生有 3 种原因。

（1）一是程序出现错误，例如向随机物理内存中写入数据，或页错误发生在 TASK_SIZE（3G）的范围外，这些情况下，虚拟地址无效，Linux 将向进程发送 SIGSEGV 信号并终止进程的运行。

（2）另一种情况是，虚拟地址有效，但其所对应的页当前不在物理内存中，即缺页错误，这时，操作系统必须从磁盘映像或交换文件（此页被换出）中将其装入物理内存。这是本节要讨论的主要内容。

（3）最后一种情况是，要访问的虚地址被写保护，即保护错误，这时，操作系统必须判断：如果是用户进程正在写当前进程的地址空间，则发 SIGSEGV 信号并终止进程的运行；如果错误发生在一旧的共享页上时，则处理方法有所不同，也就是要对这一共享页进行复制，这就是我们后面要讲的写时复制（Copy On Write 简称 COW）技术。

有关页错误的发生次数的信息可在目录 proc/stat 下找到。

### 6.5.2 页错误的定位

页错误的定位既包含虚拟地址的定位，也包含被调入页在交换文件(swapfile)或在可执行映象中的定位。

具体地说，在一个进程访问一个无效页表项时，处理器产生一个陷入并报告一个页错误，它描述了页错误发生的虚地址和访问类型，这些类型通过页的错误码 error_code 中的前 3 位来判别 ，具体如下：

```
*   bit 0 == 0 means no page found, 1 means protection fault
*   bit 1 == 0 means read, 1 means write
*   bit 2 == 0 means kernel, 1 means user-mode。
```

也就是说，如果第 0 位为 0，则错误是由访问一个不存在的页引起的（页表的表项中 present 标志为 0），否则，如果第 0 位为 1，则错误是由无效的访问权所引起的；如果第 1 位为 0，则错误是由读访问或执行访问所引起，如果为 1，则错误是由写访问所引起的；如果第 2 位为 0，则错误发生在处理器处于内核态时，否则，错误发生在处理器处于用户态时。

页错误的线性地址被存于 CR2 寄存器，操作系统必须在 vm_area_struct 中找到页错误发生时页的虚拟地址（通过红黑树或旧版本中的 AVL 树），下面通过 do_page_fault()中的一部分源代码来说明这个问题：

```
/* CR2 中包含有最新的页错误发生时的虚拟地址*/
    __asm__ ("movl %%cr2,%0":"=r" (address)) ;
        vma = find_vma (current, address) ;
```

如果没找到，则说明访问了非法虚地址，Linux 会发信号终止进程（如果必要）。否则，检查页错误类型，如果是非法类型(越界错误，段权限错误等）同样会发信号终止进程，部分源代码如下：

```
vma = find_vma (current, address) ;
     if (!vma)
          goto bad_area;
   if (vma->vm_start <= address)
          goto good_area;
   if (! (vma->vm_flags & VM_GROWSDOWN) )
          goto bad_area;
    if (error_code & 4) {  /*如是用户态进程*/
          /* 不可访问堆栈空间*/
           if (address + 32 < regs->esp)
               goto bad_area;
    }
    if (expand_stack (vma, address) )
          goto bad_area;
bad_area:                  /* 用户态的访问*/
{
    if (error_code & 4) {
          current->tss.cr2 = address;
          current->tss.error_code = error_code;
          current->tss.trap_no = 14;
          force_sig (SIGSEGV, current) ;    /* 给当前进程发杀死信号*/
          return;
        ......
```

```
    die_if_kernel ("Oops", regs, error_code) ;        /*报告内核 */
     do_exit (SIGKILL) ;                          /*强行杀死进程*/
                    }
```

### 6.5.3　进程地址空间中的缺页异常处理

对有效的虚拟地址，如果是缺页错误，Linux 必须区分页所在的位置，即判断页是在交换文件中，还是在可执行映像中。为此，Linux 通过页表项中的信息区分页所在的位置。如果该页的页表项是无效的，但非空，则说明该页处于交换文件中，操作系统要从交换文件装入页。对于有效的虚拟地址 address，do_page_fault(　)转到 good_area 标号处的语句执行：

```
good_area:
write = 0;
if  (error_code & 2)  { /* 写访问 */
    if  (! (vma->vm_flags & VM_WRITE) )
        goto bad_area;
    write++;
} else                /* 读访问 */
    if  (error_code & 1 ||
        ! (vma->vm_flags &  (VM_READ | VM_EXEC) ) )
        goto bad_area;
```

如果错误由写访问引起，函数检查这个虚拟区是否可写。如果不可写，跳到 bad_area 代码处；如果可写，把 write 局部变量置为 1。

如果错误由读或执行访问引起，函数检查这一页是否已经存在于物理内存中。如果在，错误的发生就是由于进程试图访问用户态下的一个有特权的页面（页面的 User/Supervisor 标志被清除），因此函数跳到 bad_area 代码处（实际上这种情况从不发生，因为内核根本不会给用户进程分配有特权的页面）。如果不存在物理内存，函数还将检查这个虚拟区是否可读或可执行。

如果这个虚拟区的访问权限与引起错误的访问类型相匹配，则调用 handle_mm_fault(　)函数：

```
if  (!handle_mm_fault (tsk, vma, address, write) )  {
    tsk->tss.cr2 = address;
    tsk->tss.error_code = error_code;
    tsk->tss.trap_no = 14;
    force_sig (SIGBUS, tsk) ;
    if  (! (error_code & 4) )  /* 内核态 */
        goto no_context;
}
```

如果 handle_mm_fault(　)函数成功地给进程分配一个页面，则返回 1；否则返回一个适当的错误码，以便 do_page_fault(　)函数可以给进程发送 SIGBUS 信号。handle_mm_fault(　)函数有 4 个参数：tsk 指向错误发生时正在 CPU 上运行的进程；vma 指向引起错误的虚拟地址所在虚拟区；address 为引起错误的虚拟地址；write：如果 tsk 试图向 address 写，则置为 1，如果 tsk 试图读或执行 address，则置为 0。

handle_mm_fault（）函数首先检查用来映射 address 的页中间目录和页表是否存在。即使 address 属于进程的地址空间，但相应的页表可能还没有分配，因此，在做别的事情之

前首先执行分配页目录和页表的任务：

```
pgd = pgd_offset (vma->vm_mm, address) ;
pmd = pmd_alloc (pgd, address) ;
if  (!pmd)
     return -1;
    pte = pte_alloc (pmd, address) ;
if  (!pte)
return -1;
```

pgd_offset()宏计算出 address 所在页在页目录中的目录项指针。如果有中间目录(i386 不起作用)，调用 pmd_alloc( )函数分配一个新的中间目录。然后，如果需要，调用 pte_alloc( )函数分配一个新的页表。如果这两步都成功，pte 局部变量所指向的页表表项就是引用 address 的表项。然后调用 handle_pte_fault( )函数检查 address 地址所对应的页表表项:

```
        return handle_pte_fault (tsk, vma, address, write_access, pte) ;
```

handle_pte_fault( )函数决定怎样给进程分配一个新的页面。

如果被访问的页不存在，也就是说，这个页还没有被存放在任何一个页面中，那么，内核分配一个新的页面并适当地初始化。这种技术称为请求调页。

如果被访问的页存在但是被标为只读，也就是说，它已经被存放在一个页面中，那么，内核分配一个新的页面，并把旧页面的数据拷贝到新页面来初始化它的内容。这种技术称为写时复制。

### 6.5.4 请求调页

请求调页指的是一种动态内存分配技术，它把页面的分配推迟到不能再推迟为止，也就是说，一直推迟到进程要访问的页不在物理内存时为止，由此引起一个缺页错误。

请求调页技术的引入主要是因为进程开始运行的时候并不访问其地址空间中的全部地址。事实上，有一部分地址也许进程永远不使用。此外，程序的局部性原理保证了在程序执行的每个阶段，真正使用的进程页只有一小部分，因此临时用不着的页所在的物理页面可以由其他进程来使用。因此，对于全局分配（一开始就给进程分配所需要的全部页面，直到程序结束才释放这些页面）来说，请求调页是首选的，因为它增加了系统中的空闲页面的平均数，从而更好地利用空闲内存。从另一个观点来看，在内存总数保持不变的情况下，请求调页从总体上能使系统有更大的吞吐量。

为这一切优点付出的代价是系统额外的开销：由请求调页所引发的每个“缺页”错误必须由内核处理，这将浪费 CPU 的周期。幸运的是，局部性原理保证了一旦进程开始在一组页上运行，在接下来相当长的一段时间内它会一直停留在这些页上而不去访问其他的页：这样我们就可以认为“缺页”错误是一种稀有事件。

基于以下原因，被寻址的页可以不在主存中。

(1) 进程永远也没有访问到这个页。内核能够识别这种情况，这是因为页表相应的表项被填充为 0，也就是说，pte_none 宏返回 1。

(2) 进程已经访问过这个页，但是这个页的内容被临时保存在磁盘上。内核能够识别这种情况，这是因为页表相应表项没被填充为 0（然而，由于页面不存在物理内存中，Present

为 0)。

handle_pte_fault（ ）函数通过检查与 address 相关的页表表项来区分这两种情况：

```
entry = *pte;
if (!pte_present (entry) ) {
    if (pte_none (entry) )
        return do_no_page (tsk, vma, address, write_access,
                           pte) ;
    return do_swap_page (tsk, vma, address, pte, entry,
                         write_access) ;
}
```

我们将在交换机制一节检查页被保存到磁盘上的这种情况(do_swap_page( ) 函数)。

在其他情况下，当页从未被访问时则调用 do_no_page( )函数。有两种方法装入所缺的页，这取决于这个页是否被映射到磁盘文件。该函数通过检查 vma 虚拟区描述符的 nopage 域来确定这一点，如果页与文件建立起了映射关系，则 nopage 域就指向一个把所缺的页从磁盘装入到 RAM 的函数。因此，可能的情况如下所述。

(1) vma->vm_ops->nopage 域不为 NULL。在这种情况下，某个虚拟区映射一个磁盘文件，nopage 域指向从磁盘读入的函数。这种情况涉及到磁盘文件的低层操作。

(2) 或者 vm_ops 域为 NULL，或者 vma->vm_ops->nopage 域为 NULL。在这种情况下，虚拟区没有映射磁盘文件，也就是说，它是一个匿名映射。因此，do_no_page( )调用 do_anonymous_page( )函数获得一个新的页面：

```
    if (!vma->vm_ops || !vma->vm_ops->nopage)
          return do_anonymous_page (tsk, vma, page_table,
write_access) ;
```

do_anonymous_page（ ）函数分别处理写请求和读请求：

```
    if (write_access) {
        page = __get_free_page (GFP_USER) ;
        memset ( (void *) (page) , 0, PAGE_SIZE)
       entry = pte_mkwrite (pte_mkdirty (mk_pte (page,
                vma->vm_page_prot) ) ) ;
        vma->vm_mm->rss++;
        tsk->min_flt++;
        set_pte (pte, entry) ;
         return 1;
}
```

当处理写访问时，该函数调用__get_free_page( )分配一个新的页面，并利用 memset 宏把新页面填为 0。然后该函数增加 tsk 的 min_flt 域以跟踪由进程引起的次级缺页(这些缺页只需要一个新页面)的数目，再增加进程的内存区结构 vma->vm_mm 的 rss 域以跟踪分配给进程的页面数目。然后页表相应的表项被设为页面的物理地址，并把这个页面标记为可写和脏两个标志。

相反，当处理读访问时，页的内容是无关紧要的，因为进程正在对它进行第一次寻址。给进程一个填充为 0 的页要比给它一个由其他进程填充了信息的旧页更为安全。Linux 在请求调页方面做得更深入一些。没有必要立即给进程分配一个填充为零的新页面，由于我们也可以给它一个现有的称为零页的页，这样可以进一步推迟页面的分配。零页在内核初始化期

间被静态分配，并存放在 empty_zero_page 变量中（一个有 1024 个长整数的数组，并用 0 填充）；它存放在第六个页面中（从物理地址 0x00005000 开始），并且可以通过 ZERO_PAGE 宏来引用。

因此页表表项被设为零页的物理地址：

```
entry = pte_wrprotect (mk_pte (ZERO_PAGE, vma->vm_page_prot) ) ;
set_pte (pte, entry) ;
return 1;
```

由于这个页被标记为不可写，如果进程试图写这个页，则写时复制机制被激活。当且仅当在这个时候，进程才获得一个属于自己的页并对它进行写。这种机制在下一部分进行描述。

### 6.5.5 写时复制

写时复制技术最初产生于 UNIX 系统，用于实现一种傻瓜式的进程创建：当发出 fork( ) 系统调用时，内核原样复制父进程的整个地址空间并把复制的那一份分配给子进程。这种行为是非常耗时的，因为它需要：

- 为子进程的页表分配页面；
- 为子进程的页分配页面；
- 初始化子进程的页表；
- 把父进程的页复制到子进程相应的页中。

创建一个地址空间的这种方法涉及许多内存访问，消耗许多 CPU 周期，并且完全破坏了高速缓存中的内容。在大多数情况下，这样做常常是毫无意义的，因为许多子进程通过装入一个新的程序开始它们的执行，这样就完全丢弃了所继承的地址空间。

现在的 UNIX 内核（包括 Linux），采用一种更为有效的方法称之为写时复制（或 COW）。这种思想相当简单：父进程和子进程共享页面而不是复制页面。然而，只要页面被共享，它们就不能被修改。无论父进程和子进程何时试图写一个共享的页面，就产生一个错误，这时内核就把这个页复制到一个新的页面中并标记为可写。原来的页面仍然是写保护的：当其他进程试图写入时，内核检查写进程是否是这个页面的唯一属主；如果是，它把这个页面标记为对这个进程是可写的。

Page 结构的 count 域用于跟踪共享相应页面的进程数目。只要进程释放一个页面或者在它上面执行写时复制，它的 count 域就递减；只有当 count 变为 NULL 时，这个页面才被释放。

现在我们讲述 Linux 怎样实现写时复制（COW）。当 handle_pte_fault( )确定“缺页”错误是由请求写一个页面所引起的时（这个页面存在于内存中且是写保护的），它执行以下语句：

```
if (pte_present (pte) ) {
    entry = pte_mkyoung (entry) ;
    set_pte (pte, entry) ;
    flush_tlb_page (vma, address) ;
    if (write_access) {
        if (!pte_write (entry) )
            return do_wp_page (tsk, vma, address, pte) ;
        entry = pte_mkdirty (entry) ;
        set_pte (pte, entry) ;
```

```
            flush_tlb_page (vma, address) ;
            }
        return 1;
    }
```

首先，调用 pte_mkyoung( ) 和 set_pte( )函数来设置引起错误的页所对应页表项的访问位。这个设置使页“年轻”并减少它被交换到磁盘上的机会。如果错误由违背写保护而引起的，handle_pte_fault( ) 返回由 do_wp_page( )函数产生的值；否则，则已检测到某一错误情况（例如，用户态地址空间中的页，其 User/Supervisor 标志为 0），且函数返回 1。

do_wp_page( )函数首先把 page_table 参数所引用的页表表项装入局部变量 pte，然后再获得一个新页面：

```
    pte = *page_table;
    new_page = __get_free_page (GFP_USER) ;
```

由于页面的分配可能阻塞进程，因此，一旦获得页面，这个函数就在页表表项上执行下面的一致性检查：

- 当进程等待一个空闲的页面时，这个页是否已经被交换出去（pte 和 *page_table 的值不相同）；
- 这个页是否已不在物理内存中（页表表项中页的 Present 标志为 0）；
- 页现在是否可写（页项中页的 Read/Write 标志为 1）。

如果以上情况中的任意一个发生，do_wp_page( )释放以前所获得的页面，并返回 1。

现在，函数更新次级缺页的数目，并把引起错误的页的页描述符指针保存到 page_map 局部变量中。

```
    tsk->min_flt++;
    page_map = mem_map + MAP_NR (old_page) ;
```

接下来，函数必须确定是否必须真的把这个页复制一份。如果仅有一个进程使用这个页，就无须应用写时复制技术，而且进程应该能够自由地写这个页。因此，这个页面被标记为可写，这样当试图写入的时候就不会再次引起“缺页”错误，以前分配的新的页面也被释放，函数结束并返回 1。这种检查是通过读取 page 结构的 count 域而进行的：

```
    if (page_map->count == 1) {
        set_pte (page_table, pte_mkdirty (pte_mkwrite (pte) ) ) ;
        flush_tlb_page (vma, address) ;
        if (new_page)
            free_page (new_page) ;
        return 1;
    }
```

相反，如果这个页面由两个或多个进程所共享，函数把旧页面(old_page)的内容复制到新分配的页面(new_page)中：

```
    if (old_page == ZERO_PAGE)
        memset ( (void *) new_page, 0, PAGE_SIZE) ;
    else
        memcpy ( (void *) new_page, (void *) old_page, PAGE_SIZE) ;
    set_pte (page_table, pte_mkwrite (pte_mkdirty (
            mk_pte (new_page, vma->vm_page_prot) ) ) ) ;
    flush_tlb_page (vma, address) ;
```

```
__free_page (page_map) ;
return 1;
```

如果旧页面是零页面，就使用 memset 宏把新的页面填充为 0。否则，使用 memcpy 宏复制页面的内容。不要求一定要对零页作特殊的处理，但是特殊处理确实能够提高系统的性能，因为它使用很少的地址而保护了微处理器的硬件高速缓存。

然后，用新页面的物理地址更新页表的表项，并把新页面标记为可写和脏。最后，函数调用__free_pages( )减小对旧页面的引用计数。

### 6.5.6 对本节的几点说明

(1) 通过 fork () 建立进程，开始时只有一个页目录和一页左右的可执行页，于是缺页异常会频繁发生。

(2) 虚拟地址映射到物理地址，只有在请页时才完成，这时要建立页表和更新页表（页表是动态建立的）。页表不可被换出，不记年龄，它们被内核中保留，只有在 exit 时清除。

(3) 在处理页故障的过程中，因为要涉及到磁盘访问等耗时操作，因此操作系统会选择另外一个进程进入执行状态，即进行新一轮调度。

## 6.6 交换机制

当物理内存出现不足时，Linux 内存管理子系统需要释放部分物理内存页面。这一任务由内核的交换守护进程 kswapd 完成，该内核守护进程实际是一个内核线程，它在内核初始化时启动，并周期地运行。它的任务就是保证系统中具有足够的空闲页面，从而使内存管理子系统能够有效运行。

### 6.6.1 交换的基本原理

如前所述，每个进程的可以使用的虚存空间很大（3GB），但实际使用的空间并不大，一般不会超过几 MB，大多数情况下只有几十 KB 或几百 KB。可是，当系统的进程数达到几百甚至上千个时，对存储空间的总需求就很大，在这种情况下，一般的物理内存量就很难满足要求。因此，在计算机技术的发展史上很早就有了把内存的内容与一个专用的磁盘空间交换的技术，在 Linux 中，我们把用作交换的磁盘空间叫做交换文件或交换区。

交换技术已经使用了很多年。第 1 个 UNIX 系统内核就监控空闲内存的数量。当空闲内存数量小于一个固定的极限值时，就执行换出操作。换出操作包括把进程的整个地址空间拷贝到磁盘上。反之，当调度算法选择出一个进程运行时，整个进程又被从磁盘中交换进来。

现代的 UNIX（包括 Linux）内核已经摒弃了这种方法，主要是因为当进行换入换出时，上下文切换的代价相当高。在 Linux 中，交换的单位是页面而不是进程。尽管交换的单位是页面，但交换还是要付出一定的代价，尤其是时间的代价。实际上，在操作系统中，时间和空间是一对矛盾，常常需要在二者之间作出平衡，有时需要以空间换时间，有时需要以时间

换空间，页面交换就是典型的以时间换空间。这里要说明的是，页面交换是不得已而为之，例如在时间要求比较紧急的实时系统中，是不宜采用页面交换机制的，因为它使程序的执行在时间上有了较大的不确定性。因此，Linux 给用户提供了一种选择，可以通过命令或系统调用开启或关闭交换机制。

在页面交换中，页面置换算法是影响交换性能的关键性指标，其复杂性主要与换出有关。具体说来，必须考虑 4 个主要问题：

- 哪种页面要换出；

如何在交换区中存放页面；

如何选择被交换出的页面；

何时执行页面换出操作。

请注意，我们在这里所提到的页或页面指的是其中存放的数据，因此，所谓页面的换入换出实际上是指页面中数据的换入换出。

1．哪种页面被换出

实际上，交换的最终目的是页面的回收。并非内存中的所有页面都是可以交换出去的。事实上，只有与用户空间建立了映射关系的物理页面才会被换出去，而内核空间中内核所占的页面则常驻内存。我们下面对用户空间中的页面和内核空间中的页面给出进一步的分类讨论。

可以把用户空间中的页面按其内容和性质分为以下几种：

（1）进程映像所占的页面，包括进程的代码段、数据段、堆栈段以及动态分配的“存储堆”(参见图 6.18)；

（2）通过系统调用 mmap()把文件的内容映射到用户空间；

（3）进程间共享内存区。

对于第 1 种情况，进程的代码段数据段所占的内存页面可以被换入换出，但堆栈所占的页面一般不被换出，因为这样可以简化内核的设计。

对于第 2 种情况，这些页面所使用的交换区就是被映射的文件本身。

对于第 3 种情况，其页面的换入换出比较复杂。

与此相对照，映射到内核空间中的页面都不会被换出。具体来说，内核代码和内核中的全局量所占的内存页面既不需要分配(启动时被装入)，也不会被释放，这部分空间是静态的。(相比之下，进程的代码段和全局量都在用户空间，所占的内存页面都是动态的，使用前要经过分配，最后都会被释放，中途可能被换出而回收后另行分配)

除此之外，内核在执行过程中使用的页面要经过动态分配，但永驻内存，此类页面根据其内容和性质可以分为两类。

（1）内核调用 kmalloc()或 vmalloc()为内核中临时使用的数据结构而分配的页于是立即释放。但是，由于一个页面中存放有多个同种类型的数据结构，所以要到整个页面都空闲时才把该页面释放。

（2）内核中通过调用 alloc_pages()，为某些临时使用和管理目的而分配的页面，例如，每个进程的内核栈所占的两个页面、从内核空间复制参数时所使用的页面等。这些页面也是一旦使用完毕便无保存价值，所以立即释放。

在内核中还有一种页面，虽然使用完毕，但其内容仍有保存价值，因此，并不立即释放。这类页面“释放”之后进入一个 LRU 队列，经过一段时间的缓冲让其“老化”。如果在此期间又要用到其内容了，就又将其投入使用，否则便继续让其老化，直到条件不再允许时才加以回收。这种用途的内核页面大致有以下这些：

- 文件系统中用来缓冲存储一些文件目录结构 dentry 的空间；
- 文件系统中用来缓冲存储一些索引节点 inode 的空间；
- 用于文件系统读／写操作的缓冲区。

2．如何在交换区中存放页面

我们知道物理内存被划分为若干页面，每个页面的大小为 4KB。实际上，交换区也被划分为块，每个块的大小正好等于一页，我们把交换区中的一块叫做一个页插槽（Page Slot），意思是说，把一个物理页面插入到一个插槽中。当进行换出时，内核尽可能把换出的页放在相邻的插槽中，从而减少在访问交换区时磁盘的寻道时间。这是高效的页面置换算法的物质基础。

如果系统使用了多个交换区，事情就变得更加复杂了。快速交换区（也就是存放在快速磁盘中的交换区）可以获得比较高的优先级。当查找一个空闲插槽时，要从优先级最高的交换区中开始搜索。如果优先级最高的交换区不止一个，为了避免超负荷地使用其中一个，应该循环选择相同优先级的交换区。如果在优先级最高的交换区中没有找到空闲插槽，就在优先级次高的交换区中继续进行搜索，依此类推。

3．如何选择被交换出的页面

页面交换是非常复杂的，其主要内容之一就是如何选择要换出的页面，我们以循序渐进的方式来讨论页面交换策略的选择。

策略一，需要时才交换。每当缺页异常发生时，就给它分配一个物理页面。如果发现没有空闲的页面可供分配，就设法将一个或多个内存页面换出到磁盘上，从而腾出一些内存页面来。这种交换策略确实简单，但有一个明显的缺点，这是一种被动的交换策略，需要时才交换，系统势必要付出相当多的时间进行换入换出。

策略二，系统空闲时交换。与策略一相比较，这是一种积极的交换策略，也就是，在系统空闲时，预先换出一些页面而腾出一些内存页面，从而在内存中维持一定的空闲页面供应量，使得在缺页中断发生时总有空闲页面可供使用。至于换出页面的选择，一般都采用 LRU（最近最少使用）算法。但是这种策略实施起来也有困难，因为并没有哪种方法能准确地预测对页面的访问，所以，完全可能发生这样的情况，即一个好久没有受到访问的页面刚刚被换出去，却又要访问它了，于是又把它换进来。在最坏的情况下，有可能整个系统的处理能力都被这样的换入／换出所影响，而根本不能进行有效的计算和操作。这种现象被称为页面的“抖动”。

策略三，换出但并不立即释放。当系统挑选出若干页面进行换出时，将相应的页面写入磁盘交换区中，并修改相应页表中页表项的内容（把 present 标志位置为 0），但是并不立即释放，而是将其 page 结构留在一个缓冲（Cache）队列中，使其从活跃（Active）状态转为

不活跃（Inactive）状态。至于这些页面的最后释放，要推迟到必要时才进行。这样，如果一个页面在释放后又立即受到访问，就可以从物理页面的缓冲队列中找到相应的页面，再次为之建立映射。由于此页面尚未释放，还保留着原来的内容，就不需要磁盘读入了。经过一段时间以后，一个不活跃的内存页面一直没有受到访问，那这个页面就需要真正被释放了。

策略四，把页面换出推迟到不能再推迟为止。实际上，策略三还有值得改进的地方。首先在换出页面时不一定要把它的内容写入磁盘。如果一个页面自从最近一次换入后并没有被写过（如代码），那么这个页面是“干净的”，就没有必要把它写入磁盘。其次，即使“脏”页面，也没有必要立即写出去，可以采用策略三。至于“干净”页面，可以一直缓冲到必要时才加以回收，因为回收一个“干净”页面花费的代价很小。

下面对物理页面的换入/换出给出一个概要描述，这里涉及到前面介绍的 page 结构和 free_area 结构。

（1）释放页面。如果一个页面变为空闲可用，就把该页面的 page 结构链入某个页面管理区(Zone)的空闲队列 free_area，同时页面的使用计数 count 减 1。

（2）分配页面。调用__alloc_pages()或__get_free_page()从某个空闲队列分配内存页面，并将其页面的使用计数 count 置为 1。

（3）活跃状态。已分配的页面处于活跃状态，该页面的数据结构 page 通过其队列头结构 lru 链入活跃页面队列 active_list，并且在进程地址空间中至少有一个页与该页面之间建立了映射关系。

（4）不活跃“脏”状态。处于该状态的页面其 page 结构通过其队列头结构 lru 链入不活跃“脏”页面队列 inactive_dirty_list，并且原则是任何进程的页面表项不再指向该页面，也就是说，断开页面的映射，同时把页面的使用计数 count 减 1。

（5）将不活跃“脏”页面的内容写入交换区，并将该页面的 page 结构从不活跃“脏”页面队列 inactive_dirty_list 转移到不活跃“干净”页面队列，准备被回收。

（6）不活跃“干净”状态。页面 page 结构通过其队列头结构 lru 链入某个不活跃“干净”页面队列，每个页面管理区都有个不活跃“干净”页面队列 inactive_clean_list。

（7）如果在转入不活跃状态以后的一段时间内，页面又受到访问，则又转入活跃状态并恢复映射。

（8）当需要时，就从“干净”页面队列中回收页面，也就是说或者把页面链入到空闲队列，或者直接进行分配。

以上是页面换入／换出及回收的基本思想，实际的实现代码还要更复杂一些。

4．何时执行页面换出操作

为了避免在 CPU 忙碌的时候，也就是在缺页异常发生时，临时搜索可供换出的内存页面并加以换出，Linux 内核定期地检查系统内的空闲页面数是否小于预定义的极限，一旦发现空闲页面数太少，就预先将若干页面换出，以减轻缺页异常发生时系统所承受的负担。当然，由于无法确切地预测页面的使用，即使这样做了也还可能出现缺页异常发生时内存依然没有足够的空闲页面。但是，预换出毕竟能减少空闲页面不够用的概率。并且通过选择适当的参数（如每隔多久换出一次，每次换出多少页），可以使临时寻找要换出页面的情况很少发生。为此，Linux 内核设置了一个专伺定期将页面换出的守护进程 kswapd。

### 6.6.2 页面交换守护进程 kswapd

从原理上说，kswapd 相当于一个进程，它有自己的进程控制块 task_struct 结构。与其他进程一样受内核的调度。而正因为内核将它按进程来调度，就可以让它在系统相对空闲的时候来运行。不过，与普通进程相比，kswapd 有其特殊性。首先，它没有自己独立的地址空间，所以在近代操作系统理论中把它称为“线程”以与进程相区别。那么，kswapd 的地址空间是什么？实际上，内核空间就是它的地址空间。在这一点上，它与中断服务例程相似。其次，它的代码是静态地链接在内核中的，因此，可以直接调用内核中的各种子程序和函数。

kswapd 的源代码基本上都在 mm/vmscan.c 中，图 6.19 给出了 kswapd 中与交换有关的主要函数调用关系。

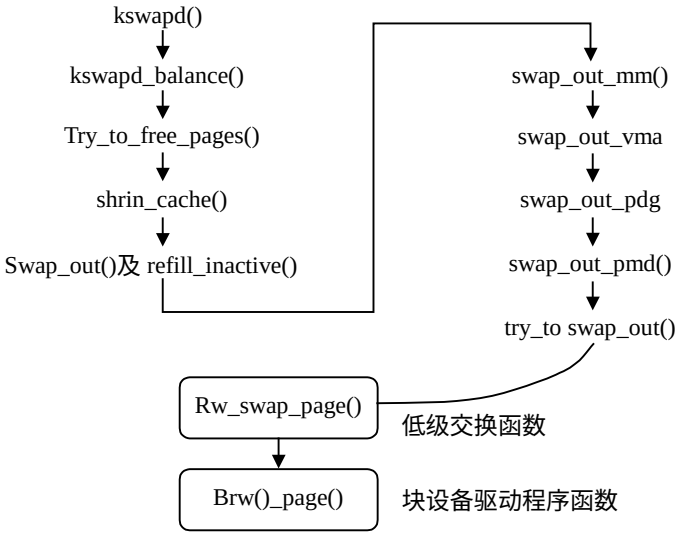


图 6.19 kswapd 的实现代码中与交换相关的主要函数的调用关系

从上面的调用关系可以看出，kswapd 的实现相当复杂，这不仅仅涉及复杂的页面交换技术，还涉及与磁盘相关的具体文件操作，因此，为了理清思路，搞清主要内容，我们对一些主要函数给予描述。

1. kswapd ()

在 Linux 2.4.10 以后的版本中对 kswapd () 的实现代码进行了模块化组织，可读性大大加强，代码如下：

```
    int kswapd (void *unused)
{
         struct task_struct *tsk = current;
        DECLARE_WAITQUEUE (wait, tsk) ;

        daemonize () ;        /*内核线程的初始化*/
        strcpy (tsk->comm, "kswapd") ;
        sigfillset (&tsk->blocked) ;  /*把进程 PCB 中的阻塞标志位全部置为 1*/
```

```
        /*
         * Tell the memory management that we're a "memory allocator",
        * and that if we need more memory we should get access to it
         * regardless  (see "__alloc_pages () ") . "kswapd" should
         * never get caught in the normal page freeing logic.
        *
        *  (Kswapd normally doesn't need memory anyway, but sometimes
         * you need a small amount of memory in order to be able to
         * page out something else, and this flag essentially protects
         * us from recursively trying to free more memory as we're
         * trying to free the first piece of memory in the first place) .
        */
        tsk->flags |= PF_MEMALLOC; /*这个标志表示给 kswapd 要留一定的内存*/

        /*
         * Kswapd main loop.
         */
        for  (;;)  {
               __set_current_state (TASK_INTERRUPTIBLE) ;
               add_wait_queue (&kswapd_wait, &wait) ; /*把 kswapd 加入等待队列*/

               mb () ;   /*增加一条汇编指令*/
                if  (kswapd_can_sleep () )     /*检查调度标志是否置位*/
                        schedule () ;     /*调用调度程序*/

            _set_current_state (TASK_RUNNING) ; /*让 kswapd 处于就绪状态*/
            remove_wait_queue (&kswapd_wait, &wait) ; /*把 kswapd 从等待队列删除*/

                /*
                 * If we actually get into a low-memory situation,
                 * the processes needing more memory will wake us
                 * up on a more timely basis.
                */
                kswapd_balance () ;   /* kswapd 的核心函数，请看后面内容*/
               run_task_queue (&tq_disk) ;  /*运行 tq_disk 队列中的例程*/
        }
}
```

kswapd 是内存管理中唯一的一个线程，其创建如下：

```
static int __init kswapd_init (void)
{
         printk ("Starting kswapd\n") ;
         swap_setup () ;
         kernel_thread (kswapd, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL) ;
        return 0;
}
```

然后，在内核启动时由模块的初始化例程调用 kswapd_init：

```
module_init(kswapd_init)
```

从上面的介绍可以看出，kswapd 成为内核的一个线程，其主循环是一个无限循环。循环一开始，把它加入等待队列，但如果调度标志为 1，就执行调度程序，紧接着就又把它从等

待队列删除，将其状态变为就绪。只要调度程序再次执行，它就会得到执行，如此周而复始进行下去。

2．kswapd_balance()函数

从该函数的名字可以看出，这是一个要求得平衡的函数，那么，求得什么样的平衡呢？在本章的初始化一节中，我们介绍了物理内存的 3 个层次，即存储节点、管理区和页面。所谓平衡就是对页面的释放要均衡地在各个存储节点、管理区中进行，代码如下：

```
static void kswapd_balance (void)
{
         int need_more_balance;
         pg_data_t * pgdat;

         do {
                 need_more_balance = 0;
                 pgdat = pgdat_list;
                do
                         need_more_balance |= kswapd_balance_pgdat (pgdat) ;
                while  ( (pgdat = pgdat->node_next) ) ;
        } while  (need_more_balance) ;
}
```

这个函数比较简单，主要是对每个存储节点进行扫描。然后又调用 kswapd_balance_pgdat () 对每个管理区进行扫描：

```
static int kswapd_balance_pgdat (pg_data_t * pgdat)
{
         int need_more_balance = 0, i;
        zone_t * zone;

         for  (i = pgdat->nr_zones-1; i >= 0; i--)  {
                zone = pgdat->node_zones + i;
                 if  (unlikely (current->need_resched) )
                         schedule () ;
                 if  (!zone->need_balance)
                         continue;
                 if  (!try_to_free_pages (zone, GFP_KSWAPD, 0) )  {
                         zone->need_balance = 0;
                        __set_current_state (TASK_INTERRUPTIBLE) ;
                         schedule_timeout (HZ) ;
                        continue;
                 }
                if  (check_classzone_need_balance (zone) )
                         need_more_balance = 1;
                 else
                         zone->need_balance = 0;
         }
```

其中，最主要的函数是 try_to_free_pages ()，能否调用这个函数取决于平衡标志 need_balance 是否为 1，也就是说看某个管理区的空闲页面数是否小于最高警戒线，这是由 check_classzone_need_balance()函数决定的。当某个管理区的空闲页面数小于其最高警戒

线时就调用 try_to_free_pages ()。

3. try_to_free_pages ()

该函数代码如下：

```
   int try_to_free_pages (zone_t *classzone, unsigned int gfp_mask, unsigned int order)
{
         int priority = DEF_PRIORITY;
         int nr_pages = SWAP_CLUSTER_MAX;

         gfp_mask = pf_gfp_mask (gfp_mask) ;
         do {
                nr_pages = shrink_caches (classzone, priority, gfp_mask, nr_pages) ;
                 if  (nr_pages <= 0)
                        return 1;
        } while  (--priority) ;

        /*
          * Hmm.. Cache shrink failed - time to kill something?
          * Mhwahahhaha! This is the part I really like. Giggle.
          */
        out_of_memory () ;
         return 0;
}
```

其中的优先级表示对队列进行扫描的长度，缺省的优先级 DEF_PRIORITY 为 6（最低优先级）。假定队列长度为 L，优先级 6 就表示要扫描的队列长度为 $L/2^6$，所以这个循环至少循环 6 次。nr_pages 为要换出的页面数，其最大值 SWAP_CLUSTER_MAX 为 32。其中主要调用的函数为 shrink_caches ()：

```
       static int shrink_caches (zone_t * classzone, int priority, unsigned int gfp_mask, int
nr_pages)
    {
             int chunk_size = nr_pages;
             unsigned long ratio;

             nr_pages -= kmem_cache_reap (gfp_mask) ;
            if  (nr_pages <= 0)
                     return 0;

            nr_pages = chunk_size;
             /* try to keep the active list 2/3 of the size of the cache */
             ratio =  (unsigned long)  nr_pages * nr_active_pages /  ( (nr_inactive_pages + 1)
* 2) ;
             refill_inactive (ratio) ;

            nr_pages = shrink_cache (nr_pages, classzone, gfp_mask, priority) ;
             if  (nr_pages <= 0)
                     return 0;

             shrink_dcache_memory (priority, gfp_mask) ;
             shrink_icache_memory (priority, gfp_mask) ;
```

```
1 #ifdef CONFIG_QUOTA
        shrink_dqcache_memory (DEF_PRIORITY, gfp_mask) ;
#endif

        return nr_pages;
}
```

其中 kmem_cache_reap () 函数“收割(reap)”由 slab 机制管理的空闲页面。如果从 slap 回收的页面数已经达到要换出的页面数 nr_pages，就不用从其他地方进行换出。refill_inactive () 函数把活跃队列中的页面移到非活跃队列。shrink_cache () 函数把一个“洗净”且未加锁的页面移到非活跃队列，以便该页能被尽快释放。

此外，除了从各个进程的用户空间所映射的物理页面中回收页面外，还调用 shrink_dcache_memory ()、shrink_icache_memory () 及 shrink_dqcache_memory () 回收内核数据结构所占用的空间。在文件系统一章将会看到，在打开文件的过程中，要分配和使用代表着目录项的 dentry 数据结构，还有代表着文件索引节点 inode 的数据结构。这些数据结构在文件关闭后并不立即释放，而是放在 LRU 队列中作为后备，以防在将来的文件操作中又用到。这样经过一段时间后，就有可能积累起大量的 dentry 数据结构和 inode 数据结构，从而占用数量可观的物理页面。这时，就要通过这些函数适当加以回收。

4．页面置换

到底哪些页面会被作为后选页以备换出，这是由 swap_out () 和 shrink_cache () 一起完成的。这个过程比较复杂，这里我们抛开源代码，以理清思路为目标。

shrink_cache () 要做很多换出的准备工作。它关注两个队列：“活跃的” LRU 队列 和“非活跃的” FIFO 队列，每个队列都是 struct page 形成的链表。该函数的代码比较长，我们把它所做的工作概述如下：

- 把引用过的页面从活跃队列的队尾移到该队列的队头（实现 LRU 策略）；
- 把未引用过的页面从活跃队列的队尾移到非活跃队列的队头（为准备换出而排队）；
- 把脏页面安排在非活跃队列的队尾准备写到磁盘；
- 从非活跃队列的队尾恢复干净页面(写出的页面就成为干净的)。

### 6.6.3 交换空间的数据结构

Linux 支持多个交换文件或设备，它们将被 swapon 和 swapoff 系统调用来打开或关闭。每个交换文件或设备都可用 swap_info_struct 结构来描述：

```
 struct swap_info_struct {
        unsigned int flags;
        kdev_t swap_device;
        spinlock_t sdev_lock;
        struct dentry * swap_file;
        struct vfsmount *swap_vfsmnt;
        unsigned short * swap_map;
        unsigned int lowest_bit;
        unsigned int highest_bit;
        unsigned int cluster_next;
```

```
        unsigned int cluster_nr;
        int prio;                       /* swap priority */
        int pages;
        unsigned long max;
        int next;                       /* next entry on swap list */
};

extern int nr_swap_pages;
```

flags 域(SWP_USED 或 SWP_WRITEOK)用作控制访问交换文件。当 swapoff 被调用（为了取消一个文件）时，SWP_WRITEOK 置成 off，使在文件中无法分配空间。如果 swapon 加入一个新的交换文件时，SWP_USED 被置位。这里还有一静态变量(nr_swapfiles)来记录当前活动的交换文件数。

域 lowest_bit，highest_bit 表明在交换文件中空闲范围的边界，这是为了快速寻址。

当用户程序 mkswap 初始化交换文件或设备时，在文件的第一个页插槽的前 10 个字节，有一个包含有位图的标志，在位图里初始化为 0，代表坏的页插槽，1 代表相关页插槽是空闲的。

当用户程序调用 swapon()时，有一页被分配给 swap_map。

swap_map 为在交换文件中每一个页插槽保留了一个字节，0 代表可用页插槽，128 代表不可用页插槽。它被用于记下交换文件中每一页插槽上的 swap 请求。

内存中的一页被换出时，调用 get_swap_page（）会得到一个一个记录换出位置的索引，然后在页表项中回填（1～31 位）此索引。这是为了在发生在缺页异常时进行处理(do_no_page)。索引的高 7 位给定交换文件，后 24 位给定设备中的页插槽号。

另外函数 swap_duplicate()被 copy_page_tables()调用来实现子进程在 fork()时继承被换出的页面，这里要增加域 swap_map 中此页面的 count 值，任何进程访问此页面时，会换入它的独立的拷贝。

swap_free()减少域 swap_map 中的 count 值，如果 count 减到 0 时，则这页面又可再次分配（get_swap_page），在把一个换出页面调入(swap_in)内存时或放弃一个页面时(free_one_table） 调用 swap_free()。

相关函数在文件 filemap.c 中。

### 6.6.4　交换空间的应用

#### 1．建立交换空间

作为交换空间的交换文件实际就是通常的文件，但文件的扇区必须是连续的，即文件中必须没有“洞”，另外，交换文件必须保存在本地硬盘上。

由于内核要利用交换空间进行快速的内存页面交换，因此，它不进行任何文件扇区的检查，而认为扇区是连续的。由于这一原因，交换文件不能包含洞。可用下面的命令建立无洞的交换文件：

```
$ dd if=/dev/zero of=/extra-swap bs=1024 count=2048
2048+0 records in
```

```
2048+0 records out
```

上面的命令建立了一个名称为 extra-swap，大小为 2048KB 的交换文件。对 i386 系统而言，由于其页面尺寸为 4KB，因此最好建立一个大小为 4K 倍数的交换文件；对 Alpha AXP 系统而言，最好建立大小为 8K 倍数的交换文件。

交换分区和其他分区也没有什么不同，可像建立其他分区一样建立交换分区。但该分区不包含任何文件系统。分区类型对内核来讲并不重要，但最好设置为 Linux Swap 类型（即类型 82）。

建立交换文件或交换分区之后，需要在文件或分区的开头写入签名，写入的签名实际是由内核使用的一些管理信息。写入签名的命令为 mkswap，如下所示：

```
$ mkswap /extra-swp 2048
Setting up swapspace, size = 2088960 bytes
$
```

这时，新建立的交换空间尚未开始使用。使用 mkswap 命令时必须小心，因为该命令不会检查文件或分区内容，因此极有可能覆盖有用的信息，或破坏分区上的有效文件系统信息。

Linux 内存管理子系统将每个交换空间的大小限制在 127MB （实际为 (4096.10)*8*4096 = 133890048 Byte = 127.6875MB）。可以在系统中同时使用 16 个交换空间，从而使交换空间总量达到 2GB。

2．使用交换空间

利用 swapon 命令可将经过初始化的交换空间投入使用。如下所示：

```
$ swapon /extra-swap
$
```

如果在 /etc/fstab 文件中列出交换空间，则可自动将交换空间投入使用：

```
/dev/hda5 none swap sw 0 0
/extra-swap none swap sw 0 0
```

实际上，启动脚本会运行 swapon －a 命令，从而将所有出现在 /etc/fstab 文件中的交换空间投入使用。

利用 free 命令，可查看交换空间的使用。如下所示：

```
$ free
total used free shared buffers
Mem: 15152 14896 256 12404 2528
-/+ buffers: 12368 2784
Swap: 32452 6684 25768
$
```

该命令输出的第一行（Mem: ） 显示了系统中物理内存的使用情况。total 列显示的是系统中的物理内存总量；used 列显示正在使用的内存数量；free 列显示空闲的内存量；shared 列显示由多个进程共享的内存量，该内存量越多越好；buffers 显示了当前的缓冲区高速缓存的大小。

输出的最后一行 （Swap:） 显示了有关交换空间的类似信息。如果该行的内容均为 0，表明当前没有活动的交换空间。

利用 top 命令或查看 /proc 文件系统中的 /proc/meminfo 文件可获得相同的信息。

利用 swapoff 命令可移去使用中的交换空间。但该命令应只用于临时交换空间，否则

有可能造成系统崩溃。

swapoff –a 命令按照 /etc/fstab 文件中的内容移去所有的交换空间，但任何手工投入使用的交换空间保留不变。

3．分配交换空间

大多数人认为，交换空间的总量应该是系统物理内存量的两倍，实际上这一规则是不正确的，正确的交换空间大小应按如下规则确定。

（1）估计需要的内存总量。运行想同时运行的所有程序，并利用 free 或 ps 程序估计所需的内存总量，只需大概估计。

（2）增加一些安全性余量。

（3）减去已有的物理内存数量，然后将所得数据取整为 MB，这就是应当的交换空间大小。

（4）如果得到的交换空间大小远远大于物理内存量，则说明需要增加物理内存数量，否则系统性能会因为过分的页面交换而下降。

（5）当计算的结果说明不需要任何交换空间时，也有必要使用交换空间。Linux 从性能的角度出发，会在磁盘空闲时将某些页面交换到交换空间中，以便减少必要时的交换时间。另外，如果在不同的磁盘上建立多个交换空间，有可能提高页面交换的速度，这是因为某些硬盘驱动器可同时在不同的磁盘上进行读写操作。

## 6.7　缓存和刷新机制

### 6.7.1　Linux 使用的缓存

不管在硬件设计还是软件设计中，高速缓存是获得高性能的常用手段。Linux 使用了多种和内存管理相关的高速缓存。

1．缓冲区高速缓存

缓冲区高速缓存中包含了由块设备使用的数据缓冲区。这些缓冲区中包含了从设备中读取的数据块或写入设备的数据块。缓冲区高速缓存由设备标识号和块标号索引，因此可以快速找出数据块。如果数据能够在缓冲区高速缓存中找到，则系统就没有必要在物理块设备上进行实际的读操作。

内核为每个缓冲区维护很多信息以有助于缓和写操作，这些信息包括一个“脏（dirty）”位，表示内存中的缓冲区已被修改，必须写到磁盘；还包括一个时间标志，表示缓冲区被刷新到磁盘之前已经在内存中停留了多长时间。因为缓冲区的有关信息被保存在缓冲区首部，所以，这些数据结构连同用户数据本身的缓冲区都需要维护。

缓冲区高速缓存的大小可以变化。当需要新缓冲区而现在又没有可用的缓冲区时，就按需分配页面。当空闲内存变得不足时，例如上一节看到的情况，就释放缓冲区并反复使用相

应的页面。

2．页面高速缓存

页面高速缓存是页面 I/O 操作访问数据所使用的磁盘高速缓存。我们在文件系统会看到，read( )、write( )和 mmap( )系统调用对常规文件的访问都是通过页面高速缓存来完成的。因为页面 I/O 操作要传输整页数据，因此高速缓存中所保留的信息单元是一个整页面。一个页面包含的数据未必是物理上相邻的磁盘块，因此就不能使用设备号和块号来标识页面。相反，页面高速缓存中一个页面的标识是通过文件的索引节点和文件中的偏移量达到的。

与页面高速缓存有关的操作主要有 3 种：①当访问的文件部分不在高速缓存中时增加一页面；②当高速缓存变得太大时删除一页面；③查找一个给定文件偏移量所在的页面。

3．交换高速缓存

只有修改后的（脏）页面才保存在交换文件中。修改后的页面写入交换文件后，如果该页面再次被交换但未被修改时，就没有必要写入交换文件，相反，只需丢弃该页面。交换高速缓存实际包含了一个页面表项链表，系统的每个物理页面对应一个页面表项。对交换出的页面，该页面表项包含保存该页面的交换文件信息，以及该页面在交换文件中的位置信息。如果某个交换页面表项非零，则表明保存在交换文件中的对应物理页面没有被修改。如果这一页面在后续的操作中被修改，则处于交换缓存中的页面表项被清零。 Linux 需要从物理内存中交换出某个页面时，它首先分析交换缓存中的信息，如果缓存中包含该物理页面的一个非零页面表项，则说明该页面交换出内存后还没有被修改过，这时，系统只需丢弃该页面。

这里给出有关交换缓存的部分函数及功能：位于/ linux/mm/swap_state.c 中。

初始化交换缓冲，设定大小，位置的函数：

```
extern unsigned long init_swap_cache (unsigned long, unsigned long) ;
```

显示交换缓冲信息的函数：

```
extern void show_swap_cache_info (void) ;
```

加入交换缓冲的函数：

```
int add_to_swap_cache (unsigned long index, unsigned long entry)
```

参数 index 是进入缓冲区的索引（index 是索引表中的某一项），entry 是‘页面表项’(即此页面在交换文件中的位置记录，这个记录类似页面表项，参见交换机制)

复制被换出的页面：

```
extern void swap_duplicate (unsigned long) ;
```

当使用 copy_page_tables()调用，来实现子进程在 fork()时继承被换出的页面，可参阅交换机制一节。

从缓冲区中移去某页面

```
delete_from_swap_cache (page_nr) ;
```

硬件高速缓存：

常见的硬件缓存是对页面表项的缓存，这一工作实际由处理器完成，其操作和具体的处理器硬件有关（但管理要由软件完成），对这一缓存接下来要做进一步描述。

### 6.7.2　缓冲区高速缓存

Linux 采用了缓冲区高速缓存机制，而不同于其他操作系统的“写透”方式，也就是说，当把一个数据写入文件时，内核将把数据写入内存缓冲区，而不是直接写入磁盘。

在这里要用到一个数据结构 buffer_head，它是用来描述缓冲区的数据结构，缓冲区的大小一般要比页面尺寸小，所以一页面中可以包含数个缓冲区，同一页面中的缓冲区用链表连接。回忆一下页面结构 page，其中有一个域 buffer_head buffer 就是用来指向缓冲区的，这个结构的详细内容请参见虚拟文件系统。

由于使用了缓冲技术，因此有可能出现这种情况：写磁盘的命令已经返回，但实际的写入磁盘的操作还未执行。

基于上述原因，应当使用正常的关机命令关机，而不应直接关掉计算机的电源。用户也可以使用 sync 命令刷新缓冲区高速缓存，从而把缓冲区中的数据强制写到磁盘中。在 Linux 系统中，除了传统的 update 守护进程之外，还有一个额外的守护进程 dbflush，这一进程可频繁运行不完整的 sync 从而可避免有时由于 sync 命令的超负荷磁盘操作而造成的磁盘冻结，一般情况下，它们在系统引导时自动执行，且每隔 30s 执行一次任务。

sync 命令使用基本的系统调用 sync()来实现。

dbflush 在 Linux 系统中由 update 启动。如果由于某种原因该进程僵死了，则内核会发送警告信息，这时需要手工启动该进程（/sbin/update）。

1．页面缓存的详细描述

经内存映射的文件每次只读取一页面内容，读取后的页面保存在页面缓存中，利用页面缓存，可提高文件的访问速度。如图 6.20 所示，页面缓存由 page_hash_table 组成，它是一个 mem_map_t(即 struct page 数据结构)的指针向量。页面缓存的结构是 Linux 内核中典型的哈希表结构。众所周知，对计算机内存的线性数组的访问是最快速的访问方法，因为线性数组中的每一个元素的位置都可以利用索引值直接计算得到，而这种计算是简单的线性计算。但是，如果要处理大量数据，有时由于受到存储空间的限制，采用线性结构是不切合实际的。但如果采用链表等非线性结构，则元素的检索性能又会大打折扣。哈希表则是一种折衷的方法，它综合了线性结构和非线性结构的优点，可以在大量数据中进行快速的查找。哈希表的结构有多种，在 Linux 内核中，常见的哈希结构和图 6.20 的结构类似。要在这种哈希表中访问某个数据，首先要利用哈希函数以目标元素的某个特征值作为函数自变量生成哈希值作为索引，然后利用该索引访问哈希表的线性指针向量。哈希线性表中的指针代表一个链表，该链表所包含的所有节点均具有相同的哈希值，在该链表中查找可访问到指定的数据。哈希函数的选择非常重要，不恰当的哈希函数可能导致大量数据映射到同一哈希值，这种情况下，元素的查找将相当耗时。但是，如果选择恰当的哈希函数，则可以在性能和空间上得到均衡效果。

在 Linux 页面缓存中，访问 page_hash_table 的索引由文件的 VFS（虚拟文件系统）索引节点 inode 和内存页面在文件中的偏移量生成。有关 VFS 索引节点的内容将在虚拟文件中讲到，在这里，应知道每个文件的 VFS 索引节点 inode 是唯一的。

当系统要从内存映射文件中读取某一未加锁的页面时，就首先要用到函数：
find_page （struct inode * inode, unsigned long offset）。
它完成如下工作。

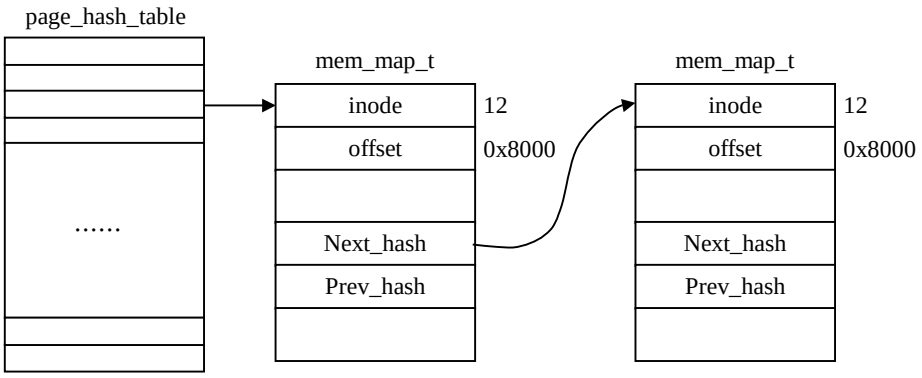


图 6.20 Linux 页面缓存示意图

首先是在“页面缓存”中查找，如果发现该页面保存在缓存中，则可以免除实际的文件读取，而只需从页面缓存中读取，这时，指向 mm_map_t 数据结构的指针被返回到页面故障的处理代码。部分代码如下：

```
for （page = page_hash（inode, offset）; page ; page = page->next_hash)

/*函数 page_hash（）是从哈希表中找页面*/
{if （page->inode != inode)
      continue;
if （page->offset != offset)
     continue;
     /* 找到了特定页面 */
     atomic_inc（&page->count）;
     set_bit（PG_referenced, &page->flags）;/*设访问位*/
     break;    }
     return page;
}
```

如果该页面不在缓存中，则必须从实际的文件系统映像中读取页面，这时 Linux 内核首先分配物理页面然后从磁盘读取页面内容。

如果可能，Linux 还会预先读取文件中下一页面内容到页面缓存中，而不等页面错误发生才去“请页面”，这样做是为了提高装入代码的速度(有关代码在 filemap.c 中，如 generic_file_readahead（）等函数)。这样，如果进程要连续访问页面，则下一页面的内容不必再次从文件中读取了，而只需从页面缓存中读取。

随着映像的读取和执行，页面缓存中的内容可能会增多，这时，Linux 可移走不再需要的页面。当系统中可用的物理内存量变小时，Linux 也会通过缩小页面缓存的大小而释放更多的物理内存页面。

2．有关页面缓存的函数

先看把读入的页面如何存于缓存，这要用到函数 add_to_page_cache()，它完成把指定的“文件页面”记入页面缓存中。

```
static inline void add_to_page_cache (struct page * page,
     struct inode * inode, unsigned long offset)
{    /*设置有关页面域，引用数，页面使用方式，页面在文件中的偏移 */
    page->count++;
     page->flags &= ~ ( (1 << PG_uptodate)  |  (1 << PG_error) ) ;
     page->offset = offset;
     add_page_to_inode_queue (inode, page) ;/* 把页面加入 inode 节点队列*/
     add_page_to_hash_queue (inode, page) ;/* 把页面加入哈唏表 page_hash_table[]*/}
```

注意：inode 的部分请看虚拟文件章节。

哈希表 page_hash_table[]的定义：

```
extern struct page * page_hash_table[PAGE_HASH_SIZE];
```

下面是有关对哈希表操作的部分代码：

```
static inline void add_page_to_inode_queue (struct inode * inode, struct page * page)
{    struct page **p = &inode->i_pages;/*指向物理页面*/
     inode->i_nrpages++;/*节点中调入内存的页面数目增 1*/
     page->inode = inode; /*指向该页面来自的文件节点结构，相互连成链*/
     page->prev = NULL;
if  ( (page->next = *p)  != NULL)
page->next->prev = page;
     *p = page;  }
```

把页面加入哈希表：

```
static inline void add_page_to_hash_queue (struct inode * inode, struct page * page)
{    struct page **p = &page_hash (inode,page->offset) ;
     page_cache_size++;/*哈希表中记录的页面数目加 1*/
     set_bit (PG_referenced, &page->flags) ;/*设置访问位*/
page->age = PAGE_AGE_VALUE;/*设缓存中的页面“年龄”为定值，为淘汰做准
备*/
     page->prev_hash = NULL;
     if  ( (page->next_hash = *p)  != NULL)
     page->next_hash->prev_hash = page;    *p = page;
}
```

有关页面的刷新函数：

```
remove_page_from_hash_queue (page) ;   /*从哈希表中去掉页面*/
remove_page_from_inode_queue (page) ;  /*从 inode 节点中去掉页面*/
```

### 6.7.3　翻译后援存储器(TLB)

页表的实现对虚拟内存系统效率是极为关键的。例如把一个寄存器的内容复制到另一个寄存器中的一条指令，在不使用分页时，只需访问内存一次取指令，而在使用分页时需要额外的内存访问去读取页表。而系统的运行速度一般是被 cpu 从内存中取得指令和数据的速率限制的，如果在每次访问内存时都要访问两次内存会使系统性能降低三分之二。

对这个问题的解决，有人提出了一个解决方案，这个方案基于这样的观察：大部分程序

倾向于对较少的页面进行大量的访问。因此，只有一小部分页表项经常被用到，其他的很少被使用。

采取的解决办法是为计算机装备一个不需要经过页表就能把虚拟地址映射成物理地址的小的硬件设备，这个设备叫做 TLB(翻译后援存储器，Translation Lookside Buffer)，有时也叫做相联存储器(Associative Memory)，如图 6.21 所示 。它通常在 MMU 内部，条目的数量较少，在这个例子中是 6 个，80386 有 32 个。

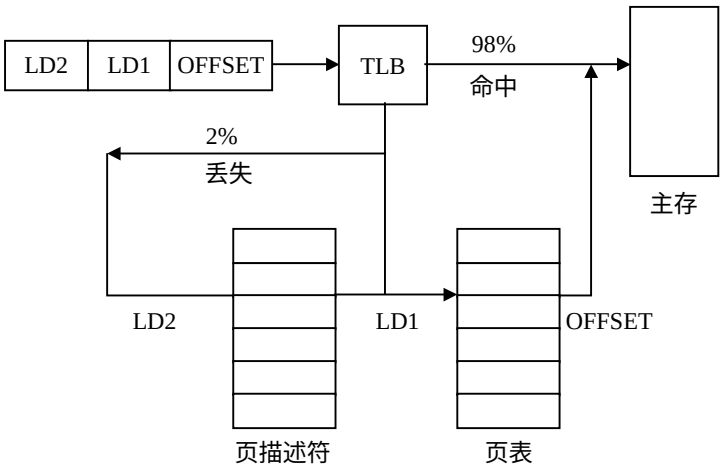


图 6.21　翻译后援存储器

每一个 TLB 寄存器的每个条目包含一个页面的信息：有效位、虚页面号、修改位、保护码和页面所在的物理页面号，它们和页面表中的表项一一对应，如图 6.22 所示。

| 段号 | 虚页面号 | 页面框 | 保护 | 年龄 | 有效位 |
|---|---|---|---|---|---|
| 4 | 1 | 7 | RW | 5 | 1 |
| 8 | 7 | 16 | RW | 1 | 1 |
| 2 | 0 | 33 | RX | 4 | 1 |
| 4 | 4 | 72 | RX | 13 | 0 |
| 5 | 8 | 17 | RW | 2 | 1 |
| 2 | 7 | 34 | RX | 2 | 1 |

图 6.22　用于加速分页面操作的 TLB

当一个虚地址被送到 MMU 翻译时，硬件首先把它和 TLB 中的所有条目同时(并行地)进行比较。如果它的虚页面号在 TLB 中，并且访问没有违反保护位，它的页面会直接从 TLB 中取出而不去访问页表；如果虚页面号在 TLB 中，但当前指令试图写一个只读的页面，这时将产

生一个缺页异常，与直接访问页表时相同。

如 MMU 发现在 TLB 中没有命中，它将随即进行一次常规的页表查找，然后从 TLB 中淘汰一个条目并把它替换为刚刚找到的页表项。因此如果这个页面很快再被用到的话，第 2 次访问时它就能在 TLB 中直接找到。在一个 TLB 条目被淘汰时，被修改的位被复制回在内存中的页表项，其他的值则已经在那里了。当 TLB 从页表装入时，所有的域都从内存中取得。

必须明确在分页机制中，TLB 中的数据和页表中的数据的相关性，不是由处理器进行维护，而是必须由操作系统来维护，高速缓存的刷新是通过装入处理器（80386）中的寄存器 CR3 来完成的（见刷新机制 flush_tlb()）。

这里提到的命中率，指一个页面在 TBL 中找到的概率。一般来说 TLB 的尺寸大可增加命中率，但会增加成本和软件的管理。所以一般都采用 8～64 个条目的数量。

假如命中率是 0.85，访问内存时间是 120 纳秒，访 TLB 时间是 15 纳秒。那么访问时间是：0.85×(15+120)+(1-0.85)×(15+120+120)=153 纳秒。

### 6.7.4　刷新机制

1．软件管理 TLB

前面我们介绍的 TLB 管理和 TLB 故障的处理都完全由 MMU 硬件完成，只有一个页面不在内存时才会陷入操作系统。

而实际上，在现代的一些 RISC 机中，包括 MIPS、Alpha、HP PA，几乎全部的这种页面管理工作都是由软件完成的。在这些机器中，TLB 条目是由操作系统显式地装入，在 TLB 没有命中时，MMU 不是到页表中找到并装入需要的页面信息，而是产生一个 TLB 故障把问题交给操作系统。操作系统必须找到页面，从 TLB 中淘汰一个条目，装入一个新的条目，然后重新启动产生异常（或故障）的指令。当然，所有这些都必须用很少指令完成，因为 TLB 不命中的频率远比页面异常大得多。

令人惊奇的是，如果 TLB 的尺寸取一个合理的较大值(比如 64 个条目)以减少不命中的频率，那么软件管理的 TLB 效率可能相当高。这里主要的收益是一个简单得多的 MMU（最后介绍），它在 CPU 芯片上为高速缓存和其他能提高性能的部件让出了相当大的面积。

人们已经使用了很多方法来提高使用软件管理 TLB 机器的性能,有一个方法既能减少 TLB 的不命中率又能减少在 TLB 不命中确实发生时的开销。为了减少 TLB 的不命中率，操作系统有时可以用它的直觉来指出那些页面可能将被使用并把他们预装入 TLB 中。例如，当一个客户进程向位于同一台机器的服务器进程发出一个 RPC 请求时，服务器很可能即将运行。知道了这一点，在客户进程因执行 RPC 陷入时，系统就可以找到服务器的代码、数据、堆栈的页面，并在 TLB 中提前为他们建立映射，以避免 TLB 故障的发生。

无论是硬件还是软件，处理 TLB 不命中的一般方法是对页表执行索引操作找出所引用的页面。用软件执行这个搜索的一个问题是保存页表的页面本身可能就不在 TLB 中，这将在处理过程中再一次引发一个 TLB 异常,这种异常可以通过保持一个大的(比如 4KB)TLB 条目的软件高速缓存而得到减少，这个高速缓存保持在固定位置，它的页面总是保持在 TLB 中，操作系统通过首先检查软件高速缓存可以大大减少 TLB 不命中的次数。

2．刷新机制

用软件来管理 TLB 和其他缓存的一个重要的要求就是保持 TLB 和其他缓存中的内容的同步性，这样必须考虑在一定条件下刷新内容。

在 Linux 中刷新机制(包括 TLB 的刷新，缓存的刷新等等）主要要用来完成以下几个工作：

(1) 保证在任何时刻内存管理硬件所看到的进程的内核映射和内核页表一致；

(2) 如果负责内存管理的内核代码对用户进程页面进行了修改，那么用户的进程在被允许继续执行前,要求必须在缓存中看到正确的数据。

例如当正在执行 write() 系统调用时，要保证页面缓存中的页面为新页，也就是要使缓存中的页面内容和写入文件的一致，就需要更新缓存中的页面。

3．通常当地址空间的状态改变时，调用适当的刷新机制来描述状态的改变

在 Linux 中刷新机制的实现是通过一系列函数（或宏）来完成的,例如常用的两个刷新函数的一般形式为：

```
flush_cache_foo ( ) ;
flush_tlb_foo ( ) ;
```

这两个函数的调用是有一定顺序的，它们的逻辑意义如下所述。

在地址空间改变前必须刷新缓存，防止缓存中存在非法的空映射。函数 flush_cache_*() 会把缓存中的映射变成无效（ 这里的缓存指的是 MMU 中的缓存，它负责虚地址到物理地址的当前映射关系;注意在这里由于各种处理器中 MMU 的内部结构不同,换存刷新函数也不尽相同。比如在 80386 处理器中这些函数是为空——i386 处理器刷新时不需要任何多余的 MMU 的信息，内核页表包含了所有的必要信息）。在刷新地址后，由于页表的改变，必须刷新 TBL 以便硬件可以把新的页表信息装入 TLB。

下面介绍一些刷新函数的作用和使用情况：

```
void flush_cache_all (void) ;
void flush_tlb_all (void) ;
```

这两个例程是用来通知相应机制，内核地址空间的映射已被改变，它意味着所有的进程都被改变了；

```
void flush_cache_mm (struct mm_struct *mm) ;
void flush_tlb_mm (struct mm_struct *mm) ;
```

它们用来通知系统被 mm_struct 结构所描述的地址空间正在改变，它们仅发生在用户空间的地址改变时；

```
flush_cache_range (struct mm_struct *mm,unsigned long start, unsigned long end) ;
flush_tlb_range (struct mm_struct *mm,unsigned long start, unsigned long end) ;
```

它们刷新用户空间中的指定范围；

```
void flush_cache_page (struct vm_area_struct *vma,unsigned long address) ;
void flush_tlb_page (struct vm_area_struct *vma,unsigned long address) ;
```

刷新一页面。

```
void flush_page_to_ram(unsigned long page);/*如果使用 i386 处理器，此函数为空，相应的刷新功能由硬件内部自动完成*/
```

这个函数一般用在写时复制，它会使虚拟缓存中的对应项无效，这是因为如果虚拟缓存

不可以自动地回写，于是会造成虚拟缓存中页面和主存中的内容不一致。

例如,如图 6.23 所示，虚拟内存 0x2000 对任务 1、任务 2、任务 3 共享，但对任务 2 只是可读，它映射物理内存 0x1000,那么如果任务 2 要对虚拟内存 0x2000 执行写操作时，会产生页面错误。内存管理系统要给它重新分配一个物理页面如 0x2600,此页面的内容是物理内存 0x1000 的拷贝，这时虚拟索引缓存中就有两项内核别名项 0x2000 分别对应两个物理地址 0x1000 和 0x2600，在任务 2 对物理页面 0x2600 的内容进行了修改后，这样内核别名即虚地址 0x2000 映射的物理页面内容不一致，任务 3 在来访问虚地址 0x2000 时就会产生不一致错误。为了避免不一致错误，使用 flush_page_to_ram 使得缓存中的内核别名无效。

一般刷新函数的使用顺序如下：

```
copy_cow_page (old_page,new_page,address) ;
flush_page_to_ram (old_page) ;
flush_page_to_ram (new_page) ;
flush_cache_page (vam,address) ;
    ….
free_page (old_page) ;
flush_tlb_page (vma,address) ;
```

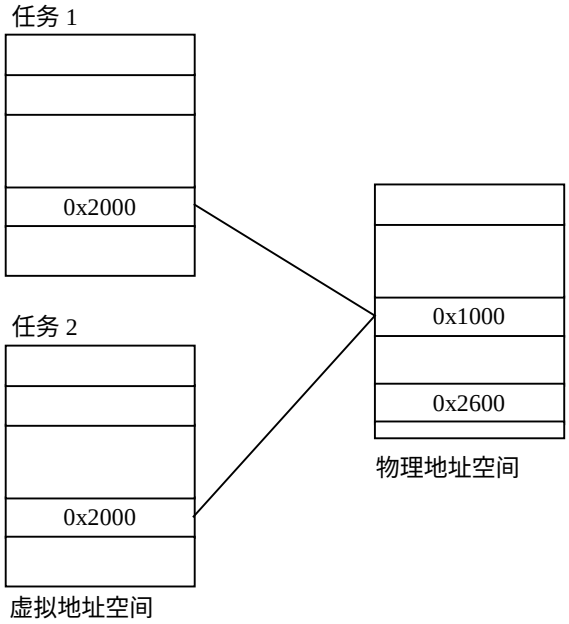


图 6.23　多个任务共享内存空间

4．函数代码简介

大部分刷新函数都在 include/asm/pttable.h 中定义，这里就 i386 中__flush_tlb()的定义给予说明：

```
#define __flush_tlb ()
     do {
            unsigned int tmpreg;
```

```
                __asm__ __volatile__ (
                        "movl %%cr3, %0;  # flush TLB \n"
                        "movl %0, %%cr3;              \n"
                        : "=r"  (tmpreg)
                         :: "memory") ;
        } while  (0)
```

这个函数比较简单，通过对 CR3 寄存的重新装入，完成对 TLB 的刷新。

# 6.8 进程的创建和执行

## 6.8.1 进程的创建

新的进程通过克隆旧的程序（当前进程）而建立。fork() 和 clone()（对于线程）系统调用可用来建立新的进程。这两个系统调用结束时，内核在系统的物理内存中为新的进程分配新的 task_struct 结构，同时为新进程要使用的堆栈分配物理页。Linux 还会为新的进程分配新的进程标识符。然后，新 task_struct 结构的地址保存在链表中，而旧进程的 task_struct 结构内容被复制到新进程的 task_struct 结构中。

在克隆进程时，Linux 允许两个进程共享相同的资源。可共享的资源包括文件、信号处理程序和虚拟内存等（通过继承）。当某个资源被共享时，该资源的引用计数值会增加 1，从而只有两个进程均终止时，内核才会释放这些资源。图 6.24 说明了父进程和子进程共享打开的文件。

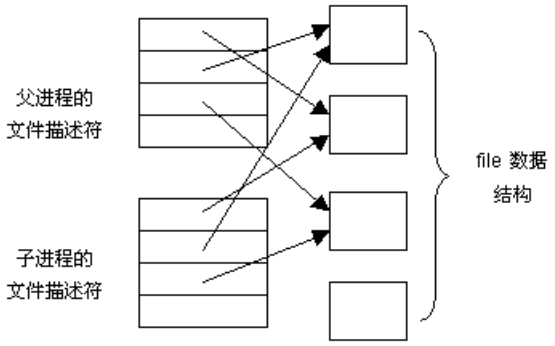


图 6.24 父进程和子进程共享打开的文件

系统对进程虚拟内存的克隆过程则更加巧妙些。新的 vm_area_struct 结构、新进程自己的 mm_struct 结构以及新进程的页表必须在一开始就准备好，但这时并不复制任何虚拟内存。如果旧进程的某些虚拟内存在物理内存中，而有些在交换文件中，那么虚拟内存的复制将会非常困难和费时。实际上，Linux 采用了称为写时复制的技术，也就是说，只有当两个进程中的任意一个向虚拟内存中写入数据时才复制相应的虚拟内存；而没有写入的任何内存

页均可以在两个进程之间共享。代码页实际总是可以共享的。

此外,内核线程是调用 kernel_thread()函数创建的,而 kernel_thread()在内核态调用了 clone()系统调用。内核线程通常没有用户地址空间，即 p->mm = NULL，它总是直接访问内核地址空间。

不管是 fork()还是 clone（）系统调用，最终都调用了内核中的 do_fork()，其源代码在 kernel/fork.c：

```
/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information  (task[nr])  and sets up the necessary registers. It also
 * copies the data segment in its entirety.  The "stack_start" and
 * "stack_top" arguments are simply passed along to the platform
 * specific copy_thread ()  routine.  Most platforms ignore stack_top.
 * For an example that's using stack_top, see
 * arch/ia64/kernel/process.c.
 */
int do_fork (unsigned long clone_flags, unsigned long stack_start,
            struct pt_regs *regs, unsigned long stack_size)
{
         int retval;
         struct task_struct *p;
         struct completion vfork;

        retval = -EPERM;

         /*
          * CLONE_PID is only allowed for the initial SMP swapper
          * calls
          */
        if  (clone_flags & CLONE_PID)  {
                if  (current->pid)
                       goto fork_out;
         }

         retval = -ENOMEM;
        p = alloc_task_struct () ;
        if  (!p)
                goto fork_out;

        *p = *current;

         retval = -EAGAIN;
         /*
          * Check if we are over our maximum process limit, but be sure to
          * exclude root. This is needed to make it possible for login and
          * friends to set the per-user process limit to something lower
          * than the amount of processes root is running. -- Rik
          */
    if  (atomic_read (&p->user->processes)  >= p->rlim[RLIMIT_NPROC].rlim_cur
                  && !capable (CAP_SYS_ADMIN)  !capable (CAP_SYS_RESOURCE) )
                goto bad_fork_free;
```

```
        atomic_inc (&p->user->__count) ;
         atomic_inc (&p->user->processes) ;

        /*
          * Counter increases are protected by
          * the kernel lock so nr_threads can't
         * increase under us  (but it may decrease) .
         */
         if  (nr_threads >= max_threads)
                 goto bad_fork_cleanup_count;

        get_exec_domain (p->exec_domain) ;

         if  (p->binfmt && p->binfmt->module)
                __MOD_INC_USE_COUNT (p->binfmt->module) ;

        p->did_exec = 0;
        p->swappable = 0;
        p->state = TASK_UNINTERRUPTIBLE;

         copy_flags (clone_flags, p) ;
        p->pid = get_pid (clone_flags) ;

        p->run_list.next = NULL;
         p->run_list.prev = NULL;

        p->p_cptr = NULL;
        init_waitqueue_head (&p->wait_chldexit) ;
         p->vfork_done = NULL;
         if  (clone_flags & CLONE_VFORK)  {
                 p->vfork_done = &vfork;
                 init_completion (&vfork) ;
         }
         spin_lock_init (&p->alloc_lock) ;

         p->sigpending = 0;
        init_sigpending (&p->pending) ;

        p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
         p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
         init_timer (&p->real_timer) ;
        p->real_timer.data =  (unsigned long)  p;

        p->leader = 0;          /* session leadership doesn't inherit */
         p->tty_old_pgrp = 0;
         p->times.tms_utime = p->times.tms_stime = 0;
          p->times.tms_cutime = p->times.tms_cstime = 0;
    #ifdef CONFIG_SMP
         {
                 int i;
                 p->cpus_runnable = ~0UL;
```

```
                p->processor = current->processor;
                 /* ?? should we just memset this ?? */
                for (i = 0; i < smp_num_cpus; i++)
                        p->per_cpu_utime[i] = p->per_cpu_stime[i] = 0;
                  spin_lock_init (&p->sigmask_lock) ;
        }
#endif
        p->lock_depth = -1;             /* -1 = no lock */
        p->start_time = jiffies;

         INIT_LIST_HEAD (&p->local_pages) ;

         retval = -ENOMEM;
         /* copy all the process information */
         if  (copy_files (clone_flags, p) )
                goto bad_fork_cleanup;
         if  (copy_fs (clone_flags, p) )
                 goto bad_fork_cleanup_files;
         if  (copy_sighand (clone_flags, p) )
                 goto bad_fork_cleanup_fs;
         if  (copy_mm (clone_flags, p) )
                 goto bad_fork_cleanup_sighand;
        retval = copy_thread (0, clone_flags, stack_start, stack_size, p, regs) ;
        if  (retval)
                 goto bad_fork_cleanup_mm;
        p->semundo = NULL;

        /* Our parent execution domain becomes current domain
            These must match for thread signalling to apply */

         p->parent_exec_id = p->self_exec_id;

         /* ok, now we should be set up.. */
        p->swappable = 1;
        p->exit_signal = clone_flags & CSIGNAL;
         p->pdeath_signal = 0;

        /*
          * "share" dynamic priority between parent and child, thus the
          * total amount of dynamic priorities in the system doesnt change,
          * more scheduling fairness. This is only important in the first
          * timeslice, on the long run the scheduling behaviour is unchanged.
         */
        p->counter =  (current->counter + 1)  >> 1;
        current->counter >>= 1;
         if  (!current->counter)
                 current->need_resched = 1;

        /*
         * Ok, add it to the run-queues and make it
         * visible to the rest of the system.
          *
```

```
         * Let it rip!
         */
         retval = p->pid;
        p->tgid = retval;
        INIT_LIST_HEAD (&p->thread_group) ;

        /* Need tasklist lock for parent etc handling! */
         write_lock_irq (&tasklist_lock) ;

        /* CLONE_PARENT and CLONE_THREAD re-use the old parent */
        p->p_opptr = current->p_opptr;
         p->p_pptr = current->p_pptr;
         if  (! (clone_flags &  (CLONE_PARENT | CLONE_THREAD) ) )  {
                p->p_opptr = current;
                 if  (! (p->ptrace & PT_PTRACED) )
                        p->p_pptr = current;
         }

        if  (clone_flags & CLONE_THREAD)  {
                 p->tgid = current->tgid;
                 list_add (&p->thread_group, &current->thread_group) ;
        }

         SET_LINKS (p) ;
         hash_pid (p) ;
        nr_threads++;
         write_unlock_irq (&tasklist_lock) ;

        if  (p->ptrace & PT_PTRACED)
                 send_sig (SIGSTOP, p, 1) ;

         wake_up_process (p) ;             /* do this last */
        ++total_forks;
         if  (clone_flags & CLONE_VFORK)
                 wait_for_completion (&vfork) ;

fork_out:
        return retval;

bad_fork_cleanup_mm:
        exit_mm (p) ;
bad_fork_cleanup_sighand:
        exit_sighand (p) ;
bad_fork_cleanup_fs:
        exit_fs (p) ; /* blocking */
bad_fork_cleanup_files:
        exit_files (p) ; /* blocking */
bad_fork_cleanup:
        put_exec_domain (p->exec_domain) ;
        if  (p->binfmt && p->binfmt->module)
                 __MOD_DEC_USE_COUNT (p->binfmt->module) ;
bad_fork_cleanup_count:
```

```
            atomic_dec (&p->user->processes) ;
            free_uid (p->user) ;
    bad_fork_free:
            free_task_struct (p) ;
          goto fork_out;
    }
```

尽管 fork () 系统调用因为传递用户堆栈和寄存器参数而与特定的平台相关，但实际上 do_fork()所做的工作还是可移植的。下面给出对以上代码的解释。

给局部变量赋初值-ENOMEM,当分配一个新的 task_struc 结构失败时就返回这个错误值。

如果在 clone_flags 中设置了 CLONE_PID 标志，就返回一个错误（-EPERM）。因为 CLONE_PID 有特殊的作用，当这个标志为 1 时，父、子进程（线程）共用一个进程号，也就是说，子进程虽然有自己的 task_struct 结构，却使用父进程的 pid。但是，只有 0 号进程（即系统中的空线程）才允许使用这个标志。

调用 alloc_task_struct()为子进程分配两个连续的物理页面，低端用来存放子进程的 task_struct 结构，高端用作其内核空间的堆栈。

用结构赋值语句*p = *current 把当前进程 task_struct 结构中的所有内容都拷贝到新进程中。稍后，子进程不该继承的域会被设置成正确的值。

在 task_struct 结构中有个指针 user，用来指向一个 user_struct 结构。一个用户常常有多个进程，所以有关用户的信息并不专属于某一个进程。这样，属于同一用户的进程就可以通过指针 user 共享这些信息。显然，每个用户有且只有一个 user_struct 结构。该结构中有一个引用计数器 count，对属于该用户的进程数量进行计数。可想而知，内核线程并不属于某个用户，所以其 task_struct 中的 user 指针为 0。每个进程 task_struct 结构中有个数组 rlim，对该进程占用各种资源的数量作出限制，而 rlim[RLIMIT_NPROC]就规定了该进程所属用户可以拥有的进程数量。所以，如果当前进程是一个用户进程，并且该用户拥有的进程数量已经达到了规定的界限值，就不允许它 fork()了。

除了检查每个用户拥有的进程数量外，接着要检查系统中的任务总数（所有用户的进程数加系统的内核线程数）是否超过了最大值 max_threads，如果是，也不允许再创建子进程。

一个进程除了属于某个用户外，还属于某个“执行域”。Linux 可以运行 X86 平台上其他 UNIX 类操作系统生成的符合 iBCS2 标准的程序。例如,一个进程所执行的程序是为 Solaris 开发的，那么这个进程就属于 Solaris 执行域 PER_SOLARIS。当然，在 Linux 上运行的绝大多数程序属于 Linux 执行域。在 task_struct 有一个指针 exec_doman，指向一个 exec_doman 结构。在 exec_doman 结构中有一个域是 module，这是指向某个 module 结构的指针。在 Linux 中，一个文件系统或驱动程序都可以作为一个单独的模块进行编译，并动态地链接到内核中。在 module 结构中有一个计数器 count，用来统计几个进程需要使用这个模块。因此，get_exec_domain(p->exec_domain)递增模块结构 module 中的计数器。

另外，每个进程所执行的程序属于某种可执行映像格式，如 a.out 格式、elf 格式，甚至 Java 虚拟机格式。对于不同格式的支持通常是通过动态安装的模块来实现的。所以，task_struct 中有一个执行 linux_binfmt 结构的指针 binfmt，而__MOD_INC_USE_COUNT () 就是对有关模块的使用计数进行递增。

紧接着为什么要把进程的状态设置成为 TASK_UNINTERRUPTIBLE？这是因为后面

get_pid()的操作必须独占，子进程可能因为一时进不了临界区而只好暂时进入睡眠状态。

copy_flags()函数将clone_flags参数中的标志位略加补充和变换,然后写入p->flags。

get_pid () 函数根据 clone_flags 中标志位 ClONE_PID 的值，或返回父进程(当前进程)的 pid，或返回一个新的 pid。

前面在复制父进程的 task_struct 结构时把父进程的所有域都照抄过来，但实际上很多域的值必须重新赋初值，因此，后面的赋值语句就是对子进程 task_struct 结构的初始化。其中 start_time 表示进程创建的时间,而全局变量 jiffies 就是从系统初始化开始至当前的是时钟滴答数。local_pages 表示属于该进程的局部页面形成一个双向链表，在此进行了初始化。

copy_files()有条件地复制已打开文件的控制结构，也就是说，这种复制只有在 clone_flags 中的 CLONE_FILES 标志为 0 时才真正进行，否则只是共享父进程的已打开文件。当一个进程有已打开文件时，task_struct 结构中的指针 files 指向一个 fiel_struct 结构，否则为 0。所有与终端设备 tty 相联系的用户进程的头 3 个标准文件 stdin、stdout 及 stderr 都是预先打开的，所以指针一般不为空。

copy_fs()也是只有在 clone_flags 中的 CLONE_FS 标志为 0 时才加以复制。在 task_struct 中有一个指向 fs_struct 结构的指针,fs_struct 结构中存放的是进程的根目录 root、当前工作目录 pwd、一个用于文件操作权限的 umask，还有一个计数器。类似地，copy_sighand () 也是只有在 CLONE_SIGHAND 为 0 时才真正复制父进程的信号结构，否则就共享父进程。信号是进程间通信的一种手段，信号随时都可以发向一个进程，就像中断随时都可以发向一个处理器一样。进程可以为各种信号设置相应的信号处理程序，一旦进程设置了信号处理程序，其 task_struct 结构中的指针 sig 就指向 signal_struct 结构（定义于 include/linux/sched.h）。关于信号的具体内容将在下一章进行介绍。

用户空间的继承是通过 copy_mm()函数完成的。进程的 task_struct 结构中有一个指针 mm,就指向代表着进程地址空间的 mm_struct 结构。对 mm_struct 的复制也是在 clone_flags 中的 CLONE_VM 标志为 0 时才真正进行，否则，就只是通过已经复制的指针共享父进程的用户空间。对 mm_struct 的复制不只限于这个数据结构本身，还包括了对更深层次数据结构的复制，其中最主要的是 vm_area_struct 结构和页表的复制，这是由同一文件中的 dum_mmap()函数完成的。

到此为止，task_struct 结构中的域基本复制好了，但是用于内核堆栈的内容还没有复制，这就是 copy_thread （） 的责任了。copy_thread （） 函数与平台相关，定义于 arch/i386/kernel/process.c 中。copy_thread （） 实际上只复制父进程的内核空间堆栈。堆栈中的内容记录了父进程通过系统调用 fork()进入内核空间、然后又进入 copy_thread () 函数的整个历程，子进程将要循相同的路线返回，所以要把它复制给子进程。但是，如果父子进程的内核空间堆栈完全相同，那返回用户空间后就无法区分哪个是子进程了，所以，复制以后还要略作调整。有兴趣的读者可以结合第三、四章内容去读该函数的源代码。

parent_exec_id 表示父进程的执行域，p->self_exec_id 是本进程（子进程）的执行域，swappable 表示本进程的页面可以被换出。exit_signal 为本进程执行 exit()系统调用时向父进程发出的信号，pdeath_signal 为要求父进程在执行 exit （） 时向本进程发出的信号。另外，counter 域的值是进程的时间片（以时钟滴达为单位），代码中将父进程的时间片分

成两半，让父、子进程各有原值的一半。

进程创建后必须处于某一组中，这是通过 task_struct 结构中的队列头 thread_group 与父进程链接起来，形成一个进程组（注意，thread 并不单指线程，内核代码中经常用 thread 通指所有的进程）。

建立进程的家族关系。先建立起子进程的祖先和双亲（当然还没有兄弟和孩子），然后通过 SET_LINKS（）宏将子进程的 task_struct 结构插入到内核中其他进程组成的双向链表中。通过 hash_pid()将其链入按其 pid 计算得的哈希表中（参看第四章进程组织方式一节）。

最后，通过 wake_up_process()将子进程唤醒，也就是将其挂入可执行队列等待被调度。

但是，还有一种特殊情况必须考虑。当参数 clone_flags 中 CLONE_VFORK 标志位为 1 时，一定要保证子进程先运行，一直到子进程通过系统调用 execve()执行一个新的可执行程序或通过系统调用 exit()退出系统时，才可以恢复父进程的执行，这是通过 wait_for_completion（）函数实现的。为什么要这样做呢？这是因为当 CLONE_VFORK 标志位为 1 时，就说明父、子进程通过指针共享用户空间（指向相同的 mm_struct 结构），那也说明父进程写入用户空间的内容同时也写入了子进程的用户空间，反之亦然。如果说，在这种情况下，父子进程对数据区的写入可能引起问题的话，那么，对堆栈区的写入可能就是致命的了。而对子程序或函数的调用肯定就是对堆栈的写入。由此可见，在这种情况下，决不能让两个进程都回到用户空间并发执行，否则，必然导致两个进程的互相“捣乱”或因非法访问而死亡。解决的办法的只能是“扣留”其中的一个进程，而让另一个进程先回到用户空间，直到两个进程不再共享它们的用户空间，或其中一个进程消亡为止（肯定是先回到用户空间的进程先消亡）。

到此为止，子进程的创建已经完成，该是从内核态返回用户态的时候了。实际上，fork()系统调用执行之后，父子进程返回到用户空间中相同的的地址，用户进程根据 fork()的返回值分别安排父子进程执行不同的代码。

### 6.8.2　程序执行

与 UNIX 类似，Linux 中的程序和命令通常由命令解释器执行，这一命令解释器称为 shell。用户输入命令之后，shell 会在搜索路径（shell 变量 PATH 中包含搜索路径）指定的目录中搜索和输入命令匹配的映像（可执行的二进制代码）名称。如果发现匹配的映像，shell 负责装载并执行该映像。shell 首先利用 fork 系统调用建立子进程，然后用找到的可执行映像文件覆盖子进程正在执行的 shell 二进制映像。

可执行文件可以是具有不同格式的二进制文件，也可以是一个文本的脚本文件。可执行映像文件中包含了可执行代码及数据，同时也包含操作系统用来将映像正确装入内存并执行的信息。Linux 使用的最常见的可执行文件格式是 ELF 和 a.out，但理论上讲，Linux 有足够的灵活性可以装入任何格式的可执行文件。

1．ELF 可执行文件

ELF 是“可执行可连接格式”的英文缩写，该格式由 UNIX 系统实验室制定。它是 Linux 中最经常使用的格式，和其他格式（例如 a.out 或 ECOFF 格式）比较起来，ELF 在装入内存时多一些系统开支，但是更为灵活。ELF 可执行文件包含了可执行代码和数据，通常也称为正文和数据。这种文件中包含一些表，根据这些表中的信息，内核可组织进程的虚拟内存。另外，文件中还包含有对内存布局的定义以及起始执行的指令位置。

下面我们分析一个简单程序在利用编译器编译并连接之后的 ELF 文件格式：

```
#include <stdio.h>

main  ()
{
printf (“Hello world!\n”) ;

}
```

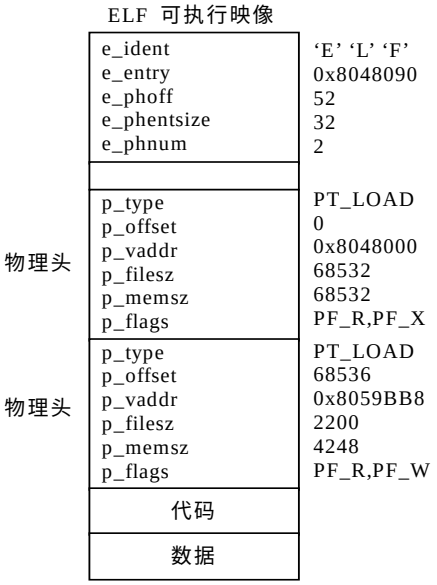


图 6.25　一个简单的 ELF 可执行文件的布局

图 6_25 所示，是上述源代码在编译连接后的ELF可执行文件的格式。从图 可以看出，ELF 可执行映象文件的开头是 3 个字符‘E’、‘L’和‘F’，作为这类文件的标识符。*e_entry* 定义了程序装入之后起始执行指令的虚拟地址。这个简单的 ELF 映像利用两个“物理头”结构分别定义代码和数据，*e_phnum* 是该文件中所包含的物理头信息个数，本例为 2。*e_phyoff* 是第一个物理头结构在文件中的偏移量，而 *e_phentsize* 则是物理头结构的大小，这两个偏移量均从文件头开始算起。根据上述两个信息，内核可正确读取两个物理头结构中的信息。

物理头结构的 p_flags 字段定义了对应代码或数据的访问属性。图中第 1 个 p_flags 字段的值为 FP_X 和 FP_R，表明该结构定义的是程序的代码；类似地，第 2 个物理头定义程序数据，并且是可读可写的。p_offset 定义对应的代码或数据在物理头之后的偏移量。p_vaddr 定义代码或数据的起始虚拟地址。p_filesz 和 p_memsz 分别定义代码或数据在文件中的大小以及在内存中的大小。对我们的简单例子，程序代码开始于两个物理头之后，而程序数据则开始于物理头之后的第 0x68533 字节处，显然，程序数据紧跟在程序代码之后。程序的代码大小为 0x68532，显得比较大，这是因为连接程序将 C 函数 printf 的代码连接到了 ELF 文件的原因。程序代码的文件大小和内存大小是一样的，而程序数据的文件大小和内存大小不一样，这是因为内存数据中，起始的 2200 字节是预先初始化的数据，初始化值来自 ELF 映象，而其后的 2048 字节则由执行代码初始化。

如前面所描述的，Linux 利用请页技术装入程序映像。当 shell 进程利用 fork ()系统调用建立了子进程之后，子进程会调用 exec ()系统调用（实际有多种 exec 调用），exec()系统调用将利用 ELF 二进制格式装载器装载 ELF 映像，当装载器检验映像是有效的 ELF 文件之后，就会将当前进程（实际就是父进程或旧进程）的可执行映像从虚拟内存中清除，同时清除任何信号处理程序并关闭所有打开的文件（把相应 file 结构中的 f_count 引用计数

减 1，如果这一计数为 0，内核负责释放这一文件对象），然后重置进程页表。完成上述过程之后，只需根据 ELF 文件中的信息将映象代码和数据的起始和终止地址分配并设置相应的虚拟地址区域，修改进程页表。这时，当前进程就可以开始执行对应的 ELF 映像中的指令了。

2．命令行参数和 shell 环境

当用户敲入一个命令时，从 shell 可以接受一些命令行参数。例如，当用户敲入命令：

```
$ ls -l /usr/bin
```

以获得在/usr/bin 目录下的全部文件列表时，shell 进程创建一个新进程执行这个命令。这个新进程装入/bin/ls 可执行文件。在这样做的过程中，从 shell 继承的大多数执行上下文被丢弃，但 3 个单独的参数 ls、-l 和 /usr/依然被保持。一般情况下，新进程可以接受任意个参数。

传递命令行参数的约定依赖于所用的高级语言。在 C 语言中，程序的 main( )函数把传递给程序的参数个数和指向字符串指针数组的地址作为参数。下面是 main()的原型：

```
int main (int argc, char *argv[])
```

再回到前面的例子，当/bin/ls 程序被调用时，argc 的值为 3，argv[0]指向 ls 字符串，argv[1]指向-l 字符串，而 argv[2]指向 /usr/bin 字符串。argv 数组的末尾处总以空指针来标记，因此，argv[3]为 NULL。

在 C 语言中传递给 main()函数的第 3 个可选参数是包含环境变量的参数。当进程用到它时，main( )的声明如下：

```
int main (int argc, char *argv[], char *envp[])
```

envp 参数指向环境串的指针数组，形式如下：

```
VAR_NAME=something
```

在这里，VAR_NAME 表示一个环境变量的名字，而“=”后面的子串表示赋给变量的实际值。envp 数组的结尾用一个空指针标记，就像 argv 数组。环境变量是用来定制进程的执行上下文，为用户或其他进程提供一般的信息，或允许进程交叉调用 execve( )系统调用保存一些信息。

命令行参数和环境串都放在用户态堆栈。图 6.26 显示了用户态堆栈底部所包含的内容。注意环境变量位于栈底附近正好在一个 NULL 的长整数之后。

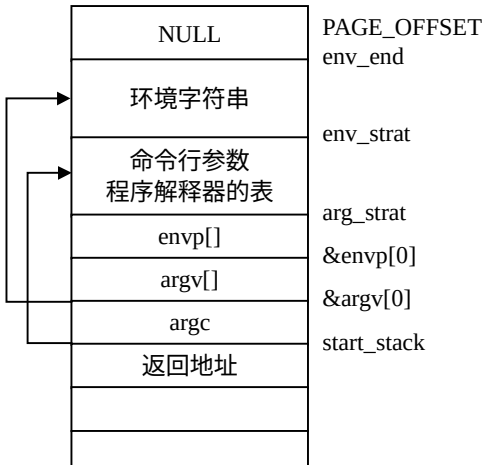

图 6.26　用户态堆栈底部所包含的内容

3．函数库

每个高级语言的源代码文件都是经过几个步骤才转化为目标文件的，目标文件中包含的是汇编语言指令的机器代码，它们和相应的高级语言指令对应。目标文件并不能被执行，因为它不包含源代码文件所用的全局外部符号名的虚拟地址。这些地址的分配或解析是由链接程序完成的，链接程序把程序所有的目标文件收集起来并构造可执行文件。链接程序还分析程序所用的库函数并把它们粘合成可执行文件。

任何程序，甚至最小的程序都会利用 C 库。请看下面的一行 C 程序：

```
void main (void) { }
```

尽管这个程序没有做任何事情，但还是需要做很多工作来建立执行环境并在程序终止时杀死这个进程。尤其是，当 main( )函数终止时，C 编译程序就把 exit( )系统调用插入到目标代码中。

实际上，一般程序对系统调用的调用通常是通过 C 库中的封装例程进行的，也就是说，C 语言函数库中的函数先调用系统调用，而我们的应用程序再调用库函数。除了 C 库，UNIX 系统中还包含很多其他的函数库。一般的 Linux 系统可能轻而易举地就有 50 个不同的库。这里仅仅列举其中的两个：数学库 libm 包含浮点操作的基本函数，而 X11 库 libX11 收集了所有 X11 窗口系统图形接口的基本底层函数。

传统 UNIX 系统中的所有可执行文件都是基于静态库的。这就意味着链接程序所产生的可执行文件不仅包括源程序的代码，还包括程序所引用的库函数的代码。

静态库的一大缺点是：它们占用大量的磁盘空间。的确，每个静态链接的可执行文件都复制库代码的一部分。因此，现代 UNIX 系统利用了共享库。可执行文件不用再包含库的目标代码，而仅仅指向库名。当程序被装入内存执行时，一个叫做程序解释器的程序就专注于分析可执行文件中的库名，确定所需库在系统目录树中的位置，并使执行进程可以使用所请求的代码。

共享库对提供文件内存映射的系统尤为方便，因为它们减少了执行一个程序所需的主内存量。当程序解释器必须把某一共享库链接到进程时，并不拷贝目标代码，而是仅仅执行一个内存映射，把库文件的相关部分映射到进程的地址空间中。这就允许共享库机器代码所在的页面由使用相同代码的所有进程进行共享。

共享库也有一些缺点。动态链接的程序启动时间通常比静态链接的长。此外，动态链接的程序的可移植性也不如静态链接的好，因为当系统中所包含的库版本发生变化时，动态链接的程序可能就不能适当地执行。

用户可以让一个程序静态地链接。例如，GCC 编译器提供-static 选项，即告诉链接程序使用静态库而不是共享库。

和静态连接库不同，动态连接库只有在运行时才被连接到进程的虚拟地址中。对于使用同一动态连接库的多个进程，只需在内存中保留一份共享库信息即可，这样就节省了内存空间。当共享库需要在运行时连接到进程虚拟地址时，Linux 的动态连接器利用 ELF 共享库中的符号表完成连接工作，符号表中定义了 ELF 映像引用的全部动态库例程。Linux 的动态连

接器一般包含在 /lib 目录中，通常为 ld.so.1、llibc.so.1 和 ld-linux.so.1。

### 6.8.3　执行函数

在执行 fork()之后，同一进程有两个拷贝都在运行，也就是说，子进程具有与父进程相同的可执行程序和数据（简称映像）。但是，子进程肯定不满足于仅仅成为父进程的“影子”，因此，父进程就要调用 execve()装入并执行子进程自己的映像。execve()函数必须定位可执行文件的映像，然后装入并运行它。当然开始装入的并不是实际二进制映像的完全拷贝，拷贝的完全装入是用请页装入机制（Demand Pageing Loading）逐步完成的。开始时只需要把要执行的二进制映像头装入内存，可执行代码的 inode 节点被装入当前进程的执行域中就可以执行了。

由于 Linux 文件系统采用了 linux_binfmt 数据结构（在/include/linux/ binfmt.h 中，见文件系统注册）来支持各种文件系统，所以 Linux 中的 exec()函数执行时，使用已注册的 linux_binfmt 结构就可以支持不同的二进制格式，即多种文件系统（EXT2,dos 等）。需要指出的是 binux_binfmt 结构中嵌入了两个指向函数的指针，一个指针指向可执行代码，另一个指向了库函数；使用这两个指针是为了装入可执行代码和要使用的库。linux_binfmt 结构描述如下，其链表结构的示意图如图 6.27 所示。

```
struct linux_binfmt {
     struct linux_binfmt * next;
       long *use_count;
     int  (*load_binary)  (struct linux_binprm *, struct  pt_regs * regs) ;/*装入二进制代码*/
     int  (*load_shlib)  (int fd) ;   /*装入公用库*/
       int  (*core_dump)  (long signr, struct pt_regs * regs) ;
```

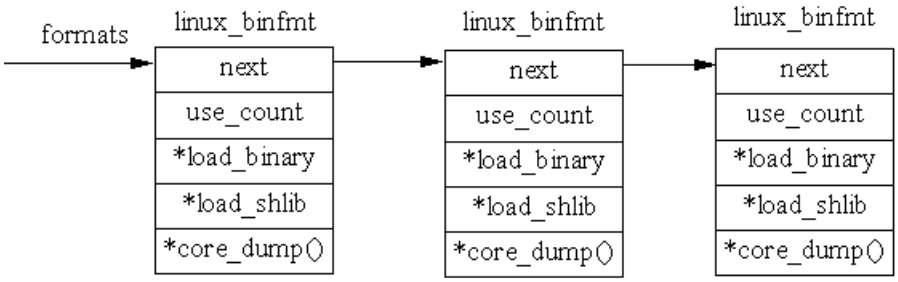


图 6.27　linux_binfmt 的链表结构

在使用这种数据结构前必须调用 vod binfmt_setup（）函数进行初始化；这个函数分别初始化了一些可执行的文件格式，如：init_elf_binfmt()；init_aout_binfmt()；init_java_binfmt()；init_script_binfmt()）。

其实初始化就是用 register_binfmt(struct linux_binfmt * fmt)函数把文件格式注册到系统中，即加入*formats 所指的链中，*formats 的定义如下：

```
static struct linux_binfmt *formats =  (struct linux_binfmt *)  NULL
```

在使用装入函数的指针时，如果可执行文件是 ELF 格式的，则指针指向的装入函数分别

是：

```
load_elf_binary (struct linux_binprm * bprm, struct pt_regs * regs) ;
static int load_elf_library (int fd) ;
```

所以 elf_format 文件格式说明将被定义成：

```
static struct linux_binfmt elf_format = {#ifndef MODULE
    NULL, NULL, load_elf_binary, load_elf_library, elf_core_dump#else
    NULL, &mod_use_count_, load_elf_binary, load_elf_library, elf_core_dump#endif }
```

其他格式文件处理很类似，相关代码请看本节后面介绍的 search_binary_handler()函数。

另外还要提的是在装入二进制时还需要用到结构 linux_binprm，这个结构保存着一些在装入代码时需要的信息：

```
struct linux_binprm{
   char buf[128];/*读入文件时用的缓冲区*/
      unsigned long page[MAX_ARG_PAGES];
      unsigned long p;
      int sh_bang;
      struct inode * inode;/*映像来自的节点*/
      int e_uid, e_gid;
   int argc, envc; /*参数数目，环境数目*/
      char * filename;  /* 二进制映像的名字，也就是要执行的文件名 */
      unsigned long loader, exec;
      int dont_iput;         /* binfmt handler has put inode */
};
```

其他域的含义在后面的 do_exec()代码中做进一步解释。

Linux 所提供的系统调用名为 execve(),可是，C 语言的程序库在此系统调用的基础上向应用程序提供了一整套的库函数，包括 execve()、execlp()、execle()、execv()、execvp()，它们之间的差异仅仅是参数的不同。下面来介绍 execve()的实现。

系统调用 execve()在内核的入口为 sys_execve()，其代码在 arch/i386/kernel/process.c：

```
 /*
  * sys_execve ()  executes a new program.
  */
asmlinkage int sys_execve (struct pt_regs regs)
{
        int error;
        char * filename;

        filename = getname ( (char *)  regs.ebx) ;
        error = PTR_ERR (filename) ;
        if  (IS_ERR (filename) )
                goto out;
error = do_execve (filename,  (char **)  regs.ecx,  (char **)  regs.edx, &regs) ;
        if  (error == 0)
                current->ptrace &= ~PT_DTRACE;
        putname (filename) ;
out:
        return error;
}
```

系统调用进入内核时，regs.ebx 中的内容为应用程序中调用相应的库函数时的第 1 个参数，这个参数就是可执行文件的路径名。但是此时文件名实际上存放在用户空间中，所以 getname()要把这个文件名拷贝到内核空间，在内核空间中建立起一个副本。然后，调用 do_execve（）来完成该系统调用的主体工作。do_execve（）的代码在 fs/exec.c 中：

```
/*
 * sys_execve ()  executes a new program.
 */
int do_execve (char * filename, char ** argv, char ** envp, struct pt_regs * regs)
{
        struct linux_binprm bprm;
        struct file *file;
        int retval;
        int i;

        file = open_exec (filename) ;

        retval = PTR_ERR (file) ;
        if  (IS_ERR (file) )
                return retval;

       bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof (void *) ;
        memset (bprm.page, 0, MAX_ARG_PAGES*sizeof (bprm.page[0]) ) ;

        bprm.file = file;
        bprm.filename = filename;
        bprm.sh_bang = 0;
        bprm.loader = 0;
        bprm.exec = 0;
        if  ( (bprm.argc = count (argv, bprm.p / sizeof (void *) ) )  < 0)  {
                allow_write_access (file) ;
                fput (file) ;
               return bprm.argc;
             }

        if  ( (bprm.envc = count (envp, bprm.p / sizeof (void *) ) )  < 0)  {
                allow_write_access (file) ;
                fput (file) ;
                return bprm.envc;
       }

        retval = prepare_binprm (&bprm) ;
        if  (retval < 0)
                goto out;

       retval = copy_strings_kernel (1, &bprm.filename, &bprm) ;
       if  (retval < 0)
                goto out;

        bprm.exec = bprm.p;
        retval = copy_strings (bprm.envc, envp, &bprm) ;
```

```
        if  (retval < 0)
                goto out;

        retval = copy_strings (bprm.argc, argv, &bprm) ;
         if  (retval < 0)
                goto out;

         retval = search_binary_handler (&bprm,regs) ;
         if  (retval >= 0)
                /* execve success */
                return retval;

out:
/* Something went wrong, return the inode and free the argument pages*/
         allow_write_access (bprm.file) ;
         if  (bprm.file)
                fput (bprm.file) ;

        for  (i = 0 ; i < MAX_ARG_PAGES ; i++)  {
                struct page * page = bprm.page[i];
               if  (page)
                       __free_page (page) ;
         }

        return retval;
}
```

参数 filename、argv、envp 分别代表要执行文件的文件名、命令行参数及环境串。下面对以上代码给予解释。

首先，将给定可执行程序的文件找到并打开，这是由 open_exec()函数完成的。open_exec() 返回一个 file 结构指针，代表着所读入的可执行文件的映像。

与可执行文件路径名的处理办法一样，每个参数的最大长度也定为一个页面（是否有点浪费？），所有 linux_binprm 结构中有一个页面指针数组，数组的大小为系统所允许的最大参数个数 MAX_ARG_PAGES（定义为 32）。memset()函数将这个指针数组初始化为全 0。

对局部变量 bprm 的各个域进行初始化。其中 bprm.p 几乎等于最大参数个数所占用的空间； bprm.sh_bang 表示可执行文件的性质，当可执行文件是一个 Shell 脚本（Shell Sript）时置为 1，此时还没有可执行 Shell 脚本，因此给其赋初值 0，还有其他两个域也赋初值 0。

函数 count()对字符串数组 argv[]中参数的个数进行计数。bprm.p / sizeof(void *) 表示所允许参数的最大值。同样，对环境变量也要统计其个数。

如果 count()小于 0，说明统计失败，则调用 fput()把该可执行文件写回磁盘，在写之前，调用 allow_write_access （）来防止其他进程通过内存映射改变该可执行文件的内容。

完成了对参数和环境变量的计数之后，又调用 prepare_binprm （）对 bprm 变量做进一步的准备工作。更具体地说，就是从可执行文件中读入开头的 128 个字节到 linux_binprm 结构的缓冲区 buf，这是为什么呢？因为不管目标文件是 ELF 格式还是 a.out 格式，或者其他格式，在其可执行文件的开头 128 个字节中都包括了可执行文件属性的信息，如图 6.25。

然后，就调用 copy_strings 把参数以及执行的环境从用户空间拷贝到内核空间的 bprm

变量中，而调用 copy_strings_kernel（）从内核空间中拷贝文件名，因为前面介绍的 get_name()已经把文件名拷贝到内核空间了。

所有的准备工作已经完成，关键是调用 search_binary_handler（）函数了，请看下面对这个函数的详细介绍。

search_binary_handler（）函数也在 exec.c 中。其中有一段代码是专门针对 alpha 处理器的条件编译，在下面的代码中跳过了这段代码：

```
  /*
  * cycle the list of binary formats handler, until one recognizes the image
  */
int search_binary_handler (struct linux_binprm *bprm,struct pt_regs *regs)
{
        int try,retval=0;
        struct linux_binfmt *fmt;
6 #ifdef __alpha__
    …..
#endif
        /* kernel module loader fixup */
        /* so we don't try to load run modprobe in kernel space. */
        set_fs (USER_DS) ;
        for  (try=0; try<2; try++)  {
              read_lock (&binfmt_lock) ;
               for  (fmt = formats ; fmt ; fmt = fmt->next)  {
                       int  (*fn)  (struct  linux_binprm  *,  struct  pt_regs  *)   =
fmt->load_binary;
                      if  (!fn)
                              continue;
                       if  (!try_inc_mod_count (fmt->module) )
                              continue;
                       read_unlock (&binfmt_lock) ;
                       retval = fn (bprm, regs) ;
                      if  (retval >= 0)  {
                             put_binfmt (fmt) ;
                             allow_write_access (bprm->file) ;
                             if  (bprm->file)
                                    fput (bprm->file) ;
                             bprm->file = NULL;
                             current->did_exec = 1;
                             return retval;
                      }
                       read_lock (&binfmt_lock) ;
                       put_binfmt (fmt) ;
                       if  (retval != -ENOEXEC)
                             break;
                       if  (!bprm->file)  {
                              read_unlock (&binfmt_lock) ;
                             return retval;
                       }
              }
               read_unlock (&binfmt_lock) ;
              if  (retval != -ENOEXEC)  {
```

```
                        break;
    #ifdef CONFIG_KMOD
                }else{
    #define printable(c) ((c)=='\t') || ((c)=='\n') || (0x20<=(c) && (c)<=0x7e))
                        char modname[20];
                        if (printable(bprm->buf[0]) &&
                            printable(bprm->buf[1]) &&
                            printable(bprm->buf[2]) &&
                           printable(bprm->buf[3]))
                                break; /* -ENOEXEC */
                        sprintf (modname, "binfmt-%04x", *(unsigned short *)
(&bprm->buf[2]));
                        request_module(modname);
    #endif
                }
            }
             return retval;
    }
```

在 exec.c 中定义了一个静态变量 formats:

```
static struct linux_binfmt *formats
```

因此，formats 就指向图 6.27 中链表队列的头，挂在这个队列中的成员代表着各种可执行文件格式。在 do_exec()函数的准备阶段，已经从可执行文件头部读入 128 字节存放在 bprm 的缓冲区中，而且运行所需的参数和环境变量也已收集在 bprm 中。search_binary_handler() 函数就是逐个扫描 formats 队列，直到找到一个匹配的可执行文件格式，运行的事就交给它。如果在这个队列中没有找到相应的可执行文件格式，就要根据文件头部的信息来查找是否有为此种格式设计的可动态安装的模块，如果有，就把这个模块安装进内核，并挂入 formats 队列，然后再重新扫描。下面对具体程序给予解释。

程序中有两层嵌套 for 循环。内层是针对 formats 队列的每个成员，让每一个成员都去执行一下 load_binary()函数，如果执行成功，load_binary()就把目标文件装入并投入运行，并返回一个正数或 0。当 CPU 从系统调用 execve()返回到用户程序时，该目标文件的执行就真正开始了，也就是，子进程新的主体真正开始执行了。如果 load_binary()返回一个负数，就说明或者在处理的过程中出错，或者没有找到相应的可执行文件格式，在后一种情况下，返回-ENOEXEC。

内层循环结束后，如果 load_binary()执行失败后的返回值为-ENOEXEC，就说明队列中所有成员都不认识目标文件的格式。这时，如果内核支持动态安装模块（取决于编译选项 CONFIG_KMOD），就根据目标文件的第 2 和第 3 个字节生成一个 binfmt 模块，通过 request_module()试着将相应的模块装入内核（参见第十章）。外层的 for 循环有两次，就是为了在安装了模块以后再来试一次。

在 linux_binfmt 数据结构中，有 3 个函数指针：load_binary、load_shlib 以及 core_dump，其中 load_binary 就是具体的装载程序。不同的可执行文件其装载函数也不同，如 a.out 格式的装载函数为 load_aout_binary(),ELF 格式的装载函数为 load_elf_binary(),其源代码分别在 fs/binfmt_aout.c 中和 fs/binfmt_elf 中。有兴趣的读者可以继续探究下去。

本章从内存的初始化开始，分别介绍了地址映射机制、内存分配与回收机制、请页机制、

交换机制、缓存和刷新机制、程序的创建及执行等 8 个方面。可以说，内存管理是整个操作系统中最复杂的一个子系统，因此，本章用大量的篇幅对相关内容进行了介绍，即使如此，也仅仅介绍了主要内容。

在本章的学习中，有一点需特别向读者强调。在 Linux 系统中，CPU 不能按物理地址访问存储空间，而必须使用虚拟地址。因此，对于 Linux 内核映像，即使系统启动时将其全部装入物理内存，也要将其映射到虚拟地址空间中的内核空间，而对于用户程序，其经过编译、链接后形成的映像文件最初存于磁盘，当该程序被运行时，先要建立该映像与虚拟地址空间的映射关系，当真正需要物理内存时，才建立地址空间与物理空间的映射关系。