分类号_____

U D C_____ 编号_____

学 位 论 文

Linux 操作系统内核分析与研究

Linux operating system kernel analysis and research

张荣亮

指导教师：余 敏 教授 江西师范大学计算机信息工程学院

申请学位级别：硕士 专业名称：计算机系统结构

论文提交日期：2007 年 5 月 答辩日期：2007 年 5 月 日

学位授予单位和日期：江西师范大学 2007 年 月 日

答辩委员会主席：_____

评阅人：_____

_____

二 00 七年五月

摘要

摘要：操作系统是计算机的重要组成部分，操作系统的优劣直接关系到计算机的整体性能。而且它是国产基础应用软件开发和国家、集体、个人信息安全的基石。由于 Linux 操作系统自由、公开、免费的特性，提供给人们一个研究国外优秀操作系统的设计思想和实现方法的机会，对于发展国产操作系统有着很重要的意义。

本文着重分析研究构成 Linux 2. 4 操作系统内核的四个基本功能：进程管理与调度机制、存储管理机制、文件系统管理机制。

1、 进程管理与调度机制描述数据结构进程控制块及 Linux 操作系统如何创建、执行、调度管理和撤销系统中的进程。

2、 存储管理机制描述物理内存管理器、内核缓冲区管理器、进程虚拟内存管理器的工作原理。

3、 文件管理机制描述 Linux 系统如何通过虚拟文件系统(VFS)支持多种不同的物理文件系统，剖析了 Linux 文件系统中一个物理文件系统的注册与注销、安装和卸载过程。

通过分析Linux 2. 4操作系统内核，掌握了操作系统的设计思想和实现方法，为以后进一步定制特定的嵌入式操作系统打下了坚实的基础；今后的工作应在操作系统内核改进上进行定性分析和理论创新。

关键词：Linux 操作系统，内核分析，进程调度，存储管理，VFS 虚拟文件系统。

Abstract

Content: The operating system is the computer important constituent, OS superior or inferior rank is directly related to the merits of the overall performance， moreover is the safe foundation stone of domestically produced foundation application software development and national, collective, individual information security. Because of open, free characteristic, Linux OS provides for the people a research overseas outstanding operating system design thought and the realization method opportunity, has the very vital significance regarding the development domestic product operating system.

This paper analyzes and researches the components of Linux 2.4 OS kernel of the four basic functions : process management and scheduling, memory management, file system management mechanisms.

1. Process management and process scheduling mechanism for describing data structures process control block and how to create, implement, management and schedul process.

2. Storage management mechanism for describing physical memory management, kernel buffer cache management, virtual memory management of process, the working principle.

3. File management mechanisms described how the Linux Virtual File System (VFS) support a wide range of physical file System, and it studies the processes of a physical file system registering、unregistering、mounting and unmounting in Linux file system.

By analyzing Linux 2. 4 Operating System Kernel, Mastered the design and implementation of the operating system, Lay a solid foundation for Custom specific embedded operating system. Meanwhile I should improve in the OS kernel performance through qualitative analysis and make theoretical innovation.

Key words: Linux operating system, Kernel analysis, Process scheduling, Memory management, Virtual File System(VFS).

目录

第一章 绪论

1.1 研究的目的与意义

Linux 操作系统自 1990 年诞生以来在高端服务器市场上获得了巨大的发展，在市场上表现出了强大的竞争力，据 IDC 资料显示，Linux 已经成为全球服务器市场上的第二大操作系统，截至 2004 年 7 月份，Linux 在服务器市场上的份额已经达到 28.3%，预计 2008 年将达到 37.6%。在嵌入式市场上，Linux 正在与 Windows CE 展开激烈的市场份额争夺战，并且已经略显优势，在亚洲地区，Linux 已经成为使用最普遍的嵌入式操作系统，统计表明，2004 年亚洲地区 63%的嵌入式项目正在使用 Linux 作为开发平台。

随着 Linux 系统的逐渐推广，它被越来越多的计算机用户所了解和应用，各国政府都在鼓励和支持 Linux 在本国的发展。越来越多的政府机构和 IT 巨头的注意力正在转向 Linux。从嵌入式设备到服务器，Linux 现在几乎可以用于所有的地方。

操作系统是计算机的重要组成部分，操作系统的优劣直接关系到计算机的整体性能。目前我国软件开发的水平比较落后。由于缺乏国产操作系统软件平台的支持，国产软件的开发主要是应用层次上的二次开发。软件市场大部分被国外软件垄断，且日益严重，这与我国发展知识经济，提高信息安全的需求相矛盾。没有自己的操作系统就好象摩天大厦没有坚固的基石，最终带给我们的是对国外的过度依赖，信息的安全性得不到保证，大大限制了我国民族软件产业发展。

以Linux为代表的开放源码软件给我国的软件业发展带来机遇。Linux在中国也发展得如火如荼，在2004年4月中日韩共同签署《开放源代码合作备忘录》，将合作致力于形成共同认可的Linux标准以实现信息交流与研究成果的共享。中国政府在最近几年的时间里，投入了大量的资金支持Linux的研究和开发，以设计具有自主版权的国产Linux 操作系统，推动国家信息产业基础软件设施的发展，最大限度地保证国家信息的安全。

我国政府是世界上公开支持开源软件运动为数不多的政府之一。这给Linux在中国的大力发展提供了广阔的宽松环境。而且，目前Internet技术和嵌入式技术在中国的发展方兴未艾，以Linux的强大技术优势定会在中国的操作系统领域有所作为。

1.2 主要工作

Linux是一个完全开放的操作系统，许多应用软件都以它作为操作系统平台，不少系统软件的改造和开发也以Linux为蓝本；作为计算机专业人员总是喜欢拥有自己的操作系统：分析源代码，编写具有自己风格的程序并重新编译生成新的Linux系统。而且作者致力于开发自己的嵌入式操作系统和商业应用，进行Linux内核的裁剪与定制研究，力求创造出具有我国自主产权的操作系统。而任何的裁减都是基于对内核的准确理解之上的，本文着重分析研究构成Linux 2.4操作系统内核的四个基本功能：进程管理与调度机制、存储管理机制、文件系统管理机制，并对Linux2.6内核在各方面所做的改进进行了描述。

1.3 论文的主要内容和结构安排

论文的组织结构如下：

第一章：绪论。简单描述了 Linux 发展现状和在中国的发展前景，以及本文的研究的意义，同时介绍了本文所做的主要工作内容和结构安排。

第二章：描述进程核心数据结构进程控制块及 Linux 操作系统如何创建、执行、调度管理和删除系统中的进程。

第三章：本章分析了 Linux 的内存管理子系统三个核心部分：物理内存管理器、内核缓冲区管理器、进程虚拟内存管理器。

第四章：本章从 Linux 内核文件管理机制——虚拟文件系统(VFS)出发，对 Linux 系统如何支持多种不同的物理文件系统进行了研究，剖析了 Linux 文件系统中一个物理文件系统的注册与注销、安装和卸载过程，以及通过 VFS 访问 Ext2 物理文件系统的内核工作机制。

第五章：总结全文，指出本文研究的不足和今后下一步进行的工作。

第二章 Linux 操作系统进程与进程管理

本章描述进程是什么以及 Linux 如何创建、管理和删除系统中的进程。

进程在操作系统中执行任务，是对正在运行的程序的一种抽象。程序是存放在磁盘上的包括一系列机器代码指令和数据的可执行的映像，它代表用户期望完成某工作的计划和步骤，只是体现在纸面，等待具体实现，因此，是一个被动的实体。进程可以看作是一个执行中的计算机程序，是计划和步骤的具体实现过程，它是动态的实体，在处理器执行机器代码指令时不断改变。除了程序的指令和数据外，进程还包含程序计数器和其它 CPU 的寄存器、包含临时数据（例如传递给某个函数的参数、函数的返回地址和保存的临时变量等）的进程堆栈、被打开的文件及输入输出设备的状态等，所有这些都随着程序指令的执行在不断变化。微处理器中当前的所有活动都在当前的执行程序环境，或者说进程的上下文中进行。

Linux 是一个多任务的操作系统，也就是说可以有多个程序同时装入内存并运行，操作系统为每个程序建立一个运行环境即创建进程。进程是分离的任务，各自拥有各自的权利和责任。如果一个进程崩溃，它不应该让系统中的另一个进程崩溃。每一个独立的进程都运行在自己的虚拟地址空间，除了通过安全的内核管理的机制之外，一个进程无法影响其它的进程。

系统中最宝贵的资源就是 CPU，通常系统只有一个 CPU。Linux 的目标之一是让系统的每一个 CPU 上一直都有进程在运行，以便充分利用 CPU 资源。如果进程数多于 CPU 数（多数是这样），其余的进程必须等到 CPU 被释放才能运行。

多进程是一个简单的思想：一个进程一直运行，直到它必须等待，通常是等待一些系统资源（如 CPU）；等它拥有了资源，它才可以再次（继续）运行。在一个多进程的系统中，同一时刻内存中有许多进程。当一个进程必须等待时，操作系统将 CPU 从这个进程拿走，并将它交给另一个更需要的进程。是调度程序选择了下一个最合适的进程。Linux 使用了一系列的调度方案来保证调度的公平性。

2.1 Linux 进程控制块

Linux的每一个进程都有自己的属性，用一个task_struct数据结构表示，即进程控制块[1]（Process Control Block，PCB）。它对进程在其生命周期内涉及的所有事件进行全面的描述，主要有进程标识符(PID)、进程状态信息、进程调度信息、进程所占的内存区域、相关文件的文件描述符、处理器环境信息、信号处理、

资源安排、同步处理等几个方面。

在一个系统中，通常可拥有数百个甚至数千个进程，相应地就有很多进程控制块。为了有效地对它们加以管理，应该用适当地方式将这些进程控制块组织起来。

进程控制块数据结构主要域定义如下（include/linux/sched.h）：

```
struct task_struct{
    volatile long state; /*进程状态*/
    unsigned long flags; /*进程标志*/
    int sigpending;      /*进程是否有信号在等待处理*/
    mm_segment_t addr_limit;
    struct exec_domain * exec_domain;
    volatile long need_resched; /*进程是否需要重新调度*/
    long counter;            /*进程剩余时间配额*/
    long nice;               /*进程优先级*/
    unsigned long policy;       /*进程调度策略*/
    struct mm_struct *mm;       /*描述进程用户空间*/
    struct list_head run_list;  /*进程可运行队列*/
    struct task_struct *next_task, *prev_task;
    struct mm_struct *active_mm;
    struct linux_binfmt *binfmt;
    pid_t pid;
    struct task_struct *p_opptr,*p_pptr,*p_cptr,*p_ysptr,*p_osptr;
    struct list_head thread_group;
    struct task_struct *pidhash_next;  /*把PCB链入散列表*/
    struct task_struct **pidhash_pprev;
    wait_queue_head_t wait_chldexit;   /*把PCB链入等待队列*/
    unsigned long rt_priority;         /*进程实时优先级*/
    long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
    uid_t uid, euid, suid, fsuid;      /*进程用户和组标识符*/
    gid_t gid, egid, sgid, fsgid;
    int ngroups;
    gid_t  groups[NGROUPS];         /*进程所属组的向量表*/
    kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
    struct user_struct *user;
```

```
    struct rlimit rlim[RLIM_NLIMITS];/*进程相关的限制信息*/
    struct fs_struct *fs;          /*描述进程所涉及到的文件系统信息*/
    struct files_struct *files; /*描述进程打开的所有文件的信息*/
    struct signal_struct *sig;       /*“信号向量表”*/
    sigset_t blocked;     /*模拟中断控制器硬件中的中断屏蔽寄存器*/
    struct sigpending pending;       /*信号队列*/
    ……
};
```

2.1.1 进程的状态

进程执行时它根据情况改变状态(state)。Linux 进程使用以下状态：

- **TASK_RUNNING 运行状态**：表示进程正在运行(是系统的当前进程)或者准备运行（等待被安排到系统的一个 CPU 上），处于这个状态的所有进程组成可运行队列。
- **TASK_INTERRUPTIBLE 可中断等待状态**：表示进程在等待一个事件或资源，当等待资源有效时被唤醒，也可被其它进程信号中断，唤醒后进入就绪状态。
- **TASK_UNINTERRUPTIBLE 不可中断等待状态**：表示进程在直接等待一个硬件条件的发生，并且在等待期间，不能被任何情况中断。
- **TASK_STOPPED 停止状态**：表示进程停止了（通常是接收到了一个信号），通过其它进程的信号才能唤醒。正在调试的进程可以在停止状态。
- **TASK_ZOMBIE 僵死状态**：表示进程已经死亡。一个终止的进程，已经释放内存、文件等大部分资源，但其 task_struct 数据结构没有被删去，在 task 向量表中仍旧占有一个条目，它不进行任何调度或状态转换，等待父进程将它彻底释放。而这种进程的状态就是僵死状态。
- **Swapping 交换状态**：虽然定义了该状态，但似乎没有使用它。

Linux 的各进程之间的状态转换及系统调用流程图如下：

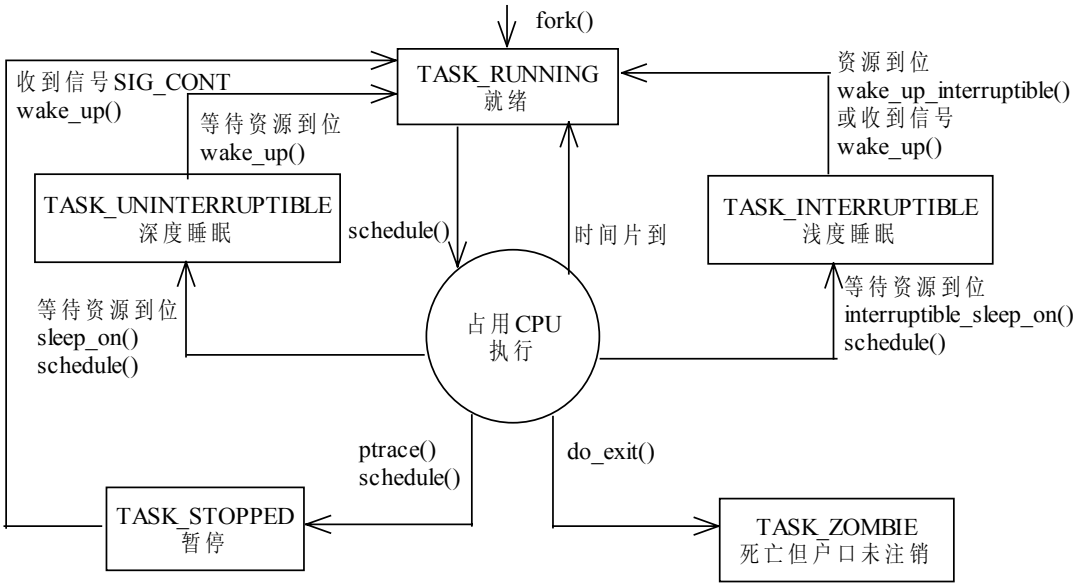

图2－1 进程状态转换图

2.1.2 进程的调度信息

调度程序需要这个信息用于公平地决定系统中的进程哪一个更应该运行。这些信息包括：need_resched（该进程是否需要重新调度）、(进程可以运行的时间量)、nice(或者 priority 优先级)、rt_priority（实时优先级），policy（进程调度策略）等。

- need_resched：调度标志，决定是否调用 schedule()函数。
- Counter：进程处于可运行状态时所剩余的时钟节拍数，每次时钟中断到来时，这个值减 1。
- 这个值将作为进程调度依据，因此又把这个域叫做进程的“动态优先级”，这样旧把时间片和优先级结合起来了。
- policy：进程使用的调度策略。Linux 有两种类型的进程：普通和实时。实时进程比所有其它进程的优先级高。如果有一个实时的进程准备运行，那么它总是先被运行。实时进程有两种调度策略：环（roundrobin）或先进先出（firstinfirstout)。在环的调度策略下，当调度程序把 CPU 分配给一个进程时，就把这个进程 PCB 放在运行队列的末尾；而在先进先出的策略下，每一个可以运行的进程按照它在调度队列中的顺序运行，并且这个顺序不会改变。普通进程使用另外的调度策略。因此 Linux 定义了三种调度策略：SCHED_OTHER（普通进程)、SCHED_FIFO（实时进程，先进先出)、SCHED_RR（实时进程，环)。
- Nice(或者 Priority)：进程的基本优先级，也叫静态优先级，也是当它允许运行的时候可以使用的时间量（jiffies)，这个值也决定 counter 的初值。

可以通过系统调用或者 renice 命令来改变一个进程的优先级。

- **rt_priority**：进程实时优先级。Linux 支持实时进程。这些实时进程比系统中其它非实时的进程拥有更高的优先级。这个域允许调度程序赋予每一个实时进程一个相对高的优先级。实时进程的优先级可以用系统调用来修改。

2.1.3 进程间通信机制

Linux 支持传统的 UNIX-IPC 机制：信号、管道和信号灯，也支持 SystemV 的 IPC 机制：共享内存、信号灯和消息队列。如下面是几个与信号相关的域：

- **int sigpending**：表示这个进程是否有信号在等待处理。
- **signal_struct *sig**：相当于一个“信号向量表”，数组中的每个元素相当于一个“信号量向”，确定了当进程接收到一个具体的信号时应该采取的行动，就好像一个中断向量指向一个中断服务程序一样。
- **sigset_t signal**：模拟中断控制器硬件中的中断状态寄存器。
- **sigset_t blocked**：模拟中断控制器硬件中的中断屏蔽寄存器。每当有一个中断请求到来时，中断状态寄存器中与之相应的某一位就被置 1，表示有相应的中断请求在等待处理，并且一直要到中断响应程序读出这个寄存器时才又被置 0。
- **sigpending pending**：信号队列。每产生一个信号就把它挂入这个队列，接收信号的进程就可以轮询所有可能的信号来源。

2.1.4 进程间的链接

在 Linux 系统中，没有一个进程是和其它进程完全无关的，总是根据不同的目的、关系和需要和其他的进程联系。从内核的角度看，则是要按不同的目的和性质将每个进程纳入不同的组织。

一、系统中的每一个进程，除了初始的进程之外，都有一个父进程。新进程不是创建的，而是从前一个进程拷贝，或者说克隆（cloned）来的。每一个进程的 task_struct 中都有指向它的父进程和兄弟进程（拥有相同的父进程的进程）以及它的子进程的指针。这些指针包括：

- p_opptr (original parentprocess);
- p_pptr (parent process);
- p_cptr (youngest child);
- p_ysptr (younger sibling);
- p_osptr (older sibling);

进程间的亲属关系图如下：

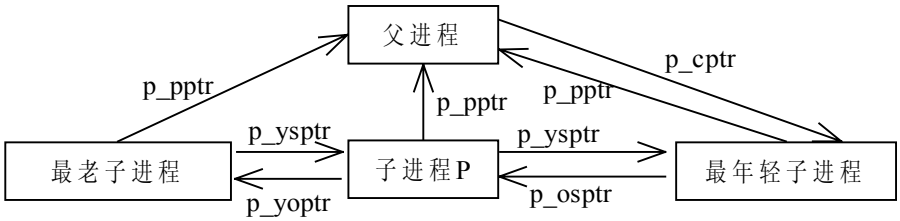

图 2－2 进程间的亲属关系

二、另外，系统中所有的进程的信息还存放在一个由 task_struct 数据结构组成的双向链表中（由指针 next_task 和 prev_task 连接），链表的头和尾都是 0 号进程 init_task 的 PCB，它的 PCB 被静态地分配到内核数据段中。这个表让 Linux 可以查到系统中的所有的进程。它需要这个表以提供对于 ps 或者 kill 等命令的支持。

要遍历整个进程链表，可用定义宏 for_each_task（p）：

```
#define for_each_task(p)\
    for(p=&init_task;(p=p->next_task)!=&init_task;)
```

三、有些情况下，内核必须能根据进程的 PID 快速定位到对应的 PCB，引入了散列表，定义了全局指针数组：

```
#define PIDHASH_SZ(4096>>2)
extern struct task_struct*pidhash[PIDHASH_SZ];
```

散列函数为：

```
#define pid_hashfn(x)    ((((x)>>8)^(x))&(PIDHASH_SZ-1))
```

当 pid 散列到相同的索引时，产生冲突，就通过 task_struct 数据结构中两个域 pidhash_next 和 pidhash_pprev 链入 hash 表 pidhash,同一双向链表中 pid 由小到大排列。

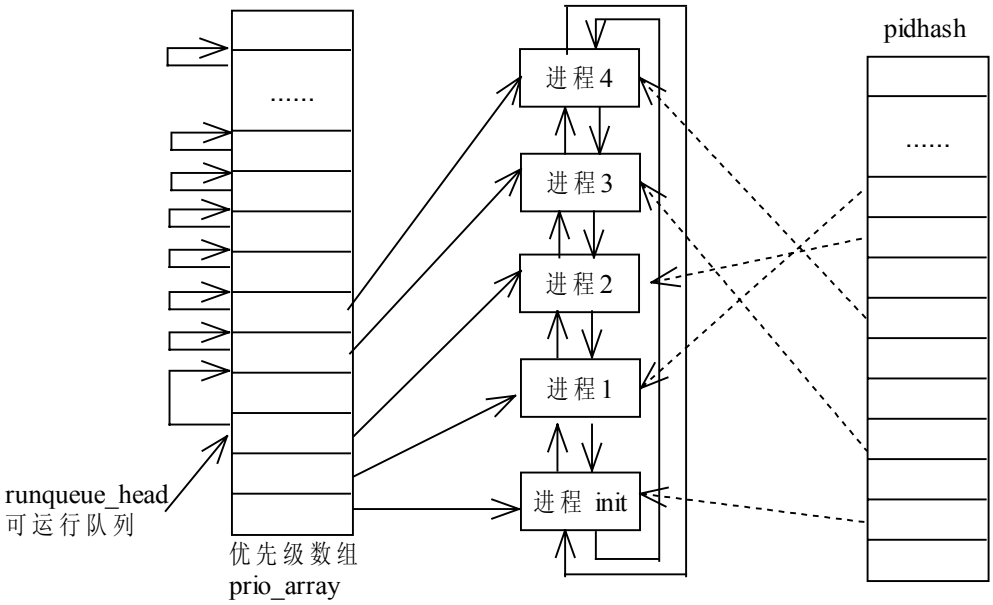

图2－3 优先级数组、hash表和进程之间的关系

四、当内核要寻找一个新的进程在CPU上运行时，必须只考虑处于可运行状态的进程，为了快速扫描整个可运行状态的进程，通过 struct list_headrun_list 将进程链入系统的一个双向循环链表中，叫**可运行队列（runqueue_head）**。Init_task起链表头和链表尾的作用，当调度程序遍历运行队列时，从init_task开始、到init_task结束。在调度程序运行过程中，队列允许加入新的可运行状态进程，并插入到可运行队列尾，这样不会影响到调度程序所要遍历的队列成员。

五、运行队列链表把处于可运行态的所有进程组织在一起，当要把其它状态的进程分组时，不同的状态要求不同的处理，如释放系统资源，或等待固定的时间间隔等；这样希望等待特定事件的进程就可通过域 wait_queue_head_t wait_chldexit 把自己放进合适的等待队列，并放弃控制权。等待队列表示一组睡眠的进程，当某一个条件变为真时，由内核唤醒它们。

2.1.5 进程的标识符

系统中的每一个进程都有一个进程标识符（pid）。进程标识符不是 task 向量表中的索引，而只是一个数字。

每一个进程都还有用户和组（user and group）的标识符（uid，euid，suid，fsuid，gid，egid，sgid，fsgid），用来控制该进程对系统中文件和设备的访问。

象所有的Unix一样，Linux使用用户和组标识符来检查对于系统中文件和映像的访问权限。Linux系统中所有的文件都有所有权和许可，这些许可描述了系统中的用户对该文件或目录拥有什么样的存取权限。基本的权限是读、写和执行。权限赋给了3类用户：文件属主、属于特定组的进程和系统中的其它进程。每一

类用户都可以拥有不同的权限，例如一个文件可以让它的属主读写、它的组读、而系统中的其它进程不能访问。

Linux 将对文件或者目录的存取权限赋予一组用户，而不是系统中的单个用户或者进程。一个进程可以属于几个组（缺省是 32 个），这些组放在每一个进程的 task_struct 结构中的 groups 向量表中。只要进程所属的其中一个组对于一个文件有访问权限，则这个进程就有对于这个文件的适当的组权限。

一个进程的 task_struct 中有 4 对进程和组标识符。

uid，gid

运行该进程的用户的用户标识符和组标识符。

euid，egid（effective uid and gid）

一些程序把执行进程的 uid 和 gid 改变为它们自己私有的 uid 和 gid（记录在描述该执行映像的 VFSI 节点的属性中），系统在运行这样的程序时，会根据修改后的 uid 及 gid 判断程序的特权。这些程序叫做 setuid 程序。这种方式是有用的，因为它是一种对服务进行限制访问的方法，特别是那些用其它人的方式运行的进程，例如网络守护进程。有效的 uid 和 gid 来自 setuid 程序，而进程的 uid 和 gid 仍旧是原来的。内核检查特权的时候检查有效 uid 和 gid。

fsuid，fsgid（Filesystem uid and gid）

通常和有效 effective uid 和 gid 相等，当检查对于文件系统的访问权限时，使用它们。通过 NFS 安装的文件系统需要它们，因为这时 NFS 服务器运行在用户态，又需要象一个特殊进程一样访问文件。此时，只有文件系统 uid 和 gid 改变（而非有效 uid 和 gid）。这避免了恶意用户向 NFS 的服务程序发送 Kill 信号。Kill 用一个特别的有效 uid 和 gid 发送给进程。

suid，sgid（Saved uid and gid）

这是 POSIX 标准的要求，让程序可以通过系统调用改变进程的 uid 和 gid。在原来的 uid 和 gid 改变之后，此处保存真正的 uid 和 gid。

2.1.6 进程的文件系统

进程可以根据需要打开或者关闭文件。进程的 task_struct 结构中用 fs_struct 数据结构来描述进程所涉及到的文件系统信息，用 files_struct 数据结构来描述进程打开的所有文件的信息。

fs_struct 数据结构中有六个指针，前三个是 dentry 目录结构指针，即 root、pwd 和 altroot.Dentry 目录数据结构里面记录着文件的各项属性，如文件名、访问权限等。Root 指向本进程的“根目录”，即当用户登录进入系统时所看到的根目录；pwd 指向进程当前工作所在的目录；altroot 为用户设置的“替换根目录”。后三个指针各自指向代表着这些“安装”的 vfsmount 数据结构。

files_struct 数据结构的主体是 file 结构数组，保存与具体打开文件有关的信息。file 结构中有一组指针指向每一个打开的文件描述符，进程就通过在 file 结构数组的下标“打开文件号”来访问这个文件。

2.1.7 进程的用户空间

多数进程都有一些虚拟内存（核心线程和核心守护进程没有），Linux 内核通过 mm_struct 结构描述进程的整个用户空间，在其中包含了进程的页表和许多跟踪记录这些虚拟内存到系统物理内存的映射关系的信息，并用 vm_area_structs 结构描述用户空间中各个虚存区。

2.1.8 限制信息

结构 rlimit 中定义了与进程相关的限制信息。包括一个用户可以拥有的进程数、一个进程可以打开的文件数、地址空间限制、文件大小、数据大小等。

2.1.9 会话和进程制

由于 Linux 是一个多用户系统，同一时刻，系统中运行有属于不同用户的多个进程。那么，当处于某个终端上的用户按下了 Ctrl+C 键时（产生 SIGINT 信号），系统如何知道将该信号发送到哪个进程，从而不影响由其它终端上用户运行的进程。

Linux 内核通过维护会话和进程组而管理多用户进程。如图 2.4 所示，每个进程是一个进程组的成员，而每个进程组又是某个会话的成员。一般而言，当用户在某个终端上登录时，一个新的会话就开始了。进程组由组中的领头进程标识，领头进程的进程标识符就是进程组的组标识符。类似地，每个会话也对应有一个领头进程。

同一会话中的进程通过该会话的领头进程和一个终端相连，该终端作为这个会话的控制终端。一个会话只能有一个控制终端，而一个控制终端只能控制一个会话。用户通过控制终端，可以向该控制终端所控制的会话中的进程发送键盘信号。同一会话中只能有一个前台进程组，属于前台进程组的进程可从控制终端获得输入，而其它进程均是后台进程，可能分属于不同的后台进程组。

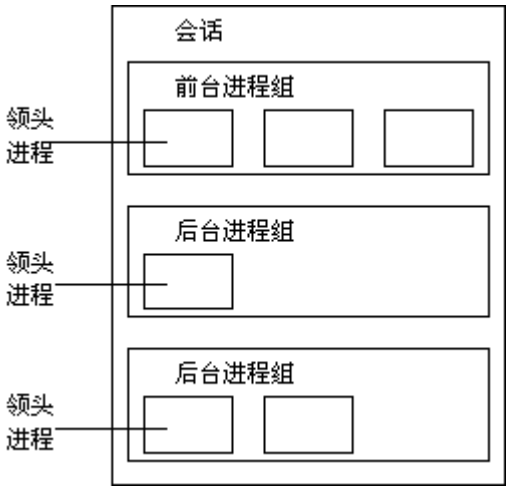

图 2.4 会话和进程、进程组

与会话和进程组相关的数据结构如下：

```
pid_t pgrp;        /*进程组标志号*/
pid_t tty_old_pgrp;
pid_t session;     /*进程组号*/
int leader;        /*进程组号*/
```

2.2 进程创建

很多操作系统都提供了产生进程的机制，其采用的方法是首先在新的地址空间里创建进程，然后读可执行文件，最后开始执行。Linux（以及 Unix）采用了不同的方式，它把进程的创建和目标程序的执行分成两步：首先通过 fork()或者 clone()复制当前进程创建一个子进程；然后，exec()函数负责读取可执行文件并将其载入进程的地址空间开始运行。

fork()是全部复制，父进程所有的资源全部都通过数据结构的复制“遗传”给子进程。而 clone()则可以将资源有选择地复制给子进程，而没有复制地数据结构则通过指针的复制让子进程共享，极端的情况下，clone()创建一个线程。

fork()和 clone()函数的主体是调用 do_fork()，下面分析 do_fork()函数所做的工作：

```
int do_fork(unsigned long clone_flags，unsigned long stack_start，
    struct pt_regs*regs，unsigned long stack_size)
```

（1） 调用 alloc_task_struct()函数向物理内存管理器申请 8KB 的 union task_union 内存区，用来存放新进程的数据结构 task_struct 和系统空间堆栈。task_struct 结构大约占 1KB，系统空间堆栈大小不能超过 7KB，

否则会覆盖 task_struct 结构。P 是指向新结构的指针。

（2） 将当前进程（current）的 task_struct 数据结构的内容拷贝到新进程的 task_struct 数据结构中。

*p=*current;

（3） 检查新创建子进程后，当前用户所拥有的进程数目有没有超出给分配的资源限制。

（4） 初始化新进程数据结构

```
p->did_exec=0;
p->swappable=0;
/*为防止信号、定时中断误唤醒未创建完毕的进程，将子进程的状态
  设成不可中断的。*/
p->state=TASK_UNINTERRUPTIBLE;
/*p->flags主要来自父进程，但根据新进程的创建标志clone_flags做了修改*/
copy_flags(clone_flags, p);
p->pid=get_pid(clone_flags);  /*为该进程获取一个有效的pid*/
p->run_list.next=NULL;        /*还没有加入到运行进程队列中*/
p->run_list.prev=NULL;
p->p_pptr=p->p_opptr=current;//父进程和原始父进程为当前进程
p->p_cptr=NULL;               /*子进程为空*/
init_waitqueue_head(&p->wait_chldexit); /*初始化等待队列*/
p->vfork_sem=NULL;
p->sigpending=0;                  /*无正等待处理的信号*/
init_sigpending(&p->pending);     /*初始化信号队列*/
p->it_real_value=p->it_virt_value=p->it_prof_value=0;//时钟
p->it_real_incr=p->it_virt_incr=p->it_prof_incr=0;
init_timer(&p->real_timer);       /*时钟队列为空*/
p->real_timer.data=(unsignedlong)p;
p->leader=0;
p->tty_old_pgrp=0;
p->times.tms_utime=p->times.tms_stime=0;
p->times.tms_cutime=p->times.tms_cstime=0;
p->lock_depth=-1;          /*-1=nolock*/
p->start_time=jiffies;     /*当前时间就是该进程的开始时刻*/
```

（5） 根据传递给 fork()的参数标志，复制或共享当前已打开的文件、文件系统信息、信号处理函数、进程的虚拟地址空间等。如果进程含有线程，则其所有线程共享这些资源，无需复制；否则，这些资源对每个进程是不同的，需要进行复制。

```
if(copy_files(clone_flags, p)) /*复制或共享当前已打开的文件*/
    gotobad_fork_cleanup;
if(copy_fs(clone_flags, p))     /*复制或共享文件系统信息*/
    gotobad_fork_cleanup_files;
if(copy_sighand(clone_flags, p))   /*复制或共享信号处理函数*/
    gotobad_fork_cleanup_fs;
if(copy_mm(clone_flags, p)) /*复制或共享进程的虚拟地址空间*/
    gotobad_fork_cleanup_sighand;
/*复制父进程的系统空间堆栈*/
retval=copy_thread(0, clone_flags, stack_start, stack_size, p, regs);
if(retval)
    gotobad_fork_cleanup_sighand;
```

（6） 让子进程和父进程平分剩余的时间片

（7） 通过 SET_LINKS(p)把新的 PCB 插入进程链表，以确保进程之间的亲属关系。

（8） 通过 hash_pid(p)散列函数把新的 PCB 散列到 pidhash 表。

（9） 通过 wake_up_process(p)把子进程 PCB 的状态域设置成 TASK_RUNNING，即将其唤醒，并把子进程插入到可运行队列链表。

（10） 返回子进程的 pid，这个 pid 最终由用户态下的父进程读取。

2.3 进程执行

一个可处于运行态的完整子进程创建完成后，应该从内核态返回到用户态，内核有意选择子进程立即投入运行，因为子进程一般会马上调用 exec()函数，启动新进程要执行的程序，这样可以避免写时复制的额外开销。

因为一个程序通常是一个存放在磁盘上的内核映象，当一个映像执行时，执行映像的内容必须放在进程的虚拟地址空间中。对于执行映像连接到的任意共享库，情况也是一样。内存映象机制将虚拟内存和文件系统结合起来，它提供了一种设施，用来将文件映射到进程的虚拟地址空间，而后就可以使用简单的内存访问指令来读写文件。此时整个进程的地址空间就是一组映射到不同数据对象的映

射。进程所有的有效虚拟地址就是这些映射到数据对象上的地址。这些对象（映射文件）为映射它的页面提供了持久性后备存储，映射使得进程可以直接寻址这些对象，被映射的对象既不会意识到也不会受映射的影响。

当一个执行映象执行时，执行文件实际上并没有真正被装到物理内存，而仅仅是被映射到了进程的虚拟内存（在内存中建立一系列数据结构）。如果进程的执行访问到了不在内存中的页面时，系统会发生页面失效异常，异常处理程序会把需要的页面逐步读入内存。因此，只有当运行程序引用映像的某一部分时，这部分映像才真正从执行文件加载到内存中。这种映像和进程虚拟地址空间的连接叫做内存映射。

执行一个新程序由函数 do_execve（）完成，其定义如下：

```
int do_execve(char*filename,char**argv,char**envp,structpt_regs*regs)
    do_execve()
    {
        先用 open_exec(filename)打开给定的可执行文件。
        /*内核将运行可执行文件时所需信息组织在一起，如参数和环境变量的
          计数等*/
        初始化数据结构 linux_binprmbprm。
        调用 prepare_binprm()进一步做数据结构 bprm 的准备工作。
        search_binary_handler(&bprm, regs);
        {
            寻找二进制文件的处理程序（handler），并用该 handler（一般是
              load_aout_binary）程序处理该二进制文件
            load_aout_binary()
            {
                异常情况判断
                flush_old_exec(bprm)/*放弃从父进程“继承”下来的全部用户
            空间*/
                {
                    通过 make_private_signals()判断子进程是否仍在共享父
                    进程信号处理表，若是，则复制过来。
                    exec_mmap()
                    {
                        flush_cache_mm(old_mm);
                        mm_release();
```

```
                exit_mmap(old_mm);    /*释放 mm_struct 下所有的虚
                vm_area_struct 数据结构,并将页面表项中的表项都设置
                成 0。*/
                flush_tlb_mm(old_mm);/*使高速缓存与内存一致*/
            }
            release_old_signals(oldsig); /*递减父进程信号处理表
            的共享计数。并且如递减后为 0 就要将其占用的空间释放*/
            flush_thread();
            /*“当前进程”已升级为进程，应脱离父进程的线程组*/
            de_thread(current);
            /*清除当前进程的服务程序的表项*/
            flush_signal_handlers(current);
            /*清除当前进程的已打开文件*/
            flush_old_files(current->files);
        }//到此为止，当前进程已被掏空，成为一个空壳。
            根据 bprm 结构更新 current 进程的控制块
        do_mmap(); //将执行文件的代码段映射到虚存
        do_mmap(); //将执行文件的数据段映射到虚存
    }
  }
}
```

2.4 进程调度与切换

Linux 进程调度由 schedule()执行，其任务是在 runqueue_head 队列中选出一个就绪进程。Linux 的进程调度时机有：

1、进程状态转换的时刻，如进程终止、进程睡眠，进程需要调用 sleep_on()或 exit()等函数。

2、当前进程的时间片用完时，需要主动放弃 CPU。

3、设备驱动程序运行时,如果每次循坏检查调度标志为假时,需要主动放弃CPU。

4、从内核态返回到用户态时，不管是从中断、异常还是系统调用返回，都要对调度标志进行检测，如果必要，则调用调度程序。

Linux中schedule()通过调用goodness()函数来衡量一个处于可运行队列状态的进程值得运行的程度。该函数综合使用调度信息 policy、counter、priority（nice）、rt_priority 几个域，给每个处于可运行状态的进程赋予一个权值

（weight），调度程序以这个权值作为选择进程的唯一依据。goodness()函数如下：

```
int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{
    int weight;
    weight = -1;
    /* 如果是普通进程，其权值取决于剩余时间片和静态优先级 */
    if (p->policy == SCHED_OTHER) {
        weight = p->counter; /* 剩余时间片 */
        if (!weight)
            goto out;
        /* 需要调度的进程恰好是当前进程，其权值奖励 1 */
        if (p->mm == this_mm || !p->mm)
            weight += 1;
        weight += 20 - p->nice; /* 静态优先级 */
        goto out;
    }
    weight = 1000 + p->rt_priority; /* 如果是实时进程，其权值取决于实时优先级 */
out:
    return weight;
}
```

如果最值得运行的进程不是当前进程，当前进程必须被挂起，运行新的进程，即进行进程切换。当一个进程运行的时候它使用了 CPU 和系统的寄存器和物理内存。每一次它调用例程都通过寄存器或者堆栈传递参数、保存数值比如调用例程的返回地址等。因此，当调度程序运行的时候它在当前进程的上下文运行。它可能是特权模式：核心态，但是它仍旧是当前运行的进程。当这个进程要挂起时，它的所有机器状态，包括程序计数器(PC)和所有的处理器寄存器，必须存到进程的 task_struct 数据结构中。然后，必须加载新进程的所有机器状态。进程切换主要通过调用 switch_mm()和 switch_to()函数来完成。

switch_mm()函数负责把虚拟内存从上一个进程映射到新进程中。

switch_to()函数负责从上一个进程的处理器状态切换到新进程的处理器状态，包括保存、恢复栈信息和寄存器信息。

2.5 进程撤销

当用户或者进程发出一个退出系统命令的时候，Linux 就调用系统调用 sys_exit。系统调用 sys_exit 的主要作用是终止当前正在运行的所有用户的应用程序，保存当前帐号的各种信息，逐步退出支撑 Linux 操作系统运行的系统子模块和子系统，把进程的状态设置为 TASK_ZOMBIE，并把其所有的子进程都托付给 Init 进程，最后调用 schedule()函数，选择一个最合适的进程投入运行。

系统调用 sys_exit 的内核函数为：NORET_TYPEvoiddo_exit(longcode)。

```
do_exit()
{   ……
    del_timer_sync(&tsk->real_timer);     /*删除当前任务定时器队列*/
    __exit_mm(current);                   /*释放进程存储空间*/
    {
        exit_mmap(mm);
        {
            for 对于当前进程 current->mm 中的每一个 VMA 块，
            {
                调用 vm_ops->unmap 释放该 VMA 块对应的虚拟空间
                调用 zap_page_range 将释放掉的虚拟空间中的 pte 页表项清
            零。
                调用 kmem_cache_free 释放该 VMA 块结构占用的物理空间。
            }
            调用 clear_page_tables（）释放页表（pagetable）。
        }
    }
    sem_exit();         /*清理进程在用户空间建立和使用的信号量队列*/
    __exit_files(tsk);  /*把当前任务所打开的文件都关闭，释放文件指针*/
    __exit_fs(tsk);         /*退出文件系统。*/
    exit_sighand(tsk);      /*释放当前任务的所有信号（signal）*/
    exit_notify();          /*向外广播退出，通知父进程释放进程
                            task_struct 数据结构和系统空间堆栈*/
    schedule();       /*继续调度，挑选一个最合适的进程投入运行*/
}
```

2.6 本章小结

进程控制块对进程在其生命周期内涉及的所有事件进行全面的描述。

每个进程都有一个生命周期：创建、执行某段程序、最后消亡。进程创建时的状态为 TASK_UNINTERRUPTIBLE，在 do_fork()结束前被父进程唤醒后，变为 TASK_RUNNING。处于 TASK_RUNNING 状态的进程被移到 runqueue_head 队列中，在适当时候由 schedule()按 CPU 调度算法选中，获得 CPU。进程执行系统调用 sys_exit()或收到 SIG_KILL 信号而调用 do_exit()时，进程状态变为 TASK_ZOMBIE，释放所申请资源，启动 schedule()切换到其它就绪进程。Linux 进程调度由 schedule()执行，其任务是在 runqueue_head 队列中选出一个就绪进程。

第三章 Linux 操作系统内存管理机制分析

内存管理子系统是操作系统的重要部分，其任务是高效地管理系统中的内存资源。

Linux 的内存管理子系统分为五个部分：物理内存管理器、内核缓冲区管理器、内核虚拟内存管理器、进程虚拟内存管理器和用户内存管理器。其中前四个是由内核完全实现和进行管理的，用户内存管理器是由内核和上层程序相结合实现的，上层程序利用内核提供基本的支持机制对用户虚拟内存资源进行管理。

每一台计算机系统都有一个高速的、随机访问的物理内存条，或称物理内存，简称内存，它所提供的内存空间定义为物理内存空间，其中每个内存单元的实际地址就是物理地址。程序可以直接访问内存中的代码和数据。将应用程序员看到的内存空间定义为虚拟地址空间（或地址空间），其中的地址就叫虚拟地址，一般用“段：偏移量”的形式来描述。线性地址空间是指一段连续的、不分段的、范围为 0～4GB 的地址空间。一个线性地址就是线性地址空间的一个绝对地址。Linux 内核将这 4GB 的空间分为两部分，较高的 1GB（虚地址 0xC0000000 到 0xFFFFFFFF）供内核使用，称为“内核空间”；而较低的 3GB（虚地址 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 内核空间由系统内的所有进程共享。从具体进程的角度来看，每个进程都可以拥有 4GB 的虚拟地址空间（或者叫虚拟内存）。

虚拟内存管理是内存管理子系统要完成的主要工作，但不是全部工作，因为虚拟内存在具体使用时必须化为实际的物理内存，因此内存管理的基础是对物理内存的管理，而各个内存管理器有都离不开内核内存管理器的支持。物理内存管理子系统负责对物理内存页的分配和回收，为了管理的方便，物理内存和虚拟内存都分成大小相等的页，因此又叫它页面分配器。页面分配器有两个主要的用户：进程虚拟内存管理子系统和内核虚拟内存管理子系统。进程虚拟内存管理子系统为用户进程提供虚拟页面的分配和回收，而内核虚拟内存管理子系统则为内核提供各种大小的物理内存块。

所以，内存管理分为三大部分：物理内存管理、进程虚拟内存管理和内核内存管理。三部分的关系如下图（图 3-1）所示。

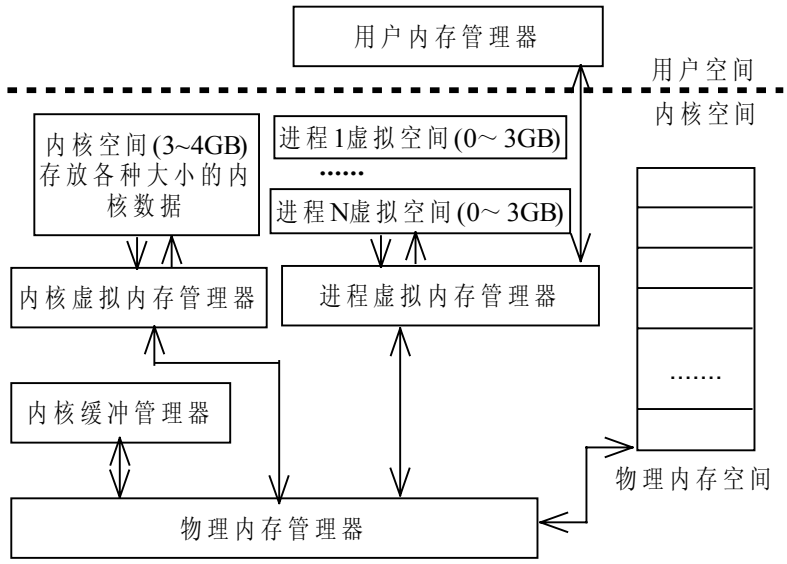

图3－1 Linux内存管理器模型及各类型地址之间的关系

3.1 物理内存管理

物理内存管理器是针对NUMA(Non-UniformMemoryArchitecture，非一致内存访问体系)架构设计的，它在多个层面上对物理页面进行管理。NUMA系统的节点通常是由一组CPU和本地内存组成的，有的节点还有I/O子系统。全系统的内存在物理上是分布的，每个节点访问本地内存和访问其他节点的远地内存的延迟是不同的。

对每块UMA物理内存都进行独立管理，每一块UMA物理内存都有一个相对应的pg_data_t数据结构实例，一个系统中可能有多个pg_data_t数据结构实例，所有的pg_data_t数据结构实例通过指针链接在链表pg_data_t*pgdat_list中。同时，它将每一个pg_data_t所代表的UMA物理内存块划分为3个zone区：ZONE_DMA，ZONE_NORMAL和ZONE_HIGHMEM，这3个区域地大小分别是：

- ZONE_DMA大小小于16MB，ISADMA能力的内存。
- ZONE_NORMAL大小范围是16MB~896MB，内核可直接映射。
- ZONE_HIGHMEM大小大于896MB，其中的页并不能永久地映射到内核地址空间，仅是页cache和用户进程使用，

在每个zone区上使用buddy（伙伴）算法对物理页面进行管理，每个zone内管理内存的基本单位是物理页面，每一个物理页面用一个structpage（或mem_map_t）结构来描述。每个节点中有zone_mem_map数组成员存放全部页的页结构。

为了实现物理内存页面换入/换出，在page数据结构中设置了所需的各种成

分，并在内核中设置了全局性的 active_list 和 inactive_dirty_list 两个 LRU 队列，在每个页面管理区中设置了一个 inactive_clean_list.。根据页面的 page 结构在这些 LRU 队列中的位置，就可知道这个页面转入不活跃后时间的长短，为回收页面提供参考。通过一个全局的 address_space 数据结构 swapper_space，把所有可交换内存页面管理起来，每个可交换内存页面的 page 数据结构通过其队列头结构 list 链入其中的一个队列。此外，为加快在暂存队列中搜索，又设置了一个杂凑表 page_hash_table。

Linux 采用 Node、Zone 和页三级结构来描述物理内存地，如下图：

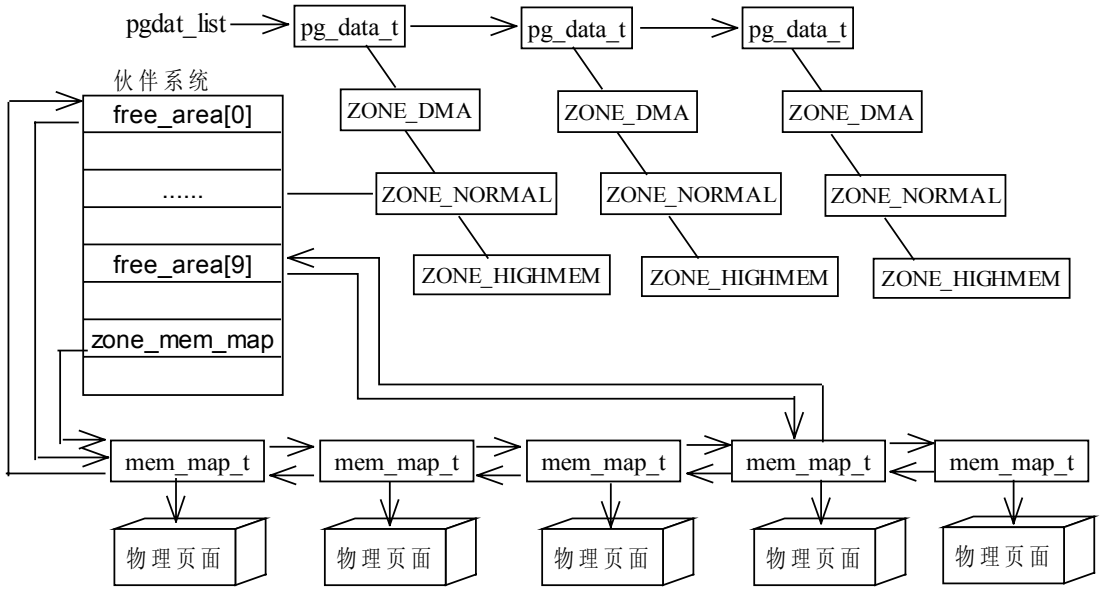

图3－2 NUMA架构的Linux物理内存管理器

图中，带箭头的联系表示指针关系，不带箭头的表示数组包含关系。

3.1.1 物理页分配与回收

Linux 的物理页分配采用链表和位图结合的方法（如图 3-3）。内核定义了一个称为 free_area 的块链表来分配和回收物理页，每个链表中的一个页块含有 2 的幂次个页面。例如，第 0 个链表中块的大小都为 1 页，第 1 个链表中块的大小都为 2 页，第 9 个链表中块的大小都为 512 页，依次类推。系统初始化时，free_area 数组也被赋了初值。也就是说，系统中所有可用的空闲物理页块都已经被加到了 free_area 数组中，并按伙伴算法尽可能地做了合并。

free_area 数组的定义如下：

```
typedef struct free_area_struct{
        struct list_head   free_list;
        unsigned int    *map;
```

}free_area_t;

free_area_t　　free_area[MAX_ORDER];

数组 free_area 的每项包含三个元素：next、prev 和 map。指针 next、prev 用于将物理页块结构 mem_map_t 连结成一个双向链表；而 map 则是记录这种页块组分配情况的位图，例如，位图的第 N 位为 1，表明第 N 个页块是空闲的。从图中也可以看到，用来记录页块组分配情况的位图大小各不相同，显然页块越小，位图越大。free_area 数组的大小为 10 或 12（可调），因此它管理的最小内存块的大小是 1 页（4K），最大内存块的大小是 512 页（2M）或 2048 页）8M。

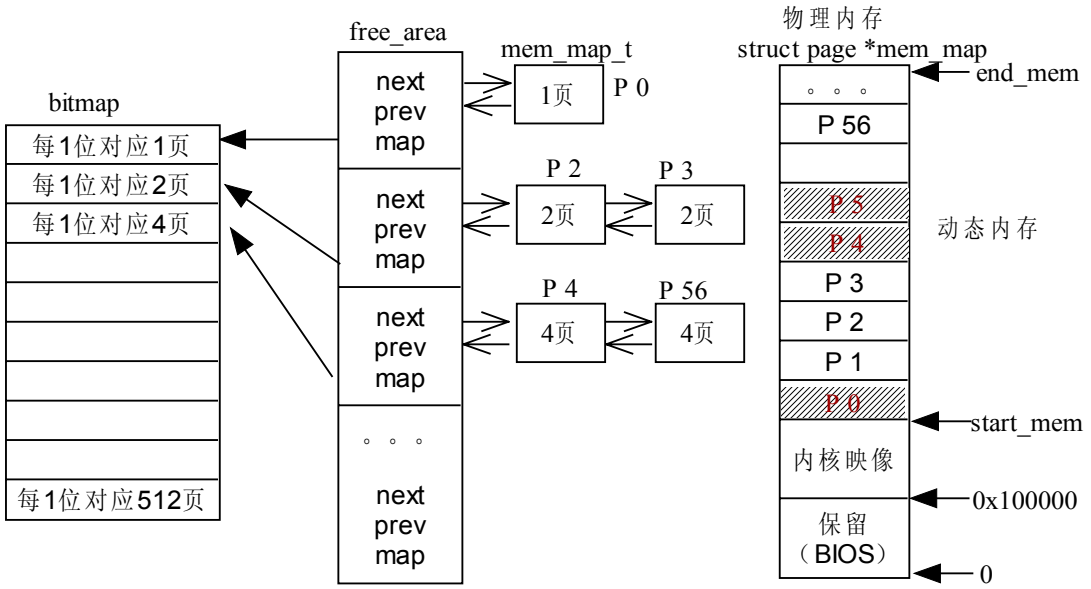

图3－3 伙伴算法使用frea_area管理物理内存图

Linux 使用 Buddy（伙伴）算法有效地分配和回收物理页块。页分配代码试图分配一个或多个连续物理页面组成的内存块。内存块的大小为 2 的幂数个页。这意味着可以分配 1 页大小、2 页大小、4 页大小的块，依此类推。Free_area 中的每一个单元都有一个与之关联的位图，该位图描述系统中大小与其自身大小相同的所有页块的占用和空闲情况。例如，数组中的第 2 个单元拥有描述 4 页大小的页块的空闲和占用的分配图。页面分配时，内存分配器在 free_area 数组中寻找一个与请求大小相同的空闲块。

buddy 算法思想为：在页面分配时，内存分配器在 free_area 数组中寻找一个与请求大小相同地空闲块。分配算法首先搜寻满足请求大小的页面，它从 free_area 数据结构的 List 域着手沿着链表来搜索空闲页面。如果没有以这样的方式请求大小的空闲页面，则它搜索两倍于请求大小的内存块。这个过程一直将持续到 free_area 被搜索完或找到满足要求的内存块为止。如果找到的页面块大于请求的块，则把它分成两块：一个与请求块匹配，另一个是空闲块。每块大小都是 2 的 N 次幂。空闲块被链接进相应大小的队列，而另一个页面被分配给调

用者。

分配页块的过程中将大的页块分为小的页块，将会使内存更为零散。页面回收时，内存分配器将检查是否有相同大小的相邻或者 buddy 内存块存在。如果有，则将他们合起来形成一个大小为原来两倍的新空闲块。每次结合完之后，代码还要检查是否可以继续合并成更大的页面。

1. 函数__get_free_pages用于物理页块的分配，其定义如下：

unsigned long __get_free_pages(int gfp_mask，unsigned long order)

其中 gfp_mask 是申请页面的优先级。

Order 是申请页数 2 的 order 次幂个物理页，也即页块在 free_area 数组中的索引。

2. 函数 free_pages 用于页块的回收，其定义如下：

voidfree_pages(unsignedlongaddr，unsignedlongorder)

这里：addr 是要回收的页块的首地址；

order 指出要回收的页块的大小为 2 的 order 次幂个物理页。

3.1.2 物理页的换出和废弃

当物理内存缺乏的时候，Linux物理内存管理器必须尝试释放某些物理页。这个任务由内核交换守护进程kswapd负责完成。内核交换守护进程（kswapd，定义在mm/vmscan.c）在系统启动时由内核的init进程创建并启动。除了被调用，kswapd进程还会定时启动。Kswapd工作分两部分。

1、检测物理内存剩余的情况，如果短缺，则按LRU（最近最少使用）策略断开active_list队列中部分可交换页面的映射，使页面变为不活跃状态，链入inactive_clean_list队列或者inactive_dirty_list队列，为页面的换出作好准备。

2、每次都执行的，把inactive_dirty_list中的页面（“脏”页面）写入交换设备，并且回收一部分inactive_clean_list（不活跃“干净”）中的页面。

Kswapd函数调用层次图如下：

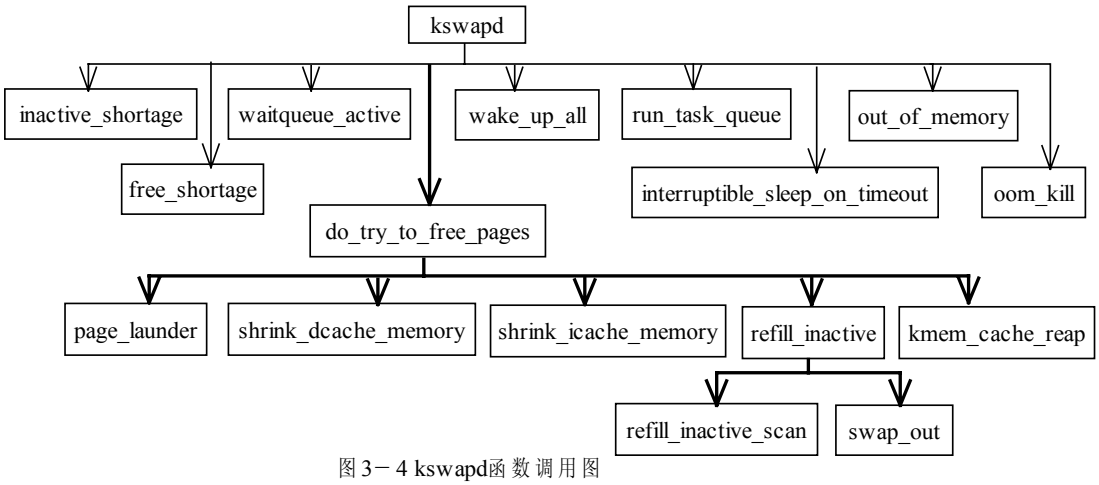

图3－4 kswapd函数调用图

一般情况下，该进程处于等待状态，它等待内核的交换计时器到期（每秒一次）。每一次计时器到期，交换守护进程就检查系统中的空闲页数是否太少。系统中应维持的物理页面供应量由两个全局量确定，那就是 inactive_target 和 free_pages.high，分别为空闲页面的数量和不活跃的数量，二者之和为正常情况下潜在的供应量。这些内存页面的来源则有三个方面。

1、当前尚存的空闲页面，这是立即就可以分配的页面。这些页面分散在各个页面管理区中，并且合并成地址连续、大小为 2、4、…、2^N 个页面的页面块，其数量由 nr_free_pages()来统计。

2、现有的不活跃“干净”页面，这些页面本质上也是马上就可以分配的页面，但页面的内容可能还会用到，所以多保留一些这样的页面有助于减少从交换设备的读入。这些页面也分散在各个页面管理区中，但不合并成块，其数量由 nr_inactive_clean_pages()来统计。

3、现有的不活跃“脏”页面，这些页面要加以“净化”，即写入交换设备后才能投入分配。这种页面全都在同一个队列中，内核中的全局量 nr_inactive_dirty_pages 记录着当前此类页的数量。

仅维持潜在的物理页面供应总量还不够，还要通过 free_shortage()检查是否有某个具体管理区中有严重的短缺，即直接可供分配的页面数量是否小于一个最低限度。

如果发现可供分配的内存页面短缺，那就要通过 do_try_to_free_pages()来释放、换出和回收若干页。回收的页通常被备份到磁盘交换文件中。为了提高效率，这里一次要回收多个页（SWAP_CLUSTER_MAX，32），并将要备份的页集中起来，一次写入磁盘上的交换文件。

下面对 page_launder()、shrink_dcache_memory()、shrink_icache_memory()、refill_inactive()、kmem_cache_reap()分别进行描

述：

一、page_launder()试图把已经转入不活跃状态的“脏”页面“洗净”，使它们变成立即可以分配的页面。将活跃页面的映射断开，使之转入不活跃状态，甚至进而换出到交换设备上，这种代价很大，应该尽量避免，所以能够不动活跃页面是最理想的。

对不活跃状态“脏”页面，依次作如下检查并作相应处理：

1、有些页面虽然已经进入不活跃“脏”页面队列，但是由于情况已经变化，或者其他错误原因而进入这个队列，就需要回到活跃页面队列中。这样的页面有：

A、页面在进入了不活跃“脏”页面队列之后又受到了访问，即发生了以次页面为目标的缺页异常，从而恢复了该页面的映射。

B、页面的生命周期还未到期。

C、页面并不用作读/写文件的缓冲，而页面的使用计数却又大于 1。

D、页面映射表中有映射。

E、页面在受到进程用户空间映射的同时又用于 ramdisk，即用内存空间模拟磁盘。

2、页面已被锁住，这表明正在对此页面进行操作，这样的页面应该留在不活跃“脏”页面队列中，但是把它移到队列的尾部。

3、如果页面仍然是“脏”的，原则上要将其写出到交换设备上。

4、如果页面不再是“脏”的，并且又是用作文件读/写缓冲的页面，则先使它脱离不活跃“脏”页面队列，再通过 try_to_free_buffers（）试图将页面释放。如果不能释放则根据返回值将其退回不活跃“脏”页面队列，或者链入活跃页面队列，或者不活跃“干净”页面队列。

5、如果页面不再是“脏”的，并且在某个 address_space 数据结构队列中，这就是已经“洗清”的页面，所以就把它转移到所属区间的不活跃“干净”页面队列中。

6、最后，如果不属于上述的任何一种情况，那就是无法处理的页面，所以把它退回活跃页面队列中。

如果经过 page_launder()以后，可分配的物理页面数量仍然不足，那就要进一设法回收页面了。不过，也并不是单纯地从各地进程的用户空间所映射的物理页面中回收，而是从四个方面回收，这也就是调用如下四个函数 shrink_dcache_memory()、shrink_icache_memory()、refill_inactive()、kmem_cache_reap()的意图。

二、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的 Inode 数据结构。这些数据结构在文件关闭以后并不

立即释放，而是放在 LRU 队列中作为后备，以防在不久将来的文件操作中又要用到。这样经过一段时间以后，就有可能积累起大量 dentry 数据结构和 inode 数据结构，占用数量可观的物理页面。这时就要通过 shrink_dcache_memory()和 shrink_icache_memory()适当加以回收，以维持这些数据结构与物理页面间的“生态平衡”。

三、在函数 refill_inactive()中，先用 kmem_cache_reap()来回收由 slab 机制管理的空闲物理页面，接着是一个 do-while 循环；在循环中做了两件事：

1、通过 refill_inactive_scan()扫描活跃页面队列，试图从中找到可以转入不活跃状态的页面；接着试试回收用于 dentry 结构和 inode 结构缓冲的页面。

2、通过 swap_out（）找到一个进程，然后扫描其映射表，从中找出可以转入不活跃状态的页面。只有在上述几种方法都不行的时候才会调用 swap_out(priority，gfp_mask)函数。

交换进程轮流检查系统中的每一个进程，试图找到一个最合适可以用于交换的。并且有可以从内存中交换出去或废弃的一个或多个页。找到了就扫描这个进程的页面映射表，将符合条件的页面暂时断开对内存页面的映射，或进一步将页面转入不活跃状态，为把这些页面换出到交换设备上作好准备。

swap_out（）函数所做的工作如下：

1) 从 init_task.next_task 开始，遍历系统中所有的进程，从中选择一个可交换的（swappable 域的值非 0）、mm->swap_cnt 值最大的进程，作为此次交换的候选进程。

2) 调用函数 swap_out_mm(best，gfp_mask)，从选定进程的地址空间中选择页，将其交换出去，将页归还给物理内存管理器。

- 进程的 mm->swap_address 域中记录了上次从该进程中交换页块时最后搜索的物理页的首地址。本次交换从该地址开始。首先确定该地址所在的内存区域（vm_area_struct）。
- 顺序搜索该内存区域所对应的页目录和页表，每次一页。先找出该页对应的 mem_map_t 数据结构。
- 如果该页不在内存或超出了物理内存界限，则查看下一页。
- 如果该页曾经被存取过（页表项的 Accessed 位为 1），则清除该位，将该页的 mem_map_t 数据结构中的 flags 标志的 PG_referenced 位置为 1，表示该页被引用，不能交换，查看下一页。
- 如果该页是保留的、锁定的或不能满足回收需求的页，则查看下一页。
- 如果该页在交换缓冲区中，则修改页表项，将其归还给物理内存管理器。

- 如果该页不是脏页（没有被写过），则修改页表项，并将其归还给物理内存管理器。不论交换成功与否，进程的 mm->swap_address 中都记录最后搜索的物理页的首地址。
- 如果该页所在区域的 vm_area_struct 结构中定义了 vm_ops->swapout 操作，则修改页表，调用该操作将页换出，将其归还给物理内存管理器。每一个进程的虚拟内存区域都可以拥有自己的交换操作（由 vm_area_struct 中的 vm_ops 指针指示），如果这样，交换守护进程会使用它特定的方法来处理脏页。
- 真正的可交换的脏页。为其分配交换项、将其加入交换缓冲区、修改页表、修改 mem_map_t 数据结构中的 flags 标志、通过物理过程将其写入交换文件、归还给物理内存管理器。即在交换文件中为其分配一页，并把此页写到该文件中。
- 只有真正交换出去了脏页才终止该函数，否则将一直继续下去，直到搜索完该进程的所有地址空间。

3) Linux 不会把选定进程的所有可以交换出去的页都交换出去，而只是去掉少量的页。如果页在内存中锁定，则不能被交换或废弃。来自于映像文件的执行映像的大部分内容可以从映象文件中重新读出来。

4) 上述过程是个多重循环，进程、内存区域、页目录、页中间项、页表。

5) 当某页被交换出去时，此页的页表条目会被一个无效的条目替换，但其中包含了此页在交换文件的信息：此页所在的交换文件以及它在文件内的偏移量。不管什么方式交换，原来的物理页被释放并被放回到 free_area 中。干净（或不脏）的页可以被废弃，放回到 free_area 中重用。

四、kmem_cache_reap()用来回收由 slab 机制管理的空闲物理页面。系统中建立的所有 cache 已用 struct list_head next 链成了一个循环链表，此处顺序搜索各 cache。每次搜索以 10 个 cache 为限，找出一个无用 slab（所有对象都空闲，即 s_inuse 为 0）最多的 cache，将它的空闲 slab 页归还给物理内存管理器。

如果从可交换进程中，交换或废弃了足够多的页，交换守护进程会重新睡眠。下一次被唤醒时，它会考虑系统中的下一个进程。这样，交换进程交换出每一个进程的物理页，直到系统重新达到平衡。这种做法比交换出整个进程更公平。

3.2 内核内存管理器

由于内核经常需要小内存用于建立各种管理机构，而物理内存管理器过于粗

放，所以Linux提供了内核内存管理器，专门负责内核中小内存的分配和回收，Linux的内核内存管理器采用slab算法。在slab机制中，每种重要的数据结构都有自己专用的动态存储区。分配、释放对象都在该类对象对应的存储区中进行。存储区中空间分配单位既为该类对象大小。存储区中没有任何内部和外部碎片。

Linux2.14中一条对象缓存链由同一种对象的若干slab块构成，而一个slab块可包含若干页面。每一个对象缓存链前端由kmem_cache_t结构控制，一个slab块由slab_t结构控制。kmem_cache_t与若干slab_t结构链接为一条双向循环链表。链表逻辑上分为三截，第一截的slab块中所有空间均已分配;第二截中的slab块空间部分已分配；第三截中的slab块为全空。kmem_cache_t结构中有一条指针firstnotfull指向第二截中的第一个slab块；在每个slab的描述结构中有一个数据项，是指向该slab上的第一个空闲对象的指针。slab_t结构中维护着本slab块中的空闲对象块链表。Slab分配器如图：

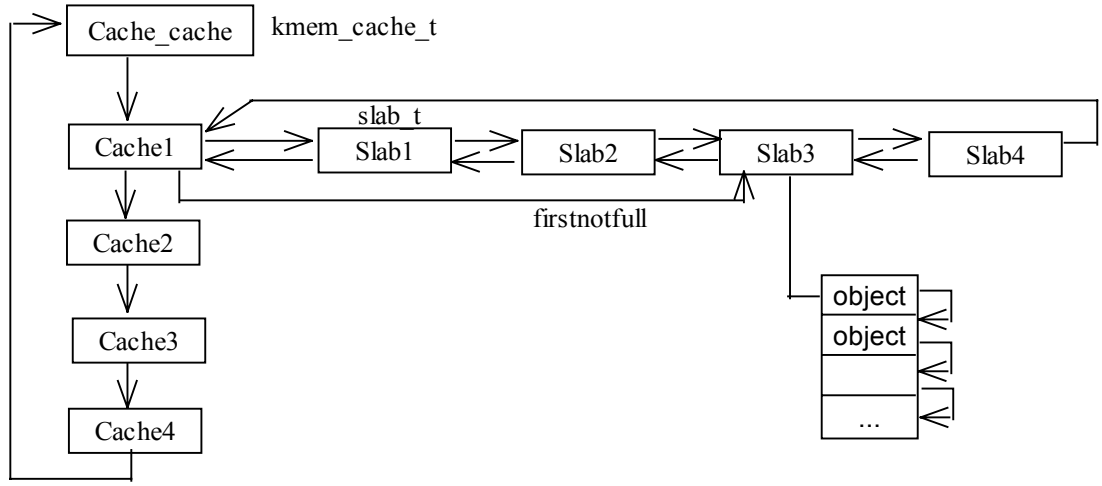

图3-5 高速缓存、slab及对象之间的关系

当申请空间时，必须明确指定需在那条对象链表中分配(即隐含指定该对象的类型)。在链表中也不需要搜索，直接将slab中的第一个空闲对象块取下即可。相应在释放对象时直接将其入链。不过由于slab链表有三截结构，在slab块全满、有空、全空时需将其在链表中的位置做相应调整。这只需调整指针，所以不需要花费很多的时间。

当对象链中所有slab块均满时不能满足请求时，将扩展一个slab块。然而slab机制不显示的收缩对象链。而将其留给内核线程kswapd周期性地在内存紧缺时收割。

3.2.1 cache 的建立

一个 cache 用一个数据结构（kmem_cache_s）描述，该数据结构中包含一个 cache 的所有信息，如：cache 名、状态标记、其中缓存的每个内存块的原始大小、对齐标准、每个 slab 占用的物理页数、每个 slab 的对象个数、当前用掉的

该 cache 的对象数等等。在使用一个 cache 之前必须先建立该 cache 对应的 kmem_cache_s 数据结构。

因为每个 cache 都要用 kmem_cache_s 数据结构描述，而且在系统中要建立许多 cache，因此最好也用一个 cache 来缓存系统中的所有 kmem_cache_s 数据结构，这个 cache 称为 cache_cache。cache_cache 是一个静态结构，在编译时已经建立起来，系统中所有的 cache 数据结构都在其中分配。

函数 kmem_cache_create 用于 kmem_cache_s 数据结构的建立。

该函数的定义如下：

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

其中的参数给出了要建立的 cache 中的关键数据。该函数所做的工作如下：

1. 检查各参数的合法性。
2. 调用函数kmem_cache_alloc，从cache_cache中申请一个内存块用于建立新cache的kmem_cache_s数据结构。
3. 根据size的值计算新cache的原始内存块的大小（必须字对齐，以提高内存访问效率）。
4. 根据对象大小将cache分为小对象（小于512字节）cache和大对象（512字节到32页）cache。对小对象cache，在创建其slab时，将slab数据结构和slab对象建在同一个物理页上；对大对象cache，为了提高内存利用率，在创建其slab时，将其slab数据结构建立在从另外的地方另申请的一块内存上。小对象和大对象cache的数据结构中有些域的值是不同的，计算并填写新cache数据结构的各个域。
5. 将新cache的数据结构连入系统中已有的kmem_cache_s数据结构链表中。

在系统初始化时，已经使用该函数为十多个常用的数据结构建立了 cache。如：slab_cache，其中的每个对象都可作一个 kmem_slab_s 数据结构；另外，创建了对象大小分别为 32、64、128、256、512、1024、2048、4096、8192、16384、32768、65536、131072 的几个 cache，以支持经常使用的内核内存分配，这几个 cache 叫通用 cache。

3.2.2 为 cache 增加 slab

当要从某个 cache 中分配内存块时，如果还没有为该 cache 建立 slab，或者该 cache 的 slab 中已经没有可用的空闲对象，则要调用函数 kmem_cache_grow

为 cache 增加 slab。

函数 kmem_cache_grow 的定义如下：

int kmem_cache_grow(kmem_cache_t *cachep, int flags)

它所做的工作如下：

1. 检查标志flags的合法性；
2. 从物理内存管理器中申请物理页，一次申请的物理页的个数由该cache的kmem_cache_s数据结构中的gfporder域指定；
3. cache的slab中每个对象的大小，对于小对象cache，在物理页的最后建立slab数据结构；对于大对象cache，在一个专用的缓冲区cache_slabp中另申请一块内存用于建立slab数据结构；
4. 对大对象cache，在一个专用的缓冲区（由域slabp_cache指出）中另申请一块内存用于建立slab对象链表（kmem_bufctl_t 数据结构）；
5. 修改申请到的每个物理页对应的mem_map_t数据结构中的next和prev指针(用于在free_area表中建立双向链表),使它们都指向该kmem_cache_s数据结构，同时设置它的标志域flags，表示该物理页正在用作slab；
6. 初始化各slab对象，将它们串成一个链表，让slab数据结构的free指针指向其第一个空闲对象，同时填写slab数据结构的其它域；
7. 将新slab加入cache的slab双向链表中，同时调整cache的firstnotfull指针。

如此以来，就根据kmem_cache_s数据结构的定义，为该cache创建了一个可用的、新的slab，以后就可以在其上分配内存对象了。

3.2.3 对象的分配

有两个函数用于slab对象的分配：

1、kmalloc(size_t size, int flags)

当只知道要申请的内存大小和标志(特殊要求)时，使用该函数。函数kmalloc首先根据申请内存的大小size确定一个通用cache（系统初始化时定义），而后调用函数__kmem_cache_alloc从指定的cache中分配一块对象。通常用于无特定目的的内存分配。

2、kmem_cache_alloc(kmem_cache_t *cachep, int flags)

当知道对象所在的cache时使用该函数。函数kmem_cache_alloc直接调用函数__kmem_cache_alloc从指定的cache中分配一块对象。通常用于有特定目的的内存分配。如，要建立一个slab数据结构，则从缓冲区cache_slabp中分配内存。由于每个专用的cache通常只用做一种数据结构，所以可以对其作一些特殊的假设（某些域的值已经初始化等）和特殊的处理，以加快数据结构创建的速度。这

也是基于cache和slab的内核内存管理器的一个突出的优点。

两种分配函数最终都规约到了对函数__kmem_cache_alloc的调用，该函数中才真正完成内核内存的分配。函数__kmem_cache_alloc的定义如下：

void*__kmem_cache_alloc(kmem_cache_t*cachep，int flags)

其中：cachep是要申请的内存块所在的缓冲区；flags是对内存块的特殊要求，如该块将要用做DMA区等。该函数所做的工作如下：

1. 如果cachep为NULL则错；
2. 找出该cache中有空闲对象的第一个slab：

   slabp = cachep->firstnotfull;
3. 如果还没有为该cache创建slab或该cache中所有slab都没有空闲对象，则调用函数kmem_cache_grow为其创建一个新的slab，创建方法如上；
4. slabp所指的slab中有空闲对象可用，调整该slab数据结构：

   slabp->s_inuse++;

   bufp=slabp->free;

   slabp->free=bufp->list.next;

   如果slabp->free为空（NULL），则说明该slab中已经没有空闲的对象，调整该cache的指针：

   cachep->firstnotfull=slabp->list.next;

   返回一个内存块的首地址：

- 小对象：

  让该对象指向它所在的slab：

  bufp->free=slabp;

  计算内存块的首地址：

  objp=((void*)bufp)-cachep->colour_off;

  返回：

  return objp;

- 大对象：

  对大对象来说，slab页是一个对象数组，每个对象的大小都一样。Slab数据结构定义在另外的地方，用一个kmem_bufctl_t数据结构数组表示该slab中对象的使用情况，该数组也定义在另外的地方，slabp->slabp_cache指向该数组。此时slabp->firstnotfull所指的是kmem_bufctl_t数据结构数组中的一个元素，该元素在kmem_bufctl_t数据结构数组中的索引就是空闲对象在slab页的对象数组中的索引。因此空闲内存块的首地址为：

```
objp = slabp->s_mem + slabp->free*cachep->objsize;
```

调整kmem_bufctl_t数据结构数组中对应元素，让它记录该首地址：

```
slabp->free=slab_bufctl(slabp)[slabp->free];
```

返回：

```
return objp;
```

3.2.4 对象的回收

Linux提供了几种回收对象的方法：

1. kmem_cache_free(kmem_cache_t*cachep，void*objp)

用于回收cache cachep中的对象objp。直接调用函数__kmem_cache_free完成具体的工作。

2. kfree(const void*objp)

如果只知道要回收的内存块的首地址，则通过此函数回收对象。所做的工作如下：

算出该地址所对应的物理页号：

```
c = GET_PAGE_CACHE(virt_to_page(objp));
__kmem_cache_free(c, (void*)objp);
```

二种对象回收操作最后都规约到对函数__kmem_cache_free的调用，该函数才完成真正的对象回收工作，其定义如下：

void__kmem_cache_free(kmem_cache_t *cachep，const void *objp)

这里：cachep是对象所属的cache，而objp是对象的首地址。

该函数所做的工作如下：

1. 小对象回收：
   1) 该对象的尾部有一个指针，它指向该对象所属的slab，找到该slab：

   ```
   unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
   slab_bufctl(slabp)[objnr] = slabp->free;
   slabp->free = objnr;
   ```

   2) 将该对象加入slab的空闲对象队列中：

   ```
   slabp->s_inuse--;
   list_del(&slabp->list);
   list_add_tail(&slabp->list, &cachep->slabs);
   ```

   3) 如果回收以前该slab没有全部被分完，且回收以后该slab不是全部空闲，则正常返回。
   4) 如果回收以后该slab全部空闲，则调用函数kmem_cache_full_free，调整其cache的空闲队列。即将该slab移到cache的slab队列的队尾，

以便归还给物理内存管理器。返回。

5) 如果回收以前该slab已被全部分完，则调用函数kmem_cache_one_free，调整其cache的空闲队列。即将该slab后移，以保证cache的slab队列的顺序，必要时调整cache的firstnotfull指针。返回。

2. 大对象回收：

1) 根据对象地址objp，找到对象所在的物理页，并据此找到该对象所属的slab：

```
slabp = GET_PAGE_SLAB(virt_to_page(objp));
```

2) 从该slab中算出对象的首地址：

```
unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
```

转小对象2)。

3.2.5 回收指定cache的空闲slab

与函数kmem_cache_grow相对应，函数kmem_cache_shrink用于收缩一个cache的空间，即将其中完全空闲的slab归还给物理内存管理器。该函数的定义如下：

```
int kmem_cache_shrink(kmem_cache_t*cachep)
```

其中cachep是要收缩的cache。该函数所做工作如下：

1. 从cache_cache开始搜索cache循环链表，找出cachep的前一个cache：

```
searchp=&cache_cache;
for(;searchp->next!=&cache_cache;searchp=searchp->next){
        if(searchp->next!=cachep)
            continue;
        if(cachep==clock_searchp)
            clock_searchp=cachep->next;
        goto found;
}
```

2. 从cachep的最后一个slab开始，顺序向前搜索，将完全空闲的slab归还给物理内存管理器：

```
while(!cachep->growing){//该cache不是正在增长
    slabp=cachep->slabs.prev;   //最后一个slab
    if(slabp->inuse||slabp==kmem_slab_end(cachep))
        break;
    kmem_slab_unlink(slabp);//将slab从链表中摘下
```

```
spin_unlock_irq(&cachep->spinlock);
kmem_slab_destroy(cachep, slabp); //将slab归还给物理内存管理器
spin_lock_irq(&cachep->spinlock);
}
```

因为一个cache的空闲slab总在队列的最后，所以这段程序可以将该cache的所有空闲slab一起释放。

3. 函数kmem_slab_destroy所做的工作如下：
   1) 如果该cache定义有析构函数（cachep->dtor），则对要释放的slab中的每个对象都调用一次析构函数。
   2) 清除该slab所占用的物理页对应的mem_map_t数据结构中的PG_Slab标志，将这些物理页归还给物理内存管理器。
   3) 对大对象，调用函数kmem_cache_free，回收其slab数据结构和slabp_cache指针数组。

3.2.6 回收 cache 中的空闲 slab

函数 kmem_cache_reap 从系统的前 10 个 cache 中选择一个空闲 slab 最多的 cache，将它的空闲 slab 归还给物理内存管理器。该函数的定义如下：

```
void kmem_cache_reap(int gfp_mask)
```

其中 gfp_mask 是对要归还的 slab 的特殊要求。该函数所做的工作如下：

1、从 clock_searchp 开始顺序搜索各 cache，满足以下条件的 cache 不予考虑：标志为 SLAB_NO_REAP 的 cache、正在增长的 cache、已经损坏的 cache、不满足 gfp_mask 要求的 cache。

2、对满足条件的 cache，从它的最后一个 slab 开始统计其中空闲 slab 的数量，找出一个空闲 slab 最多的 cache。

3、从该cache的最后一个slab开始，顺序向前，逐个调用函数kmem_slab_destroy，将各 slab 所占用的物理内存归还给物理内存管理器。

4、如果 gfp_mask 标志中要求的是 DMA 内存，则只归还标志为 GFP_DMA 的空闲 slab。

5、该函数通常由内核交换守护进程调用，用于从 cache 中回收物理页。

3.2.7 slab 分配模式应用实例

分配和释放数据结构是所有内核中最普遍得操作之一。为了便于数据的频繁分配和回收，编程者常常会用到一个空闲链表。该空闲链表包含有可使用的、已经分配好的数据结构块。当代码需要一个新的数据结构实例时，就可以从空闲链

表中抓取一个，而不需要分配内存，再把数据放进去。当不再需要这个数据结构实例时，就把它放回空闲链表，而不是释放掉它。因此，空闲链表相当于对象高速缓存区以便快速存储频繁使用的对象类型。Linux 把高速对象缓存区分为专用和通用。它们分别用于不同的目的。

一、slab 专用对象缓存区的建立和释放

专用对象缓存区主要用于频繁使用的数据结构，如 task_struct、mm_struct、vm_area_struct、file、dentry、inode 等。

下面考察 task_struct 结构使用的例子。

1、首先，内核用一个全局变量存放指向 task_struct 缓冲区的指针：

```
static kmem_cache_t  * task_struct_cachep;
```

内核初始化期间，在 fork_init()中如下创建缓冲区：

```
/* create a slab on which task_structs can be allocated */
task_struct_cachep = kmem_cache_create("task_struct", sizeof(struct
      task_struct),  ARCH_MIN_TASKALIGN, SLAB_PANIC, NULL, NULL);
```

这样就创建了一个名为 task_struct_cachep 的缓冲区，其中存放的就是类型为 struct task_struct 的对象。对象被创建在 slab 中默认的偏移量处，并完全按缓冲区对齐。没有构造函数和析构函数。

2、每当进程调用 fork()时，一定会创建一个新的进程控制块，这是在 dup_task_struct()中完成的：

```
# define alloc_task_struct()
    kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
struct task_struct *tsk;
tsk = alloc_task_struct();
```

3、进程执行完后，如果没有子进程在等待，它的进程控制块就会释放，并返回给 task_struct_cachep slab 缓冲区，这是在 free_task_struct()中执行的（tsk 是指向现有进程的控制块指针）：

```
# define free_task_struct(tsk)
    kmem_cache_free(task_struct_cachep, (tsk));
free_task_struct(tsk);
```

4、由于进程程控制块是内核的核心组成部分，时刻都要用到，因此 task_struct_cachep 缓冲区绝不能销毁。

如果当内核卸载一个模块时，应当同时撤销为这个模块中的数据结构所建立的缓冲区，可通过下列函数：

```
int kmem_cache_destroy (kmem_cache_t * cachep);
```

如果要频繁创建很多相同类型的对象，那么就应该考虑使用 slab 缓冲区。而不要去手工实现空闲链表。

二、通用对象缓存区的建立和释放

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。通用缓冲区类似于物理页面分配中的大小分区,最小为 32 字节,然后依次为 64、128、...，直到 128KB（即 32 个页面），但是，对通用缓冲区的管理又采用 slab 方式。通用缓冲区中分配和释放缓冲区的函数为：

```
void * kmalloc(size_t size, gfp_t flags);
void kfree(const void *objp);
```

当一个数据结构的使用频度很低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 kmalloc()进行分配。如果数据结构的大小接近一个页面，则通过__get_free_pages()为之分配一个页面。

在内核中，尤其是驱动程序中，又大量的数据结构仅仅是一次性使用，而且所占用内存只有几十个字节，因此一般情况下调用 kmalloc()给内核数据结构分配内存就足够了。

下面看一个中断处理程序中分配内存的例子。假设中断处理程序想分配一个缓冲区来保存输入数据。BUF_SIZE 预定义为缓冲区长度，应大于两个字节。

```
char * buf;
buf = kmalloc(BUF_SIZE, GFP_ATOMIC);
if (!buf)
    /* 内存分配出错 */
```

当不再需要这个内存时，应该释放它：

```
kfree(buf).
```

3.3 进程虚拟内存管理器

在 IntelCPU 硬件平台上，Linux 操作系统对每个进程支持 4G 的虚拟地址空间，其中 0 到 3G 是每个进程私有的，用来存放进程的正文段、数据段和堆栈段等，进程可以直接对其访问，各个进程空间互不干扰。3G 到 4G 是所有进程共享的，用来存放内核的正文段、数据段和栈段等等，用户态进程不可访问。它们之间的关系如图所示：

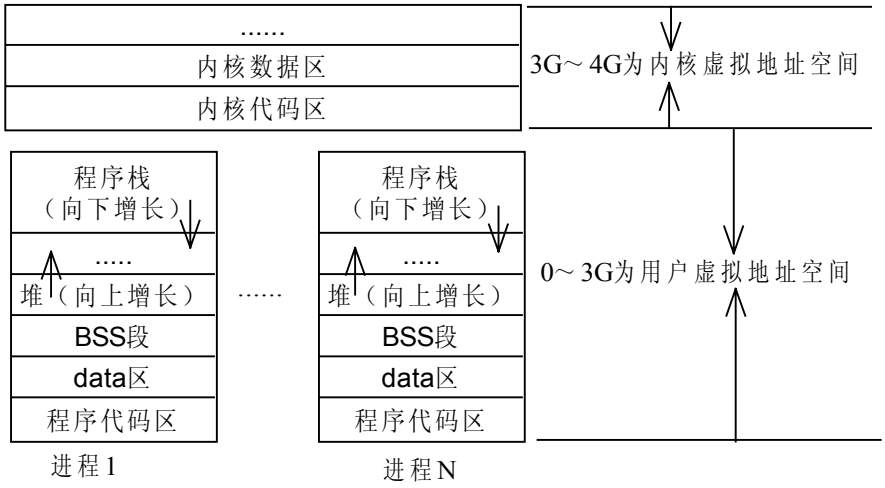

图3－6 Linux下进程地址空间

3.3.1 虚拟内存抽象模型

一、进程空间的核心数据结构

每个进程都有各自的虚存空间。每个进程的虚拟内存用一个mm_struct来表示，在内核代码中，用于这个数据结构（指针）的变量名通常是mm。mm_struct数据结构是整个进程用户空间的抽象，也是总的控制结构。它描述了一个进程的页目录，有关进程的上下文信息，以及数据、代码、堆栈在其虚拟地址空间中的起始、结束地址，还有虚拟存储区的数目以及内存区域结构的链表和树。

mm_struct数据结构（include\linux\sched.h）定义如下：

```
struct mm_struct{
    struct vm_area_struct *mmap;    /*虚存区间结构的单链表*/
    struct vm_area_struct *mmap_avl; /*虚存区间结构的平衡二叉树*/
    struct vm_area_struct *mmap_cache;/*最近用到的虚存区间结构*/
    pgd_t *pgd;                     /*指向该进程的页面目录*/
    atomic_t mm_users;              /*用户空间的用户个数*/
    atomic_t mm_count;              /*对这个结构的应用计数*/
    int map_count;                  /*虚存区间的个数*/
    struct semaphore mmap_sem;      /*用于P、V操作的信号量*/
    spinlock_t page_table_lock;
    struct list_head mmlist;        /*激活的mm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_cnt;  /*numberofpagestoswaponnextpass*/
    unsigned long swap_address;
    mm_context_t context;
};
```

其中的页目录域pgd指出了该进程的页目录所在的物理内存地址，通过它可以方便地访问到该进程的页目录进而访问到它的所有页表。

指示该进程内存区域的有三个指针：

- mmap所指是一个vm_area_struct数据结构链表，它按地址从小到大的顺序将一个进程的所有vm_area_struct数据结构连接了起来。
- mmap_avl 所指是vm_area_struct数据结构的一个AVL（Adelson-VelskiiandLandis）树，利用该树型结构可以快速地查找到一个vm_area_struct结构。当一个进程的内存区域块较多时（如大于32）才建立AVL树。
- mmap_cache指针指向find_vma()最后一次查找到的vm_area_struct结构。下一次再查找vm_area_struct结构时，首先看该指针所指的结构是否满足要求。根据内存访问的局域性，这种查找通常有很高的命中率（35%）。

vm_area_struct 数据结构（include\linux\mm.h）定义如下：

```
struct vm_area_struct{
    struct mm_struct *vm_mm; /*所属的地址空间*/
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct *vm_next;
    pgprot_t vm_page_prot;
    unsigned short vm_flags;
    short vm_avl_height;
    struct vm_area_struct *vm_avl_left;
    struct vm_area_struct *vm_avl_right;
    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;
    struct vm_operations_struct *vm_ops;
```

```
    unsigned long vm_offset;
    struct file *vm_file;
    unsigned long vm_pte;           /*sharedmem*/
};
```

一个进程的虚拟地址空间通常被划分成一组内存区域，一个vm_area_struct数据结构描述了一块这样的区域，对同一块区域的处理有着相同的规则。一个进程的所有内存区域（vm_area_struct数据结构）合在一起就描述了该进程的所有有效虚拟地址，在这些区域外的所有地址都是非法的。

该结构的两个域：vm_start和vm_end描述了该区域在其虚拟地址空间中的位置；域vm_file记录了该区域所映射的文件，而vm_offset表示该区域在映射文件中的开始位置；域vm_page_prot是属于该内存区域的页的缺省保护权限（只读、只写、执行等）；域vm_flags是该区域的属性（读、写、共享等）；域vm_ops是一张表，表示在特殊情况下对该区域的页应采取的操作（如，nopage处理缺页、swapout换出页、swapin换入页等）。

Linux 在加载进程的时候，把进程的不同部分加载到进程私有虚拟地址空间的不同部分中。表示进程有效虚拟地址空间的数据结构是 vm_area_struct，每个进程的地址空间由多个 vm_area_struct 组成，这些 vm_area_struct 通过指针按地址降序链接起来形成进程的虚拟地址空间链，每个 vm_area_struct 实例表示的虚拟地址空间都是连续的。为了可以快速定位某个虚存区，Linux 还利用了 AVL 平衡树来组织 vm_area_struct[3][4][5][6]。

mm_struct、vm_area_struct 和页目录、页表是描述虚拟内存的主要数据结构，它们和有效虚拟地址空间 VMA 之间的关系如下：

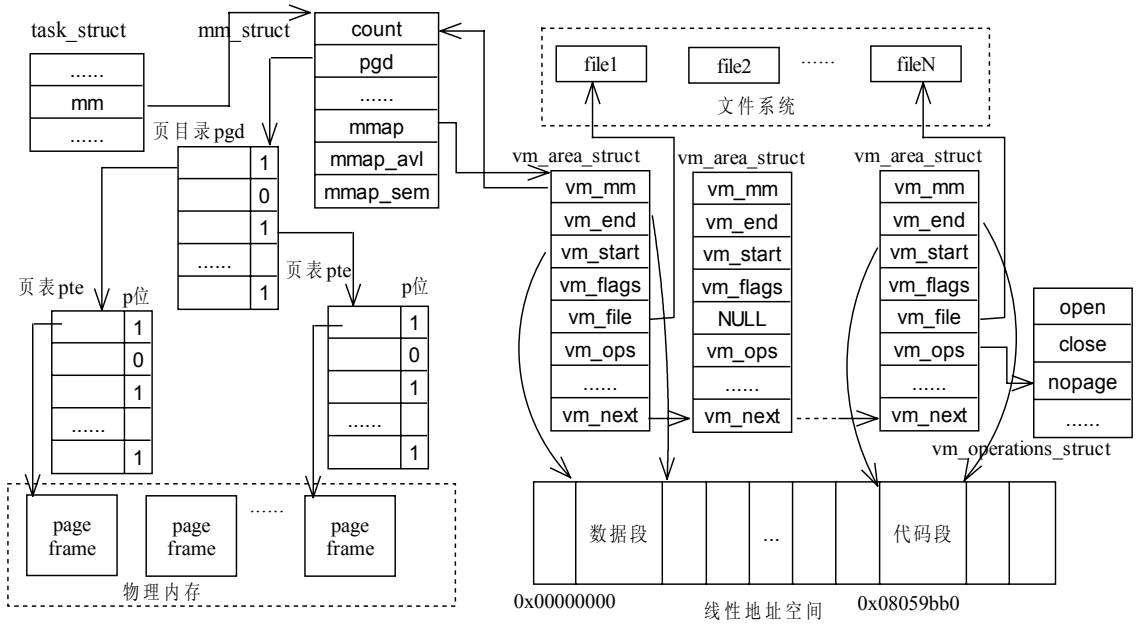

图3－7 虚拟内存数据结构之间的关系

进程的有效虚拟地址空间 VMA 区，在操作系统加载进程的时候大小就基本确定，但在进程执行的过程中，可以动态的增长，有时候需要交换扩展。每个 VMA 区要么与磁盘上的普通文件相关联，要么与交换设备或交换文件相关联。

二、Linux 的分页机制

当应用程序访问一个虚拟地址时，首先必须将虚拟地址转化成物理地址，然后处理器才能解析访问请求，内存管理子系统的主要工作之一就是完成虚实地址映射。在实际程序执行时，处理器硬件使用由操作系统维护的一组表格（页目录、页表等）中所保存的信息将虚拟地址转换为物理地址。

为了保证可移值性，Linux64 位操作系统中采用三级页表来保存这种映射关系[7][8][9][10][11][12][13]。

- 1、顶级页表是页目录(PGD)，PGD 包含了一个 pgd_t 类型数组，其中的表项为无符号长整型。PGD 中的表项指向二级页目录中的表项：PMD。其大小为 4K 的表，每一个 process 只有一个页目录，以 4 字节为一个表项，分成 1024 个表项(或称入口点)；该表项的值为所指页表的始地址。32 位虚拟地址的第 1 个子位段共 10 位，其值的范围从 0 到 1023，对应于页目录的一个入口点。
- 2、二级页表是中间页目录(PMD)。PMD 是个 pmd_t 类型数组，其中的表项指向物理页面 PTE 中的表项。
- 3、最后一级的页表简称为页表（PTE），其中包含了 pte_t 类型的页表项，该页表项指向物理页面。页表（PTE）的每一个入口点的值为此表项所指的一页框（pageframe），32 位虚拟地址的第 2 个子位段共 10 位，其值的范围从 0 到 1023。

  页框（pageframe）并不是物理页，它指是虚存的一个地址空间。

为了让这种地址转换更简单，Linux 将虚拟内存和物理内存都分为适当大小的块，叫做页（page）。每一页都赋予一个唯一编号：PFN 页编号（pageframenumber)。

标准 Linux 中使用三级页表完成地址转换，转换关系图如下：

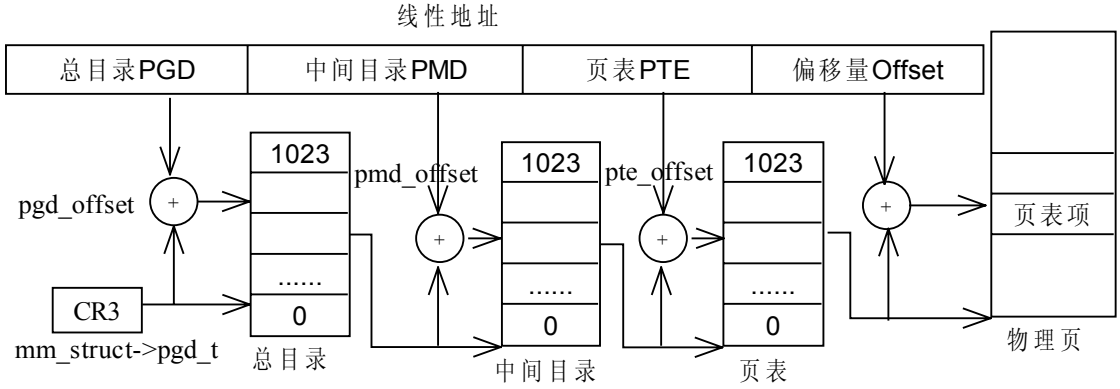

图3－8 Linux的三级分页模式

为了将虚拟地址转换为物理地址，处理器首先从虚拟地址中分离出各级页编号和偏移量，然后：

1、pgd_offset(mm_struct，address)返回 mm_struct 地址空间中某个地址 address 映射所在的页全局目录 PGD，

2、pmd_offset(pdg，address)返回页全局目录 PGD 中包含 address 映射的页中间目录 PMD，

3、最后通过 pte_offset(pmd，address)返回保存该 address 映射关系的页表项 PTE。

4、将物理页的基地址加上偏移量，就得到了该虚拟地址对应的物理地址。

第一级页表的基地址通常在系统的 CR3 寄存器，在进程运行以前已经设置好。

在只支持两级页表的硬件中，也可以使用三级系统，i386 机器上 Linux 的页表结构就为两级，PGD 和 PMD 页表是合二为一。在这种分页模型下，虚拟地址由两部分组成：虚拟页号和页内偏移量。假如页大小是 4K，则虚拟地址的 0 到 11 位是页内偏移量，第 12 以上的位是页编号。每当处理器遇到虚拟地址时，它必须从中提取出偏移量和虚拟页编号，而后将虚拟页编号转换为物理页编号，再根据偏移量访问物理页中的正确物理内存单元。

Linux 运行的每一个平台都必须提供一组转换宏，让内核能处理特定进程的页表。这样，内核就不需要知道页表条目的具体结构和组织情况，这组宏隐藏了具体硬件的差别。

在 Intel 处理器的 pgtable.h 文件中，定义了这组宏。其中第一级页表对应为页目录，第二级页表为空，第三级页表对应为处理器的页表，三级的页表转换宏实际只有两个在起作用。相当于折叠起第二个页表。

页表是一个数组，它的每个元素称为一个页表项（PTE）[14][15][16][17][18][19]。在不同的系统中，页表项的长度不一样，其内容也不一样。在 Intel 系统上，页表项长度为 32 位，4 个字节。

图 3.9 显示了 Intel 的 PTE 结构。

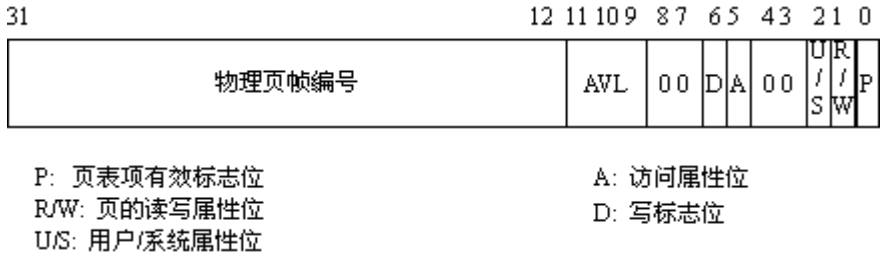

图 3-9 i386 系统中的页表项格式

每一个进程都拥有自己的页表（页目录+页表）。这些页表将进程的虚拟页映射到内存的物理页。

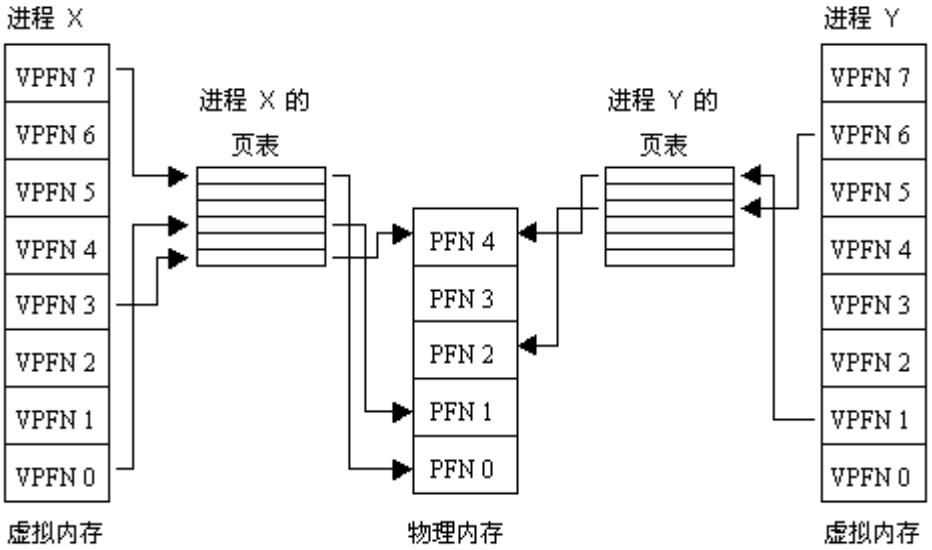

图 3-10 虚拟地址到物理地址的映射模型

图 3-9 显示了两个进程（进程 X 和进程 Y）的虚拟地址空间，及其各自的页表。进程 X 的虚拟页号 0 映射到物理页号 1，而进程 Y 的虚拟页号 1 映射到物理页号 4。

要将虚拟地址转换到物理地址，处理器首先找出虚拟地址的页编号和页内偏移量。对一个有效的页表项，处理器取出其中的物理页号，乘以页的大小，得到物理内存中本页的基地址，再加上它需要的指令或数据的偏移量，就可得到最终的物理地址。

比如，进程 Y 的虚拟页编号 1 映射到了物理页编号 4（起始于 0x4000=4*0x1000），加上偏移 0x194，得到了最终的物理地址 0x4194。

虚拟内存使多个进程可以方便地共享内存。系统中所有的内存访问都要通过页表，而且每一个进程都有自己的页表。当一个物理页编号出现在两个进程的页表中时，这两个进程就都可以访问到该物理内存页，从而共享了该物理内存页。

例如进程 X 和进程 Y 共享物理页号 4 的情形。进程 X 通过虚拟页号 3，而进程 Y 通过虚拟页号 1 都可以访问到同一个物理页 4。

三、高速缓存

高速缓存是获得高性能的常用手段。Linux 使用了多种与内存管理相关的高速缓存，缓冲区类型有缓冲区高速缓存、页高速缓存、交换高速缓存和硬件高速缓存。

缓冲区高速缓存 BufferCache：它包含了块设备驱动程序使用的数据缓冲区。这些缓冲区大小固定（例如 512 字节），其中包括刚从块设备读出的数据或者将要写到块设备的数据。块设备是只能通过读写固定大小的数据块来访问的设备。缓冲区高速缓存由设备号和块标志号索引，来快速定位数据块。块设备只能通过 buffercache 存取。如果数据可以在 buffercache 中找到，那就不需要从物理块设备如硬盘上读取，从而使访问加快。

页高速缓存 PageCache：用来加快对磁盘上映像和数据的访问。它用于缓存文件的逻辑内容，一次一页，并可通过文件和文件内的偏移量来访问。当数据页从磁盘读到内存中时，它们被缓存到页高速缓存中。

交换高速缓存 SwapCache:只有改动过的（或脏 dirty）页才被存储到交换文件中。只要它们写到交换文件之后没有再次被修改，下一次这些页需要交换出去的时候，就不需要再写到交换文件中，因为该页已经在交换文件中，直接废弃该页就可以了。在一个交换比较厉害的系统中，这会节省许多不必要和高代价的磁盘操作。

硬件高速缓存 HardwareCache:硬件高速缓存的常见的实现方法是在处理器里面：PTE 的高速缓存。这种情况下，处理器不需要总是直接读取页表，而是在需要时把页转换表放在缓存区里。CPU 里有转换表缓冲区(TLBTranslationLook-asideBuffers)，其中缓存了系统中一个或多个进程的页表条目的拷贝。

当引用虚拟地址时，处理器试图在 TLB 中寻找匹配的条目。如果找到了，就可用它直接将虚拟地址转换到物理地址，进而对数据执行正确的操作。如果找不到，则需要操作系统的帮助。处理器用信号通知操作系统，发生了 TLBmissing。一个和系统相关的机制将这个异常转给操作系统的相应代码去处理。操作系统为这个地址映射生成新的 TLB 条目。当异常清除之后，处理器再次尝试转换虚拟地址，这一次将会成功因为 TLB 中该地址有了一个有效的条目。

3.3.2 线性空间的建立

Linux在进程通过fork()系统调用来生成子进程时，就为这个进程创建了一个

完整的用户空间。为提高效率，内核调用copy_mm()函数，仅仅为子进程生成一个新的mm_struct结构、vm_area_struct结构、以及页目录和所有页表（见图3-6），而父子进程的页表项完全相同并把所有的页表项都置为写保护。这样，当父子进中任何一个执行写操作时，需要分配一个新页，使子父进程页表分别指向各自的页面，以保证它们的正确性和独立性，这种技术叫做**写时复制（copy-on-write）。**

当通过exec()系统调用开始执行一个进程时，进程的可执行映像（包括代码段、数据段等）必须装入到进程的线性空间。当可执行映像通过do_mmap()函数映射到进程的线性空间时，将产生一组vm_area_struct结构来描述各个虚拟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，可能是可执行码，可能是初始化的变量和未初始化的数据，也可能是刚打开的一个文件。do_mmap()主要供内核使用，它的主要功能是将可执行文件的映象映射到虚拟内存中，或将别的进程中的内存信息映射到该进程的虚拟空间中；用户空间可以通过mmap()系统调用建立由一个已打开文件的内容到用户空间虚存区的映射。

不同的程序空间（代码区和数据区、堆、堆栈）因为其行为和特点不同，Linux采取不同的方法来实现它们的线性空间的建立，其中代码区和数据区通过mmap()，堆通过 brk()，堆栈通过 expand_stack()来实现。

mmap()通过一些简单的 flags 检查后，将调用进来的参数通过一定的转换后变为统一的 do_mmap_pgoff 所需要的参数，然后调用 do_mmap_pgoff()来进行实际的线性空间建立。

do_*map*_pgoff()主要包括以下几个步骤：

1、参数有效性检查，包括参数地址与长度，已经映射的次数等。

2、调用 get_unmapped_area()函数在进程虚拟地址空间中找一块满足申请虚拟内存长度的空闲空间。

3、检查这次 mmap 的 flags 属性和该地址空间所属的文件系统的属性是否一致。

4、如果已经存在一个线性区包含该次申请的空间，则对该线性区进行修改；如果该次分配的线性空间可以通过某个线性区的扩展来实现，则通过 vma_merge 来完成线区的修改。

5、如果步骤 4 不成立，则调用 kmem_cache_alloc()函数在核心内存区申请一块长度为 structvm_area_struct 的内存，用于管理刚申请的虚拟地址空间，并初始化 vma 的一些域。

6、调用 do_munmap()函数清除刚申请的虚拟内存块的旧的映象。

7、如果该地址所属的文件系统定义了自己特定的 mmap，就执行之，来完成一些与该文件系统相关的特殊的操作，并根据文件描述符设置好 vma_area_struct 中的跟文件系统相关的 vm_ops、vm_pgoff 和 vm_file 域。

8、调用 insert_vm_struct()函数将 vma 所管理的虚拟内存空间插入到 vm_struct 的 AVL 树中去，调用 vma_merge()函数将相邻的可以合并的虚拟内存块合并为一个新的更大的内存空间，并用 vma_link()将线性区链接到内核管理虚空间的线性区链表中。

9、如果刚申请的内存的标志字中只了上锁标志，则将上锁的内存区的长度加上新申请的内存的长度，通过 make_pages_present（）为新增的区间建立起对内存页面的映射，并返回得到的虚拟内存的起始地址。

mmap()建立的线性地址空间主要有两种：一种是线性区和文件系统关联，需要该线性区的数据需要从文件得到，这种线性区再分配页时，需要通过页缓冲这条途径；另一种是线性区无文件系统无关，其页分配通过匿名页分配这条途径。brk()用于堆线性空间的分配和释放，因为堆是动态建立和使用的，所以其不需要从文件系统中得到初始数据，其建立的线性区与文件系统无关，该线性区的页分配全部通过匿名页这条途径，所以其操作步骤比 mmap()简单，相当于一个子集，即不需要 mmap 操作中步骤 3 和步骤 7。

expand_stack()用于栈空间的建立，栈空间是不需要用户显示来申请，其由操作系统自动来建立，所以操作系统没有系统调用来实现栈空间的建立，其线性区的建立在缺页异常处理的时候通过调用 expand_stack()来扩展现有的栈空间线性区。

3.3.3 线性空间的释放与修改

Linux 内核通过 do_munmap()从当前进程的地址空间中删除一个线性地址区间。要删除的区间并不总是对应一个线性区：它或许是一个线性区的一部分，或许跨越两个或多个线性区。该函数主要步骤为：

1、通过 find_vma_prev 根据 addr 找到第一块 vma->end>addr 的 vma 块 mpnt。

2、由于解除一部分空间的映射有可能使原来的区间一分为二，所以先调用 kmem_cache_alloc()函数在核心内存区申请一个 vm_area_struct 结构 extra

3、要解除映射的那部分空间也有可能跨越好几个区间，所以要把所有涉及的区间转移到一个临时队列 free 中，如果建立了 AVL 树，则也要把这些区间的 vm_area_struct 结构从 AVL 树中删除。

4、用一个 while 循坏，把 free 队列里的所有虚存区间通过 zap_page_range()释放该线性区所映射的若干连续页面的映射，并且释放所映射的内存页面，或对交换设备上物理页面的引用

5、每释放一个虚存区间的操作，就要调用 unmap_fixup（）对虚存区间的 vm_area_struct 数据结构和进程的 mm_struct 数据结构作出调整。如果整个虚存区间解除了映射，则要释放原有的 vm_area_struct 数据结构；如果原来的区间一分为二，则需要插入一个 vm_area_struct 数据结构。

6、当循坏结束时，解除了一些页面的映射，通过 free_pgtables()释放页表中的内容。

除 do_munmap()会修改线性区，修改页表外。do_mlock()，do_mprotect()，do_mremap()这些系统调用也会页表进行修改。它们功能分别是线性区加锁，修改线性区的属性和移动线性区的位置，其主要步骤基本分为两个阶段:第一阶段，扫描线性区，对相关的线性区进行修改。第二阶段，更新页表。

3.4 缺页（page fault exception）异常处理

当某个可执行映像映射到进程用户空间中并开始执行时，因为只有很少一部分虚存页面装入到了物理内存，当进程指令访问一个不存在于物理内存的地址时，CPU 则产生一个缺页异常（page fault exception）[20] [21] [22]，并将这个虚地址存入 CR2 寄存器，缺页异常的处理程序 do_page_fault()对产生缺页的原因进行区分。激发缺页异常的情况通常有三种：

1、被访问的物理页由于是第一次访问，所以还在磁盘上或由于访问后被置换时因为没有发生写操作，而未写到交换文件中。

2、虚地址有效，但其所对应的物理页由于太长时间没有访问而被 kswapd 置换到磁盘或交换文件中。

3、COW（写时复制）型中断，当某个进程进行写操作时，如果写的页是多进程在使用时，为了不影响其它进程的正常运行，通常需要将该页复制一份，供执行写操作的进程单独使用。

该函数根据一些条件来判断该产生异常的地址是否是进程正常产生的地址，如果不是，则该系统将会给进程发出一个结束信号，结束进程的执行；如果是进程正常产生的地址，需要内核对其进行分配新页，该函数调用handle_mm_fault()来进行实际的页分配，然后从磁盘(交换文件 / 执行文件映象)把所需的页面读入内存，然后重启该指令的执行。

handle_mm_fault()通过查找页表，找到与该异常地址相关的页表项，然后通过调用 handle_pte_fault()来进行页分配并将虚实的映射关系填入该页表项中。不同的异常原因会调用不同的页分配函数来进行页分配处理。

1、如果页表项无效，且页表项为空，则该异常地址对应的物理页还没有分配过，通过 do_no_page()来实现页分配，这种技术成为“请求调页”，对应于第一种缺页异常。

2、如果页表项有效，则产生这次异常的原因是写了只读页，通过 do_wp_page()分配写一个新的页面，并把旧页面的数据复制到新页面来对其进行初始化，这种技术称为“写时复制”；对应于第二种缺页异常。

3、如果页表项无效，但是页表项不为空，则说明该页被换出到交换分区中，通过 do_swap_page()该页重新读入内存中；这个过程称为“页的换入和换出”，对应于第三种缺页异常。

我们这着重介绍一下 do_no_page()函数，Linux 中页分配相关的主要有两种页：匿名页和页缓冲区页。do_no_page()的主要包括以下几个步骤：

1、如果 vma 中具有文件系统相关的 vm_ops 且 vm_ops 内的 nopage()域不为 NULL，说明某个虚存区映射了一个磁盘文件，则转到步骤 3；否则转到步骤 2。

2、虚存区没有映射磁盘文件，用 do_anonymous_age()进行匿名页的分配，然后返回。

3、调用 vma->vm_ops->nopage()来进行特定的缺页异常的处理函数，在 Linux 中一般文件系统的该函数指针指向的函数是 filemap_nopage()。该函数来实现与文件系统相关的页缓冲区的页分配。

4、判断这次产生异常的操作是否为写操作，如果是则需要进行写时复制，如果不是转到步骤 6。

5、进行写时复制，申请一个新页，把 filemap_nopage()返回的数据复制给新页。

6、建立页和产生异常的地址之间的映射，保存在页表项，并且通过__update_tlb()来更新该地址所匹配的 TLB 表项的内容。

7、结束返回。

do_anonymous_age()用来实现匿名页的分配，因为其不需要和文件系统打交道，所以其操作比较简单，只要包括下面几个步骤:

1、通过 alloc_page()调用__get_free_page()申请一个新的页面，并把新页面初始化为 0。

2、建立页和产生异常的地址之间的映射，保存在页表中，并且通过一 update_tlb()来更新该地址所匹配的 TLB 表项的内容。

3、结束返回。

Filemap_nopage()用来实现与文件系统相关的页分配，它先建立页缓冲页，然后调用文件系统和设备驱动将所需的数据读到该页中，它主要包括以下几个步骤：

1、用线性区保存的 mapping 和该数据所在文件的偏移量 pgoff 进行哈希查找，如果有页存在，则说明与文件数据相对应的页已经存在于页缓冲之中，则直接转到步骤 3。

2、通过 page_cache_read()来分配一个新页，将它与该文件数据进行关联，并将该页通过 add_to_page_cache_unique()添加页缓冲的哈希表中。再调用文件系统 readpage()去读取文件中的数据，然后转到步骤 1。文件系统在完成该页的

读取之后，会将该页的 PG_updated 标志位置 1 通知正在等待该页那些服务。

3、如果页的 PG_updated 位没有置为 1，则说明该页数据还在读取中，转到步骤 5。

4、返回该页缓冲页。

5、等待文件系统和设备驱动完成页的读取。这时该次服务进入休眠，将系统控制权让给其它服务，比如网络收发包等等。

6、文件系统和设备驱动完成页的读取后，唤醒该次服务，转到步骤 4。

3.5 本章小结

内存管理系统是操作系统中最为重要的部分，其任务是高效地管理系统中的内存资源。虚拟内存管理是内存管理子系统要完成的主要工作。

虚拟内存可以提供以下的功能：

1、广阔的地址空间：系统的虚拟内存可以比系统的实际内存大很多倍。

2、进程的保护：系统中的每一个进程都有自己的虚拟地址空间。这些虚拟地址空间是完全分开的，这样一个进程的运行不会影响其他进程。并且，硬件上的虚拟内存机制是被保护的，内存不能被写入，这样可以防止迷失的应用程序覆盖代码的数据。

3、内存映射：内存映射用来把文件映射到进程的地址空间。在内存映射中，文件的内容直接连接到进程的虚拟地址空间。

4、公平的物理内存分配：内存管理系统允许系统中每一个运行的进程都可以公平地得到系统的物理内存。

5、共享虚拟内存：虽然虚拟内存允许进程拥有自己单独的虚拟地址空间，但有时可能会希望进程共享内存。

物理内存管理子系统负责对物理内存页的分配和回收；进程虚拟内存管理子系统为用户进程提供虚拟页面的分配和回收；而内核虚拟内存管理子系统则为内核提供各种大小的物理内存块。内存管理的基础是对物理内存的管理，而各个内存管理器有都离不开内核内存管理器的支持。

第四章Linux 操作系统文件系统管理机制

Linux文件系统是由两大模块构成的：（一）VFS(VirtrualFileSystem)虚拟文件系统[23] [24] [25] [26] [27] [28] [29]。Linux操作系统支持多种不同的文件系统就是通过VFS实现的，它是Linux提供给用户进程统一的文件系统操作界面，用户可通过虚拟文件系统访问多种具体的文件系统，而不管实际的文件系统是如何组织的。（二）具体的文件系统，而Ext2(theSecondExtended)文件系统是Linux的缺省文件系统。

VFS文件系统要管理在任何时间装配的所有不同的文件系统，不同的物理文件系统具有不同的组织结构和不同的处理方式，为了能够处理各种不同的物理文件系统，操作系统必须把它们所具有的特性进行抽象，以建立一个面向各种物理文件系统的转换机制，并通过这个机制把各种不同的物理文件系统转换为一个具有统一共性的虚拟文件系统，使得不同的物理文件系统看起来都是相同的。如当用户进程对文件系统提出操作请求时，虚拟文件系统将内存的数据结构与具体文件系统的数据结构关联起来，同时调用具体的文件系统的操作函数，启动设备的输入/输出操作，实现设备上文件的读取、写回、查找、更改、更新等操作。同时虚拟文件系统提供的内存节点缓冲区、内存目录项缓冲区、数据块缓冲区提供了内存中操作节点、目录、数据块的手段，实现文件系统尽量在内存中处理文件，减少读取外设的操作次数.在操作完成之后，文件系统在适当的时机将调用虚拟文件系统的更新例程，将改变的数据从内存中全部写回外部设备。文件系统整体结构如图1所示：

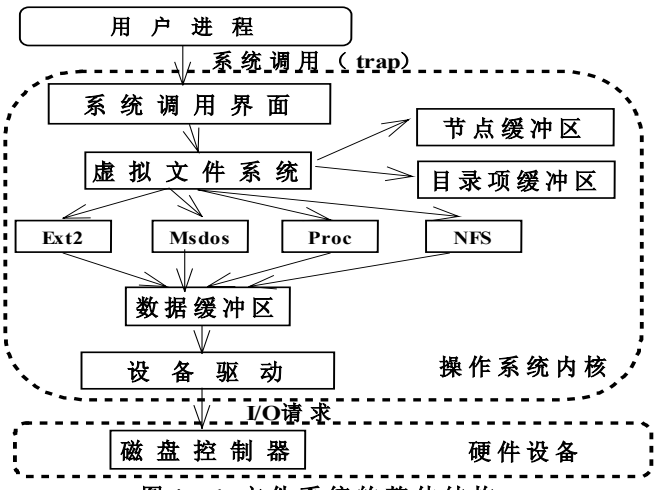

图 4－1 文件系统的整体结构

4.1 Linux 文件系统核心数据结构

VFS 不是一种物理文件系统，而是一套转换机制，它在系统启动时建立，在系统关闭时消失，并仅存在于内存空间，它用内存中的数据结构和函数例程处理多种具体文件系统的数据结构和函数例程，所以，VFS 并不具有一般物理文件系统的实体。在 VFS 提供的接口中包含向各种物理文件系统转换用的一系列数据结构，如 VFS 超级块、VFS 的 inode 等；同时还包含对不同物理文件系统进行处理的各种操作函数的转换入口，如 super_operations、file_operations、inode_operations 等，为用户程序提供了一个统一的、抽象的、虚拟的文件系统界面。

4.1.1 VFS 超级块与 Ext2 超级块

对于每个已安装的文件系统，在内存中都由一个与其对应的VFS超级块来表示，VFS超级块的作用是把在各种文件系统中表示整体组织结构的信息转换成统一的格式，各文件系统VFS超级块中各成员的数据则来自该文件系统本身的管理信息结构。VFS超级块是各种逻辑文件系统在安装时建立，并在这些文件系统卸载时自动删除的。

VFS超级块主要包含文件系统的组织信息、具体文件系统类型、超级块操作函数、文件系统的注册和安装信息、等待队列、打开文件列表，不同物理文件系统特有的信息则由联合体U的各个成员项表示，从而实现多种文件系统的兼容。这些数据是在文件系统安装时，由读超级块例程read_super()把具体文件系统的管理信息写入它的VFS超级块中。当文件系统访问某一个具体的文件系统时，首先操作系统要获取该文件系统的超级块，从而解析出必要的文件系统信息。

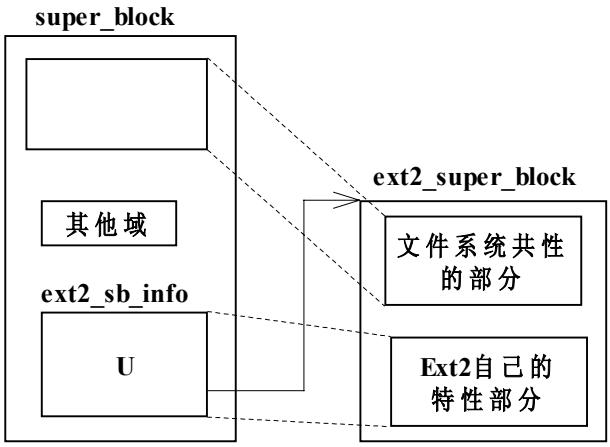

图4－2 VFS超级块与EXT2超级块的关系示意图

而对于具体物理文件系统Ext2，其超级块是用来描述Ext2文件系统整体信息的数据结构，主要描述文件在逻辑分区中的静态分布情况，以及描述文件系统的

各种组成结构的尺寸、数量等信息，如给定块大小、inode节点总数、每组内inode节点数、空闲块和空闲inode节点数等。在Linux启动时，根设备中的Ext2超级块经函数ext2_fill_super读入后，又在内存中建立一个映象super_block.u.ext2_sb_info，这样通过VFS的超级块可以访问Ext2的超级块。VFS超级块与具体文件系统Ext2超级块的关系示意图如图4-2。

4.1.2 VFS inode 与 Ext2 inode

在进程使用一个文件时，无论是哪种文件系统的文件，首先需要打开这个文件，也就是在内核的内存中建立一个且只有一个该文件的VFS inode，并且只要VFSinode对系统有用，就一直保存在VFS inode cache缓冲队列中。每一个VFSinode中的信息，都是使用文件系统相关的例程从底层文件系统中获取的。VFS inode的作用是把不同文件系统的活动文件的管理信息，如文件类型、文件尺寸、文件位置等转换成统一的格式，它除了包括具体文件系统inode中的若干静态管理信息外，又增加了许多文件的动态管理信息，如设备标识符、索引节点编号(表明一个VFS inode惟一地对应一个文件)、VFS inode的双向链表指针、节点操作函数指针、文件操作函数指针，在联合体U中的各个成员项给出了不同文件系统的个性信息、锁定标志等，从而实现多种文件系统的兼容，所以访问inode是文件系统定位一个文件的基本途径。

在Ext2树型结构文件系统中所有的普通文件、设备特殊文件、FIFO文件、链接文件和目录文件都是用inode节点唯一表示和标识的，不同类型的文件是通过节点的文件类型属性区分，inode是管理文件系统的最基本单位，是文件系统连接任何子目录、任何文件的桥梁。在一个文件系统中每个文件的节点号是唯一的。同一个块组中的文件所对应的inode组合到一起形成一个块组的inode表。Ext2文件系统的inode的作用是记录与文件管理有关的各种静态信息，如文件的类型和访问权限、文件的尺寸、文件的位置、文件的时间信息等。Ext2磁盘节点的结构如图4-3。

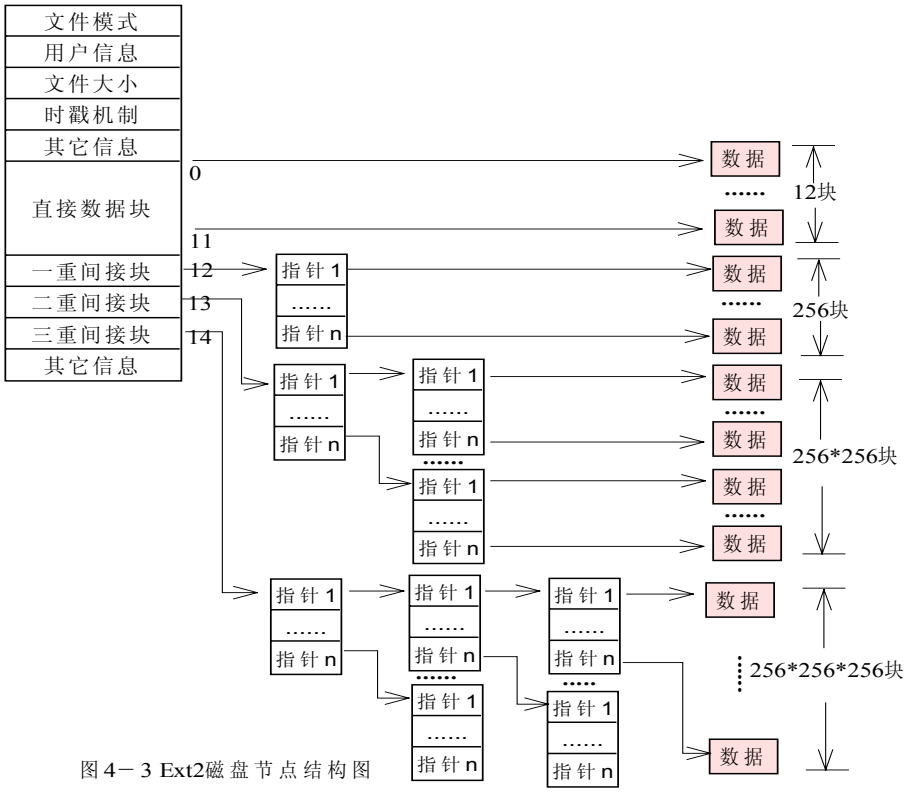

图4－3 Ext2磁盘节点结构图

VFS的索引节点结构存有文件或目录文件的信息，具体文件系统的索引节点根据文件系统不同处理方法不同，Ext2文件系统的inode是存储在磁盘上的，是一种静态结构，要使用它，必须调入内存，用它填写VFS的索引节点，因此VFS索引节点是动态节点；通过VFS的inode可以访问Ext2的inode。VFSinode与Ext2 inode的关系示意图如图4-4。

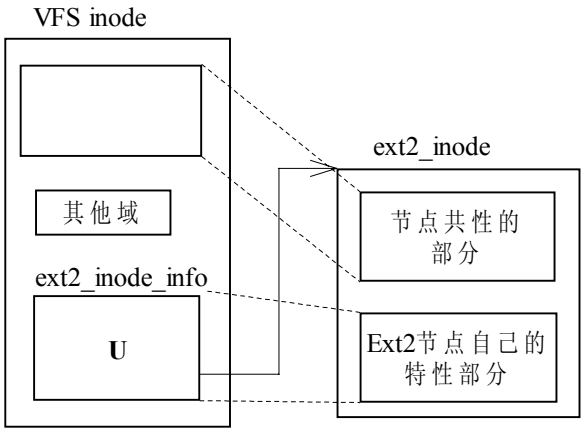

图4－4 VFS inode与Ext2 inode的关系示意图

4.1.3 VFS dentry 目录项与 Ext2 ext2_dir_entry_2

在文件系统中，用inode号表示一个文件是方便的，但对用户来说，更习惯用文件名来表示文件，因此用文件系统目录机制完成文件名到inode号之间的转

换。文件系统中的所有信息节点都是通过dentry名称目录项进行访问的，名称目录项属于实际文件系统，它是动态生成的，读入内存后存储在目录项缓存dcache中，主要有目录项名及对应的inode号。

dentry通过表关系形成一个dentry树，也就是用户看到的树型目录结构描述（如图5所示）。每个文件或目录都有一个dentry结构，dentry结构包括了路径信息，各种目录链表，还指向了inode和超级块，它描述了文件或目录在文件系统目录树中的关系及状态。文件系统由文件或目录路径名可得到dentry，进而可找到对应的inode结构，反过来，由dentry结构也可得到文件或目录的路径名；dentry将文件系统的路径名与inode节点联系起来了。

Ext2的目录是简单的、具有固定格式的特殊文件，它们也用inode描述。一个目录就是一个目录项结构的数组，每一个目录项由一个ext2_dir_entry_2结构进行描述。该结构最主要的内容就是该目录下的文件名或子目录到inode号的对应关系。

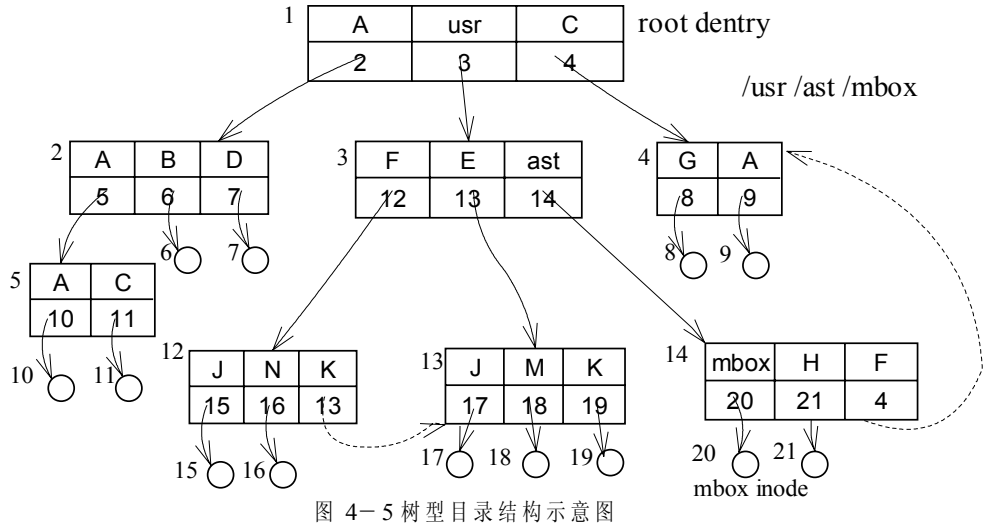

图 4－5 树型目录结构示意图

4.1.4 系统有关操作函数集的结构

VFS 文件系统是虚拟的，无法涉及到具体文件系统的细节，所以，必然在 VFS 和具体结构文件之间有一些结构。VFS 的数据结构就好像是一个标准，具体文件系统要想被 linux 支持，就必须提供 VFS 标准应具有的结构及操作函数。实际上，具体文件系统在使用前，必须将自己的结构及操作函数映射到 VFS 中，这样才能被访问到。下表分析 VFS

中的操作函数集的结构。

VFS 中核心操作函数结构	Ext2 的操作函数	各操作函数结构体的作用
Struct super_operations	Ext2 超级块的操作数据结构的实例定义为:ext2_sops.	超级块操作函数实现的是对超级块的节点的读取、写回、释放、删除等操作和超级块的释放、写回、获取文件系统状态、再次安装等操作。
Struct inode_operations	普通文件节点操作: ext2_file_inode_operations. 目录文件节点操作: ext2_dir_inode_operations 符号链接文件节点操作: ext2_symlink_inode_operations. 块设备文件和字符设备文件节点操作: blkdev_inode_operations chardev_inode_operations.	节点的操作函数是实现节点的创建、查找、删除等操作的数据结构。 该数据结构是对普通文件、目录文件、设备特有文件等所有节点的操作的统一体。
Struct file_operations	普通文件操作： ext2_file_operations 目录文件操作： ext2_dir_operations 块设备文件和字符设备文件： def_blk_fops def_chr_fops	文件操作数据实现对文件的操作包括：移动文件指针、打开文件、释放文件、搜索、读写文件、更新文件等。它是对普通文件、目录文件、符号链接文件、设备特有文件等的所有文件操作的统一体
Struct dentry_operations	目录文件节点操作: ext2_dir_inode_operations 目录文件操作： ext2_dir_operations	目录项操作函数是目录缓冲区常用的操作函数。

4.2 VFS 实现机制

4.2.1 文件系统的建立

如果要使一个文件系统在操作系统中可被用户访问，首先必须建立文件系统。文件系统的建立要经历下面两个步骤:

1. 文件系统要在操作系统内核中进行注册。
2. 文件系统的安装挂接。通过安装挂接具体文件系统，VFS 读取它的超级块，得到具体文件系统的拓扑结构，并将这些信息映射到 VFS 超级块结构中，将设备文件与文件系统的一个空闲目录相关联；这样，该设备文件系统的访问就可通过访问该目录来实现。

4.2.2 文件系统的注册与注销

向系统内核注册也有两种方式:一种是在编译核心系统时确定，并在系统初始化时通过内嵌的函数调用向注册表登记;另一种则利用Linux的模块块(module)特征，把某个文件系统当作一个模块。装入该模块时(通过kerneld或用insmod命令)向注册表登记它的类型，卸装该模块时则从注册表注销。

各种文件系统的注册是通过内核提供的文件系统初始化函数实现的，对不同文件系统使用不同的初始化函数；在各种文件系统初始化函数中，把该文件系统的注册结构体作为参数，调用由内核提供的文件注册函数register_filesystem()；该函数的功能是把文件系统注册结构体加入到注册链表中，从而完成注册任务。

如果某文件系统是被内核支持的，则该文件系统的注册在操作系统的文件系统初始化例程中进行。例如Ext2文件系统的初始化函数定义如下：

```
static int__init init_ext2_fs(void)
{
return register_filesystem(&ext2_fs_type);
}
```

如果该文件系统作为可加载模块加载时，通过调用Ext2文件系统的init_module()函数进行文件系统注册。在该函数中，调用了Ext2文件系统的init_ext2_fs()，以后的过程同上。

文件系统的注册就是将该文件系统的内核支持函数与操作系统内核相关联的过程。每一种文件系统的初始化例程都向虚拟文件系统注册，每一种文件系统有一个file_system_type类型的数据结构表示，它包括了该文件系统的名称和一个指向它的虚拟文件系统超级块的指针。所有已注册文件系统的file_system_type结构体组成一个链表，称为注册链表，并用全局变量

file_systems来表示，通过该全局变量可以访问所有注册的文件系统。如图6所示。

文件系统的注销是文件系统注册的反过程。某个文件系统的注销过程就是在文件系统类型链表中删除该文件系统类型结构变量并释放占用的内存空间的过程。文件系统的注销可使用可加载模块卸载的方法实现。以Ext2文件系统为例，注销该文件系统通过调用该文件系统的clear_module()函数来实现。在该函数中，调用了Ext2文件系统的exit_ext2_fs()，最终调用了虚拟文件系统的unregister_filesystem()函数实现。

```
staticvoid__exitexit_ext2_fs(void)
{
    unregister_filesystem(&ext2_fs_type);
}
```

4.2.3 文件系统的安装与卸载

实际的文件系统是内建到系统内核中，也可作为可载入模块在需要时进行装载。每个装载的文件系统由vfsmount数据结构来表示，它们被放在由vfsmntlist指向的队列链表中；另一个指针vfsmnttail指向表中最后一项；mru_vfsmnt指针指向最近被使用的文件系统。每个vfsmount结构包括记录该文件系统的块设备的设备号、文件系统的装配目录和文件系统装配时分配的VFS超级块的指针。

由于Ext2是Linux的标准文件系统，所以在系统启动时自动把Ext2文件系统的磁盘分区作为Linux文件系统的根文件系统，但超级用户在安装其它类型文件系统时，要指定三种信息:文件系统的名称、包含文件系统的物理块设备和文件系统在已有文件系统中的安装点。例如:#mount-tiso9660/dev/cdrom/mnt/cdrom，其中iso9660是要安装的文件系统类型，/dev/hdc是文件系统所在的设备，/mnt/usr是安装点。装载例程执行如下几个步骤：

(1) 寻找对应的文件系统信息，VFS通过filesystems在file_system_type组成的链表中根据指定的文件系统名称搜索文件系统类型信息.若未找到则返回错误码.

(2) 检查物理块设备是否存在且还没有安装，是则找到装载点的目录inode节点，检查是否是目录类型，且尚未装载其它的文件系统。

(3) 在super_blocks中搜索空闲的super_block结构，若没有搜索到，则分配一个超级块结构，将它初始化，然后加入到super_blocks向量中。

(4) 运行所装载文件系统的读超级块例程，这个读例程读取物理设备上的超级块信息，并填充VFS超级块的各个域。

(5) 增加新的安装点，分配新的装载点结构，进行初始化设置，然后加入到vfsmntlist表中，并修改vfsmnttial指针。

(6) 如果成功安装，则所有已安装的文件系统形成图6所示的数据结构。

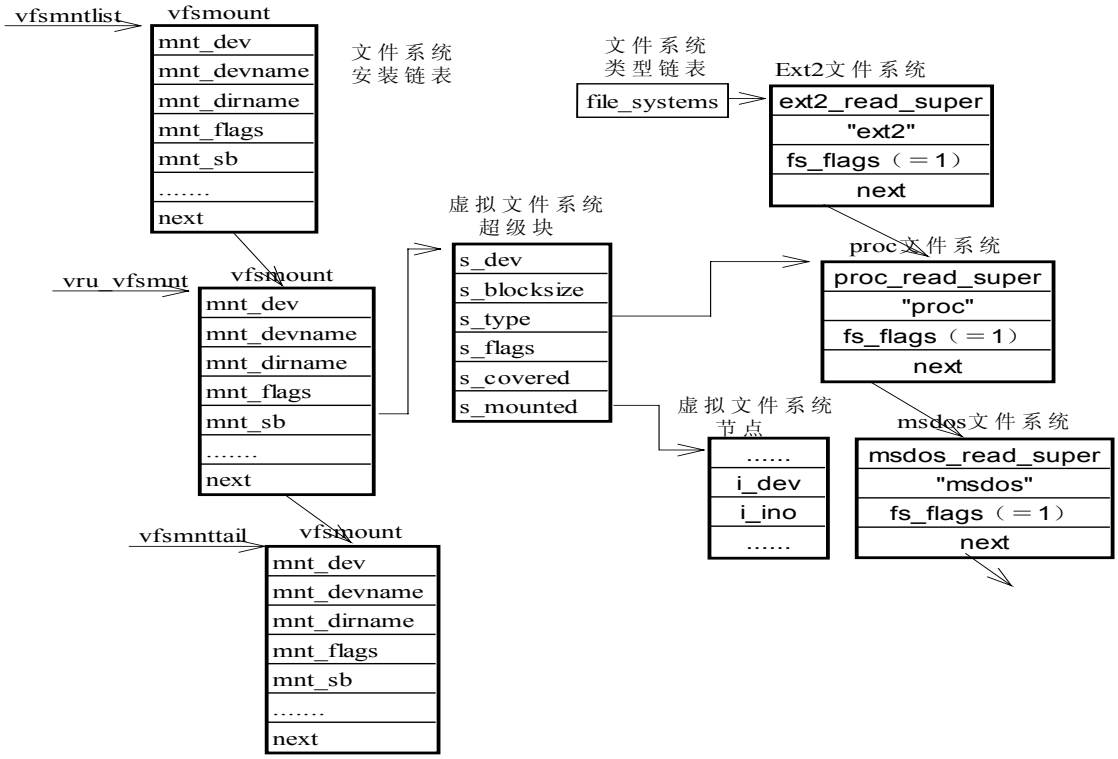

图4－ 6 文件系统安装结构图

文件系统的卸载是文件系统安装的反过程。文件系统的卸载过程为：文件系统卸载程序首先检查该文件系统是否在使用，若在使用就不可以卸载。然后检查整个 VFS 文件系统的节点链表，查找属于该文件系统的节点是否修改过。如果该文件系统的超级块被修改过，必须全部写回磁盘。然后，释放该文件系统超级块所占用的内存，返回到空闲内存缓冲池中。最后，该文件系统在 VFS 文件系统安装链表中的 vfsmount 数据结构实例被删除，同时释放占用的内存空间。

4.3 通过虚拟文件系统访问 Ext2 中的文件

在Linux的VFS中提供了具体文件系统统一的操作界面，以及依赖于VFS的多种操作函数的数据结构。在这些结构体中，文件系统共用的多个操作函数的函数指针组成结构体。属性的赋值由各个文件系统根据需要进行，以保证不同的文件系统都可以利用该数据结构实现具体的操作。

如当进程访问Ext2物理文件系统的文件时，并不直接访问该物理文件系统，而是通过VFS访问它在内存中的VFS超级块和Ext2inode，获得Ext2文件系统和文件的管理信息，然后使用这些信息访问磁盘上的文件。这样，进程通过VFS接口就可以访问各种不同的物理文件系统。VFS的功能及其工作原理，可以通过进程

访问不同物理文件系统的过程来说明。下图给出了进程访问Ext2文件系统时，经VFS实现转换的示意图如下：

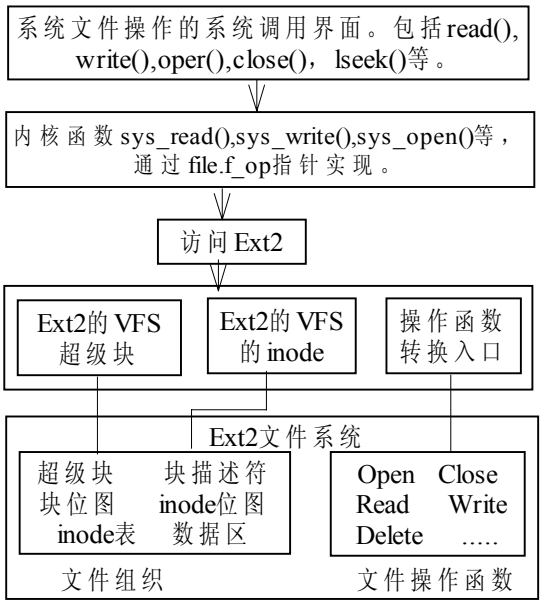

图4－7 由VFS实现转换示意图

4.4 本章小结

本章从 Linux 的内核文件管理机制——虚拟文件系统(VFS)出发，对 Linux 系统如何支持多种不同的物理文件系统进行了研究，并详细分析了整个过程中采用的数据结构，剖析了 Linux 文件系统中一个物理文件系统的注册与注销、安装和卸载过程，以及通过 VFS 访问 Ext2 物理文件系统的内核工作机制。

第五章 linux2.6 与 linux2.4 的对比分析

相对Linux2.4而言，Linux 2.6在进程调度、内存管理、文件系统、网络、系统安全、设备驱动等方面都有很大的改进,本章对进程管理与调度、内存管理、文件系统管理作简单的归纳分析：

5.1 进程管理的改进

1、新调度器算法：Linux2.6内核使用了新的调度器算法，它在高负载的情况下执行得非常出色，并在有多个处理器时能够很好地扩展[30][31][32][33][34]。

2、高效的调度程序：Linux2.6内核中，进程调度经过重新编写，调度程序不需每次都扫描所有的任务，而是在一个任务变成就绪状态时将其放到一个名为“当前”的队列中。

由于在Linux2.4中，所有就绪进程都放在一个以runqueue_head为头的双向链表，而且它们之间没有顺序，所以要选择一个优先级最高的就绪进程作为候选进程，就要遍历整个就绪队列的所有就绪进程. 所以，在Linux 2.4中调度器选择候选进程next的时间复杂度是O(n) [35][36][37][38]。

在Linux2.6中，每个CPU维护一个就绪队列，就绪队列又分为active队列和expired队列两部分，就绪进程在active队列和expired队列中是按动态优先级从高到低有序插入到其Hash表queue的相应链表中的，相同优先级的进程又按入队的先后顺序被排列在同一链表中，并且每个Hash表queue都有一个位映射数组bitmap与之对应，根据此bitmap不必遍历整个就绪队列，只要经过简单的位运算和地址运算就能在恒定时间内找到就绪队列中优先级最高的进程. 所以，在Linux2.6中O(1)调度器选择候选进程next的时间复杂度是O(1)，与当前系统负载无关，实时性能更好。

3、新的同步措施：Linux2.6内核支持futex (FastUser—SpaceMutex)。Futex是一种快速的互斥量。用户空间的互斥量的基本思想是：对互斥量的操作主要在用户态的库函数完成，只有在线程因加锁失败需要睡眠或被唤醒这些情况下才调用系统调用。

5.2 内存管理的改进

1、对内核地址空间所做的优化：在Linux2.4 内核中,缺省情况下会将虚地址空间的3GB 分给用户进程,1GB 分给内核进程。Linux2.6 内核在保持原有地址

空间分配不变的情况下,对内核空间所存放的页描述符进行了重新组织。它引入了一种称为“页集群“(page clustering) 的技术, 即通过一个页描述符来控制一组页。通过减少mem-map数组中页描述符的数量,从而减低了mem-map对于内核空间的消耗。

2、抢占式内存：Linux2.6版内核中内存是可抢占的。这将显著地降低用户交互式应用程序、多媒体应用程序等类似程序的延迟。

3、内存池机制：Linux2.6版内核引入内存池以满足不间断地进行内存分配。

4、反向映射：反向映射技术为每一个内存页面的Page结构增加了如下内容:

```
union {
    struct pte_chain * chain;
    pte_addr_t direct;
    } pte;
```

Pte为一个链表或者指针,包含了指向该页的每一个进程的页表条目(Page_Table Entries, PTE)的指针。

反向映射的采用给页面换出带来的好处很明显:不需要遍历每一个进程的页表来判断本页是否被这个进程使用;可以优先换出希望换出的内存页面;只需要扫描不活跃的页面。

5、改善虚拟内存：Linux2. 6版内核融合了RikvanRiel的R—MAP技术，将显著改善虚拟内存在一定程度负载下的性能。

6、分节点的页面管理：Linux 2. 6中页面队列从全局变成从属于某个节点的管理区,这样有利于释放特定管理区的内存,不需要使用全局锁,而且减少了NUMA机器节点间缓冲。内存收集也被设计成可以针对某个节点。

7、改善共享内存:Linux2.6内核为多程序提供NUMA(Non_Uniform Memory Access)技术共享内存，当出现竞争时，对就近内存有更高使用权。

5.3 文件系统的改进

1、扩展属性的支持，也即给指定的文件在文件系统中嵌入一些元数据。

2、POSIX访问控制是标准UNIX权限控制的超集，支持更细粒度的访问控制。

3、Linux对文件系统层还进行了大量的改进以兼容PC机的主流操作系统。

第六章　结束语

操作系统是计算机的重要组成部分，操作系统的优劣直接关系到计算机的整体性能；而且是基础应用软件开发和信息安全的基石。现在，越来越多的商业领域、个人计算机爱好者开始使用 Linux 操作系统，并投入到 Linux 操作系统的深入研究和开发中去。Linux 操作系统的发展带来整个操作系统领域的全面进步。Linux 操作系统的最大特点是开源、免费，提供了完整的内核源程序，性能优越、稳定性好、安全性高。

正是由于Linux操作系统自由、公开、免费的特性，提供给人们一个研究国外优秀操作系统的设计思想和实现方法的机会，对于发展国产操作系统有着很重要的意义。本文的研究工作正是在这个背景下产生的。

本文着重分析研究构成Linux 2. 4操作系统内核的四个基本功能：进程管理与调度机制、存储管理机制、文件系统管理机制。通过对Linux内核的分析，自认为从中获益不少：

一、对计算机工作原理有进一步的理解，对操作系统如何管理处理机、存储器、文件、I/O设备等有个更具体的理解。

二、为以后工作中进行嵌入式设备驱动开发，或者嵌入式操作系统裁减和移植打下坚实的基础，对设计大型系统软件也能做到整体把握。

三、为将来做进一步的计算机科学研究与开发工作提供了开阔的视野和研究方法。

在完成本文之后，自己对操作系统理解更深入和全面了，但又深深感觉自己掌握的计算机知识还太少了，很多计算机工作原理和机制还处于一知半解状态，今后还有很多工作需要进一步深入研究，如进行操作系统裁减和移值，或者系统软件的开发，要经过大项目的锻炼才能真正理解掌握操作系统的艺术精髓。

参 考 文 献

[1]毛德操，胡希明.Linux内核源代码情景分析[M].杭州:浙江大学出版社，2001.

[2]陈莉君。Linux操作系统内核分析[M]。北京：人民邮电出版社，2000.

[3]林伟，linux内存管理子系统在龙芯2号上的优化[M].硕士学位论文.北京：中国科学院计算技术研究所2005.

[4]DanielP.Bovet,Marco Cesati. Understanding the Linux Ihernel, 2nd Edition [M]. O'Relilly&Associates, Inc., 2002.

[5]Mel Gorman. Understanding The Linux Virtual Memory Manager [M].Prentice Hall PTR, Inc., 2004.

[6]Simon Winwood, Yefim Shuf&Hubertus Franke. Multiple Page Size Support in the Linux Kernel [J]. Linux Symposium, 2002.

[7]刘生平.桌面Linux内存管理性能优化技术研究与实现[M].硕士学位论文.北京:清华大学2004年.

[8]Drawson R. Engler, Sandeep K.Gutpta, M.Frans Kaashoek . AVM: Application Virtual Memory, Proceedings of the 6th Australasian conference on Computer systems architecture,2001, Pages: 3 – 10.

[9]Charles D. Cranor . Design and Implementation of the UVM Virtual Memory System, Ph.D.thesis,Department of Computer cienceWashington University St. Louis, MO 63130 , 1998.

[10]Charles D. Cranor ,Gurudatta M. Parulkar . The UVM Virtual Memory System, 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, Pp.117–130.

[11]Kirean Harty and David R Cheriton, Application-Controlled Physical Memory Using External Page-Cache Manament, Proceedings of the fifth international conference On Architectural support for programming languages and operating systems 1992, Pages: 187 – 197.

[12]Tobias G. Dussa . dm_phys:A Dataspace Manager for Physical Memory, System Architecture Group Faculty for Computer Science University of Karlsruhe, 2002-03-05

[13]Michael Clarke, Operating System Support for Emerging Application Domains, Ph.D. thesis.

[14]陈莉君，康华。Linux操作系统原理与应用[M]。清华大学出版社，2006

[15]Martin J Bligh,David Hansen. Linux Memory Management on Larger Machines [EB /OL]. http://archive.Linuxsymposium.org/ols2003/

[16] Proceedings/All2Reprints/Reprint2Bligh2OLS2003.pdf, 2003207.[3] Rik Van Riel. Towards an O (1) VM [EB /OL]. http: //www.sur2riel.com /lectures/Ols2003/,2003.

[17]D. (Daniele) Bovet and Marco Cesati. Understanding the Linux kernel[M]. O'Reilly , 2000.

[18]D. (Daniele) Bovet and Marco Cesati. Understanding the Linux kernel (2nd Edition) [M] . O'Reilly , 2003.

[19] Andrew S. Tanenbaum. Modern Operating Systems (2nd Edi2 tion) [M] . Prentice - Hall , 2001.

[20]Martin J . Bligh and David Hansen. Linux memory management on larger machines [Z] . Proceedings of the Linux Symposium.2003. 55～68

[21]Mel Gorman. Understanding The Linux Virtual Memory Manager[Z].http ://www. skynet . ie/ ～ mel/ projects/ vm/ guide/ pdf/understand. pdf . 2004.

[22] Paul Larson . Kernel comparison : Improved memory management in the 2. 6 kernel. IBM[Z] . http :/ / www - 106. ibm. com/ devel2 operworks/ Linux/ library/ l - mem26/ 2004.

[23]倪继利。Linux内核分析及编程[M]。电子工业出版社，2006

[24]夏煜。Linux操作系统的文件系统研究[M].硕士学位论文.西安：西北工业大学，2000

[25]史芳丽,周亚莉.Linux 系统中虚拟文件系统内核机制研究[J];陕西师范大学学报,2005

[26]Linux 的文件系统[EB/ OL] . http :/ / www. ironwareinfo.com.cn , 2004.

[27]文件系统[EB/ OL] . http :/ / www. todayhero. net , 2004.

[28]Evi Nemeth , Garth Snyder , Trent R Hein. Linux 系统管理技术手册[M] . 张辉译. 北京: 人民邮电出版社,2004. 53～66 , 113～152.

[29]杜聪, 徐志伟. COSMOS 文件系统的性能分析[J].计算机学报,2001 , (7) :702～709.

[30]栾建海，李众立，黄晓芳。Linux2。6内核分析【J】；兵工自动化软件技术，2005.2.24

[31]William Stallings. 操作系统一精髓与设计原理【M】. 北京：清华大学出版社，2002.

[32]陈燕晖，罗宇.Linux 2. 6存储管理子系统新特性分析[J]；计算机工程与应用2005。

[33]Linus Torvalds. Linux Kernel 2. 6. 10[DB / OL]. http://www.kernel.org,2004

[34]Roberto Arcomano berto. Linux documents[DB / OL]. http://www.tldp.org，2004

[35]谢长生, 刘志斌 Linux 2. 6 内存管理研究【J】；计算机应用研究，2005

[36]李善平，陈丈智，等. 边学边干一Linux 内核指导[M]. 浙江：浙江大学出版社，2002.

[37]William Stallings. 操作系统一精髓与设计原理[M]. 北京：清华大学出版社，2002.

[38]李善平，等. Linux 内核 2. 4 版源代码分析大全[M]. 机械工业出版社. 2002.

致　谢

美好的时光总是过得飞快，在师大三年的研究生学习、研究和生活就要结束了，首先我要深深地感谢我的导师余敏教授。余老师开明的思想、渊博的学识、朴实严谨的治学态度、辛勤认真的工作作风无形中感染了我，是我在学习和工作上努力的方向，是我学习的榜样；在余老师的指导和帮助下受益良多，我自认为这三年成长不少，为我将来工作中进一步发展打下了坚实的基础，因此我很感激余老师给我的培养；而且我有个心愿，就是在我三年之后有自己的事业了，能回报余老师和计算机学院给我提供的良好学习环境和宽容的学术氛围。在此，谨向敬爱的余老师表示最崇高的敬意。

感谢我年近六十岁的父母，为了支持我能继续读研而仍然日夜辛苦劳动着。二老的奉献和牺牲我永远会牢记在心的。祝愿父母身体健康，幸福一生。

感谢章志明、张正球师兄在我研一的时候，给我的指导，让我明确了自己的研究方向和兴趣，在学习和研究的过程中少走了许多弯路。感谢彭雅丽、邬可可、姜样兰、许生模同学，他们各自的优点对我的成长影响很大，和他们交流让我获益匪浅。

感谢我的朋友和实验室的师弟师妹们，和他们在实验室一起学习的日子真的很愉快，也衷心祝愿他们在后继的生活、学习、研究中，飞得更高，飞得更远。

最后，祝福各位事业有成，生活快乐，爱情美满，一切顺利。

独 创 性 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：　　签字日期：　　年　月　日

学位论文版权使用授权书

本学位论文作者完全了解江西师范大学研究生学院有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权江西师范大学研究生学院可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：　　导师签名：

签字日期：　　年　月　日　　签字日期：　　年　月　日