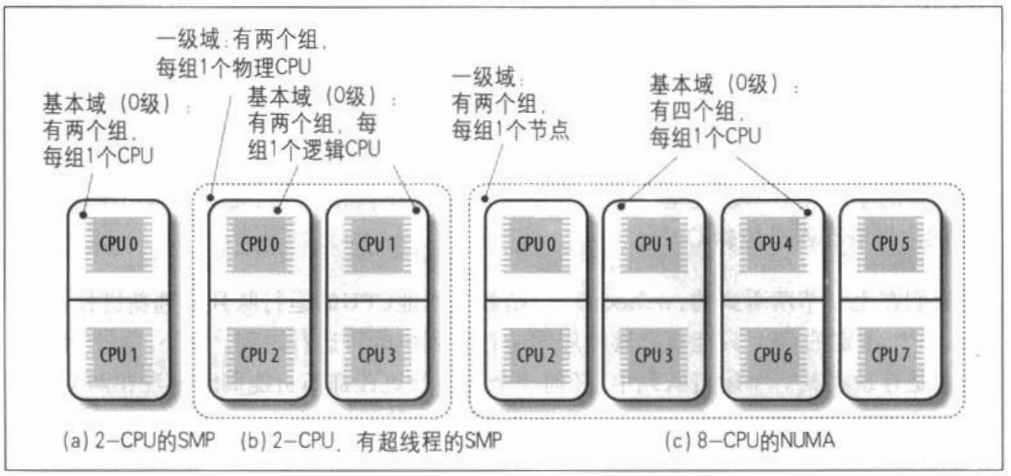


图 7-2：调度域分层的 3 个实例

图7-2(a)表示具有两个CPU的标准多处理器体系结构中由单个调度域组成的一个层次结构，该调度域包括两个组，每个组有一个 CPU。

图 7-2 (b)表示一个两层的层次结构，用在使用超线程技术、有两个 CPU 的多处理器结构中。最上层的调度域包括了系统中所有四个逻辑CPU，它由两个组构成。上层域的每个组对应一个子调度域并包括一个物理 CPU。底层的调度域(也被称为基本调度域)包括两个组，每个组一个逻辑 CPU。

最后，图 7-2(c)表示有两个节点，每个节点有四个 CPU 的 8-CPU NUMA 体系结构上的两层层次结构。最上层的域由两个组构成，每个组对应一个不同的节点。每个基本调度域包括一个节点内的 CPU，包括四个组，每个组包括一个 CPU。

每个调度域由一个 sched_domain描述符表示，而调度域中的每个组由 sched_group描述符表示。每个 sched_domain描述符包括一个 groups字段，它指向组描述符链表中的第一个元素。此外，sched_domain结构的 parent 字段指向父调度域的描述符（如果有的话）。

系统中所有物理 CPU 的 sched_domain描述符都存放在每 CPU 变量 phys_domains 中。如果内核不支持超线程技术，这些域就在域层次结构的最底层，运行队列描述符的sd字段指向它们，即它们是基本调度域。相反，如果内核支持超线程技术，则底层调度域存放在每 CPU 变量 cpu_domains 中。

rebalance_tick()函数

为了保持系统中运行队列的平衡，每次经过一次时钟节拍，scheduler_tick()就调用rebalance_tick()函数。它接收的参数有：本地CPU的下标this_cpu、本地运行队列的地址this_rq以及一个标志idle，该标志可以取下面的值：

SCHED_IDLE
: CPU当前空闲，即current是*swapper*进程。

NOT_IDLE
: CPU当前不空闲，即current不是*swapper*进程。

rebalance_tick()函数首先确定运行队列中的进程数，并更新运行队列的平均工作量，为了完成这个工作，函数要访问运行队列描述符的nr_running和cpu_load字段。

随后，rebalance_tick()开始在所有调度域上的循环，其路径是从基本域（本地运行队列描述符的sd字段所引用的域）到最上层的域。在每次循环中，函数确定是否已到调用函数load_balance()的时间，从而在调度域上执行重新平衡的操作。由存放在sched_domain描述符中的参数和idle值决定调用load_balance()的频率。如果idle等于SCHED_IDLE，那么运行队列为空，rebalance_tick()就以很高的频率调用load_balance()(大概每一到两个节拍处理一次对应于逻辑和物理CPU的调度域)。相反，如果idle等于NOT_IDLE，rebalance_tick()就以很低的频率调度load_balance()(大概每10ms处理一次逻辑CPU对应的调度域，每100ms处理一次物理CPU对应的调度域)。

load_balance()函数

load_balance()函数检查是否调度域处于严重的不平衡状态。更确切地说，它检查是否可以通过把最繁忙的组中的一些进程迁移到本地CPU的运行队列来减轻不平衡的状况。如果是，函数尝试实现这个迁移。它接收四个参数：

this_cpu
: 本地CPU的下标。

this_rq
: 本地运行队列的描述符的地址。

sd
: 指向被检查的调度域的描述符。

idle

取值为SCHED_IDLE（本地CPU空闲）或NOT_IDLE。

函数执行下面的操作：

1. 获取this_rq->lock自旋锁。
2. 调用find_busiest_group()函数分析调度域中各组的工作量。函数返回最繁忙的组的sched_group描述符的地址，假设这个组不包括本地CPU，在这种情况下，函数还返回为了恢复平衡而被迁移到本地运行队列中的进程数。另一方面，如果最繁忙的组包括本地CPU或所有的组本来就是平衡的，函数返回NULL。这个过程不是微不足道的，因为函数试图过滤掉统计工作量中的波动。
3. 如果find_busiest_group()在调度域中没有找到既不包括本地CPU又非常繁忙的组，就释放this_rq->lock自旋锁，调整调度域描述符的参数，以延迟本地CPU下一次对load_balance()的调度，然后函数终止。
4. 调用find_busiest_queue()函数以查找在第2步中找到的组中最繁忙的CPU，函数返回相应运行队列的描述符地址busiest。
5. 获取另一个自旋锁，也就是busiest->lock自旋锁。为了避免死锁，这一操作必须非常小心：首先释放this_rq->lock，然后通过增加CPU下标获得这两个锁。
6. 调用move_tasks()函数，尝试从最繁忙的运行队列中把一些进程迁移到本地运行队列this_rq中（见下一节）。
7. 如果函数move_task()没有成功地把某些进程迁移到本地运行队列，那么调度域还是不平衡。把busiest->active_balance标志设置为1，并唤醒*migration*内核线程，它的描述符存放在busiest->migration_thread中。*Migration*内核线程顺着调度域的链搜索－从最繁忙运行队列的基本域到最上层域，寻找空闲CPU。如果找到一个空闲CPU，该内核线程就调用move_tasks()把一个进程迁移到空闲运行队列。
8. 释放busiest->lock和this_rq->lock自旋锁。
9. 函数结束。

move_tasks()函数

move_tasks()函数把进程从源运行队列迁移到本地运行队列。它接收6个参数：this_rq和this_cpu（本地运行队列描述符和本地CPU下标）、busiest（源运行队列描述符）、max_nr_move(被迁移进程的最大数)、sd(在其中执行平衡操作的调度域的描述符地址)

以及idle标志（除了可以被设置为SCHED_IDLE和NOT_IDLE以外，在函数被idle_balance()间接调用时，该标志还可以被设置为NEWLY_IDLE。见本章前面“schedule()函数”一节）。

函数首先分析busiest运行队列的过期进程，从优先级高的进程开始。当扫描完所有过期进程后，函数扫描busiest运行队列的活动进程。函数对所有的候选进程调用can_migrate_task()，如果下列条件都满足，则can_migrate_task()返回1：

- 进程当前没有在远程CPU上执行。
- 本地CPU包含在进程描述符的cpus_allowed位掩码中。
- 至少满足下列条件之一：
  - 本地CPU空闲。如果内核支持超线程技术，则所有本地物理芯片中的逻辑CPU必须空闲。
  - 内核在平衡调度域时因反复进行进程迁移都不成功而陷入困境。
  - 被迁移的进程不是“高速缓存命中”的（最近不曾在远程CPU上执行，因此可以设想远程CPU上的硬件高速缓存中没有该进程的数据）。

如果can_migrate_task()返回1，move_tasks()就调用pull_task()函数把候选进程迁移到本地运行队列中。实际上，pull_task()执行dequeue_task()从远程运行队列删除进程，然后执行enqueue_task()把进程插入本地运行队列，最后，如果刚被迁移的进程比当前进程拥有更高的动态优先级，就调用resched_task()抢占本地CPU的当前进程。

与调度相关的系统调用

已经介绍的几个系统调用允许进程改变它们的优先级及调度策略。作为一般原则，总是允许用户降低其进程的优先级。然而，如果他们想修改属于其他某一用户进程的优先级，或者如果他们想增加自己进程的优先级，那么，他们必须拥有超级用户的特权。

nice()系统调用

nice()（注3）系统调用允许进程改变它们的基本优先级。包含在increment参数中的整数值用来修改进程描述符的nice字段。在Unix中的nice命令（允许用户用修改的调度优先级来运行程序）就是基于这个系统调用的。

注3：因为这个系统调用用来降低进程的优先级，因此为了自己的优先级而调用它的用户对其他用户来说就是“美好的（nice）”。

sys_nice()服务例程处理nice()系统调用。尽管increment参数可以有任何值，但是大于40的绝对值会被截为40。从传统上来说，负值相当于请求优先级增加，并请求超级用户特权，而正值相当于请求优先级减少。在负增加的情况下，调用capable()函数核实进程是否有CAP_SYS_NICE权能。而且，函数调用security_task_setnice()安全勾。我们将在第二十章讨论那个函数。如果用户想用请求的权能来改变优先级，sys_nice()就把current->static_prio转换到nice值的范围，再加上increment的值，并调用set_user_nice()函数。然后，set_user_nice()函数获得本地运行队列锁，更新current进程的静态优先级，调用resched_task()函数以允许其他进程抢占current进程，并释放运行队列锁。

nice()系统调用只维持向后兼容；它已经被下面描述的setpriority()系统调用取代。

getpriority()和 setpriority()系统调用

nice()系统调用只影响调用它的进程，而另外两个系统调用getpriority()和setpriority()则作用于给定组中所有进程的基本优先级。getpriority()返回20减去给定组中所有进程之中最低nice字段的值（即所有进程中的最高优先级）；setpriority()把给定组中所有进程的基本优先级都设置为一个给定的值。

内核对这两个系统调用的实现是通过sys_getpriority()和sys_setpriority()服务例程完成的。这两个服务例程本质上作用于一组相同的参数：

which

指定进程组的值。它采用下列值之一：

PRIO_PROCESS

根据进程的ID选择进程（进程描述符的pid字段）

PRIO_PGRP

根据组ID选择进程（进程描述符的pgrp字段）

PRIO_USER

根据用户ID选择进程（进程描述符的uid字段）

who

用pid、pgrp或uid字段的值（取决于which的值）选择进程。如果who是0，则把它的值设置为current进程相应字段的值。

niceval

新的基本优先级值（仅被sys_setpriority()所需要）。它的取值范围应该在-20（最高优先级）～+19（最小优先级）之间。

正如以前提到的，只有具有CAP_SYS_NICE权能的进程才允许增加它们自己的基本优先级或修改其他进程的优先级。

正如我们将在第十章看到的，只有当出现了某些错误时，系统调用才返回一个负值。由于这个原因，getpriority()不返回-20～+19之间正常的nice值，而是1~40之间的一个非负值。

sched_getaffinity()和sched_setaffinity()系统调用

sched_getaffinity()和sched_setaffinity()系统调用分别返回和设置CPU进程亲和力掩码，也就是允许执行进程的CPU的位掩码。该掩码存放在进程描述符的cpus_allowed字段中。

sys_sched_getaffinity()系统调用服务例程通过调用find_task_by_pid()搜索进程描述符，返回的值为相应字段cpus_allowed与可用CPU位图做与运算的结果。

系统调用sys_sched_setaffinity()有一点复杂。除了寻找目标进程的描述符并更新cpus_allowed字段以外，该函数还必须检查进程所属的运行队列，其对应的CPU亲和力掩码是否不再是最新值。在这种糟糕的情况下，必须把进程从一个运行队列迁移到另一个运行队列。为了避免死锁和竞争条件的问题，由migration内核线程(每个CPU有一个这样的线程)完成这个工作。一旦必须把进程从运行队列rq1迁移到运行队列rq2，sys_sched_setaffinity()系统调用就唤醒rq1的迁移线程(rq1->migration_thread)，该线程从rq1中删除被迁移的进程，然后把它插入rq2。

与实时进程相关的系统调用

现在我们介绍一组系统调用，它们允许进程改变自己的调度规则，尤其是可以变为实时进程。进程为了修改任何进程（包括自己）的描述符的rt_priority和policy字段，同样必须具有CAP_SYS_NICE权能。

sched_getscheduler()和sched_setscheduler()系统调用

sched_ getscheduler()查询由pid参数所表示的进程当前所用的调度策略。如果pid等于0，将检索调用进程的策略。如果成功，这个系统调用为进程返回策略：SCHED_FIFO、SCHED_RR或SCHED_NORMAL(后者也称为SCHED_OTHER)。相应的sys_sched_getscheduler()服务例程调用find_task_by_pid()，后一个函数确定给定pid所对应的进程描述符，并返回其policy字段的值。

sched_setscheduler()系统调用既设置调度策略，也设置由参数pid所表示进程的相关参数。如果pid等于0，调用进程的调度程序参数将被设置。

相应的sys_sched_setscheduler()系统调用服务例程简单地调用do_sched_setscheduler()函数。后者检查由参数policy指定的调度策略和由参数param->sched_priority指定的新优先级是否有效。它还检查进程是否具有CAP_SYS_NICE权能，或者进程的拥有者是否有超级用户的权限。如果每个条件都满足，就把进程从它的运行队列（如果进程是可运行的）中删除；更新进程的静态优先级、实时优先级和动态优先级；把进程插回到运行队列；最后，在需要的情况下，调用resched_task()函数抢占运行队列的当前进程。

sched_getparam()和sched_setparam()系统调用

sched_getparam()系统调用为pid所表示的进程检索调度参数。如果pid是0，则current进程的参数被检索。正如你所期望的，相应的sys_sched_getparam()服务例程找到与pid相关的进程描述符指针，把它的rt_priority字段存放在类型为sched_param的局部变量中，并调用copy_to_user()把它拷贝到进程地址空间中由param参数指定的地址。

sched_setparam()系统调用类似于sched_setscheduler()，它与后者的不同在于不让调用者设置policy字段的值（注4）。相应的sys_sched_setparam()服务例程用几乎与sys_sched_setscheduler()相同的参数调用do_sched_setscheduler()。

sched_yield()系统调用

sched_ yield()系统调用允许进程在不被挂起的情况下自愿放弃CPU，进程仍然处于TASK_RUNNING状态，但调度程序把它放在运行队列的过期进程集合中（如果进程是普通进程），或放在运行队列链表的末尾（如果进程是实时进程）。随后调用schedule()函数。在这种方式下，具有相同动态优先级的其他进程将有机会运行。这个调用主要由SCHED_FIFO实时进程使用。

sched_get_priority_min()和sched_get_priority_max()系统调用

sched_ get_priority_min()和sched_ get_priority_max()系统调用分别返回最小和最大实时静态优先级的值，这个值由policy参数所标识的调度策略来使用。

注4： POSIX标准的一个特殊要求造成了这种异常情况。

如果current是实时进程，则sys_sched_get_priority_min()服务例程返回1，否则返回0。

如果current是实时进程，则sys_sched_get_priority_max()服务例程返回99（最高优先级），否则返回0。

sched_rr_get_interval()系统调用

sched_rr_ get_interval()系统调把参数pid表示的实时进程的轮转时间片写入用户态地址空间的一个结构中。如果pid等于0，系统调用就写当前进程的时间片。

相应的sys_sched_rr_get_interval()服务例程同样调用find_process_by_pid()检索与pid相关的进程描述符。然后，把所选中进程的基本时间片转换为秒数和纳秒数，并把它们拷贝到用户态的结构中。通常，FIFO实时进程的时间片等于0。

第八章



内存管理

在第二章中我们已看到，Linux 如何有效地利用 80x86 的分段和分页单元把逻辑地址转换为物理地址。我们还提到，RAM的某些部分永久地分配给内核，并用来存放内核代码以及静态内核数据结构。

RAM的其余部分称为动态内存（dynamic memory），这不仅是进程所需的宝贵资源，也是内核本身所需的宝贵资源。实际上，整个系统的性能取决于如何有效地管理动态内存。因此，现在所有多任务操作系统都在尽力优化对动态内存的使用，也就是说，尽可能做到当需要时分配，不需要时释放。图 8-1 显示了用作动态内存的页框，详细内容请参见第二章的“物理内存布局”一节。

本章主要通过三部分内容描述内核如何给自己分配动态内存。“页框管理”和“内存区管理”两节分别介绍对连续物理内存区处理的两种不同技术。而“非连续内存区管理”一节介绍了处理非连续内存区的第三种技术。在这几节中我们的主题将涉及诸如内存管理区、内核映射、伙伴系统、slab 高速缓存和内存池。

页框管理

在第二章“硬件中的分页”一节中我们曾介绍过，Intel的Pentium处理器可以采用两种不同的页框大小：4KB和4MB（或者如果PAE被激活，则为2MB —— 参见第二章“物理地址扩展（PAE）分页机制”一节）。Linux采用4KB页框大小作为标准的内存分配单元。基于以下两个原因，这会使事情变得简单：

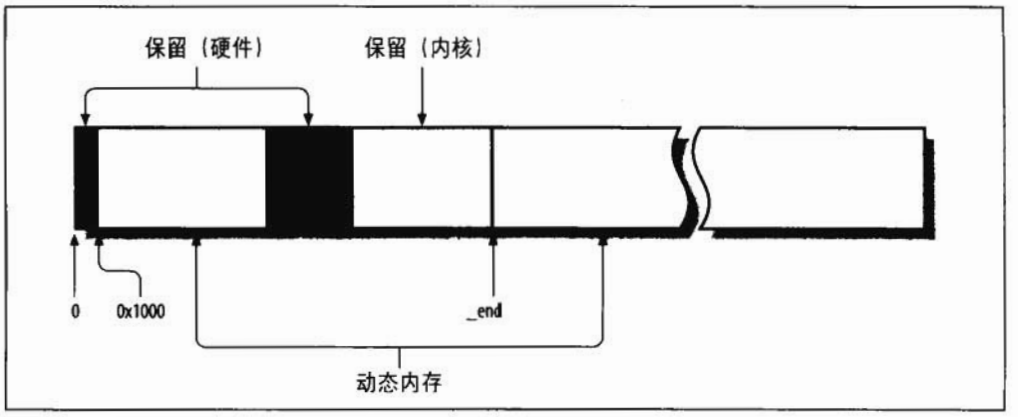


图 8-1：动态内存

- 由分页单元引发的缺页异常很容易得到解释，或者是由于请求的页存在但不允许进程对其访问，或者是由于请求的页不存在。在第二种情况下，内存分配器必须找到一个 4KB 的空闲页框，并将其分配给进程。
- 虽然 4KB 和 4MB 都是磁盘块大小的倍数，但是在绝大多数情况下，当主存和磁盘之间传输小块数据时更高效。

页描述符

内核必须记录每个页框当前的状态。例如，内核必须能区分哪些页框包含的是属于进程的页，而哪些页框包含的是内核代码或内核数据。类似地，内核还必须能够确定动态内存中的页框是否空闲。如果动态内存中的页框不包含有用的数据，那么这个页框就是空闲的。在以下情况下页框是不空闲的：包含用户态进程的数据、某个软件高速缓存的数据、动态分配的内核数据结构、设备驱动程序缓冲的数据、内核模块的代码等等。

页框的状态信息保存在一个类型为 page 的页描述符中，其中的字段如表 8-1 所示。所有的页描述符存放在 mem_map 数组中。因为每个描述符长度为 32 字节，所以 mem_map 所需要的空间略小于整个 RAM 的 1%。virt_to_page(addr) 宏产生线性地址 addr 对应的页描述符地址。pfn_to_page(pfn) 宏产生与页框号 Pfn 对应的页描述符地址。

表 8-1：页描述符的字段

类型	名字	说明
unsigned long	flags	一组标志（参见表8-2）。也对页框所在的管理区进行编号
atomic_t	_count	页框的引用计数器
atomic_t	_mapcount	页框中的页表项数目（如果没有则为-1）
unsigned long	private	可用于正在使用页的内核成分（例如，在缓冲页的情况下它是一个缓冲器头指针；参见第十五章的“块缓冲区和缓冲区首部”一节）。如果页是空闲的，则该字段由伙伴系统使用（参见本章后面）
struct address_space *	mapping	当页被插入页高速缓存中时使用（参见第十五章“页高速缓存”一节），或者当页属于匿名区时使用（参见第十七章的“匿名页的反向映射”一节）
unsigned long	index	作为不同的含义被几种内核成分使用。例如，它在页磁盘映像或匿名区中标识存放在页框中的数据的位置（参见第十五章），或者它存放一个换出页标识符（第十七章）
struct list_head	lru	包含页的最近最少使用（LRU）双向链表的指针

你不必现在就完全理解页描述符所有字段的作用。在接下来的章节中，我们常常会回到页描述符的字段。此外，有几个字段的含义还取决于页框是否空闲及什么样的内核成分在使用页框。

让我们较详细地描述以下两个字段：

_count

页的引用计数器。如果该字段为-1，则相应页框空闲，并可被分配给任一进程或内核本身；如果该字段的值大于或等于0，则说明页框被分配给了一个或多个进程，或用于存放一些内核数据结构。page_count()函数返回_count加1后的值，也就是该页的使用者的数目。

flags

包含多达32个用来描述页框状态的标志（参见表8-2）。对于每个PG_*xyz*标志，内核都定义了操纵其值的一些宏。通常，Page*Xyz*宏返回标志的值，而SetPage*Xyz*和ClearPage*Xyz*宏分别设置和清除相应的位。

表 8-2：描述页框状态的标志

标志名	含义
PG_locked	页被锁定。例如，在磁盘 I/O 操作中涉及的页
PG_error	在传输页时发生 I/O 错误
PG_referenced	刚刚访问过的页
PG_uptodate	在完成读操作后置位，除非发生磁盘 I/O 错误
PG_dirty	页已经被修改（参见第十七章的“PFRA 实现”一节）
PG_lru	页在活动或非活动页链表中[参见第十七章的“最近最少使用（LRU）链表”一节]
PG_active	页在活动页链表中[参看第十七章的“最近最少使用（LRU）链表”一节]
PG_slab	包含在 slab 中的页框（参见本章后面“内存区管理”一节）
PG_highmem	页框属于 ZONE_HIGHMEM 管理区(参见本章后面“非一致内存访问（NUMA）”一节)
PG_checked	由一些文件系统（如 Ext2 和 Ext3）使用的标志 (参见第十八章)
PG_arch_1	在 80x86 体系结构上没有使用
PG_reserved	页框留给内核代码或没有使用
PG_private	页描述符的 private 字段存放了有意义的数据
PG_writeback	正在使用 writepage 方法将页写到磁盘上（参见第十六章）
PG_nosave	系统挂起 / 唤醒时使用
PG_compound	通过扩展分页机制处理页框（参见第二章的“扩展分页”一节）
PG_swapcache	页属于对换高速缓存（参见第十七章的“交换高速缓存”一节）
PG_mappedtodisk	页框中的所有数据对应于磁盘上分配的块
PG_reclaim	为回收内存对页已经做了写入磁盘的标记
PG_nosave_free	系统挂起 / 恢复时使用

非一致内存访问（NUMA）

我们习惯上认为计算机内存是一种均匀、共享的资源。在忽略硬件高速缓存作用的情况下，我们期望不管内存单元处于何处，也不管 CPU 处于何处，CPU 对内存单元的访问都需要相同的时间。可惜，这种假设在某些体系结构上并不总是成立。例如，对于某些多处理器 Alpha 或 MIPS 计算机，这就不成立。

Linux 2.6 支持非一致内存访问（*Non-Uniform Memory Access*，*NUMA*）模型，在这

种模型中，给定CPU对不同内存单元的访问时间可能不一样。系统的物理内存被划分为几个节点（*node*）。在一个单独的节点内，任一给定CPU访问页面所需的时间都是相同的。然而，对不同的CPU，这个时间可能就不同。对每个CPU而言，内核都试图把耗时节点的访问次数减到最少，这就要小心地选择CPU最常引用的内核数据结构的存放位置（注1）。

每个节点中的物理内存又可以分为几个管理区（Eone），这我们将在下一节介绍。每个节点都有一个类型为pg_data_t的描述符，它的字段如表8-3所示。所有节点的描述符存放在一个单向链表中，它的第一个元素由pgdat_list变量指向。

表8-3：节点描述符的字段

类型	名字	说明
struct zone_t[ ]	node_zones	节点中管理区描述符的数组
struct zonelist_t[ ]	node_zonelists	页分配器使用的zonelist数据结构的数组（参见后面“内存管理区”一节）
int	nr_zones	节点中管理区的个数
struct page *	node_mem_map	节点中页描述符的数组
struct bootmem_data *	bdata	用在内核初始化阶段
unsigned long	node_start_pfn	节点中第一个页框的下标
unsigned long	node_present_pages	内存节点的大小，不包括洞（以页框为单位）
unsigned long	node_spanned_pages	节点的大小，包括洞（以页框为单位）
int	node_id	节点标识符
pg_data_t *	pgdat_next	内存节点链表中的下一项
wait_queue_head_t	kswapd_wait	*kswapd*页换出守护进程使用的等待队列（参见第十七章的“周期回收”一节）
struct task_struct *	kswapd	指针指向*kswapd*内核线程的进程描述符
int	kswapd_max_order	*kswapd*将要创建的空闲块大小取对数的值

注1：另外，Linux内核甚至在一些特殊的单处理器系统上使用NUMA，这些系统的物理地址空间中拥有巨大的“洞”。内核通过将有效物理地址的连续附属区域分配给不同的内存节点来处理这些体系结构。

我们同样只关注80x86体系结构。IBM兼容PC使用一致访问内存（UMA）模型，因此，并不真正需要NUMA的支持。然而，即使NUMA的支持没有编译进内核，Linux还是使用节点，不过，这是一个单独的节点，它包含了系统中所有的物理内存。因此，pgdat_list变量指向一个链表，此链表是由一个元素组成的，这个元素就是节点0描述符，它被存放在contig_page_data变量中。

在80x86结构中，把物理内存分组在一个单独的节点中可能显得没有用处；但是，这种方式有助于内存代码的处理更具有可移植性，因为内核假定在所有的体系结构中物理内存都被划分为一个或多个节点（注2）。

内存管理区

在一个理想的计算机体系结构中，一个页框就是一个内存存储单元，可用于任何事情：存放内核数据和用户数据、缓冲磁盘数据等等。任何种类的数据页都可以存放在任何页框中，没有什么限制。

但是，实际的计算机体系结构有硬件的制约，这限制了页框可以使用的方式。尤其是，Linux内核必须处理80x86体系结构的两种硬件约束：

- ISA总线的直接内存存取（DMA）处理器有一个严格的限制：它们只能对RAM的前16MB寻址。
- 在具有大容量RAM的现代32位计算机中，CPU不能直接访问所有的物理内存，因为线性地址空间太小。

为了应对这两种限制，Linux 2.6把每个内存节点的物理内存划分为3个管理区（zone）。在80x86 UMA体系结构中的管理区为：

ZONE_DMA
: 包含低于16 MB的内存页框

ZONE_NORMAL
: 包含高于16 MB且低于896 MB的内存页框

ZONE_HIGHMEM
: 包含从896MB开始高于896 MB的内存页框

注2： 我们还有这种设计选择的另外一个例子：即使硬件体系结构仅定义了两级页表，Linux也使用四级（参见第二章“Linux中的分页”一节）。

ZONE_DMA区包含的页框可以由老式基于ISA的设备通过DMA使用[第十三章“直接内存访问（DMA）”一节将进一步给出详细的信息]。

ZONE_DMA和ZONE_NORMAL区包含内存的“常规”页框，通过把它们线性地映射到线性地址空间的第4个GB，内核就可以直接进行访问（参见第二章的“内核页表”一节）。相反，ZONE_HIGHMEM区包含的内存页不能由内核直接访问，尽管它们也线性地映射到了线性地址空间的第4个GB（参见本章后面“高端内存页框的内核映射”一节）。在64位体系结构上ZONE_HIGHMEM区总是空的。

每个内存管理区都有自己的描述符。它的字段如表8-4所示。

表8-4：管理区描述符的字段

类型	名称	说明
unsigned long	free_pages	管理区中空闲页的数目
unsigned long	pages_min	管理区中保留页的数目（参见本章后面的“保留的页框池”一节）
unsigned long	pages_low	回收页框使用的下界；同时也被管理区分配器作为阈值使用（参见本章后面的“管理区分配器”一节）
unsigned long	pages_high	回收页框使用的上界；同时也被管理区分配器作为阈值使用
Unsigned long[]	lowmem_reserve	指明在处理内存不足的临界情况下每个管理区必须保留的页框数目
struct per_cpu_pageset[]	pageset	数据结构用于实现单一页框的特殊高速缓存（参见本章后面的“每CPU页框高速缓存”一节）
spinlock_t	lock	保护该描述符的自旋锁
struct free_area[]	free_area	标识出管理区中的空闲页框块（参见本章后面的“伙伴系统算法”一节）
spinlock_t	lru_lock	活动以及非活动链表使用的自旋锁
struct list head	active_list	管理区中的活动页链表（参见第十七章）
struct list head	inactive_list	管理区中的非活动页链表（参见第十七章）

表 8-4：管理区描述符的字段（续）

类型	名称	说明
unsigned long	nr_scan_active	回收内存时需要扫描的活动页数目（参见第十七章的“内存不足时回收”一节）
unsigned long	nr_scan_inactive	回收内存时需要扫描的非活动页数目
unsigned long	nr_active	管理区的活动链表上的页数目
unsigned long	nr_inactive	管理区的非活动链表上的页数目
unsigned long	pages_scanned	管理区内回收页框时使用的计数器
int	all_unreclaimable	在管理区中填满不可回收页时此标志被置位
int	temp_priority	临时管理区的优先级（回收页框时使用）
int	prev_priority	管理区优先级，范围在 12 和 0 之间（由回收页框算法使用，参见第十七章的“内存紧缺回收”一节）
wait_queue_head_t*	wait_table	进程等待队列的散列表，这些进程正在等待管理区中的某页
unsigned long	wait_table_size	等待队列散列表的大小
unsigned long	wait_table_bits	等待队列散列表数组大小，值为 2^{order}
struct pglist_data*	zone_pgdat	内存节点［参见前面的“非一致内存访问（NUMA）”一节］
struct page *	zone_mem_map	指向管理区的第一个页描述符的指针
unsigned long	zone_start_pfn	管理区第一个页框的下标
unsigned long	spanned_pages	以页为单位的管理区的总大小，包括洞
unsigned long	present_pages	以页为单位的管理区的总大小，不包括洞
char*	name	指针指向管理区的传统名称：“DMA”，“NORMAL”或“HighMem”

管理区结构中的许多字段用于回收页框，相关内容将在第十七章中描述。

每个页描述符都有到内存节点和到节点内管理区（包含相应页框）的链接。为节省空间，这些链接的存放方式与典型的指针不同，而是被编码成索引存放在 flags 字段的高位。

实际上，刻画页框的标志的数目是有限的，因此保留flags字段的最高位来编码特定内存节点和管理区号总是可能的（注3）。page_zone()函数接收一个页描述符的地址作为它的参数；它读取页描述符中flags字段的最高位，然后通过查看zone_table数组来确定相应管理区描述符的地址。在启动时用所有内存节点的所有管理区描述符的地址初始化这个数组。

当内核调用一个内存分配函数时，必须指明请求页框所在的管理区。内核通常指明它愿意使用哪个管理区。例如，如果一个页框必须直接映射在线性地址的第4个GB，但它又不用于ISA DMA的传输，那么，内核不是在ZONE_NORMAL区就是在ZONE_DMA区请求一个页框。当然，如果ZONE_NORMAL没有空闲页框，那么，应该从ZONE_DMA获取页框。为了在内存分配请求中指定首选管理区，内核使用zonelist数据结构，这就是管理区描述符指针数组。

保留的页框池

可以用两种不同的方法来满足内存分配请求。如果有足够的空闲内存可用，请求就会被立刻满足。否则，必须回收一些内存，并且将发出请求的内核控制路径阻塞，直到有内存被释放。

不过，当请求内存时，一些内核控制路径不能被阻塞——例如，这种情况发生在处理中断或在执行临界区内的代码时。在这些情况下，一条内核控制路径应当产生原子内存分配请求（使用GFP_ATOMIC标志；参见稍后的“分区页框分配器”一节）。原子请求从不被阻塞：如果没有足够的空闲页，则仅仅是分配失败而已。

尽管无法保证一个原子内存分配请求决不失败，但是内核会设法尽量减少这种不幸事件发生的可能性。为做到这一点，内核为原子内存分配请求保留了一个页框池，只有在内存不足时才使用。

保留内存的数量（以KB为单位）存放在min_free_kbytes变量中。它的初始值在内核初始化时设置，并取决于直接映射到内核线性地址空间第4个GB的物理内存的数量——也就是说，取决于包含在ZONE_DMA和ZONE_NORMAL内存管理区内的页框数目：

注3：为索引保留的位的数目取决于内核是否支持NUMA模型以及flags字段的大小。如果不支持NUMA，那么flags字段中管理区索引占两位，节点索引占一位（通常设为0）。在NUMA 32位体系结构上，flags中管理区索引占两位，节点数目占六位。最后，在NUMA 64位体系结构上，64位的flags字段中管理区索引占两位，节点数目占十位。

保留池的大小 $=\lfloor\sqrt{16 \times \text{直接映射内存}}\rfloor$（KB）

但是，min_free_kbytes的初始值不能小于128也不能大于65536（注4）。

ZONE_DMA和ZONE_NORMAL内存管理区将一定数量的页框贡献给保留内存，这个数目与两个管理区的相对大小成比例。例如，如果ZONE_NORMAL管理区比ZONE_DMA大8倍，那么页框的7/8从ZONE_NORMAL获得，而1/8从ZONE_DMA获得。

管理区描述符的pages_min字段存储了管理区内保留页框的数目。正如我们将在第十七章看到的，这个字段和pages_low、pages_high字段一起还在页框回收算法中起作用。pages_low字段总是被设为pages_min的值的5/4，而pages_high总是被设为pages_min的值的3/2。

分区页框分配器

被称作分区页框分配器（*zoned page frame allocator*）的内核子系统，处理对连续页框组的内存分配请求。它的主要组成如图8-2所示。

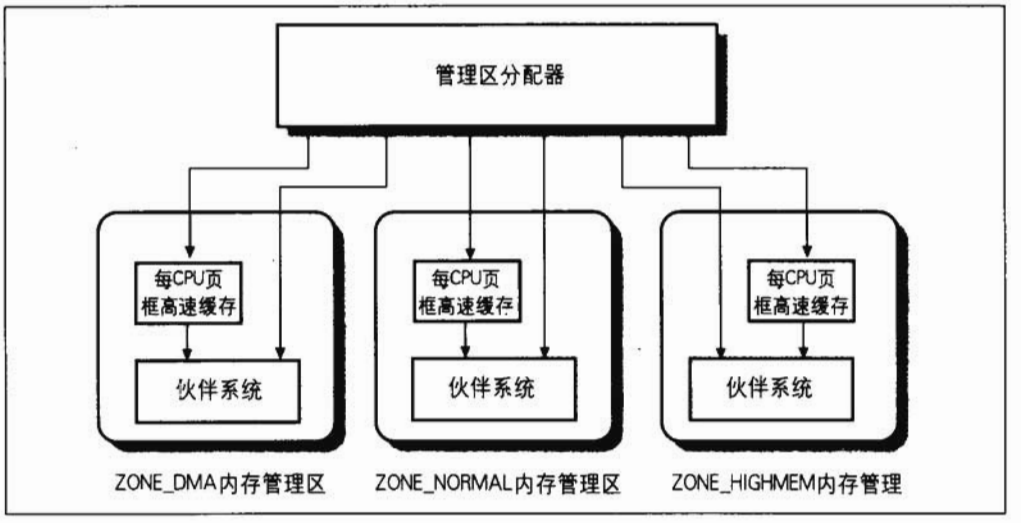


图8-2：分区页框分配器的组成

其中，名为“管理区分配器”部分接受动态内存分配与释放的请求。在请求分配的情况下，该部分搜索一个能满足所请求的一组连续页框内存的管理区（参见后面的“管理区

注4：稍后系统管理员可以通过写入*/proc/sys/vm/min_free_kbytes*文件或通过发出一个适当的sysctl()系统调用来更改保留内存的数量。

分配器”一节)。在每个管理区内，页框被名为伙伴系统”(参见后面的“伙伴系统算法”一节)的部分来处理。为达到更好的系统性能，一小部分页框保留在高速缓存中用于快速地满足对单个页框的分配请求(参见后面的“每CPU页框高速缓存”一节)。

请求和释放页框

可以通过6个稍有差别的函数和宏请求页框。除非另作说明，一般情况下，它们都返回第一个所分配页的线性地址，或者如果分配失败，则返回NULL。

alloc_pages(gfp_mask,order)

用这个函数请求2^{order}个连续的页框。它返回第一个所分配页框描述符的地址，或者如果分配失败，则返回NULL。

alloc_page(gfp_mask)

用于获得一个单独页框的宏；它扩展为：

```
alloc_pages(gfp_mask, 0)
```

它返回所分配页框描述符的地址，或者如果分配失败，则返回NULL。

__get_free_pages(gfp_mask,order)

该函数类似于alloc_pages()，但它返回第一个所分配页的线性地址。

__get_free_page(gfp_mask)

用于获得一个单独页框的宏；它扩展为：

```
__get_free_pages(gfp_mask, 0)
```

get_zeroed_page(gfp_mask)

函数用来获取填满0的页框；它调用：

```
alloc_pages(gfp_mask | __GFP_ZERO, 0)
```

然后返回所获取页框的线性地址。

__get_dma_pages(gfp_mask, order)

用这个宏获得适用于DMA的页框，它扩展为：

```
__get_free_pages(gfp_mask | __GFP_DMA, order)
```

参数gfp_mask是一组标志，它指明了如何寻找空闲的页框。能在gfp_mask中使用的标志如表8-5所示。

表8-5：用于请求页框的标志

标志	说明
__GFP_DMA	所请求的页框必须处于ZONE_DMA管理区。等价于GFP_DMA
__GFP_HIGHMEM	所请求的页框处于ZONE_HIGHMEM管理区
__GFP_WAIT	允许内核对等待空闲页框的当前进程进行阻塞
__GFP_HIGH	允许内核访问保留的页框池
__GFP_IO	允许内核在低端内存页上执行I/O传输以释放页框
__GFP_FS	如果清0，则不允许内核执行依赖于文件系统的操作
__GFP_COLD	所请求的页框可能为“冷的”（参见稍后的“每CPU页框高速缓存”一节）
__GFP_NOWARN	一次内存分配失败将不会产生警告信息
__GFP_REPEAT	内核重试内存分配直到成功
__GFP_NOFAIL	与_ _GFP_REPEAT相同
__GFP_NORETRY	一次内存分配失败后不再重试
__GFP_NO_GROW	slab分配器不允许增大slab高速缓存（参见稍后的“slab分配器”一节）
__GFP_COMP	属于扩展页的页框（参见第二章的“扩展分页”一节）
__GFP_ZERO	任何返回的页框必须被填满0

实际上，Linux使用预定义标志值的组合，如表8-6所示，组名就是你在6个页框分配函数中所遇到的参数。

表8-6：用于请求页框的一组标志值

组名	相应标志
GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_WAIT
GFP_NOFS	__GFP_WAIT | __GFP_IO
GFP_KERNEL	__GFP_WAIT | __GFP_IO | __GFP_FS
GFP_USER	__GFP_WAIT | __GFP_IO | __GFP_FS
GFP_HIGHUSER	__GFP_WAIT | __GFP_IO | __GFP_FS_ | _GFP_HIGHMEM

__GFP_DMA和__GFP_HIGHMEM标志被称作管理区修饰符；它们标示寻找空闲页框时内核所搜索的管理区。contig_page_data节点描述符的node_zonelists字段是一个管

理区描述符链表的数组，它代表后备管理区：对管理区修饰符的每一个设置，相应的链表包含的内存管理区能在原来的管理区缺少页框的情况下被用于满足内存分配请求。在80x86 UMA体系结构中，后备管理区如下：

- 如果__GFP_DMA标志被置位，则只能从ZONE_DMA内存管理区获取页框。
- 否则，如果__GFP_HIGHMEM标志没有被置位，则只能按优先次序从ZONE_NORMAL和ZONE_DMA内存管理区获取页框。
- 否则（__GFP_HIGHMEM标志被置位），则可以按优先次序从ZONE_HIGHMEM、ZONE_NORMAL和ZONE_DMA内存管理区获得页框。

下面4个函数和宏中的任一个都可以释放页框：

__free_pages(page, order)

该函数先检查page指向的页描述符；如果该页框未被保留（PG_reserved标志为0），就把描述符的count字段减1。如果count值变为0，就假定从与page对应的页框开始的2^{order}个连续页框不再被使用。在这种情况下，该函数释放页框，正如后面的“管理区分配器”一节描述的那样。

free_pages(addr, order)

这个函数类似于__free_pages(page, order)，但是它接收的参数为要释放的第一个页框的线性地址addr。

__free_page(page)

这个宏释放page所指描述符对应的页框；它扩展为：

```
__free_pages(page, 0)
```

free_page(addr)

该宏释放线性地址为addr的页框。它扩展为：

```
free_pages(addr,0)
```

高端内存页框的内核映射

与直接映射的物理内存末端、高端内存的始端所对应的线性地址存放在high_memory变量中，它被设置为896MB。896MB边界以上的页框并不映射在内核线性地址空间的第4个GB，因此，内核不能直接访问它们。这就意味着，返回所分配页框线性地址的页分配器函数不适用于高端内存，即不适用于ZONE_HIGHMEM内存管理区内的页框。

例如，假定内核调用__get_free_pages(GFP_HIGHMEM,0)在高端内存分配一个页框。如果分配器在高端内存确实分配了一个页框，那么__get_free_pages()不能返回它的

线性地址，因为它根本就不存在，因此，函数返回NULL。依次类推，内核不能使用这个页框；甚至更坏的情况下，也不能释放该页框，因为内核已经丢失了它的踪迹。

在64位硬件平台上不存在这个问题，因为可使用的线性地址空间远大于能安装的RAM大小，简言之，这些体系结构的ZONE_HIGHMEM管理区总是空的。但是在32位平台上，如80x86体系结构，Linux设计者不得不找到某种方法来允许内核使用所有可使用的RAM，达到PAE所支持的64GB。采用的方法如下：

- 高端内存页框的分配只能通过alloc_pages()函数和它的快捷函数alloc_page()。这些函数不返回第一个被分配页框的线性地址，因为如果该页框属于高端内存，那么这样的线性地址根本不存在。取而代之，这些函数返回第一个被分配页框的页描述符的线性地址。这些线性地址总是存在的，因为所有页描述符一旦被分配在低端内存中，它们在内核初始化阶段就不会改变。
- 没有线性地址的高端内存中的页框不能被内核访问。因此，内核线性地址空间的最后128MB中的一部分专门用于映射高端内存页框。当然，这种映射是暂时的，否则只有128MB的高端内存可以被访问。取而代之，通过重复使用线性地址，使得整个高端内存能够在不同的时间被访问。

内核可以采用三种不同的机制将页框映射到高端内存：分别叫做永久内核映射、临时内核映射及非连续内存分配。在本节中，我们集中于前两种技术；第三种技术将在本章后面“非连续内存区管理”一节进行讨论。

建立永久内核映射可能阻塞当前进程；这发生在空闲页表项不存在时，也就是在高端内存上没有页表项可以用作页框的“窗口”时。因此，永久内核映射不能用于中断处理程序和可延迟函数。相反，建立临时内核映射决不会要求阻塞当前进程；不过，它的缺点是只有很少的临时内核映射可以同时建立起来。

使用临时内核映射的内核控制路径必须保证当前没有其他的内核控制路径在使用同样的映射。这意味着内核控制路径永远不能被阻塞，否则其他内核控制路径有可能使用同一个窗口来映射其他的高端内存页。

当然，这些技术中没有一种可以确保对整个RAM同时进行寻址。毕竟，只有128MB的线性地址留给映射高端内存，尽管PAE支持系统高达64GB RAM。

永久内核映射

永久内核映射允许内核建立高端页框到内核地址空间的长期映射。它们使用主内核页表中一个专门的页表，其地址存放在pkmap_page_table变量中。页表中的表项数由LAST_PKMAP宏产生。页表照样包含512或1024项，这取决于PAE是否被激活[参见

第二章“物理地址扩展（PAE）分页机制”一节]；因此，内核一次最多访问2MB或4MB的高端内存。

该页表映射的线性地址从PKMAP_BASE开始。pkmap_count数组包含LAST_PKMAP个计数器，pkmap_page_table页表中的每一项都有一个。我们区分以下三种情况：

计数器为0

对应的页表项没有映射任何高端内存页框，并且是可用的。

计数器为1

对应的页表项没有映射任何高端内存页框，但是它不能使用，因为自从它最后一次使用以来，其相应的TLB表现还未被刷新。

计数器为n（远大于1）

相应的页表项映射一个高端内存页框，这意味着正好有$n-1$个内核成分在使用这个页框。

为了记录高端内存页框与永久内核映射包含的线性地址之间的联系，内核使用了page_address_htable散列表。该表包含一个page_address_map数据结构，用于为高端内存中的每一个页框进行当前映射。而该数据结构还包含一个指向页描述符的指针和分配给该页框的线性地址。

page_address()函数返回页框对应的线性地址，如果页框在高端内存中并且没有被映射，则返回NULL。这个函数接受一个页描述符指针page作为其参数，并区分以下两种情况：

1. 如果页框不在高端内存中（PG_highmem标志为0），则线性地址总是存在并且是通过计算页框下标，然后将其转换成物理地址，最后根据相应的物理地址得到线性地址。这是由下面的代码完成的：

```
__va((unsigned long)(page - mem_map) << 12)
```

2. 如果页框在高端内存(PG_highmem标志为1)中，该函数就到page_address_htable散列表中查找。如果在散列表中找到页框，page_address()就返回它的线性地址，否则返回NULL。

kmap()函数建立永久内核映射。本质上它等价于下列代码：

```
void * kmap(struct page * page)
{
   if (!PageHighMem(page))
      return page_address(page);
   return kmap_high(page);
}
```

如果页框确实属于高端内存，则调用kmap_high()函数。这个函数本质上等价于下列代码：

```
void * kmap_high(struct page * page)
{
   unsigned long vaddr;
   spin_lock(&kmap_lock);
   vaddr = (unsigned long) page->virtual;
   if (!vaddr)
      vaddr = map_new_virtual(page);
   pkmap_count[(vaddr-PKMAP_BASE) >> PAGE_SHIFT]++;
   spin_unlock(&kmap_lock);
   return (void *) vaddr;
}
```

该函数获取kmap_lock自旋锁，以保护页表免受多处理器系统上的并发访问。注意，没有必要禁止中断，因为中断处理程序和可延迟函数不能调用kmap()。接下来，kmap_high()函数检查页框是否已经通过调用page_address()被映射。如果不是，该函数调用map_new_virtual()函数把页框的物理地址插入到pkmap_page_table的一个项中并在page_address_htable散列表中加入一个元素。然后，kmap_high()使页框的线性地址所对应的计数器加1来将调用该函数的新内核成分考虑在内。最后，kmap_high()释放kmap_lock自旋锁并返回对该页框进行映射的线性地址。

map_new_virtual()函数本质上执行两个嵌套循环：

```
for (;;) {
    int count;
    DECLARE_WAITQUEUE(wait, current);
    for (count = LAST_PKMAP; count >0; --count) {
        last_pkmap_nr = (last_pkmap_nr + 1) & (LAST_PKMAP - 1);
        if (!last_pkmap_nr) {
            flush_all_zero_pkmaps();
            count = LAST_PKMAP;
}
if (!pkmap_count[last_pkmap_nr]) {
            unsigned long vaddr = PKMAP_BASE +
                                  (last_pkmap_nr << PAGE_SHIFT);
            set_pte(&(pkmap_page_table[last_pkmap_nr]),
                    mk_pte(page, _  _pgprot(0x63)));
            pkmap_count[last_pkmap_nr] = 1;
            set_page_address(page, (void *) vaddr);
            return vaddr;
        }
    }
    current->state = TASK_UNITERRUPTIBLE;
    add_wait_queue(&pkmap_map_wait, &wait);
    spin_unlock(&kmap_lock);
    schedule();
    remove_wait_queue(&pkmap_map_wait, &wait);
    spin_lock(&kmap_lock);
    if (page_address(page))
```

```
        return (unsigned long) page_address(page);
    }
```

在内循环中，该函数扫描pkmap_count中的所有计数器直到找到一个空值。当在pkmap_count中找到了一个未使用的项时，大的if代码块运行。这段代码确定该项对应的线性地址，为它在pkmap_page_table页表中创建一个项，将count置1，因为该项现在已经被使用了，调用set_page_address()函数插入一个新元素到page_address_htable散列表中，并返回线性地址。

函数从上次停止的地方开始，穿越pkmap_count数组执行循环。这是函数通过将pkmap_page_table页表中上次使用过页表项的索引保存在一个名为last_pkmap_nr的变量中做到的。因此，搜索从上次因调用map_new_virtual()函数而跳出的地方重新开始。

当在pkmap_count中搜索到最后一个计数器时，就又从下标为0的计数器重新开始搜索。不过，在继续之前，map_new_virtual()调用flush_all_zero_pkmaps()函数来开始寻找计数器为1的另一趟扫描。每个值为1的计数器都表示在pkmap_page_table页表中表项是空闲的，但不能使用，因为相应的TLB表项还没有被刷新。flush_all_zero_pkmaps()把它们的计数器重置为0，删除page_address_htable散列表中对应的元素，并在pkmap_page_table的所有项上进行TLB刷新。

如果内循环在pkmap_count中没有找到空的计数器，map_new_virtual()函数就阻塞当前进程，直到某个进程释放了pkmap_page_table页表中的一个表项。通过把current插入到pkmap_map_wait等待队列，把current状态设置为TASK_UNINTERRUPTIBLE并调用schedule()放弃CPU来达到此目的。一旦进程被唤醒，该函数就通过调用page_address()检查是否存在另一个进程已经映射了该页；如果还没有其他进程映射该页，则内循环重新开始。

kunmap()函数撤销先前由kmap()建立的永久内核映射。如果页确实在高端内存中，则调用kunmap_high()函数，它本质上等价于下列代码：

```
void kunmap_high(struct page * page)
{
    spin_lock(&kmap_lock);
    if ((--pkmap_count[((unsigned long)page_address(page)
                        -PKMAP_BASE)>>PAGE_SHIFT]) == 1)
       if (waitqueue_active(&pkmap_map_wait))
           wake_up(&pkmap_map_wait);
    spin_unlock(&kmap_lock);
}
```

中括号内的表达式从页的线性地址计算出pkmap_count数组的索引。计数器被减1并与1相比。匹配成功表明没有进程在使用页。该函数最终能唤醒由map_new_virtual()添加在等待队列中的进程（如果有的话）。

临时内核映射

临时内核映射比永久内核映射的实现要简单；此外，它们可以用在中断处理程序和可延迟函数的内部，因为它们从不阻塞当前进程。

在高端内存的任一页框都可以通过一个“窗口”（为此而保留的一个页表项）映射到内核地址空间。留给临时内核映射的窗口数是非常少的。

每个CPU都有它自己的包含13个窗口的集合，它们用enum km_type数据结构表示。该数据结构中定义的每个符号，如KM_BOUNCE_READ、KM_USER0或KM_PTE0，标识了窗口的线性地址。

内核必须确保同一窗口永不会被两个不同的控制路径同时使用。因此，km_type结构中的每个符号只能由一种内核成分使用，并以该成分命名。最后一个符号KM_TYPE_NR本身并不表示一个线性地址，但由每个CPU用来产生不同的可用窗口数。

在km_type中的每个符号（除了最后一个）都是固定映射的线性地址的一个下标（参见第二章“固定映射的线性地址”一节）。enum fixed_addresses数据结构包含符号FIX_KMAP_BEGIN和FIX_KMAP_END；把后者赋给下标FIX_KMAP_BEGIN+(KM_TYPE_NR*NR_CPUS)-1。在这种方式下，系统中的每个CPU都有KM_TYPE_NR个固定映射的线性地址。此外，内核用fix_to_virt(FIX_KMAP_BEGIN)线性地址对应的页表项的地址初始化kmap_pte变量。

为了建立临时内核映射，内核调用kmap_atomic()函数，它本质上等价于下列代码：

```
void * kmap_atomic(struct page * page, enum km_type type)
{
    enum fixed_addresses idx;
    unsigned long vaddr;

    current_thread_info()->preempt_count++;
    if (!PageHighMem(page))
        return page_address(page);
    idx = type + KM_TYPE_NR * smp_processor_id();
    vaddr = fix_to_virt(FIX_KMAP_BEGIN + idx);
    set_pte(kmap_pte-idx, mk_pte(page, 0x063));
    __flush_tlb_single(vaddr);
    return (void *) vaddr;
}
```

type参数和CPU标识符（通过smp_processor_id()）指定必须用哪个固定映射的线性地址映射请求页。如果页框不属于高端内存，则该函数返回页框的线性地址；否则，用页的物理地址及Present、Accessed、Read/Write和Dirty位建立该固定映射的线性地址对应的页表项。最后，该函数刷新适当的TLB项并返回线性地址。

为了撤销临时内核映射，内核使用kunmap_atomic()函数。在80x86结构中，这个函数减少当前进程的preempt_count；因此，如果在请求临时内核映像之前能抢占内核控制路径，那么在同一个映射被撤销后可以再次抢占。此外，kunmap_atomic()检查当前进程的TIF_NEED_RESCHED标志是否被置位，如果是，就调用schedule()。

伙伴系统算法

内核应该为分配一组连续的页框而建立一种健壮、高效的分配策略。为此，必须解决著名的内存管理问题，也就是所谓的外碎片（*external fragmentation*）。频繁地请求和释放不同大小的一组连续页框，必然导致在已分配页框的块内分散了许多小块的空闲页框。由此带来的问题是，即使有足够的空闲页框可以满足请求，但要分配一个大块的连续页框就可能无法满足。

从本质上说，避免外碎片的方法有两种：

- 利用分页单元把一组非连续的空闲页框映射到连续的线性地址区间。
- 开发一种适当的技术来记录现存的空闲连续页框块的情况，以尽量避免为满足对小块的请求而分割大的空闲块。

基于以下三种原因，内核首选第二种方法：

- 在某些情况下，连续的页框确实是必要的，因为连续的线性地址不足以满足请求。一个典型的例子就是给DMA处理器分配缓冲区的内存请求（参见第十三章）。因为当在一次单独的I/O操作中传送几个磁盘扇区的数据时，DMA忽略分页单元而直接访问地址总线，因此，所请求的缓冲区就必须位于连续的页框中。
- 即使连续页框的分配并不是很必要，但它在保持内核页表不变方面所起的作用也是不容忽视的。修改页表会怎样呢？从第二章我们知道，频繁地修改页表势必导致平均访问内存次数的增加，因为这会使CPU频繁地刷新转换后援缓冲器（TLB）的内容。
- 内核通过4MB的页可以访问大块连续的物理内存。这样减少了转换后援缓冲器的失效率，因此提高了访问内存的平均速度［参见第二章“转换后援缓冲器（TLB）”一节］。

Linux采用著名的伙伴系统（*buddy system*）算法来解决外碎片问题。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1, 2, 4, 8, 16, 32, 64, 128, 256, 512和1024个连续的页框。对1024个页框的最大请求对应着4MB大小的连续RAM块。每个块的第一个页框的物理地址是该块大小的整数倍。例如，大小为16个页框的块，其起始地址是 16×2^{12}（$2^{12} = 4096$，这是一个常规页的大小）的倍数。

我们通过一个简单的例子来说明该算法的工作原理。

假设要请求一个256个页框的块（即1MB）。算法先在256个页框的链表中检查是否有一个空闲块。如果没有这样的块，算法会查找下一个更大的页块，也就是，在512个页框的链表中找一个空闲块。如果存在这样的块，内核就把256的页框分成两等份，一半用作满足请求，另一半插入到256个页框的链表中。如果在512个页框的块链表中也没找到空闲块，就继续找更大的块——1024个页框的块。如果这样的块存在，内核把1024个页框块的256个页框用作请求，然后从剩余的768个页框中拿512个插入到512个页框的链表中，再把最后的256个插入到256个页框的链表中。如果1024个页框的链表还是空的，算法就放弃并发出错信号。

以上过程的逆过程就是页框块的释放过程，也是该算法名字的由来。内核试图把大小为b的一对空闲伙伴块合并为一个大小为$2b$的单独块。满足以下条件的两个块称为伙伴：

- 两个块具有相同的大小，记作b。
- 它们的物理地址是连续的。
- 第一块的第一个页框的物理地址是$2 \times b \times 2^{12}$的倍数。

该算法是迭代的，如果它成功合并所释放的块，它会试图合并$2b$的块，以再次试图形成更大的块。

数据结构

Linux 2.6为每个管理区使用不同的伙伴系统。因此，在80x86结构中，有三种伙伴系统：第一种处理适合ISA DMA的页框，第二种处理“常规”页框，第三种处理高端内存页框。每个伙伴系统使用的主要数据结构如下：

- 前面介绍过的mem_map数组。实际上，每个管理区都关系到mem_map元素的子集。子集中的第一个元素和元素的个数分别由管理区描述符的zone_mem_map和size字段指定。
- 包含有11个元素、元素类型为free_area的一个数组，每个元素对应一种块大小。该数组存放在管理区描述符的free_area字段中。

我们考虑管理区描述符中free_area数组的第k个元素，它标识所有大小为2^k的空闲块。这个元素的free_list字段是双向循环链表的头，这个双向循环链表集中了大小为2^k页的空闲块对应的页描述符。更精确地说，该链表包含每个空闲页框块（大小为2^k）的起始页框的页描述符；指向链表中相邻元素的指针存放在页描述符的lru字段中（注5）。

注5：　正如我们稍后将看到的，当页不空闲时页描述符的lru字段可被用于其他目的。

除了链表头外，free_area数组的第k个元素同样包含字段nr_free，它指定了大小为2^k页的空闲块的个数。当然，如果没有大小为2^k的空闲页框块，则nr_free等于0且free_list为空（free_list的两个指针都指向它自己的free_list字段）。

最后，一个2^k的空闲页块的第一个页的描述符的private字段存放了块的order，也就是数字k。正是由于这个字段，当页块被释放时，内核可以确定这个块的伙伴是否也空闲，如果是的话，它可以把两个块结合成大小为2^{k+1}页的单一块。应当注意的是，直到Linux 2.6.10，内核使用了10组标志来对这种信息进行编码。

分配块

__rmqueue()函数用来在管理区中找到一个空闲块。该函数需要两个参数：管理区描述符的地址和order，order表示请求的空闲页块大小的对数值（0表示一个单页块，1表示一个两页块，依次类推）。如果页框被成功分配，__rmqueue()函数就返回第一个被分配页框的页描述符。否则，函数返回NULL。

__rmqueue()函数假设调用者已经禁止了本地中断并获得了保护伙伴系统数据结构的zone->lock自旋锁。从所请求order的链表开始，它扫描每个可用块链表（由不指向自己的链表项表示）进行循环搜索，如果需要搜索更大的order，就继续搜索：

```
struct free_area *area;
unsigned int current_order;

for (current_order=order; current_order<11; ++current_order) {
    area = zone->free_area + current_order;
    if (!list_empty(&area->free_list))
        goto block_found;
}
return NULL;
```

如果直到循环结束还没有找到合适的空闲块，那么__rmqueue()就返回NULL。否则，找到了一个合适的空闲块，在这种情况下，从链表中删除它的第一个页框描述符，并减少管理区描述符中的free_ pages的值：

```
block_found:
    page = list_entry(area->free_list.next, struct page, lru);
    list_del(&page->lru);
    ClearPagePrivate(page);
    page->private = 0;
    area->nr_free--;
    zone->free_pages -= 1UL << order;
```

如果从curr_order链表中找到的块大于请求的order，就执行一个while循环。这几行代码蕴含的原理如下：当为了满足2^h个页框的请求而有必要使用2^k个页框的块时（$h <$

k)，程序就分配前面的2^h个页框，而把后面2^k-2^h个页框循环再分配给free_area链表中下标在h到k之间的元素：

```
size = 1 << curr_order;
while (curr_order > order) {
    area--;
    curr_order--;
    size >>= 1;
    buddy = page + size;
    /* 插入伙伴作为链表中第一个元素 */
    list_add(&buddy->lru, &area->free_list);
    area->nr_free++;
    buddy->private = curr_order;
    SetPagePrivate(buddy);
}
return page;
```

因为__rmqueue()函数已经找到了合适的空闲块，所以它返回所分配的第一个页框对应的页描述符的地址page。

释放块

__free_pages_bulk()函数按照伙伴系统的策略释放页框。它使用3个基本输入参数(注6)：

page

被释放块中所包含的第一个页框描述符的地址。

zone

管理区描述符的地址。

order

块大小的对数。

该函数假设调用者已经禁止本地中断并获得了保护伙伴系统数据结构的zone->lock自旋锁。__free_pages_bulk()首先声明和初始化一些局部变量：

```
struct page * base = zone->zone_mem_map;
unsigned long buddy_idx, page_idx = page - base;
struct page * buddy, * coalesced;
int order_size = 1 << order;
```

page_idx局部变量包含块中第一个页框的下标，这是相对于管理区中的第一个页框而言的。

注6：由于性能的原因，这个内联函数还使用另一个参数；但是它的值可以由正文中说明的3个基本参数决定。

order_size局部变量用于增加管理区中空闲页框的计数器：

```
zone->free_pages += order_size;
```

现在函数开始执行循环，最多循环(10-order)次，每次都尽量把一个块和它的伙伴进行合并。函数以最小的块开始，然后向上移动到顶部：

```
while (order < 10) {
    buddy_idx = page_idx ^ (1 << order);
    buddy = base + buddy_idx;
    if (!page_is_buddy(buddy, order))
        break;
    list_del(&buddy->lru);
    zone->free_area[order].nr_free--;
    ClearPagePrivate(buddy);
    buddy->private = 0;
    page_idx &= buddy_idx;
    order++;
}
```

在循环体内，函数寻找块的下标buddy_idx，它是拥有page_idx页描述符下标的块的伙伴。结果这个下标可以被简单地如下计算：

```
buddy_idx = page_idx ^ (1 << order);
```

实际上，使用（1 < order）掩码的异或（XOR）转换page_idx第order位的值。因此，如果这个位原先是0，buddy_idx就等于page_idx + order_size；相反，如果这个位原先是1，buddy_idx就等于page_idx - order_size。

一旦知道了伙伴块下标，就可以通过下式很容易地获得伙伴块的页描述符：

```
buddy = base + buddy_idx;
```

现在函数调用page_is_buddy()来检查buddy是否描述了大小为order_size的空闲页框块的第一个页。

```
int page_is_buddy(struct page *page, int order)
{
    if (PagePrivate(buddy) && page->private == order &&
          !PageReserved(buddy) && page_count(page) ==0)
        return 1;
    return 0;
}
```

正如所见，buddy的第一个页必须为空闲（_count字段等于－1），它必须属于动态内存（PG_reserved位清零），它的private字段必须有意义（PG_private位置位），最后private字段必须存放将要被释放的块的order。

如果所有这些条件都符合，伙伴块就被释放，并且函数将它从以order排序的空闲块链表上删除，并再执行一次循环以寻找两倍大小的伙伴块。

如果page_is_buddy()中至少有一个条件没有被满足，则该函数跳出循环，因为获得的空闲块不能再和其他空闲块合并。函数将它插入适当的链表并以块大小的order更新第一个页框的private字段。

```
coalesced = base + page_idx;
coalesced->private = order;
SetPagePrivate(coalesced);
list_add(&coalesced->lru, &zone->free_area[order].free_list);
zone->free_area[order].nr_free++;
```

每CPU页框高速缓存

正如我们将在本章稍后看到的，内核经常请求和释放单个页框。为了提升系统性能，每个内存管理区定义了一个“每CPU”页框高速缓存。所有“每CPU”高速缓存包含一些预先分配的页框，它们被用于满足本地CPU发出的单一内存请求。

实际上，这里为每个内存管理区和每个CPU提供了两个高速缓存：一个热高速缓存，它存放的页框中所包含的内容很可能就在CPU硬件高速缓存中；还有一个冷高速缓存。

如果内核或用户态进程在刚分配到页框后就立即向页框写，那么从热高速缓存中获得页框就对系统性能有利。实际上，每次对页框存储单元的访问将都会导致从另一个页框中给硬件高速缓存“窃取”一行——当然，除非硬件高速缓存包含有一行：它映射刚被访问的“热”页框单元。

反过来，如果页框将要被DMA操作填充，那么从冷高速缓存中获得页框是方便的。在这种情况下，不会涉及到CPU，并且硬件高速缓存的行不会被修改。从冷高速缓存获得页框为其他类型的内存分配保存了热页框储备。

实现每CPU页框高速缓存的主要数据结构是存放在内存管理区描述符的pageset字段中的一个per_cpu_pageset数组数据结构。该数组包含为每个CPU提供的一个元素；这个元素依次由两个per_cpu_pages描述符组成，一个留给热高速缓存而另一个留给冷高速缓存。per_cpu_pages描述符的字段在表8-7中列出。

表8-7：per_cpu_pages描述符的字段

类型	名称	描述
int	count	高速缓存中的页框个数
int	low	下界，表示高速缓存需要补充

表8-7：per_cpu_pages描述符的字段（续）

类型	名称	描述
int	high	上界，表示高速缓存用尽
int	batch	在高速缓存中将要添加或被删去的页框个数
strut list_head	list	高速缓存中包含的页框描述符链表

内核使用两个位标来监视热高速缓存和冷高速缓存的大小：如果页框个数低于下界low，内核通过从伙伴系统中分配batch个单一页框来补充对应的高速缓存；否则，如果页框个数高过上界high，内核从高速缓存中释放batch个页框到伙伴系统中。值batch、low和high本质上取决于内存管理区中包含的页框个数。

通过每CPU页框高速缓存分配页框

buffered_rmqueue()函数在指定的内存管理区中分配页框。它使用每CPU页框高速缓存来处理单一页框请求。

参数为内存管理区描述符的地址，请求分配的内存大小的对数order，以及分配标志gfp_flags。如果gfp_flags中的__GFP_COLD标志被置位，那么页框应当从冷高速缓存中获取，否则它应从热高速缓存中获取（此标志只对单一页框请求有意义）。该函数本质上执行如下操作：

1. 如果order不等于0，每CPU页框高速缓存就不能被使用：函数跳到第4步。
2. 检查由__GFP_COLD标志所标识的内存管理区本地每CPU高速缓存是否需要补充（per_cpu_pages描述符的count字段小于或等于low字段）。在这种情况下，它执行如下子步骤：
   a. 通过反复调用__rmqueue()函数从伙伴系统中分配batch个单一页框。
   b. 将已分配页框的描述符插入高速缓存链表中。
   c. 通过给count增加实际被分配页框的个数来更新它。
3. 如果count值为正，则函数从高速缓存链表获得一个页框，count减1并跳到第5步。（注意，每CPU页框高速缓存有可能为空，当在第2a步调用__rmqueue()函数而分配页框失败时就会发生这种情况。）
4. 到这里，内存请求还没有被满足，或者是因为请求跨越了几个连续页框，或者是因为被选中的页框高速缓存为空。调用__rmqueue()函数从伙伴系统中分配所请求的页框。

5. 如果内存请求得到满足，函数就初始化（第一个）页框的页描述符：清除一些标志，将private字段置0，并将页框引用计数器置1。此外，如果gfp_flags中的__GPF_ZERO标志被置位，则函数将被分配的内存区域填充0。
6. 返回（第一个）页框的页描述符地址，如果内存分配请求失败则返回NULL。

释放页框到每CPU页框高速缓存

为了释放单个页框到每CPU页框高速缓存，内核使用free_hot_page()和free_cold_page()函数。它们都是free_hot_cold_page()函数的简单封装，接收的参数为将要释放的页框的描述符地址page和cold标志（指定是热高速缓存还是冷高速缓存）。

free_hot_cold_page()函数执行如下操作：

1. 从page->flags字段获取包含该页框的内存管理区描述符地址［参见前面的“非一致内存访问（NUMA）”一节］。
2. 获取由cold标志选择的管理区高速缓存的per_cpu_pages描述符的地址。
3. 检查高速缓存是否应该被清空：如果count值高于或等于high，则调用free_pages_bulk()函数，将管理区描述符、将被释放的页框个数（batch字段）、高速缓存链表的地址以及数字0（为0到order个页框）传递给该函数。free_pages_bulkl()函数依次反复调用__free_pages_bulk()函数来释放指定数量的（从高速缓存链表获得的）页框到内存管理区的伙伴系统中。
4. 把释放的页框添加到高速缓存链表上，并增加count字段。

应该注意的是，在当前的Linux 2.6内核版本中，从没有页框被释放到冷高速缓存中：至于硬件高速缓存，内核总是假设被释放的页框是热的。当然，这并不意味着冷高速缓存是空的：当达到下界时通过buffered_rmqueue()补充冷高速缓存。

管理区分配器

管理区分配器是内核页框分配器的前端。该成分必须分配一个包含足够多空闲页框的内存管理区，使它能满足内存请求。这个任务并不像第一眼看上去那么简单，因为管理区分配器必须满足几个目标：

- 它应当保护保留的页框池（参见前面的“保留的页框池”一节）。
- 当内存不足且允许阻塞当前进程时，它应当触发页框回收算法（参见第十七章）；一旦某些页框被释放，管理区分配器将再次尝试分配。
- 如果可能，它应当保存小而珍贵的ZONE_DMA内存管理区。例如，如果是对

ZONE_NORMAL或ZONE_HIGHMEM页框的请求，那么管理区分配器会不太愿意分配ZONE_DMA内存管理区中的页框。

我们在前面的“分区页框分配器”一节中已经看到，对一组连续页框的每次请求实质上是通过执行alloc_pages宏来处理的。接着，这个宏又依次调用__alloc_pages()函数，该函数是管理区分配器的核心。它接收以下3个参数：

gfp_mask

在内存分配请求中指定的标志（参见前面的表8-5）。

order

将要分配的一组连续页框数量的对数（即要分配2^{order}个连续的页框）。

zonelist

指向zonelist数据结构的指针，该数据结构按优先次序描述了适于内存分配的内存管理区。

__alloc_pages()函数扫描包含在zonelist数据结构中的每个内存管理区。实现代码如下：

```
.
for (i = 0; (z=zonelist->zones[i]) != NULL; i++) {
    if (zone_watermark_ok(z, order, ...)) {
       page = buffered_rmqueue(z, order, gfp_mask);
       if (page)
           return page;
    }
}
```

对于每个内存管理区，该函数将空闲页框的个数与一个阈值作比较，该阈值取决于内存分配标志、当前进程的类型以及管理区被函数检查过的次数。实际上，如果空闲内存不足，那么每个内存管理区一般会被检查几遍，每一遍在所请求的空闲内存最低量的基础上使用更低的阈值扫描。因此前面一段代码在__alloc_pages()函数体内被复制了几次，每次变化很小。在前面的“每CPU页框高速缓存”一节中已经对buffered_rmqueue()函数作了描述：它返回第一个被分配的页框的页描述符；如果内存管理区没有所请求大小的一组连续页框，则返回NULL。

zone_watermark_ok()辅助函数接收几个参数，它们决定内存管理区中空闲页框个数的阈值min。特别是，如果满足下列两个条件则该函数返回值1：

1. 除了被分配的页框外，在内存管理区中至少还有min个空闲页框，不包括为内存不足保留的页框（管理区描述符的lowmem_reserve字段）。
2. 除了被分配的页框外，这里在order至少为k的块中起码还有$min/2^k$个空闲页框，

其中，对于每个 k，取值在 1 和分配的 order 之间。因此，如果 order 大于 0，那么在大小至少为 2 的块中起码还有 min/2 个空闲页框；如果 order 大于 1，那么在大小至少为 4 的块中起码还有 min/4 个空闲页框；依此类推。

阈值 min 的值由 zone_watermark_ok() 确定，具体如下：

- 作为函数参数被传递的基本值可以是内存管理区界值 pages_min、pages_low 和 pages_high 中的任意一个（参见本章前面的“保留的页框池”一节）。
- 如果作为参数传递的 gfp_high 标志被置位，那么 base 值被 2 除。通常，如果 gfp_mask 中的 __GFP_WAIT 标志被置位（也就是说，如果能从高端内存中分配页框），则这个标志等于 1。
- 如果作为参数传递的 can_try_harder 标志被置位，则阈值将会再减少四分之一。如果 gfp_mask 中的 __GFP_WAIT 标志被置位，或者如果当前进程是一个实时进程并且在进程上下文中（在中断处理程序和可延迟函数之外）已经完成了内存分配，则 can_try_harder 标志等于 1。

__alloc_pages() 函数本质上执行如下步骤：

1. 执行对内存管理区的第一次扫描（参见前面列出的代码）。在第一次扫描中，阈值 min 被设为 z->pages_low，其中的 z 指向正在被分析的管理区描述符（参数 can_try_harder 和 gfp_high 被设为 0）。
2. 如果函数在上一步没有终止，那么没有剩下多少空闲内存：函数唤醒 *kswapd* 内核线程来异步地开始回收页框（参见第十七章）。
3. 执行对内存管理区的第二次扫描，将值 z->pages_min 作为阈值 base 传递。正如前面解释的，实际阈值由 the can_try_harder 和 gfp_high 标志决定。这一步与第 1 步相似，但该函数使用了较低的阈值。
4. 如果函数在上一步没有终止，那么系统内存肯定不足。如果产生内存分配请求的内核控制路径不是一个中断处理程序或一个可延迟函数，并且它试图回收页框（或者是 current 的 PF_MEMALLOC 标志被置位，或者是它的 PF_MEMDIE 标志被置位），那么函数随即执行对内存管理区的第三次扫描，试图分配页框并忽略内存不足的阈值，也就是说，不调用 zone_watermark_ok()。唯有在这种情况下才允许内核控制路径耗用为内存不足预留的页（由管理区描述符的 lowmem_reserve 字段指定）。其实，在这种情况下产生内存请求的内核控制路径最终将试图释放页框，因此只要有可能它就应当得到它所请求的。如果没有任何内存管理区包含足够的页框，函数就返回 NULL 来提示调用者发生了错误。
5. 在这里，正在调用的内核控制路径并没有试图回收内存。如果 gfp_mask 的

__GFP_WAIT标志没有被置位，函数就返回NULL来提示该内核控制路径内存分配失败：在这种情况下，如果不阻塞当前进程就没有办法满足请求。

6. 在这里当前进程能够被阻塞：调用cond_resched()检查是否有其它的进程需要CPU。

7. 设置current的PF_MEMALLOC标志来表示进程已经准备好执行内存回收。

8. 将一个指向reclaim_state数据结构的指针存入current->reclaim_state。这个数据结构只包含一个字段reclaimed_slab，被初始化为0（我们将在本章后面的“slab分配器与分区页框分配器的接口”一节看到如何使用这个字段）。

9. 调用try_to_free_pages()寻找一些页框来回收（参见第十七章的“内存紧缺回收”一节）。后一个函数可能阻塞当前进程。一旦函数返回，__alloc_pages()就重设current的PF_MEMALLOC标志并再次调用cond_resched()。

10. 如果上一步已经释放了一些页框，那么该函数还要执行一次与第3步相同的内存管理区扫描。如果内存分配请求不能被满足，那么函数决定是否应当继续扫描内存管理区：如果__GFP_NORETRY标志被清除，并且内存分配请求跨越了多达8个页框或__GFP_REPEAT和__GFP_NOFAIL标志其中之一被置位，那么函数就调用blk_congestion_wait()使进程休眠一会儿（参见第十四章），并且跳回到第6步。否则，函数返回NULL来提示调用者内存分配失败了。

11. 如果在第9步中没有释放任何页框，就意味着内核遇到很大的麻烦，因为空闲页框已经少到了危险的地步，并且不可能回收任何页框。也许到了该作出重要决定的时候了。如果允许内核控制路径执行依赖于文件系统的操作来杀死一个进程（gfp_mask中的__GFP_FS标志被置位）并且__GFP_NORETRY标志为0，那么执行如下子步骤：

    a. 使用等于z->pages_high的阈值再一次扫描内存管理区。

    b. 调用out_of_memory()通过杀死一个进程开始释放一些内存（参见第十七章的“内存不足删除程序”一节）。

    c. 跳回第1步。

    因为第11a步使用的界值远比前面扫描时使用的界值要高，所以这个步骤很容易失败。实际上，只有当另一个内核控制路径已经杀死一个进程来回收它的内存后，第11a步才会成功执行。因此，第11a步避免了两个无辜的进程（而不是一个）被杀死。

释放一组页框

管理区分配器同样负责释放页框；幸运的是，释放内存比分配它要简单得多。

在前面的“分区页框分配器”一节描述的用来释放页框的所有内核宏和函数都依赖于__free_pages()函数。它接收的参数为将要释放的第一个页框的页描述符的地址(page)和将要释放的一组连续页框的数量的对数（order)。该函数执行如下步骤：

1. 检查第一个页框是否真正属于动态内存（它的PG_reserved标志被清0)；如果不是，则终止。
2. 减少page->_count使用计数器的值；如果它仍然大于或等于0，则终止。
3. 如果order等于0，那么该函数调用free_hot_page()来释放页框给适当内存管理区的每CPU热高速缓存（参见前面的“每CPU页框高速缓存”一节)。
4. 如果order大于0，那么它将页框加入到本地链表中，并调用free_pages_bulk()函数把它们释放到适当内存管理区的伙伴系统中（参见前面的“每CPU页框高速缓存”一节中描述的free_hot_cold_page()的第3步)。

内存区管理

本节关注内存区（*memory area*)，也就是说，关注具有连续的物理地址和任意长度的内存单元序列。

伙伴系统算法采用页框作为基本内存区，这适合于对大块内存的请求，但我们如何处理对小内存区的请求呢，比如说几十或几百个字节？

显然，如果为了存放很少的字节而给它分配一个整页框，这显然是一种浪费。取而代之的正确方法就是引入一种新的数据结构来描述在同一页框中如何分配小内存区。但这样也引出了一个新的问题，即所谓的内碎片（*internal fragmentation*)。内碎片的产生主要是由于请求内存的大小与分配给它的大小不匹配而造成的。

一种典型的解决方法（早期Linux版本采用）就是提供按几何分布的内存区大小，换句话说，内存区大小取决于2的幂而不取决于所存放的数据大小。这样，不管请求内存的大小是多少，我们都可保证内碎片小于50%。为此，内核建立了13个按几何分布的空闲内存区链表，它们的大小从32到131072字节。伙伴系统的调用既为了获得存放新内存区所需的额外页框，也为了释放不再包含内存区的页框。用一个动态链表来记录每个页框所包含的空闲内存区。

slab分配器

在伙伴算法之上运行内存区分配算法没有显著的效率。一种更好的算法源自*slab*分配器模式，该模式最早用于Sun公司的Solaris 2.4操作系统中。新算法基于下列前提：

- 所存放数据的类型可以影响内存区的分配方式。例如，当给用户态进程分配一个页框时，内核调用 get_zeroed_page()函数用0填充这个页。

  slab分配器概念扩充了这种思想，并把内存区看作对象（*object*），这些对象由一组数据结构和几个叫做构造（*constructor*）或析构（*destructor*）的函数（或方法）组成。前者初始化内存区，而后者回收内存区。

  为了避免重复初始化对象，slab分配器并不丢弃已分配的对象，而是释放但把它们保存在内存中。当以后又要请求新的对象时，就可以从内存获取而不用重新初始化。

- 内核函数倾向于反复请求同一类型的内存区。例如，只要内核创建一个新进程，它就要为一些固定大小的表[如进程描述符、打开文件对象等等（参见第三章）]分配内存区。当进程结束时，包含这些表的内存区还可以被重新使用。因为进程的创建和撤消非常频繁，在没有slab分配器时，内核把时间浪费在反复分配和回收那些包含同一内存区的页框上；slab分配器把那些页框保存在高速缓存中并很快地重新使用它们。

- 对内存区的请求可以根据它们发生的频率来分类。对于预期频繁请求一个特定大小的内存区而言，可以通过创建一组具有适当大小的专用对象来高效地处理，由此以避免内碎片的产生。另一种情况，对于很少遇到的内存区大小，可以通过基于一系列几何分布大小（如早期Linux版本所使用的2的幂次方大小）的对象的分配模式来处理，即使这种方法会导致内碎片的产生。

- 在引入的对象大小不是几何分布的情况下，也就是说，数据结构的起始地址不是物理地址值的2的幂次方，事情反倒好办。这可以借助处理器硬件高速缓存而导致较好的性能。

- 硬件高速缓存的高性能又是尽可能地限制对伙伴系统分配器调用的另一个理由，因为对伙伴系统函数的每次调用都“弄脏”硬件高速缓存，所以增加了对内存的平均访问时间。内核函数对硬件高速缓存的影响就是所谓的函数“足迹（*footprint*）”，其定义为函数结束时重写高速缓存的百分比。显而易见，大的“足迹”导致内核函数刚执行之后较慢的代码执行，因为硬件高速缓存此时填满了无用的信息。

slab分配器把对象分组放进高速缓存。每个高速缓存都是同种类型对象的一种“储备”。例如，当一个文件被打开时，存放相应“打开文件”对象所需的内存区是从一个叫做*filp*（“文件指针”）的slab分配器的高速缓存中得到的。

包含高速缓存的主内存区被划分为多个*slab*，每个slab由一个或多个连续的页框组成，这些页框中既包含已分配的对象，也包含空闲的对象（如图8-3所示）。

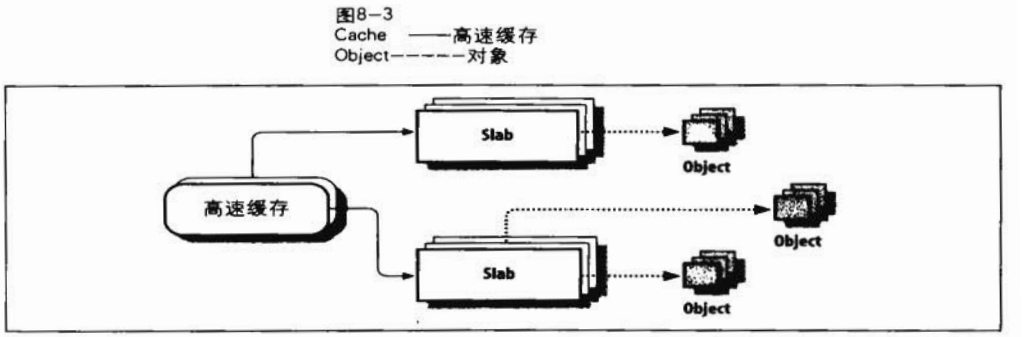


图 8-3：slab 分配器的组成

我们将在第十七章看到，内核周期性地扫描高速缓存并释放空 slab 对应的页框。

高速缓存描述符

每个高速缓存都是由 kmem_cache_t（等价于 struct kmem_cache_s 类型）类型的数据结构来描述的，表 8-8 列出了它的字段。我们在表中省略了用于收集统计数信息和调试的几个字段。

表 8-8：kmem_cache_t 描述符的字段

类型	名称	说明
struct array_cache * []	array	每 CPU 指针数组指向包含空闲对象的本地高速缓存（参见本章后面的闲“空闲 slab 对象的本地高速缓存”一节）
unsigned int	batchcount	要转移进本地高速缓存或从本地高速缓存中转移出的大批对象的数量
unsigned int	limit	本地高速缓存中空闲对象的最大数目。这是可调的
struct kmem_list3	lists	参见下一个表
unsigned int	objsize	高速缓存中包含的对象的大小
unsigned int	flags	描述高速缓存永久属性的一组标志
unsigned int	num	封装在一个单独 slab 中的对象个数（高速缓存中的所有 slab 具有相同的大小）
unsigned int	free_limit	整个 slab 高速缓存中空闲对象的上限
spinlock_t	spinlock	高速缓存自旋锁
unsigned int	gfporder	一个单独 slab 中包含的连续页框数目的对数

表8-8：kmem_cache_t描述符的字段（续）

类型	名称	说明
unsigned int	gfpflags	分配页框时传递给伙伴系统函数的一组标志
size_t	colour	slab使用的颜色个数（参见本章后面的“slab着色”一节）
unsigned int	colour_off	slab中的基本对齐偏移
unsigned int	colour_next	下一个被分配的slab使用的颜色
kmem_cache_t *	slabp_cache	指针指向包含slab描述符的普通slab高速缓存（如果使用了内部slab描述符，则这个字段为NULL，参见下一节）
unsigned int	slab_size	单个slab的大小
unsigned int	dflags	描述高速缓存动态属性的一组标志
void *	ctor	指向与高速缓存相关的构造方法的指针
void *	dtor	指向与高速缓存相关的析构方法的指针
const char *	name	存放高速缓存名字的字符数组
struct list_head	next	高速缓存描述符双向链表使用的指针

kmem_cache_t描述符的lists字段又是一个结构体，它的字段在表8-9中列出。

表8-9：kmem_list3结构的字段

类型	名称	说明
struct list_head	slabs_partial	包含空闲和非空闲对象的slab描述符双向循环链表
struct list_head	slabs_full	不包含空闲对象的slab描述符双向循环链表
struct list_head	slabs_free	只包含空闲对象的slab描述符双向循环链表
unsigned long	free_objects	高速缓存中空闲对象的个数
int	free_touched	由slab分配器的页回收算法使用（参见第十七章）
unsigned long	next_reap	由slab分配器的页回收算法使用（参见第十七章）
struct array_cache *	shared	指向所有CPU共享的一个本地高速缓存的指针（参见后面的“空闲slab对象的本地高速缓存”一节）

slab 描述符

高速缓存中的每个 slab 都有自己的类型为 slab 的描述符，如表 8-10 所示。

表 8-10：slab 描述符的字段

类型	名称	说明
struct list_head	list	slab 描述符的三个双向循环链表中的一个（在高速缓存描述符的kmem_list3结构中的 slabs_full、slabs_partial 或 slabs_free 链表）使用的指针
unsigned long	colouroff	slab 中第一个对象的偏移（参见本章后面的“slab 着色”一节）
void *	s_mem	slab 中第一个对象（或者已被分配，或者空闲）的地址
unsigned int	inuse	当前正在使用的（非空闲）slab 中的对象个数
unsigned int	free	slab 中下一个空闲对象的下标，如果没有剩下空闲对象则为 BUFCTL_END（参见本章后面的“对象描述符”一节）

slab 描述符可以存放在两个可能的地方：

外部 slab 描述符

存放在 slab 外部，位于 cache_sizes 指向的一个不适合 ISA DMA 的普通高速缓存中（参见下一节）。

内部 slab 描述符

存放在 slab 内部，位于分配给 slab 的第一个页框的起始位置。

当对象小于 512MB 时，或者当内碎片在 slab 内部为 slab 描述符将在后面介绍（及对象描述符）留下足够的空间时，slab 分配器选择第二种方案。如果 slab 描述符存放在 slab 外部，那么高速缓存描述符的 flags 字段中的 CFLGS_OFF_SLAB 标志被置 1；否则它被置 0。

图 8-4 显示了高速缓存描述符和slab描述符之间的主要关系。全满的 slab、部分满的 slab 及空闲的 slab 链接在不同的链表中。

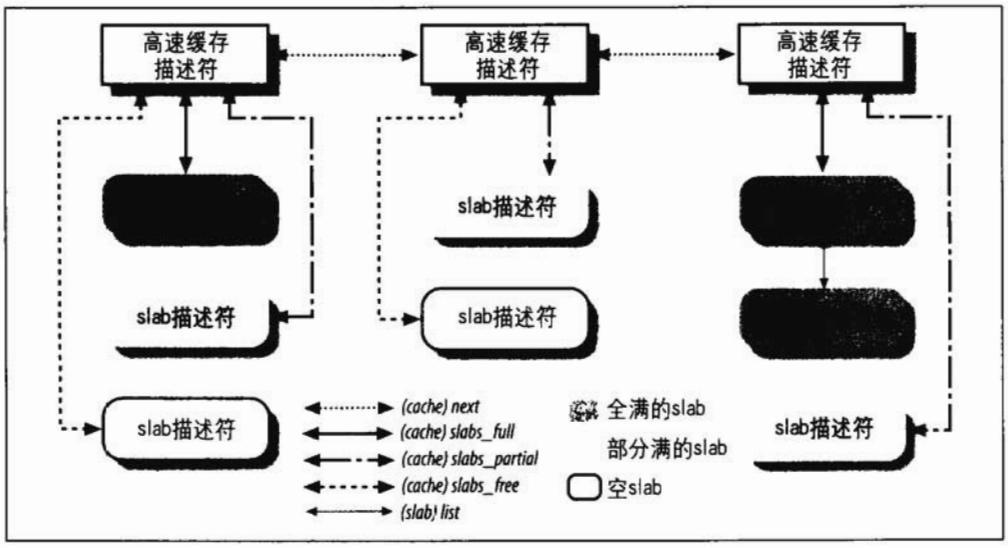


图 8-4：高速缓存描述符与 slab 描述符之间的关系

普通和专用高速缓存

高速缓存被分为两种类型：普通和专用。普通高速缓存只由slab分配器用于自己的目的，而专用高速缓存由内核的其余部分使用。

普通高速缓存是：

- 第一个高速缓存叫做*kmem_cache*，包含由内核使用的其余高速缓存的高速缓存描述符。cache_cache 变量包含第一个高速缓存的描述符。
- 另外一些高速缓存包含用作普通用途的内存区。内存区大小的范围一般包括 13 个几何分布的内存区。一个叫做 malloc_sizes 的表（其元素类型为 cache_sizes）分别指向 26 个高速缓存描述符，与其相关的内存区大小为 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 和 131072 字节。对于每种大小，都有两个高速缓存：一个适用于 ISA DMA 分配，另一个适用于常规分配。

在系统初始化期间调用 kmem_cache_init() 和 kmem_cache_sizes_init() 来建立普通高速缓存。

专用高速缓存是由 kmem_cache_create() 函数创建的。这个函数首先根据参数确定处理新高速缓存的最佳方法（例如，是在 slab 的内部还是外部包含 slab 描述符）。然后它从cache_cache普通高速缓存中为新的高速缓存分配一个高速缓存描述符，并把这个描

述符插入到高速缓存描述符的cache_chain链表中(当获得了用于保护链表避免被同时访问的cache_chain_sem信号量后，插入操作完成)。

还可以调用kmem_cache_destroy()撤销一个高速缓存并将它从cache_chain链表上删除。这个函数主要用于模块中，即模块装入时创建自己的高速缓存，卸载时撤销高速缓存。为了避免浪费内存空间，内核必须在撤销高速缓存本身之前就撤销其所有的slab。kmem_cache_shrink()函数通过反复调用slab_destroy()撤销高速缓存中所有的slab(参见后面“从高速缓存中释放slab”一节)。

所有普通和专用高速缓存的名字都可以在运行期间通过读取*/proc/slabinfo*文件得到。这个文件也指明每个高速缓存中空闲对象的个数和已分配对象的个数。

slab分配器与分区页框分配器的接口

当slab分配器创建新的slab时，它依靠分区页框分配器来获得一组连续的空闲页框。为了达到此目的，它调用kmem_getpages()函数，在UMA系统上该函数本质上等价于如下代码片段：

```
void * kmem_getpages(kmem_cache_t *cachep, int flags)
{
   struct page *page;
   int i;

   flags |= cachep->gfpflags;
   page = alloc_pages(flags, cachep->gfporder);
   if (!page)
      return NULL;
   i = (1 << cache->gfporder);
   if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
      atomic_add(i, &slab_reclaim_pages);
    while (i--)
            SetPageSlab(page++);
    return page_address(page);
}
```

两个参数的含义如下：

cachep

指向需要额外页框的高速缓存的高速缓存描述符（请求页框的个数由存放在cachep->gfporder字段中的order决定)。

flags

说明如何请求页框（参见本章前面“分区页框分配器”一节)。这组标志与存放在高速缓存描述符的gfpflags字段中的专用高速缓存分配标志相结合。

内存分配请求的大小由高速缓存描述符的gfporder字段指定，该字段将高速缓存中slab的大小编码（注7）。如果已经创建了slab高速缓存并且SLAB_RECLAIM_ACCOUNT标志已经置位，那么当内核检查是否有足够的内存来满足一些用户态请求时，分配给slab的页框将被记录为可回收的页。函数还将所分配页框的页描述符中的PG_slab标志置位。

在相反的操作中，通过调用kmem_freepages()函数可以释放分配给slab的页框（参见本章后面“从高速缓存中释放slab”一节）：

```
void kmem_freepages(kmem_cache_t *cachep, void *addr)
{
   unsigned long i = (1<<cachep->gfporder);
   struct page *page = virt_to_page(addr);

   if (current->reclaim_state)
       current->reclaim_state->reclaimed_slab += i;
   while (i--)
       ClearPageSlab(page++);
   free_pages((unsigned long) addr, cachep->gfporder);
   if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
      atomic_sub(1<<cachep->gfporder, &slab_reclaim_pages);
}
```

这个函数从线性地址addr开始释放页框，这些页框曾分配给由cachep标识的高速缓存中的slab。如果当前进程正在执行内存回收（current->reclaim_state字段非NULL），reclaim_state结构的reclaimed_slab字段就被适当地增加，于是刚被释放的页就能通过页框回收算法（参见第十七章的“内存紧缺回收”一节）被记录下来。此外，如果SLAB_RECLAIM_ACCOUNT标志置位（参见上面），slab_reclaim_pages变量则被适当地减少。

给高速缓存分配slab

一个新创建的高速缓存没有包含任何slab，因此也没有空闲的对象。只有当以下两个条件都为真时，才给高速缓存分配slab：

- 已发出一个分配新对象的请求。
- 高速缓存不包含任何空闲对象。

注7：注意不可能从ZONE_HIGHMEM内存管理区分配页框，因为kmem_getpages()函数返回由page_address()函数产生的线性地址；正如在本章前面的“高端内存页框的内核映射”一节解释的那样，该函数为未映射的高端内存页框返回NULL。

当这些情况发生时，slab分配器通过调用cache_grow()函数给高速缓存分配一个新的slab。而这个函数调用kmem_getpages()从分区页框分配器获得一组页框来存放一个单独的slab，然后又调用alloc_slabmgmt()获得一个新的slab描述符。如果高速缓存描述符的CFLGS_OFF_SLAB标志置位，则从高速缓存描述符的slabp_cache字段指向的普通高速缓存中分配这个新的slab描述符；否则，从slab的第一个页框中分配这个slab描述符。

给定一个页框，内核必须确定它是否被slab分配器使用，如果是，就迅速得到相应高速缓存和slab描述符的地址。因此，cache_ grow()扫描分配给新slab的页框的所有页描述符，并将高速缓存描述符和slab描述符的地址分别赋给页描述符中lru字段的next和prev子字段。这项工作不会出错，因为只有当页框空闲时伙伴系统的函数才会使用lru字段，而只要涉及伙伴系统，slab分配器函数所处理的页框就不空闲并将PG_slab标志置位（注8）。相反的问题——在高速缓存中给定一个slab，用哪些页框来实现它？这个问题可以通过使用slab描述符的s_mem字段和高速缓存描述符的gfporder字段（slab的大小）来回答。

接着，cache_grow()调用cache_init_objs()，它将构造方法（如果定义了的话）应用到新slab包含的所有对象上。

最后，cache_ grow()调用list_add_tail()来将新得到的slab描述符*slabp，添加到高速缓存描述符*cachep的全空slab链表的末端，并更新高速缓存中的空闲对象计数器：

```
list_add_tail(&slabp->list, &cachep->lists->slabs_free);
cachep->lists->free_objects += cachep->num;
```

从高速缓存中释放slab

在两种条件下才能撤销slab：

- slab高速缓存中有太多的空闲对象。
- 被周期性调用的定时器函数确定是否有完全未使用的slab能被释放（参见第十七章）。

在两种情况下，调用slab_destroy()函数撤销一个slab，并释放相应的页框到分区页框分配器：

```
void slab_destroy(kmem_cache_t *cachep, slab_t *slabp)
{
   if (cachep->dtor) {
```

注8：我们将在第十七章看到，lru字段还被页框回收算法使用。

```
            int i;
        for (i = 0; i < cachep->num; i++) {
            void* objp = slabp->s_mem+cachep->objsize*i;
            (cachep->dtor)(objp, cachep, 0);
        }
    }
    kmem_freepages(cachep, slabp->s_mem - slabp->colouroff);
    if (cachep->flags & CFLGS_OFF_SLAB)
        kmem_cache_free(cachep->slabp_cache, slabp);
    }
```

这个函数检查高速缓存是否为它的对象提供了析构方法（dtor字段不为NULL），如果是，就使用析构方法释放slab中的所有对象。objp局部变量记录当前已检查的对象。接下来，又调用kmem_freepages()，该函数把slab使用的所有连续页框返回给伙伴系统。最后，如果slab描述符存放在slab的外面，那么，就从slab描述符的高速缓存释放这个slab描述符。

实际上，该函数稍微更复杂一些。例如，可以使用SLAB_DESTROY_BY_RCU标志来创建slab高速缓存，这就意味着应使用call_rcu()函数注册一个回调来延期释放slab[参见第五章的“读—拷贝—更新(RCU)”一节]。正如前面描述的主要情形，回调函数接着调用kmem_freepages()，也可能调用kmem_cache_free()。

对象描述符

每个对象都有类型为kmem_bufctl_t的一个描述符。对象描述符存放在一个数组中，位于相应的slab描述符之后。因此，与slab描述符本身类似，slab的对象描述符也可以用两种可能的方式来存放，如图8-5所示。

外部对象描述符

存放在slab的外面，位于高速缓存描述符的slabp_cache字段指向的一个普通高速缓存中。内存区的大小（尤其是存放对象描述符的普通高速缓存的大小）取决于在slab中所存放的对象个数（高速缓存描述符的num字段）。

内部对象描述符

存放在slab内部，正好位于描述符所描述的对象之前。

数组中的第一个对象描述符描述slab中的第一个对象，依次类推。对象描述符只不过是一个无符号整数，只有在对象空闲时才有意义。它包含的是下一个空闲对象在slab中的下标，因此实现了slab内部空闲对象的一个简单链表。空闲对象链表中的最后一个元素的对象描述符用常规值BUFCTL_END (0xffff)标记。

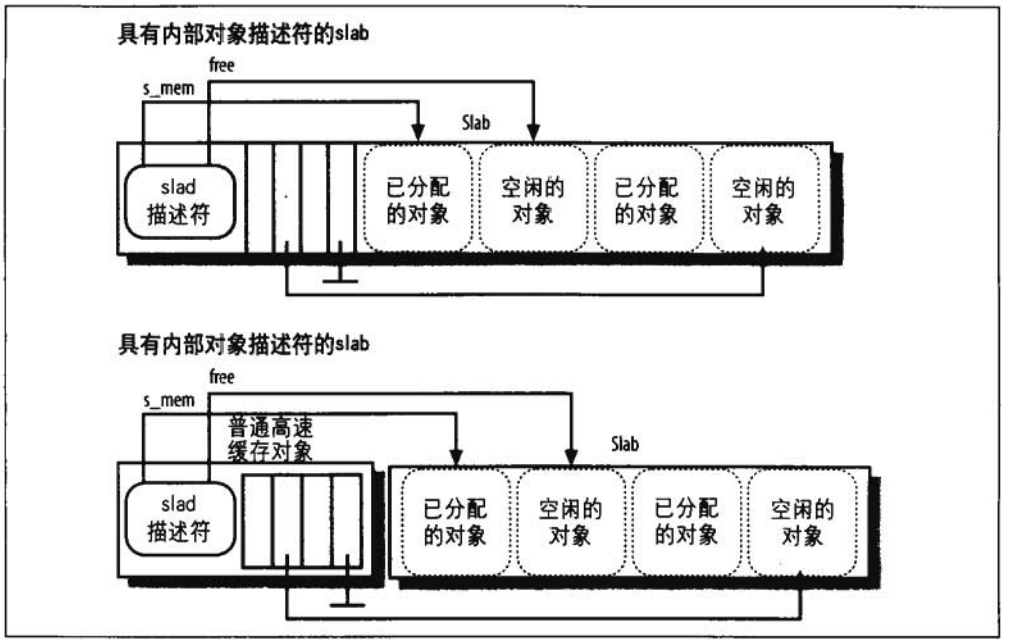


图 8-5：slab 描述符与对象描述符之间的关系

对齐内存中的对象

slab分配器所管理的对象可以在内存中进行对齐，也就是说，存放它们的内存单元的起始物理地址是一个给定常量的倍数，通常是2的倍数。这个常量就叫对齐因子(*alignment factor*)。

slab分配器所允许的最大对齐因子是4096，即页框大小。这就意味着通过访问对象的物理地址或线性地址就可以对齐对象。在这两种情况下，只有最低的12位才可以通过对齐来改变。

通常情况下，如果内存单元的物理地址是字大小（即计算机的内部内存总线的宽度）对齐的，那么，微机对内存单元的存取会非常快。因此，缺省情况下，kmem_cache_create()函数根据BYTES_PER_WORD宏所指定的字大小来对齐对象。对于80x86处理器，这个宏产生的值为4，因为字长是32位。

当创建一个新的slab高速缓存时，就可以让它所包含的对象在第一级硬件高速缓存中对齐。为了做到这点，设置SLAB_HWCACHE_ALIGN高速缓存描述符标志。kmem_cache_create()函数按如下方式处理请求：

- 如果对象的大小大于高速缓存行(cache line)的一半，就在RAM中根据L1_CACHE_BYTES的倍数（也就是行的开始）对齐对象。
- 否则，对象的大小就是L1_CACHE_BYTES的因子取整。这可以保证一个小对象不会横跨两个高速缓存行。

显然，slab分配器在这里所做的事情就是以内存空间换取访问时间，即通过人为地增加对象的大小来获得较好的高速缓存性能，由此也引起额外的内碎片。

slab着色

从第二章我们知道，同一硬件高速缓存行可以映射RAM中很多不同的块。在本章我们也已看到，相同大小的对象倾向于存放在高速缓存内相同的偏移量处。在不同的slab内具有相同偏移量的对象最终很可能映射在同一高速缓存行中。高速缓存的硬件可能因此而花费内存周期在同一高速缓存行与RAM内存单元之间来来往往传送两个对象，而其他的高速缓存行并未充分使用。slab分配器通过一种叫做slab着色（*slab coloring*）的策略，尽量降低高速缓存的这种不愉快行为：把叫做颜色（*color*）的不同随机数分配给slab。

在讨论slab着色之前，我们先看一下高速缓存内对象的布局。让我们考虑某个高速缓存，它的对象在RAM中被对齐。这就意味着对象的地址肯定是某个给定正数值（比如说*aln*）的倍数。连对齐的约束也考虑在内，在slab内放置对象就有很多种可能的方式。方式的选择取决于对下列变量所做的决定：

num

可以在slab中存放的对象个数（其值在高速缓存描述符的num字段中）。

osize

对象的大小，包括对齐的字节。

dsize

slap描述符的大小加上所有对象描述符的大小，就等于硬件高速缓存行大小的最小倍数。如果slab描述符和对象描述符都存放在slap的外部，那么这个值等于0。

free

在slab内未用字节（没有分配给任一对象的字节）的个数。

一个slab中的总字节长度可以表示为如下表达式：

$$slab\ 的长度 = (num \times osize) + dsize + free$$

*free*总是小于*osize*，因为否则的话，就有可能把另外的对象放在slab内。不过，*free*可以大于*aln*。

slab分配器利用空闲未用的字节*free*来对slab着色。术语“着色”只是用来再细分slab，并允许内存分配器把对象展开在不同的线性地址之中。这样的话，内核从微处理器的硬件高速缓存中可能获得最好性能。

具有不同颜色的slab把slab的第一个对象存放在不同的内存单元，同时满足对齐约束。可用颜色的个数是*free/aln*（这个值存放在高速缓存描述符的colour字段）。因此，第一个颜色表示为0，最后一个颜色表示为(*free/aln*)－1。（一种特殊情况是，如果*free*比*aln*小，那么*colour*被设为0，不过所有slab都使用颜色0，因此颜色真正的个数为1。）

如果用颜色*col*对一个slab着色，那么，第一个对象的偏移量（相对于slab的起始地址）就等于*col* × *aln*+*dsize*字节。图8-6显示了slab内对象的布局对slab颜色的依赖情况。着色本质上导致把slab中的一些空闲区域从末尾移到开始。

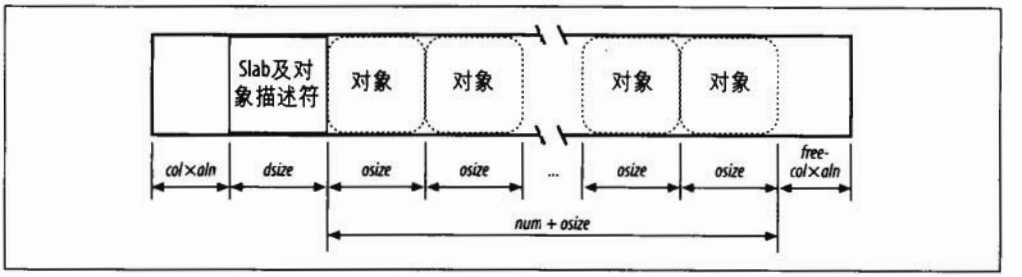


图8-6：具有颜色col与对齐aln的slab

只有当*free*足够大时，着色才起作用。显然，如果对象没有请求对齐，或者如果slab内的未用字节数小于所请求的对齐（*free* ≤ *aln*），那么，唯一可能着色的slab就是具有颜色0的slab，也就是说，把这个slab的第一个对象的偏移量赋为0。

通过把当前颜色存放在高速缓存描述符的colour_next字段，就可以在一个给定对象类型的slab之间平等地发布各种颜色。cache_grow()函数把colour_next所表示的颜色赋给一个新的slab，并递增这个字段的值。当colour_next的值变为colour后，又从0开始。这样，每个新创建的slab都与前一个slab具有不同的颜色，直到最大可用颜色。此外，cache_grow()函数从高速缓存描述符的colour_off字段获得值aln，根据slab内对象的个数计算dsize，最后把col×aln+dsize的值存放到slab描述符的colouroff字段中。

空闲Slab对象的本地高速缓存

Linux 2.6对多处理器系统上slab分配器的实现不同于Solaris 2.4上最初的实现。为了减少处理器之间对自旋锁的竞争并更好地利用硬件高速缓存，slab分配器的每个高速缓

存包含一个被称作slab本地高速缓存的每CPU数据结构，该结构由一个指向被释放对象的小指针数组组成。slab对象的大多数分配和释放只影响本地数组，只有在本地数组下溢或上溢时才涉及slab数据结构。该技术非常类似于本章前面的“每CPU页框高速缓存”一节中的技术。

高速缓存描述符的array字段是一组指向array_cache数据结构的指针，系统中的每个CPU对应于一个元素。每个array_cache数据结构是空闲对象的本地高速缓存的一个描述符，它的字段显示在表8-11中。

表8-11：array_cache结构的字段

类型	名称	说明
unsigned int	avail	指向本地高速缓存中可使用对象的指针的个数。它同时也作为高速缓存中第一个空槽的下标
unsigned int	limit	本地高速缓存的大小，也就是本地高速缓存中指针的最大个数
unsigned int	batchcount	本地高速缓存重新填充或腾空时使用的块大小
unsigned int	touched	如果本地高速缓存最近已经被使用过，则该标志设为1

注意，本地高速缓存描述符并不包含本地高速缓存本身的地址；事实上，它正好位于描述符之后。当然，本地高速缓存存放的是指向已释放对象的指针，而不是对象本身，对象本身总是位于高速缓存的slab中。

当创建一个新的slab高速缓存时，kmem_cache_create()函数决定本地高速缓存的大小（将这个值存放在高速缓存描述符的limit字段中）、分配本地高速缓存，并将它们的指针存放在高速缓存描述符的array字段。这个大小取决于存放在slab高速缓存中对象的大小，范围从1（相对于非常大的对象）到120（相对于小对象）。此外，batchcount字段的初始值，也就是从一个本地高速缓存的块里添加或删除的对象的个数，被初始化为本地高速缓存大小的一半（注9）。

在多处理器系统中，小对象使用的slab高速缓存同样包含一个附加的本地高速缓存，它的地址被存放在高速缓存描述符的lists.shared字段中。共享的本地高速缓存正如它的名字暗示的那样，被所有CPU共享，它使得将空闲对象从一个本地高速缓存移动到另一个高速缓存的任务更容易（参见下一节）。它的初始大小等于batchcount字段的值的8倍。

注9：系统管理员通过写入*/proc/slabinfo*文件可以为每个高速缓存调整本地高速缓存的大小以及batchcount字段的值。

分配 slab 对象

通过调用 kmem_cache_alloc()函数可以获得新对象。参数 cachep 指向高速缓存描述符，新空闲对象必须从该高速缓存描述符获得，而参数 flag表示传递给分区页框分配器函数的标志，该高速缓存的所有 slab 应当是满的。

该函数本质上等价于下列代码：

```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
{
    unsigned long save_flags;
    void *objp;
    struct array_cache *ac;

    local_irq_save(save_flags);
    ac = cache_p->array[smp_processor_id()];
    if (ac->avail) {
        ac->touched = 1;
        objp = ((void **)(ac+1))[--ac->avail];
    } else
        objp = cache_alloc_refill(cachep, flags);
    local_irq_restore(save_flags);
    return objp;
}
```

函数首先试图从本地高速缓存获得一个空闲对象。如果有空闲对象，avail字段就包含指向最后被释放的对象的项在本地高速缓存中的下标。因为本地高速缓存数组正好存放在 ac 描述符后面，所以((void**)(ac+1))[--ac->avail]获得那个空闲对象的地址并递减 ac->avail 的值。当本地高速缓存中没有空闲对象时，调用 cache_alloc_refill()函数重新填充本地高速缓存并获得一个空闲对象。

cache_alloc_refill()函数本质上执行如下步骤：

1. 将本地高速缓存描述符的地址存放在 ac 局部变量中：

   ```
   ac = cachep->array[smp_processor_id()];
   ```

2. 获得 cachep->spinlock。

3. 如果slab高速缓存包含共享本地高速缓存，并且该共享本地高速缓存包含一些空闲对象，函数就通过从共享本地高速缓存中上移ac->batchcount个指针来重新填充 CPU 的本地高速缓存。然后，函数跳到第 6 步。

4. 函数试图填充本地高速缓存，填充值为高速缓存的 slab 中包含的多达 ac->batchcount 个空闲对象的指针：

   a. 查看高速缓存描述符的slabs_partial和slabs_free链表，并获得slab描述

符的地址slabp，该slab描述符的相应slab或者部分被填充，或者为空。如果不存在这样的描述符，则函数转到第5步。

b. 对于slab中的每个空闲对象，函数增加slab描述符的inuse字段，将对象的地址插入本地高速缓存，并更新free字段使得它存放了slab中下一个空闲对象的下标：

```
slabp->inuse++;
((vcid**)(ac+1))[ac->avail++] =
                slabp->s_mem + slabp->free * cachep->obj_size;
slabp->free = ((kmem_bufctl_t*)(slabp+1))[slabp->free];
```

c. 如果必要，将清空的slab插入到适当的链表上，可以是slab_full链表，也可以是slab_partial链表。

5. 在这一步，被加到本地高速缓存上的指针个数被存放在ac->avail字段：函数递减同样数量的kmem_list3结构的free_objects字段来说明这些对象不再空闲。

6. 释放cachep->spinlock。

7. 如果现在ac->avail字段大于0（一些高速缓存再填充的情况发生了），函数将ac->touched字段设为1，并返回最后插入到本地高速缓存的空闲对象指针：

```
return ((void**)(ac+1))[--ac->avail];
```

8. 否则，没有发生任何高速缓存再填充情况：调用cache_grow()获得一个新slab，从而获得了新的空闲对象。

9. 如果cache_grow()失败了，则函数返回NULL；否则它返回到第1步重复该过程。

释放Slab对象

kmem_cache_free()函数释放一个曾经由slab分配器分配给某个内核函数的对象。它的参数为cachep和objp，前者是高速缓存描述符的地址，而后者是将被释放对象的地址：

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
{
   unsigned long flags;
   struct array_cache *ac;

   local_irq_save(flags);
   ac = cachep->array[smp_processor_id()];
   if (ac->avail == ac->limit)
      cache_flusharray(cachep, ac);
   ((void**)(ac+1))[ac->avail++] = objp;
   local_irq_restore(flags);
}
```

函数首先检查本地高速缓存是否有空间给指向一个空闲对象的额外指针。如果有，该指针就被加到本地高速缓存然后函数返回。否则，它首先调用cache_flusharray()来清空本地高速缓存，然后将指针加到本地高速缓存。

cache_flusharray()函数执行如下操作：

1. 获得cachep->spinlock自旋锁。
2. 如果slab高速缓存包含一个共享本地高速缓存，并且如果该共享本地高速缓存还没有满，函数就通过从CPU的本地高速缓存中上移ac->batchcount个指针来重新填充共享本地高速缓存。
3. 调用free_block()函数将当前包含在本地高速缓存中的ac->batchcount个对象归还给slab分配器。对于在地址objp处的每个对象，函数执行如下步骤：
   a. 增加高速缓存描述符的lists.free_objects字段。
   b. 确定包含对象的slab描述符的地址：

      ```
      slabp = (struct slab *)(virt_to_page(objp)->lru.prev);
      ```

      （请记住，slab页的描述符的lru.prev字段指向相应的slab描述符。）
   c. 从它的slab高速缓存链表（cachep->lists.slabs_partial或是cachep->lists.slabs_full）上删除slab描述符。
   d. 计算slab内对象的下标：

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

   e. 将slabp->free的当前值存放在对象描述符中，并将对象的下标放入slabp->free（最后被释放的对象将再次成为首先被分配的对象）：

      ```
      ((kmem_bufctl_t *)(slabp+1))[objnr] = slabp->free;
      slabp->free = objnr;
      ```

   f. 递减slabp->inuse字段。
   g. 如果slabp->inuse等于0（也就是slab中所有对象空闲），并且整个slab高速缓存中空闲对象的个数(cachep->lists.free_objects)大于cachep->free_limit字段中存放的限制，那么函数将slab的页框释放到分区页框分配器：

      ```
      cachep->lists.free_objects -= cachep->num;
      slab_destroy(cachep, slabp);
      ```

      存放在cachep->free_limit字段中的值通常等于cachep->num+(1+N)×cachep->batchcount，其中N代表系统中CPU的个数。
   h. 否则，如果slab->inuse等于0，但整个slab高速缓存中空闲对象的个数小于

cachep->free_limit，函数就将slab描述符插入到cachep->lists.slabs_free链表中。

i. 最后，如果slab->inuse大于0，slab被部分填充，则函数将slab描述符插入到cachep->lists.slabs_partial链表中。

4. 释放cachep->spinlock自旋锁。

5. 通过减去被移到共享本地高速缓存或被释放到slab分配器的对象的个数来更新本地高速缓存描述符的avail字段。

6. 移动本地高速缓存数组起始处的那个本地高速缓存中的所有指针。这一步是必需的，因为已经把第一个对象指针从本地高速缓存上删除，因此剩下的指针必须上移。

通用对象

正如“伙伴系统算法”一节中所描述的那样，如果对存储区的请求不频繁，就用一组普通高速缓存来处理，普通高速缓存中的对象具有几何分布的大小，范围为32～131072字节。

调用kmalloc()函数就可以得到这种类型的对象，函数等价于下列代码片段：

```
void * kmalloc(size_t size, int flags)
{
    struct cache_sizes *csizep = malloc_sizes;
    kmem_cache_t * cachep;
    for (; csizep->cs_size; csizep++) {
        if (size > csizep->cs_size)
            continue;
        if (flags & __GFP_DMA)
            cachep = csizep->cs_dmacachep;
        else
            cachep = csizep->cs_cachep;
        return kmem_cache_alloc(cachep, flags);
    }
    return NULL;
}
```

该函数使用malloc_sizes表为所请求的大小分配最近的2的幂次方大小的内存。然后，调用kmem_cache_alloc()分配对象，传递的参数或者为适用于ISA DMA页框的高速缓存描述符，或者为适用于“常规”页框的高速缓存描述符，这取决于调用者是否指定了__GFP_DMA标志。

调用kmalloc()所获得的对象可以通过调用kfree()来释放：

```
void kfree(const void *objp)
{
    kmem_cache_t * c;
```

```
    unsigned long flags;
    if (!objp)
       return;
    local_irq_save(flags);
    c = (kmem_cache_t *)(virt_to_page(objp)->lru.next);
    kmem_cache_free(c, (void *)objp);
    local_irq_restore(flags);
}
```

通过读取内存区所在的第一个页框描述符的lru.next子字段，就可确定出合适的高速缓存描述符。通过调用kmem_cache_free()来释放相应的内存区。

内存池

内存池（*memory pool*）是Linux 2.6的一个新特性。基本上讲，一个内存池允许一个内核成分，如块设备子系统，仅在内存不足的紧急情况下分配一些动态内存来使用。

不应该将内存池与前面“保留的页框池”一节中描述的保留页框混淆。实际上这些页框只能用于满足中断处理程序或内部临界区发出的原子内存分配请求。而内存池是动态内存的储备，只能被特定的内核成分（即池的“拥有者”）使用。拥有者通常不使用储备；但是，如果动态内存变得极其稀有以至于所有普通内存分配请求都将失败的话，那么作为最后的解决手段，内核成分就能调用特定的内存池函数提取储备得到所需的内存。因此，创建一个内存池就像手头存放一些罐装食物作为储备，当没有新鲜食物时就使用开罐器。

一个内存池常常叠加在slab分配器之上——也就是说，它被用来保存slab对象的储备。但是一般而言，内存池能被用来分配任何一种类型的动态内存，从整个页框到使用kmalloc()分配的小内存区。因此，我们一般将内存池处理的内存单元看作“内存元素”。

内存池由mempool_t对象描述，它的字段如表8-12所示。

表8-12：mempool_t对象的字段

类型	名称	说明
spinlock_t	lock	用来保护对象字段的自旋锁
int	min_nr	内存池中元素的最大个数
int	curr_nr	当前内存池中元素的个数
void **	elements	指向一个数组的指针，该数组由指向保留元素的指针组成
void *	pool_data	池的拥有者可获得的私有数据
mempool_alloc_t *	alloc	分配一个元素的方法

表8-12：mempool_t对象的字段（续）

类型	名称	说明
mempool_free_t *	free	释放一个元素的方法
wait_queue_head_t	wait	当内存池为空时使用的等待队列

min_nr字段存放了内存池中元素的初始个数。换句话说，存放在该字段中的值代表了内存元素的个数，内存池的拥有者确信能从内存分配器得到这个数目。curr_nr字段总是低于或等于min_nr，它存放了内存池中当前包含的内存元素个数。内存元素自身被一个指针数组引用，指针数组的地址存放在elements字段中。

alloc和free方法与基本的内存分配器进行交互，分别用于获得和释放一个内存元素。两个方法可以是拥有内存池的内核成分提供的定制函数。

当内存元素是slab对象时，alloc和free方法一般由mempool_alloc_slab()和mempool_free_slab()函数实现，它们只是分别调用kmem_cache_alloc()和kmem_cache_free()函数。在这种情况下，mempool_t对象的pool_data字段存放了slab高速缓存描述符的地址。

mempool_create()函数创建一个新的内存池，它接收的参数为内存元素的个数min_nr、实现alloc和free方法的函数的地址和赋给pool_data字段的任意值。该函数分别为mempool_t对象和指向内存元素的指针数组分配内存，然后反复调用alloc方法来得到min_nr个内存元素。相反地，mempool_destroy()函数释放池中所有内存元素，然后释放元素数组和mempool_t对象自己。

为了从内存池分配一个元素，内核调用mempool_alloc()函数，将mempool_t对象的地址和内存分配标志传递给它（参见本章前面的表8-5和表8-6）。函数本质上依据参数所指定的内存分配标志，试图通过调用alloc方法从基本内存分配器分配一个内存元素。如果分配成功，函数返回获得的内存元素而不触及内存池。否则，如果分配失败，就从内存池获得内存元素。当然，在内存不足的情况下过多的分配会用尽内存池：在这种情况下，如果__GFP_WAIT标志没有置位，则mempool_alloc()阻塞当前进程直到有一个内存元素被释放到内存池中。

相反地，为了释放一个元素到内存池，内核调用mempool_free()函数。如果内存池未满（curr_min小于min_nr），则函数将元素加到内存池中。否则，mempool_free()调用free方法来释放元素到基本内存分配器。

非连续内存区管理

从前面的讨论中我们已经知道，把内存区映射到一组连续的页框是最好的选择，这样会充分利用高速缓存并获得较低的平均访问时间。不过，如果对内存区的请求不是很频繁，那么，通过连续的线性地址来访问非连续的页框这样一种分配模式就会很有意义。这种模式的主要优点是避免了外碎片，而缺点是必须打乱内核页表。显然，非连续内存区的大小必须是4096的倍数。Linux在几个方面使用非连续内存区，例如，为活动的交换区分配数据结构（参见第十七章中的“激活和禁用交换区”一节），为模块分配空间（参见附录二），或者给某些I/O驱动程序分配缓冲区。此外，非连续内存区还提供了另一种使用高端内存页框的方法（参见后面的“分配非连续内存区”一节）。

非连续内存区的线性地址

要查找线性地址的一个空闲区，我们可以从PAGE_OFFSET开始查找（通常为0xc0000000，即第4个GB的起始地址）。图8-7显示了如何使用第4个GB的线性地址：

- 内存区的开始部分包含的是对前896MB RAM进行映射的线性地址（参见第二章“进程页表”一节）；直接映射的物理内存末尾所对应的线性地址保存在high_memory变量中。
- 内存区的结尾部分包含的是固定映射的线性地址（参见第二章“固定映射的线性地址”一节）。
- 从PKMAP_BASE开始，我们查找用于高端内存页框的永久内核映射的线性地址（参见本章前面“高端内存页框的内核映射”一节）。
- 其余的线性地址可以用于非连续内存区。在物理内存映射的末尾与第一个内存区之间插入一个大小为8MB（宏VMALLOC_OFFSET）的安全区，目的是为了“捕获”对内存的越界访问。出于同样的理由，插入其他4KB大小的安全区来隔离非连续的内存区。

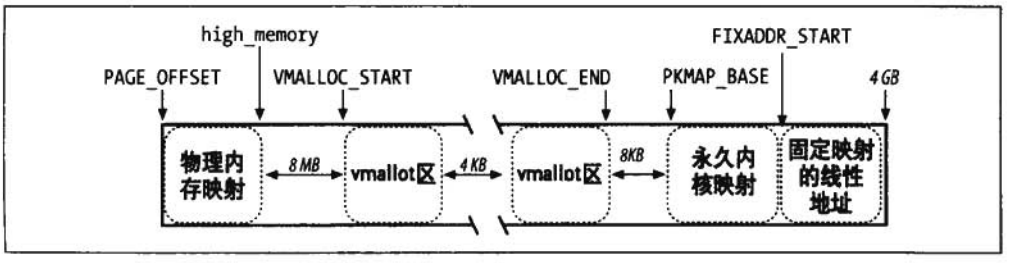


图8-7：从PAGE_OFFSET开始的线性地址区间

为非连续内存区保留的线性地址空间的起始地址由VMALLOC_START宏定义，而末尾地址由 VMALLOC_END 宏定义。

非连续内存区的描述符

每个非连续内存区都对应着一个类型为vm_struct 的描述符，表8-13列出了它的字段。

表 8-13：vm_struct 描述符的字段

类型	名称	说明
void *	addr	内存区内第一个内存单元的线性地址
unsigned long	size	内存区的大小加4096（内存区之间的安全区间的大小）
unsigned long	flags	非连续内存区映射的内存的类型
struct page **	pages	指向 nr_pages 数组的指针，该数组由指向页描述符的指针组成
unsigned int	nr_pages	内存区填充的页的个数
unsigned long	phys_addr	该字段设为 0，除非内存已被创建来映射一个硬件设备的 I/O 共享内存
struct vm_struct *	next	指向下一个 vm_struct 结构的指针

通过 next 字段，这些描述符被插入到一个简单的链表中，链表第一个元素的地址存放在vmlist变量中。对这个链表的访问依靠 vmlist_lock 读/写自旋锁来保护。flags字段标识了非连续区映射的内存的类型：VM_ALLOC 表示使用 vmalloc()得到的页，VM_MAP 表示使用 vmap()映射的已经被分配的页（参见下一节），而 VM_IOREMAP 表示使用 ioremap()映射的硬件设备的板上内存（参见第十三章）。

get_vm_area()函数在线性地址 VMALLOC_START 和 VMALLOC_END 之间查找一个空闲区域。该函数使用两个参数：将被创建的内存区的字节大小（size）和指定空闲区类型（参见上面）的标志（flag）。步骤执行如下：

1. 调用 kmalloc()为 vm_struct 类型的新描述符获得一个内存区。
2. 为写得到 vmlist_lock 锁，并扫描类型为 vm_struct 的描述符链表来查找线性地址一个空闲区域，至少覆盖 size + 4096 个地址（4096 是内存区之间的安全区间大小）。
3. 如果存在这样一个区间，函数就初始化描述符的字段，释放 vmlist_lock 锁，并以返回这个非连续内存区的起始地址而结束。

4. 否则，get_vm_area()释放先前得到的描述符，释放vmlist_lock，然后返回NULL。

分配非连续内存区

vmalloc()函数给内核分配一个非连续内存区。参数size表示所请求内存区的大小。如果这个函数能够满足请求，就返回新内存区的起始地址；否则，返回一个NULL指针：

```
void * vmalloc(unsigned long size)
{
   struct vm_struct *area;
   struct page **pages;
   unsigned int array_size, i;
   size = (size + PAGE_SIZE - 1) & PAGE_MASK;
   area = get_vm_area(size, VM_ALLOC);
   if (!area)
       return NULL;
   area->nr_pages = size >> PAGE_SHIFT;
   array_size = (area->nr_pages * sizeof(struct page *));
   area->pages = pages = kmalloc(array_size, GFP_KERNEL);
   if (!area_pages) {
       remove_vm_area(area->addr);
       kfree(area);
       return NULL;
 }
 memset(area->pages, 0, array_size);
 for (i=0; i<area->nr_pages; i++) {
       area->pages[i] = alloc_page(GFP_KERNEL|_  _GFP_HIGHMEM);
       if (!area->pages[i]) {
          area->nr_pages = i;
 fail:    vfree(area->addr);
          return NULL;
      }
 }
 if (map_vm_area(area, _  _pgprot(0x63), &pages))
     goto fail;
 return area->addr;
}
```

函数首先将参数size设为4096（页框大小）的整数倍。然后，vmalloc()调用get_vm_area()来创建一个新的描述符，并返回分配给这个内存区的线性地址。描述符的flags字段被初始化为VM_ALLOC标志，该标志意味着通过使用vmalloc()函数，非连续页框将被映射到一个线性地址区间。然后vmalloc()函数调用kmalloc()来请求一组连续页框，这组连续页框足够包含一个页描述符指针数组。调用memset()函数来将所有这些指针设为NULL。接着重复调用alloc_page()函数，每一次为区间中nr_pages个页的每一个分配一个页框，并把对应页描述符的地址存放在area->pages数组中。注意，必须使用area->pages数组是因为页框可能属于ZONE_HIGHMEM内存管理区，所以此时它们不必被映射到一个线性地址上。

现在到了棘手的部分。直到这里，已经得到了一个新的连续线性地址区间，并且已经分配了一组非连续页框来映射这些线性地址。最后至关重要的步骤是修改内核使用的页表项，以此表明分配给非连续内存区的每个页框现在对应着一个线性地址，这个线性地址被包含在vmalloc()产生的非连续线性地址区间中。这就是map_vm_area()所要做的。

map_vm_area()函数使用以下3个参数：

area
: 指向内存区的vm_struct描述符的指针。

prot
: 已分配页框的保护位。它总是被置为0x63，对应着Present、Accessed、Read/Write及Dirty。

pages
: 指向一个指针数组的变量的地址，该指针数组的指针指向页描述符(因此，struct page ***被当作数据类型使用！)。

函数首先把内存区的开始和末尾的线性地址分别分配给局部变量address和end：

```
address = area->addr;
end = address + (area->size - PAGE_SIZE);
```

请记住，area->size存放的是内存区的实际地址加上4KB内存之间的安全区间。然后函数使用pgd_offset_k宏来得到在主内核页全局目录中的目录项，该项对应于内存区起始线性地址，然后获得内核页表自旋锁：

```
pgd = pgd_offset_k(address);
spin_lock(&init_mm.page_table_lock);
```

然后，函数执行下列循环：

```
int ret = 0;
for (i = pgd_index(address); i < pgd_index(end-1); i++) {
    pud_t *pud = pud_alloc(&init_mm, pgd, address);
    ret = -ENOMEM;
    if (!pud)
        break;
    next = (address + PGDIR_SIZE) & PGDIR_MASK;
    if (next < address || next > end)
        next = end;
    if (map_area_pud(pud, address, next, prot, pages))
        break;
    address = next;
    pgd++;
    ret = 0;
}
```

```
    spin_unlock(&init_mm.page_table_lock);
    flush_cache_vmap((unsigned long)area->addr, end);
    return ret;
```

每次循环都首先调用pud_alloc()来为新内存区创建一个页上级目录，并把它的物理地址写入内核页全局目录的合适表项。然后调用alloc_area_pud()为新的页上级目录分配所有相关的页表。接下来，把常量2^{30}（在PAE被激活的情况下，否则为2^{22}）与address的当前值相加（2^{30}就是一个页上级目录所跨越的线性地址范围的大小），最后增加指向页全局目录的指针pgd。

循环结束的条件是：指向非连续内存区的所有页表项全被建立。

map_area_pud()函数为页上级目录所指向的所有页表执行一个类似的循环：

```
    do {
        pmd_t * pmd = pmd_alloc(&init_mm, pud, address);
        if (!pmd)
            return -ENOMEM;
        if (map_area_pmd(pmd, address, end-address, prot, pages))
            return -ENOMEM;
        address = (address + PUD_SIZE) & PUD_MASK;
        pud++;
    } while (address < end);
```

map_area_pmd()函数为页中间目录所指向的所有页表执行一个类似的循环：

```
    do {
        pte_t * pte = pte_alloc_kernel(&init_mm, pmd, address);
        if (!pte)
            return -ENOMEM;
        if (map_area_pte(pte, address, end-address, prot, pages))
            return -ENOMEM;
        address = (address + PMD_SIZE) & PMD_MASK;
        pmd++;
    } while (address < end);
```

pte_alloc_kernel()函数（参见第二章的“页表处理”一节）分配一个新的页表，并更新页中间目录中相应的目录项。接下来，map_area_pte()为页表中相应的表项分配所有的页框。address值增加2^{22}（2^{22}就是一个页表所跨越的线性地址区间的大小），并且循环反复执行。

map_area_pte()的主循环为：

```
    do {
        struct page * page = **pages;
        set_pte(pte, mk_pte(page, prot));
        address += PAGE_SIZE;
        pte++;
```

```
        (*pages)++;
    } while (address < end);
```

将被映射的页框的页描述符地址page是从地址pages处的变量指向的数组项读得的。通过set_pte和mk_pte宏，把新页框的物理地址写进页表。把常量4096（即一个页框的长度）加到address上之后，循环又重复执行。

注意，map_vm_area()并不触及当前进程的页表。因此，当内核态的进程访问非连续内存区时，缺页发生，因为该内存区所对应的进程页表中的表项为空。然而，缺页处理程序要检查这个缺页线性地址是否在主内核页表中(也就是init_mm.pgd页全局目录和它的子页表，参见第二章“内核页表”一节)。一旦处理程序发现一个主内核页表含有这个线性地址的非空项，就把它的值拷贝到相应的进程页表项中，并恢复进程的正常执行。这种机制将在第九章“缺页异常处理程序”一节描述。

除了vmalloc()函数之外，非连续内存区还能由vmalloc_32()函数分配，该函数与vmalloc()很相似，但是它只从ZONE_NORMAL和ZONE_DMA内存管理区中分配页框。

Linux 2.6还特别提供了了一个vmap()函数，它将映射非连续内存区中已经分配的页框：本质上，该函数接收一组指向页描述符的指针作为参数，调用get_vm_area()得到一个新vm_struct描述符，然后调用map_vm_area()来映射页框。因此该函数与vmalloc()相似，但是它不分配页框。

释放非连续内存区

vfree()函数释放vmalloc()或vmalloc_32()创建的非连续内存区，而vunmap()函数释放vmap()创建的内存区。两个函数都使用同一个参数——将要释放的内存区的起始线性地址address；它们都依赖于__vunmap()函数来做实质性的工作。

__vunmap()函数接收两个参数：将要释放的内存区的起始地址的地址addr，以及标志deallocate_pages，如果被映射到内存区内的页框应当被释放到分区页框分配器（调用vfree()）中，那么这个标志被置位，否则被清除（vunmap()被调用）。该函数执行以下操作：

1. 调用remove_vm_area()函数得到vm_struct描述符的地址area，并清除非连续内存区中的线性地址对应的内核的页表项。
2. 如果deallocate_pages被置位，函数扫描指向页描述符的area->pages指针数组；对于数组的每一个元素，调用__free_page()函数释放页框到分区页框分配器。此外，执行kfree(area->pages)来释放数组本身。

3. 调用kfree(area)来释放vm_struct描述符。

remove_vm_area()函数执行如下循环：

```
write_lock(&vmlist_lock);
for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
    if (tmp->addr == addr) {
        unmap_vm_area(tmp);
        *p = tmp->next;
        break;
    }
}
write_unlock(&vmlist_lock);
return tmp;
```

内存区本身通过调用unmap_vm_area()来释放。这个函数接收单个参数，即指向内存区的vm_struct描述符的指针area。它执行下列循环以进行map_vm_area()的反向操作：

```
address = area->addr;
end = address + area->size;
pgd = pgd_offset_k(address);
for (i = pgd_index(address); i <= pgd_index(end-1); i++) {
    next = (address + PGDIR_SIZE) & PGDIR_MASK;
    if (next <= address || next > end)
        next = end;
    unmap_area_pud(pgd, address, next - address);
    address = next;
    pgd++;
}
```

unmap_area_pud()依次在循环中执行map_area_pud()的反操作：

```
do {
    unmap_area_pmd(pud, address, end-address);
    address = (address + PUD_SIZE) & PUD_MASK;
    pud++;
} while (address && (address < end));
```

unmap_area_pmd()函数在循环中执行map_area_pmd()的反操作：

```
do {
    unmap_area_pte(pmd, address, end-address);
    address = (address + PMD_SIZE) & PMD_MASK;
    pmd++;
} while (address < end);
```

最后，unmap_area_pte()在循环中执行map_area_pte()的反操作：

```
do {
    pte_t page = ptep_get_and_clear(pte);
    address += PAGE_SIZE;
```

```
        pte++;
        if (!pte_none(page) && !pte_present(page))
            printk("Whee... Swapped out page in kernel page table\n");
    } while (address < end);
```

在每次循环过程中，ptep_get_and_clear 宏将 pte 指向的页表项设为0。

与vmalloc()一样，内核修改主内核页全局目录和它的子页表中的相应项（参见第二章“内核页表”一节），但是映射第4个GB的进程页表的项保持不变。这是在情理之中的，因为内核永远也不会收回扎根于主内核页全局目录中的页上级目录、页中间目录和页表。

例如，假定内核态的进程访问一个随后要释放的非连续内存区。进程的页全局目录项等于主内核页全局目录中的相应项，由于在第九章“缺页异常处理程序”一节中所描述的机制，这些目录项指向相同的页上级目录、页中间目录和页表。unmap_area_pte()函数只清除页表中的项（不回收页表本身）。进程对已释放非连续内存区的进一步访问必将由于空的页表项而触发缺页异常。但是，缺页处理程序会认为这样的访问是一个错误，因为主内核页表不包含有效的表项。



第九章

进程地址空间

在前一章我们已经看到，内核中的函数以相当直接了当的方式获得动态内存，这是通过调用以下几种函数中的一个达到的：__get_free_pages()或alloc_pages ()从分区页框分配器中获得页框，kmem_cache_alloc()或kmalloc()使用slab分配器为专用或通用对象分配块，而vmalloc()或vmalloc_32()获得一块非连续的内存区。如果所请求的内存区得以满足，这些函数都返回一个页描述符地址或线性地址（即所分配动态内存区的起始地址）。

使用这些简单方法是基于以下两个原因：

- 内核是操作系统中优先级最高的成分。如果某个内核函数请求动态内存，那么，必定有正当的理由发出那个请求，因此，没有道理试图推迟这个请求。
- 内核信任自己。所有的内核函数都被假定是没有错误的，因此内核函数不必插入针对编程错误的任何保护措施。

当给用户态进程分配内存时，情况完全不同：

- 进程对动态内存的请求被认为是不紧迫的。例如，当进程的可执行文件被装入时，进程并不一定立即对所有的代码页进行访问。类似地，当进程调用malloc()以获得请求的动态内存时，也并不意味着进程很快就会访问所有所获得的内存。因此，一般来说，内核总是尽量推迟给用户态进程分配动态内存。
- 由于用户进程是不可信任的，因此，内核必须能随时准备捕获用户态进程引起的所有寻址错误。

本章我们会看到，内核使用一种新的资源成功实现了对进程动态内存的推迟分配。当用户态进程请求动态内存时，并没有获得请求的页框，而仅仅获得对一个新的线性地址区间的使用权，而这一线性地址区间就成为进程地址空间的一部分。这一区间叫做“线性区”（memory region）（译注1）。

在下一节，我们讨论进程是怎样看待动态内存的。然后，在“线性区”一节中描述进程地址空间的基本组成。接下来，我们仔细分析缺页异常处理程序在推迟给进程分配页框中所起的作用。然后，我们阐述内核怎样创建和删除进程的整个地址空间。最后，我们讨论与进程的地址空间管理有关的API和系统调用。

进程的地址空间

进程的地址空间（*address space*）由允许进程使用的全部线性地址组成。每个进程所看到的线性地址集合是不同的，一个进程所使用的地址与另外一个进程所使用的地址之间没有什么关系。后面我们会看到，内核可以通过增加或删除某些线性地址区间来动态地修改进程的地址空间。

内核通过所谓线性区的资源来表示线性地址区间，线性区是由起始线性地址、长度和一些访问权限来描述的。为了效率起见，起始地址和线性区的长度都必须是4096的倍数，以便每个线性区所识别的数据完全填满分配给它的页框。下面是进程获得新线性区的一些典型情况：

- 当用户在控制台输入一条命令时，shell进程创建一个新的进程去执行这个命令。结果是，一个全新的地址空间（也就是一组线性区）分配给了新进程（参见本章后面的“创建和删除进程的地址空间”一节和第二十章）。
- 正在运行的进程有可能决定装入一个完全不同的程序。在这种情况下，进程标识符仍然保持不变，可是在装入这个程序以前所使用的线性区却被释放，并有一组新的线性区被分配给这个进程（参见第二十章中的“exec函数”一节）。
- 正在运行的进程可能对一个文件（或它的一部分）执行“内存映射”。在这种情况下，内核给这个进程分配一个新的线性区来映射这个文件（参见第十六章中的“内存映射”一节）。
- 进程可能持续向它的用户态堆栈增加数据，直到映射这个堆栈的线性区用完为止。

译注1：“memory region”字面含义为内存区，但实际含义为线性地址空间中的一个区字段，在此把它译为线性区更贴切一些。其实际含义就是通常所指的虚拟内存中的一个区间，可以称为“虚存区”（Virtual Memory Area，VMA）。

在这种情况下，内核也许会决定扩展这个线性区的大小（参见本章后面的“缺页异常处理程序”一节）。

- 进程可能创建一个IPC共享线性区来与其他合作进程共享数据。在这种情况下，内核给这个进程分配一个新的线性区以实现这个方案（参见第十九章中的“IPC共享内存”一节）。
- 进程可能通过调用类似malloc()这样的函数扩展自己的动态区（堆）。结果是，内核可能决定扩展给这个堆所分配的线性区（参见本章后面的“堆的管理”一节）。

表9-1显示了与前面提到的任务相关的一些系统调用。除brk()在本章的最后进行讨论外，其余的系统调用在其他章节阐述。

表9-1：与创建、删除线性区相关的系统调用

系统调用	说明
brk()	改变进程堆的大小
execve()	装入一个新的可执行文件，从而改变进程的地址空间
_exit()	结束当前进程并撤销它的地址空间
fork()	创建一个新进程，并为它创建新的地址空间
mmap(), mmap2()	为文件创建一个内存映射，从而扩大进程的地址空间
mremap()	扩大或缩小线性区
remap_file_pages()	为文件创建非线性映射（参见第十六章）
munmap()	撤销对文件的内存映射，从而缩小进程的地址空间
shmat()	创建一个共享线性区
shmdt()	撤消一个共享线性区

我们会在“缺页异常处理程序”一节中看到，确定一个进程当前所拥有的线性区（即进程的地址空间）是内核的基本任务，因为这可以让缺页异常处理程序有效地区分引发这个异常处理程序的两种不同类型的无效线性地址：

- 由编程错误引发的无效线性地址。
- 由缺页引发的无效线性地址，即使这个线性地址属于进程的地址空间，但是对应于这个地址的页框仍然有待分配。

从进程的观点来看，后一种地址不是无效的，内核要利用这种缺页以实现请求调页：内核通过提供页框来处理这种缺页，并让进程继续执行。

内存描述符

与进程地址空间有关的全部信息都包含在一个叫做内存描述符（*memory descriptor*）的数据结构中（译注2），这个结构的类型为mm_struct，进程描述符的mm字段就指向这个结构。内存描述符的字段如表9-2所示：

表9-2：内存描述符中的字段

类型	字段	说明
struct vm_area_struct *	mmap	指向线性区对象的链表头
struct rb_root	mm_rb	指向线性区对象的红－黑树的根
struct vm_area_struct *	mmap_cache	指向最后一个引用的线性区对象
unsigned long (*)()	get_unmapped_area	在进程地址空间中搜索有效线性地址区间的方法
void (*)()	unmap_area	释放线性地址区间时调用的方法
unsigned long	mmap_base	标识第一个分配的匿名线性区或文件内存映射的线性地址（参见第二十章“程序段和进程的线性区”一节）
unsigned long	free_area_cache	内核从这个地址开始搜索进程地址空间中线性地址的空闲区间
pgd_t *	pgd	指向页全局目录
atomic_t	mm_users	次使用计数器
atomic_t	mm_count	主使用计数器
int	map_count	线性区的个数
struct rw_semaphore	mmap_sem	线性区的读/写信号量
spinlock_t	page_table_lock	线性区的自旋锁和页表的自旋锁
struct list_head	mmlist	指向内存描述符链表中的相邻元素
unsigned long	start_code	可执行代码的起始地址
unsigned long	end_code	可执行代码的最后地址
unsigned long	start_data	已初始化数据的起始地址
unsigned long	end_data	已初始化数据的最后地址
unsigned long	start_brk	堆的起始地址

译注2：实际上就是描述进程虚拟内存的数据结构。

表 9-2：内存描述符中的字段（续）

类型	字段	说明
unsigned long	brk	堆的当前最后地址
unsigned long	start_stack	用户态堆栈的起始地址
unsigned long	arg_start	命令行参数的起始地址
unsigned long	arg_end	命令行参数的最后地址
unsigned long	env_start	环境变量的起始地址
unsigned long	env_end	环境变量的最后地址
unsigned long	rss	分配给进程的页框数
unsigned long	anon_rss	分配给匿名内存映射的页框数
unsigned long	total_vm	进程地址空间的大小（页数）
unsigned long	locked_vm	“锁住”而不能换出的页的个数（参见第十七章）
unsigned long	shared_vm	共享文件内存映射中的页数
unsigned long	exec_vm	可执行内存映射中的页数
unsigned long	stack_vm	用户态堆栈中的页数
unsigned long	reserved_vm	在保留区中的页数或在特殊线性区中的页数
unsigned long	def_flags	线性区默认的访问标志
unsigned long	nr_ptes	this 进程的页表数
unsigned long []	saved_auxv	开始执行 ELF 程序时使用（参见第二十章）
unsigned int	dumpable	表示是否可以产生内存信息转储的标志
cpumask_t	cpu_vm_mask	用于懒惰 TLB 交换的位掩码（参见第二章）
mm_context_t	context	指向有关特定体系结构信息的表（例如，在 80x86 平台上的 LDT 地址）
unsigned long	swap_token_time	进程在这个时间将有资格获得交换标记（参见第十七章“交换标记”一节）
char	recent_pagein	如果最近发生了主缺页，则设置该标志
int	core_waiters	正在把进程地址空间的内容卸载到转储文件中的轻量级进程的数量（参见本章后面“删除进程的地址空间”一节）
struct completion *	core_startup_done	指向创建内存转储文件时的补充原语（参见第五章“补充原语”一节）

表 9-2：内存描述符中的字段（续）

类型	字段	说明
struct completion	core_done	创建内存转储文件时使用的补充原语
rwlock_t	ioctx_list_lock	用于保护异步 I/O 上下文链表的锁（参见第十六章）
struct kioctx *	ioctx_list	异步 I/O 上下文链表（参见第十六章）
struct kioctx	default_kioctx	默认的异步 I/O 上下文（参见第十六章）
unsigned long	hiwater_rss	进程所拥有的最大页框数
unsigned long	hiwater_vm	进程线性区中的最大页数

所有的内存描述符存放在一个双向链表中。每个描述符在 mmlist 字段存放链表相邻元素的地址。链表的第一个元素是 init_mm 的 mmlist 字段，init_mm 是初始化阶段进程 0 所使用的内存描述符。mmlist_lock 自旋锁保护多处理器系统对链表的同时访问。

mm_users 字段存放共享 mm_struct 数据结构的轻量级进程的个数（参见第三章“clone()、fork()及 vfork()系统调用”一节）。mm_count 字段是内存描述符的主使用计数器，在 mm_users 次使用计数器中的所有用户在 mm_count 中只作为一个单位。每当 mm_count 递减时，内核都要检查它是否变为 0，如果是，就要解除这个内存描述符，因为不再有用户使用它。

我们用一个例子来解释 mm_users 和 mm_count 之间的不同。考虑一个内存描述符由两个轻量级进程共享。它的 mm_users 字段通常存放的值为 2，而 mm_count 字段存放的值为 1（两个所有者进程算作一个）。

如果把内存描述符暂时借给一个内核线程（参见下一节），那么，内核就增加 mm_count。这样，即使两个轻量级进程都死亡，且 mm_users 字段变为 0，这个内存描述符也不被释放，直到内核线程使用完为止，因为 mm_count 字段仍然大于 0。

如果内核想确保内存描述符在一个长操作的中间不被释放，那么，就应该增加 mm_users 字段而不是 mm_count 字段的值（这正是函数 try_to_unuse()所做的事，参见第十七章“激活和禁用交换区”一节）。最终的结果是相同的，因为 mm_users 的增加确保了 mm_count 不变为 0，即使拥有这个内存描述符的所有轻量级进程全部死亡。

mm_alloc()函数用来获得一个新的内存描述符。由于这些描述符被保存在 slab 分配器高速缓存中，因此，mm_alloc()调用 kmem_cache_alloc()来初始化新的内存描述符，并把 mm_count 和 mm_users 字段都置为 1。

相反，mmput()函数递减内存描述符的mm_users字段。如果该字段变为0，这个函数就释放局部描述符表、线性区描述符（参见本章后面的部分）及由内存描述符所引用的页表，并调用mmdrop()。后一个函数把mm_count字段减1，如果该字段变为0，就释放mm_struct数据结构。

mmap、mm_rb、mmlist和mmap_cache字段将在下一节进行讨论。

内核线程的内存描述符

内核线程仅运行在内核态，因此，它们永远不会访问低于TASK_SIZE（等于PAGE_OFFSET，通常为 0xc0000000）的地址。与普通进程相反，内核线程不用线性区，因此，内存描述符的很多字段对内核线程是没有意义的。

因为大于TASK_SIZE线性地址的相应页表项都应该总是相同的，因此，一个内核线程到底使用什么样的页表集根本就没有什么关系。为了避免无用的TLB和高速缓存刷新，内核线程使用一组最近运行的普通进程的页表。结果，在每个进程描述符中包含了两种内存描述符指针：mm和active_mm。

进程描述符中的mm字段指向进程所拥有的内存描述符，而active_mm字段指向进程运行时所使用的内存描述符。对于普通进程而言，这两个字段存放相同的指针。但是，内核线程不拥有任何内存描述符，因此，它们的mm字段总是为NULL。当内核线程得以运行时，它的active_mm字段被初始化为前一个运行进程的active_mm值（参看第七章“schedule()函数”一节）。

然而，事情有点复杂。只要处于内核态的一个进程为“高端”线性地址（高于TASK_SIZE）修改了页表项，那么，它就也应当更新系统中所有进程页表集合中的相应表项。事实上，一旦内核态的一个进程进行了设置，那么，映射应该对内核态的其他所有进程都有效。触及所有进程的页表集合是相当费时的操作，因此，Linux采用一种延迟方式。

我们在第八章“非连续内存区管理”一节已经提到这种延迟方式：每当一个高端地址必须被重新映射时（一般是通过vmalloc()或vfree()），内核就更新根目录在swapper_pg_dir主内核页全局目录（参见第二章“内核页表”一节）中的常规页表集合。这个页全局目录由主内存描述符（*master memory descriptor*）的pgd字段所指向，而主内存描述符存放于init_mm变量（注1）。

注1：我们在第三章“内核线程”一节中提到，*swapper*内核线程在初始化阶段使用init_mm。但是，一旦初始化完成，swapper再不使用这个内存描述符。

在随后的“处理非连续内存区访问”一节，我们将描述缺页处理程序如何在非常必要时维护存放在常规页表中的扩展信息。

线性区

Linux通过类型为vm_area_struct的对象实现线性区，它的字段如表9-3所示（注2）。

表9-3：线性区对象的字段

类型	字段	说明
struct mm_struct *	vm_mm	指向线性区所在的内存描述符
unsigned long	vm_start	线性区内的第一个线性地址
unsigned long	vm_end	线性区之后的第一个线性地址
struct vm_area_struct *	vm_next	进程链表中的下一个线性区
pgprot_t	vm_page_prot	线性区中页框的访问许可权
unsigned long	vm_flags	线性区的标志
struct rb_node	vm_rb	用于红－黑树的数据（参见本章后面）
union	shared	链接到反映射所使用的数据结构（参见第十七章“对映射页的反映射”一节）
struct list_head	anon_vma_node	指向匿名线性区链表的指针（参见第十七章“映射页的反向映射”一节）
struct anon_vma *	anon_vma	指向anon_vma数据结构的指针（参见第十七章“映射页的反向映射”一节）
struct vm_operations_struct *	vm_ops	指向线性区的方法
unsigned long	vm_pgoff	在映射文件中的偏移量（参见第十六章）。对匿名页，它等于0或vm_start/PAGE_SIZE（参见第十七章）
struct file *	vm_file	指向映射文件的文件对象（如果有的话）
void *	vm_private_data	指向内存区的私有数据
unsigned long	vm_truncate_count	释放非线性文件内存映射中的一个线性地址区间时使用

注2： 我们对NUMA系统中使用的一些附加字段不予说明。

每个线性区描述符表示一个线性地址区间。vm_start字段包含区间的第一个线性地址，而vm_end字段包含区间之外的第一个线性地址。vm_end - vm_start表示线性区的长度。vm_mm字段指向拥有这个区间的进程的mm_struct内存描述符。我们稍后将描述vm_area_struct的其他字段。

进程所拥有的线性区从来不重叠，并且内核尽力把新分配的线性区与紧邻的现有线性区进行合并。如果两个相邻区的访问权限相匹配，就能把它们合并在一起。

如图9-1所示，当一个新的线性地址区间加入到进程的地址空间时，内核检查一个已经存在的线性区是否可以扩大（情况a）。如果不能，就创建一个新的线性区（情况b）。类似地，如果从进程的地址空间删除一个线性地址区间，内核就要调整受影响的线性区大小（情况c）。有些情况下，调整大小迫使一个线性区被分成两个更小的部分（情况d）（注3）。

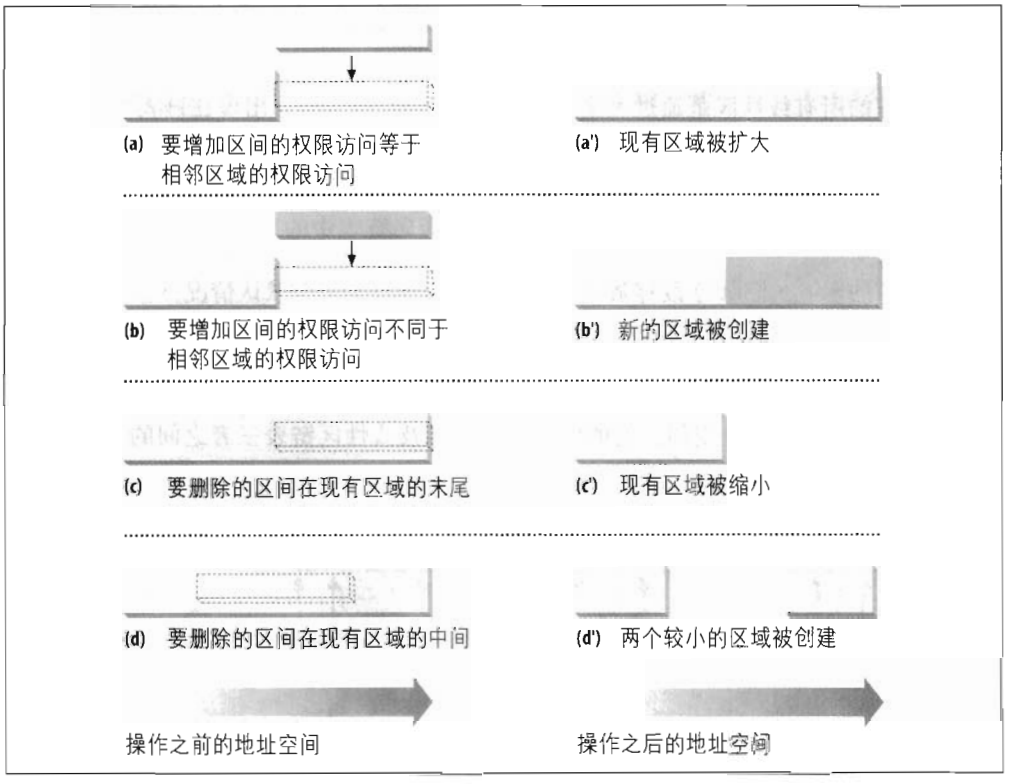


图9-1：增加或删除一个线性地址区间

注3： 从理论上说，删除一个线性地址区间可能会失败，因为没有空闲的内存给新的内存描述符使用。

vm_ops字段指向vm_operations_struct数据结构，该结构中存放的是线性区的方法。只有如表9-4所示的4种方法可应用于UMA系统。

表9-4：作用于线性区的方法

方法	说明
open	当把线性区增加到进程所拥有的线性区集合时调用
close	当从进程所拥有的线性区集合删除线性区时调用
nopage	当进程试图访问RAM中不存在的一个页，但该页的线性地址属于线性区时，由缺页异常处理程序调用（参见后面“缺页异常处理程序”一节）
populate	设置线性区的线性地址（预缺页）所对应的页表项时调用。主要用于非线性文件内存映射

线性区数据结构

进程所拥有的所有线性区是通过一个简单的链表链接在一起的。出现在链表中的线性区是按内存地址的升序排列的；不过，每两个线性区可以由未用的内存地址区隔开。每个vm_area_struct元素的vm_next字段指向链表的下一个元素。内核通过进程的内存描述符的mmap字段来查找线性区，其中mmap字段指向链表中的第一个线性区描述符。

内存描述符的map_count字段存放进程所拥有的线性区数目。默认情况下，一个进程可以最多拥有65536个不同的线性区，系统管理员可以通过写*/proc/sys/vm/max_map_count*文件来修改这个限定值。

图9-2显示了进程的地址空间、它的内存描述符以及线性区链表三者之间的关系。

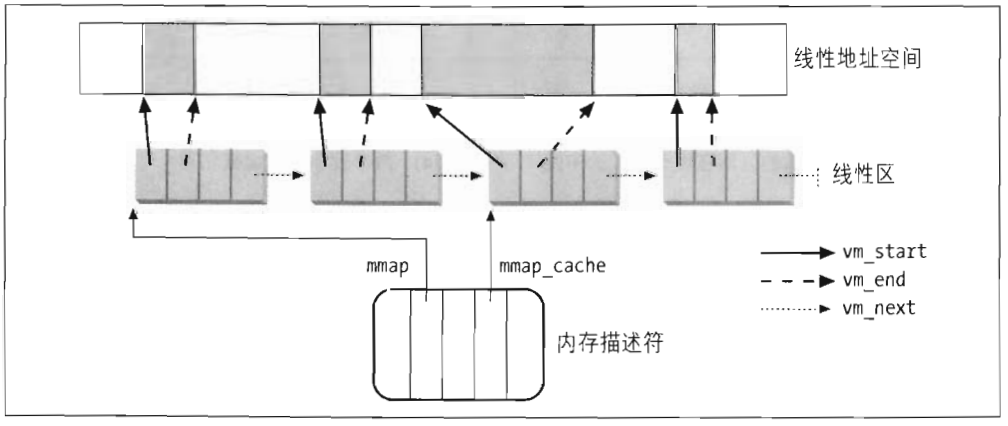


图9-2：与进程地址空间相关的描述符

内核频繁执行的一个操作就是查找包含指定线性地址的线性区。由于链表是经过排序的，因此，只要在指定线性地址之后找到一个线性区，搜索就可以结束。

然而，仅当进程的线性区非常少时使用这种链表才是很方便的，比如说只有一二十个线性区。在链表中查找元素、插入元素、删除元素涉及许多操作，这些操作所花费的时间与链表的长度成线性比例。

尽管多数的Linux进程使用的线性区非常少，但是诸如面向对象的数据库，或malloc()的专用调试器那样过于庞大的大型应用程序可能会有成百上千的线性区。在这种情况下，线性区链表的管理变得非常低效，因此，与内存相关的系统调用的性能就降低到令人无法忍受的程度。

因此，Linux 2.6把内存描述符存放在叫做红－黑树（red-black tree）的数据结构中。在红－黑树中，每个元素（或节点）通常有两个孩子：左孩子和右孩子。树中的元素被排序。对每个节点N，N的左子树上的所有元素都排在N之前，相反，N的右子树上的所有元素都排在N之后［如图9-3（a）所示］；节点的关键字被写入节点内部。此外，红－黑树必须满足下列4条规则：

1. 每个节点必须或为黑或为红。
2. 树的根必须为黑。
3. 红节点的孩子必须为黑。
4. 从一个节点到后代叶子节点的每个路径都包含相同数量的黑节点。当统计黑节点个数时，空指针也算作黑节点。

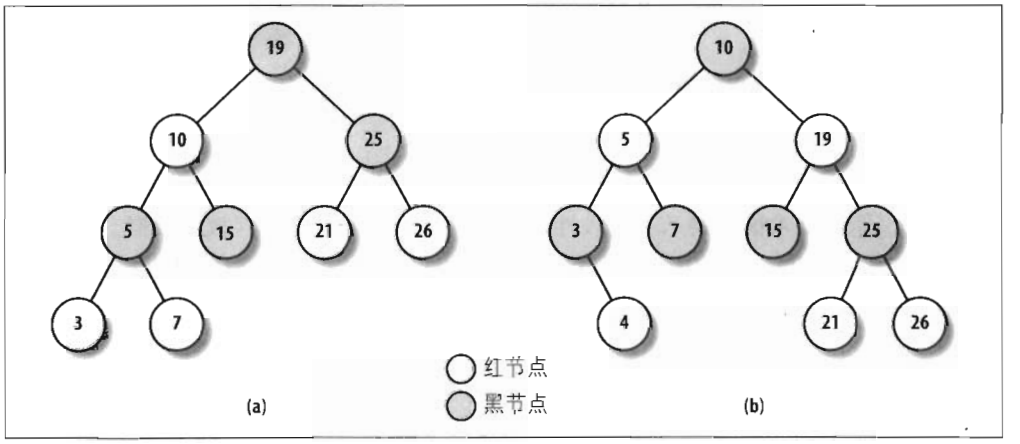


图9-3：红－黑树实例

这4条规则确保具有n个内部节点的任何红－黑树其高度最多为$2 \times \log (n+1)$。

在红－黑树中搜索一个元素因此而变得非常高效，因为其操作的执行时间与树大小的对数成线性比例。换句话说，双倍的线性区个数只多增加一次循环。

在红－黑树中插入和删除一个元素也是高效的，因为算法能很快地遍历树以确定插入元素的位置或删除元素的位置。任何新节点必须作为一个叶子插入并着成红色。如果操作违背了上述规则，就必须移动或重新着色树的几个节点。

例如，假如值为4的一个元素必须插入到图9-3（a）所示的红－黑树中。它的正确位置是关键值为3的节点的右孩子，但是，一旦把它插入，值为3的红节点就具有红孩子，因此而违背了规则3。为了满足这条规则，值为3、4、7的节点颜色就得改变。但是，这种操作又会违背规则4，因此，算法在以关键值为19的节点为根节点的子树上执行“旋转”操作，产生如图9-3（b）所示的新红－黑树。这看起来较复杂，但是，在红－黑树上插入或删除一个元素只需要少量的操作——这个数与树大小的对数成线性比例。

因此，为了存放进程的线性区，Linux既使用了链表，也使用了红－黑树。这两种数据结构包含指向同一线性区描述符的指针，当插入或删除一个线性区描述符时，内核通过红－黑树搜索前后元素，并用搜索结果快速更新链表而不用扫描链表。

链表的头由内存描述符的mmap字段所指向。任何线性区对象都在vm_next字段存放指向链表下一个元素的指针。红－黑树的首部由内存描述符的mm_rb字段所指向。任何线性区对象都在类型为rb_node的vm_rb字段中存放节点颜色以及指向双亲、左孩子和右孩子的指针。

一般来说，红－黑树用来确定含有指定地址的线性区，而链表通常在扫描整个线性区集合时来使用。

线性区访问权限

在讲述下一部分以前，我们先阐明页与线性区之间的关系。正如第二章中所提到的，我们使用“页”这个术语既表示一组线性地址又表示这组地址中所存放的数据。尤其是，我们把介于0～4095之间的线性地址区间称为第0页，介于4096～8191之间的线性地址区间称为第1页，依此类推。因此每个线性区都由一组号码连续的页所构成。

在前几章我们已经讨论了与页相关的两种标志：

- 在每个页表项中存放的几个标志，如：Read/Write、Present或User/Supervisor（参见第二章中的“常规分页”一节）。

- 存放在每个页描述符 flags 字段中的一组标志（参见第八章中的“页框管理”一节）。

第一种标志由80x86硬件用来检查能否执行所请求的寻址类型；第二种标志由Linux用于许多不同的目的（见表8-2）。

现在介绍第三种标志，即与线性区的页相关的那些标志。它们存放在 vm_area_struct 描述符的 vm_flags 字段中（见表9-5）。一些标志给内核提供有关这个线性区全部页的信息，例如它们包含有什么内容，进程访问每个页的权限是什么。另外的标志描述线性区自身，例如它应该如何增长。

表9-5：线性区标志

标志名	说明
VM_READ	页是可读的
VM_WRITE	页是可写的
VM_EXEC	页是可执行的
VM_SHARED	页可以由几个进程共享
VM_MAYREAD	可以设置 VM_READ 标志
VM_MAYWRITE	可以设置 VM_WRITE 标志
VM_MAYEXEC	可以设置 VM_EXEC 标志
VM_MAYSHARE	可以设置 VM_SHARE 标志
VM_GROWSDOWN	线性区可以向低地址扩展
VM_GROWSUP	线性区可以向高地址扩展
VM_SHM	线性区用于 IPC 的共享内存
VM_DENYWRITE	线性区映射一个不能打开用于写的文件
VM_EXECUTABLE	线性区映射一个可执行文件
VM_LOCKED	线性区中的页被锁住，且不能换出
VM_IO	线性区映射设备的 I/O 地址空间
VM_SEQ_READ	应用程序顺序地访问页
VM_RAND_READ	应用程序以真正的随机顺序访问页
VM_DONTCOPY	当创建一个新进程时不拷贝线性区
VM_DONTEXPAND	通过 mremap() 系统调用禁止线性区扩展
VM_RESERVED	线性区是特殊的（如：它映射某个设备的 I/O 地址空间），因此它的页不能被交换出去
VM_ACCOUNT	创建 IPC 共享线性区时检查是否有足够的空闲内存用于映射（参见第十九章）

表9-5：线性区标志（续）

标志名	说明
VM_HUGETLB	通过扩展分页机制处理线性区中的页（参见第二章“扩展分页”一节）
VM_NONLINEAR	线性区实现非线性文件映射

线性区描述符所包含的页访问权限可以任意组合。例如，存在这样一种可能性，允许一个线性区中的页可以执行但是不可以读取。为了有效地实现这种保护方案，与线性区的页相关的访问权限（读、写及执行）必须被复制到相应的所有表项中，以便由分页单元直接执行检查。换句话说，页访问权限表示何种类型的访问应该产生一个缺页异常。稍后我们会看到，Linux委派缺页处理程序查找导致缺页的原因，因为缺页处理程序实现了许多页处理策略。

页表标志的初值（正如我们看到的，同一线性区所有页标志的初值必须一样）存放在vm_area_struct描述符的vm_page_prot字段中。当增加一个页时，内核根据vm_page_prot字段的值设置相应页表项中的标志。

然而，并不能把线性区的访问权限直接转换成页保护位，这是因为：

- 在某些情况下，即使由相应线性区描述符的vm_flags字段所指定的某个页的访问权限允许对该页进行访问，但是，对该页的访问还是应当产生一个缺页异常。例如，我们在本章后面的“写时复制”一节会看到，内核可能决定把属于两个不同进程的两个完全一样的可写私有页（它的VM_SHARE标志被清0）存入同一个页框中；在这种情况下，无论哪一个进程试图改动这个页都应当产生一个异常。
- 正如在第二章中提到的，80x86处理器的页表仅有两个保护位，即Read/Write和User/Supervisor标志。此外，一个线性区所包含的任何一个页的User/Supervisor标志必须总置为1，因为用户态进程必须总能够访问其中的页。
- 启用PAE的新近Intel Pentium 4微处理器，在所有64位页表项中支持NX(No eXecute)标志。

如果内核没有被编译成支持PAE，那么Linux采取以下规则以克服80x86微处理器的硬件限制：

- 读访问权限总是隐含着执行访问权限，反之亦然。
- 写访问权限总是隐含着读访问权限。

反之，如果内核被编译成支持PAE，而且CPU有NX标志，Linux就采取不同的规则：

- 执行访问权限总是隐含着读访问权限。
- 写访问权限总是隐含着读访问权限。

此外，为了做到在“写时复制”技术中适当地推迟页框的分配（参见本章后面的内容），只要相应的页不是由多个进程所共享，那么，这种页框都是写保护的。

因此，要根据以下规则精简由读、写、执行和共享访问权限的16种可能组合：

- 如果页具有写和共享两种访问权限，那么，Read/Write位被设置为1。
- 如果页具有读或执行访问权限，但是既没有写也没有共享访问权限，那么，Read/Write位被清0。
- 如果支持NX位，而且页没有执行访问权限，那么，把NX位设置为1。
- 如果页没有任何访问权限，那么，Present位被清0，以便每次访问都产生一个缺页异常。然而，为了把这种情况与真正的页框不存在的情况相区分，Linux还把Page size位置为1（注4）。

访问权限的每种组合所对应的精简后的保护位存放在protection_map数组的16个元素中。

线性区的处理

对控制内存处理所用的数据结构和状态信息有了基本理解以后，我们来看一组对线性区描述符进行操作的低层函数。这些函数应当被看作简化了do_map()和do_unmap()实现的辅助函数。这两个函数将在本章后面的“分配线性地址区间”和“释放线性地址区间”两节中进行描述，它们分别扩大或者缩小进程的地址空间。这两个函数所处的层次比我们在这里所考虑函数的层次要高一些，它们并不接受线性区描述符作为参数，而是使用一个线性地址区间的起始地址、长度和访问限权作为参数。

查找给定地址的最邻近区：find_vma()

find_vma()函数有两个参数：进程内存描述符的地址mm和线性地址addr。它查找线性区的vm_end字段大于addr的第一个线性区的位置，并返回这个线性区描述符的地址；如果没有这样的线性区存在，就返回一个NULL指针。注意由find_vma()函数所选择的线性区并不一定要包含addr，因为addr可能位于任何线性区之外。

注4：你可能认为Page size位的这种用法并不正当，因为这个位本来是表示实际页的大小。但是，Linux可以侥幸逃脱这种骗局，因为80 × 86芯片在页目录项中检查Page size位，而不是在页表的表项中检查该位。

每个内存描述符包含一个mmap_cache字段，这个字段保存进程最后一次引用线性区的描述符地址。引进这个附加的字段是为了减少查找一个给定线性地址所在线性区而花费的时间。程序中引用地址的局部性使下面这种情况出现的可能性很大：如果检查的最后一个线性地址属于某一给定的线性区，那么，下一个要检查的线性地址也属于这一个线性区。

因此，该函数一开始就检查由mmap_cache所指定的线性区是否包含addr。如果是，就返回这个线性区描述符的指针：

```
vma = mm->mmap_cache;
if (vma && vma->vm_end > addr && vma->vm_start <= addr)
    return vma;
```

否则，必须扫描进程的线性区，并在红－黑树中查找线性区：

```
rb_node = mm->mm_rb.rb_node;
vma = NULL;
while (rb_node) {
    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;
```

函数使用的宏rb_entry,从指向红－黑树中一个节点的指针导出相应线性区描述符的地址。

函数find_vma_prev()与find_vma()类似，不同的是它把函数选中的前一个线性区描述符的指针赋给附加参数ppre。

最后，函数find_vma_prepare()确定新叶子节点在与给定线性地址对应的红－黑树中的位置，并返回前一个线性区的地址和要插入的叶子节点的父节点的地址。

查找一个与给定的地址区间相重叠的线性区：find_vma_intersection()

find_vma_intersection()函数查找与给定的线性地址区间相重叠的第一个线性区。mm参数指向进程的内存描述符，而线性地址start_addr和end_addr指定这个区间。

```
vma = find_vma(mm,start_addr);
if (vma && end_addr <= vma->vm_start)
    vma = NULL;
return vma;
```

如果没有这样的线性区存在，函数就返回一个NULL指针。准确地说，如果find_vma()函数返回一个有效的地址，但是所找到的线性区是从这个线性地址区间的末尾开始的，vma就被置为NULL。

查找一个空闲的地址区间：get_unmapped_area()

函数get_unmapped_area()搜查进程的地址空间以找到一个可以使用的线性地址区间。len参数指定区间的长度，而非空的addr参数指定必须从哪个地址开始进行查找。如果查找成功，函数返回这个新区间的起始地址；否则返回错误码-ENOMEM。

如果参数addr不等于NULL，函数就检查所指定的地址是否在用户态空间并与页边界对齐。接下来，函数根据线性地址区间是否应该用于文件内存映射或匿名内存映射，调用两个方法（get_unmapped_area文件操作和内存描述符的get_unmapped_area方法）中的一个。在前一种情况下，函数执行get_unmapped_area文件操作，在第十六章将对此进行讨论。

在第二种情况下，函数执行内存描述符的get_unmapped_area方法。根据进程的线性区类型，由函数arch_get_unmapped_area()或arch_get_unmapped_area_topdown()实现get_unmapped_area方法。在第二十章“程序段和进程的线性区”一节，我们将会看到通过系统调用mmap()，每个进程都可能获得两种不同形式的线性区：一种从线性地址0x40000000开始并向高端地址增长，另一种正好从用户态堆栈开始并向低端地址增长。

现在我们讨论函数arch_get_unmapped_area()，在分配从低端地址向高端地址移动的线性区时使用这个函数。它本质上等价于下面的代码片段：

```
if (len > TASK_SIZE)
    return -ENOMEM;
addr = (addr + 0xfff) & 0xfffff000;
if (addr && addr + len <= TASK_SIZE) {
    vma = find_vma(current->mm, addr);
    if (!vma || addr + len <= vma->vm_start)
        return addr;
}
start_addr = addr = mm->free_area_cache;
for (vma = find_vma(current->mm, addr); ; vma = vma->vm_next) {
    if (addr + len > TASK_SIZE) {
        if (start_addr == (TASK_SIZE/3+0xfff)&0xfffff000)
            return -ENOMEM;
        start_addr = addr = (TASK_SIZE/3+0xfff)&0xfffff000;
        vma = find_vma(current->mm, addr);
    }
    if (!vma || addr + len <= vma->vm_start) {
        mm->free_area_cache = addr + len;
        return addr;
```

```
        }
        addr = vma->vm_end;
    }
```

函数首先检查区间的长度是否在用户态下线性地址区间的限长TASK-SIZE（通常为3GB）之内。如果addr不为0，函数就试图从addr开始分配区间。为了安全起见，函数把addr的值调整为4KB的倍数。

如果addr等于0或前面的搜索失败，函数arch_get_unmapped_area()就扫描用户态线性地址空间以查找一个可以包含新区的足够大的线性地址范围，但任何已有的线性区都不包括这个地址范围 。为了提高搜索的速度，让搜索从最近被分配的线性区后面的线性地址开始。把内存描述符的字段mm->free_area_cache初始化为用户态线性地址空间的三分之一（通常是1GB），并在以后创建新线性区时对它进行更新。如果函数找不到一个合适的线性地址范围，就从用户态线性地址空间的三分之一的开始处重新开始搜索：其实，用户态线性地址空间的三分之一是为有预定义起始线性地址的线性区（典型的是可执行文件的正文段、数据段和bss段，参见第二十章）而保留的。

函数调用find_vma()以确定搜索起点之后第一个线性区终点的位置。可能出现三种情况：

- 如果所请求的区间大于正待扫描的线性地址空间部分(addr + len > TASK-SIZE)，函数就从用户态地址空间的三分之一处重新开始搜索，如果已经完成第二次搜索，就返回-ENOMEM（没有足够的线性地址空间来满足这个请求）。
- 刚刚扫描过的线性区后面的空闲区没有足够的大小(vma != NULL && vma->vm_start < addr + len)。此时，继续考虑下一个线性区。
- 如果以上两种情况都没有发生，则找到一个足够大的空闲区，此时，函数返回addr。

向内存描述符链表中插入一个线性区：insert_vm_struct()

insert_vm_struct()函数在线性区对象链表和内存描述符的红－黑树中插入一个vm_area_struct结构。这个函数使用两个参数：mm指定进程内存描述符的地址，vmp指定要插入的vm_area_struct对象的地址。线性区对象的vm_start和vm_end字段必定已经初始化过。该函数调用find_vma_prepare()在红－黑树mm->mm_rb中查找vma应该位于何处。然后，insert_vm_struct()又调用vma_link()函数，后者依次执行以下操作：

1. 在mm->mmap所指向的链表中插入线性区。
2. 在红－黑树mm->mm_rb中插入线性区。

3. 如果线性区是匿名的，就把它插入以相应的anon_vma数据结构作为头节点的链表中（参见第十七章“匿名页的反向映射一节”）。

4. 递增mm->map_count计数器。

如果线性区包含一个内存映射文件，则vma_link()函数执行在第十七章描述的其他任务。

__vma_unlink()函数接收的参数为一个内存描述符地址mm和两个线性区对象地址vma和prev。两个线性区都应当属于mm，prev应当在线性区的排序中位于vma之前。该函数从内存描述符链表和红－黑树中删除vma，如果mm->mmap_cache（存放刚被引用的线性区）字段指向刚被删除的线性区，则还要对mm->mmap_cache进行更新。

分配线性地址区间

现在让我们讨论怎样分配一个新的线性地址区间。为了做到这点，do_mmap()函数为当前进程创建并初始化一个新的线性区。不过，分配成功之后，可以把这个新的线性区与进程已有的其他线性区进行合并。

do_mmap()函数使用下面的参数：

file 和 offset
: 如果新的线性区将把一个文件映射到内存，则使用文件描述符指针file和文件偏移量offset。这个主题将在第十六章进行讨论。在这一节中，我们假定不需要内存映射，并且file和offset都为空。

addr
: 这个线性地址指定从何处开始查找一个空闲的区间。

len
: 线性地址区间的长度。

prot
: 这个参数指定这个线性区所包含页的访问权限。可能的标志有PROT_READ、PROT_WRITE、PROT_EXEC和PROT_NONE。前三个标志与标志VM_READ、VM_WRITE及VM_EXEC的意义一样。PROT_NONE表示进程没有以上三个访问权限中的任意一个。

flag
: 这个参数指定线性区的其他标志：

  MAP_GROWSDOWN、MAP_LOCKED、MAP_DENYWRITE和MAP_EXECUTABLE

  它们的含义与表9-5中所列出标志的含义相同。

MAP_SHARED和MAP_PRIVATE

前一个标志指定线性区中的页可以被几个进程共享；后一个标志作用相反。这两个标志都指向vm_area_struct描述符中的VM_SHARED标志。

MAP_FIXED

区间的起始地址必须是由参数addr所指定的。

MAP_ANONYMOUS

没有文件与这个线性区相关联（参见第十六章）。

MAP_NORESERVE

函数不必预先检查空闲页框的数目。

MAP_POPULATE

函数应该为线性区建立的映射提前分配需要的页框。该标志仅对映射文件的线性区（参见第十六章）和IPC共享的线性区（参见第十九章）有意义。

MAP_NONBLOCK

只有在MAP_POPULATE标志置位时才有意义：提前分配页框时，函数肯定不阻塞。

do_mmap()函数对offset的值进行一些初步检查，然后执行do_mmap_pgoff()函数。本章假设新的线性地址区间映射的不是磁盘文件(在第十六章将详细讨论文件内存映射)。这里仅对实现匿名线性区的do_mmap_pgoff()函数进行说明。

1. 检查参数的值是否正确，所提的请求是否能被满足。尤其是要检查以下不能满足请求的条件：
   - 线性地址区间的长度为0或者包含的地址大于TASK_SIZE。
   - 进程已经映射了过多的线性区，因此mm内存描述符的map_count字段的值超过了允许的最大值。
   - flag参数指定新线性地址区间的页必须被锁在RAM中，但不允许进程创建上锁的线性区，或者进程加锁页的总数超过了保存在进程描述符signal->rlim[RLIMIT_MEMLOCK].rlim_cur字段中的阈值。

   如果以上情况中的任何一个成立，则do_mmap_pgoff()函数终止并返回一个负值。如果线性地址区间的长度为0，则函数不执行任何操作就返回。
2. 调用get_unmapped_area()获得新线性区的线性地址区间（参见上一节“线性区的处理”）
3. 通过把存放在prot和flags参数中的值进行组合来计算新线性区描述符的标志：

```
vm_flags = calc_vm_prot_bits(prot,flags) |
           calc_vm_flag_bits(prot,flags) |
           mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
if (flags & MAP_SHARED)
   vm_flags |= VM_SHARED | VM_MAYSHARE;
```

只有在prot中设置了相应的PROT_READ、PROT_WRITE和PROT_EXEC标志，calc_vm_prot_bits()函数才在vm_flags中设置VM_READ、VM_WRITE和VM_EXEC标志；只有在flags设置了相应的MAP_GROWSDOWN、MAP_DENYWRITE、 MAP_EXECUTABLE和MAP_LOCKED标志，calc_vm_flag_bits()也才在vm_flags中设置VM_GROWSDOWN、VM_DENYWRITE、VM_EXECUTABLE和VM-LOCKED标志。在vm_flags中还有几个标志被置为1：VM_MAYREAD，VM_MAYWRITE，VM_MAYEXEC，在mm->def_flags（注5）中所有线性区的默认标志，以及如果线性区的页与其他进程共享时的VM_SHARED和VM_MAYSHARE。

4. 调用find_vma_prepare()确定处于新区间之前的线性区对象的位置，以及在红－黑树中新线性区的位置：

```
for (;;) {
    vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
    if (!vma || vma->vm_start >= addr + len)
        break;
    if (do_munmap(mm, addr, len))
        return -ENOMEM;
}
```

find_vma_prepare()函数也检查是否还存在与新区间重叠的线性区。这种情况发生在函数返回一个非空的地址，这个地址指向一个线性区，而该区的起始位置位于新区间结束地址之前的时候。在这种情况下，do_mmap_pgoff()调用do_munmap()删除新的区间，然后重复整个步骤（参见下一节“释放线性地址区间”）。

5. 检查插入新的线性区是否引起进程地址空间的大小(mm->total_vm<<PAGE_SHIFT)+len超过存放在进程描述符signal->rlim[RLIMIT_AS].rlim_cur字段中的阈值。如果是，就返回出错码-ENOMEM。注意，这个检查只在这里进行，而不在第1步与其他检查一起进行，因为一些线性区可能在第4步就被删除。

6. 如果在flags参数中没有设置MAP_NORESERVE标志，新的线性区包含私有可写页，并且没有足够的空闲页框，则返回出错码-ENOMEM；这最后一个检查是由security_vm_enough_memory()函数实现的。

7. 如果新区间是私有的（没有设置VM_SHARED），且映射的不是磁盘上的一个文件，那么，调用vma_merge()检查前一个线性区是否可以以这样的方式进行扩展来包

注5： 实际上，内存描述符的def_flags字段只能由mlockall()系统调用修改，这个系统调用可以设置VM_LOCKED标志，由此而锁住RAM中调用进程的未来所有页。

含新的区间。当然，前一个线性区必须与在vm_flags局部变量中存放标志的那些线性区具有完全相同的标志。如果前一个线性区可以扩展，那么，vma_merge()也试图把它与随后的线性区进行合并（这发生在新区间填充两个线性区之间的空洞，且三个线性区全部具有相同的标志的时候）。万一在扩展前一个线性区时获得成功，则跳到第12步。

8. 调用slab分配函数kmem_cache_alloc()为新的线性区分配一个vm_area_struct数据结构。

9. 初始化新的线性区对象（由vma指向）：

```
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_next = NULL;
INIT_LIST_HEAD(&vma->shared);
```

10. 如果MAP_SHARED标志被设置（以及新的线性区不映射磁盘上的文件），则该线性区是一个共享匿名区：调用shmem_zero_setup()对它进行初始化。共享匿名区主要用于进程间通信（参见第十九章“IPC共享内存”一节）。

11. 调用vma_link()把新线性区插入到线性区链表和红－黑树中（参见前面“线性区的处理”一节）。

12. 增加存放在内存描述符total_vm字段中的进程地址空间的大小。

13. 如果设置了VM_LOCKED标志，就调用make_pages_present()连续分配线性区的所有页，并把它们锁在RAM中：

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

make-pages-present()函数按如下方式调用get_user_pages()：

```
write = (vma->vm_flags & VM_WRITE) != 0;
get_user_pages(current, current->mm, addr, len, write, 0, NULL,
NULL);
```

get_user_pages()函数在addr和addr+len之间页的所有起始线性地址上循环；对于其中的每个页，该函数调用follow_page()检查在当前页表中是否有到物理页的映射。如果没有这样的物理页存在，则get_user_pages()调用handle_mm_fault()，

我们将在“处理地址空间内的错误地址”一节看到，后一个函数分配一个页框并根据内存描述符的vm_flags字段设置它的页表项。

14. 最后，函数通过返回新线性区的线性地址而终止。

释放线性地址区间

内核使用do_munmap()函数从当前进程的地址空间中删除一个线性地址区间。参数为：进程内存描述符的地址mm，地址区间的起始地址start和它的长度len。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。

do_munmap()函数

该函数经过两个主要的阶段。第一阶段（第1～6步），扫描进程所拥有的线性区链表，并把包含在进程地址空间的线性地址区间中的所有线性区从链表中解除链接。第二阶段（第7～12步），更新进程的页表，并把第一阶段找到并标识出的线性区删除。函数利用稍后要说明的split_vma()和unmap_region()函数。do_munmap()执行下面的步骤：

1. 对参数值进行一些初步检查：如果线性地址区间所含的地址大于TASK_SIZE，如果start不是4096的倍数，或者如果线性地址区间的长度为0，则函数返回一个错误代码-EINVAL。

2. 确定要删除的线性地址区间之后第一个线性区mpnt的位置(mpnt->end> start)，如果有这样的线性区：

```
mpnt = find_vma_prev(mm, start, &prev);
```

3. 如果没有这样的线性区，也没有与线性地址区间重叠的线性区，就什么都不做，因为在该区间上没有线性区：

```
end = start + len;
if (!mpnt || mpnt->vm_start >= end)
    return 0;
```

4. 如果线性区的起始地址在线性区mpnt内，就调用split_vma()(在下面说明）把线性区mpnt划分成两个较小的区：一个区在线性地址区间外部，而另一个在区间内部。

```
if (start > mpnt->vm_start) {
    if (split_vma(mm, mpnt, start, 0))
        return -ENOMEM;
    prev = mpnt;}
```

更新局部变量prev，以前它存储的是指向线性区mpnt前面一个线性区的指针，现在

要让它指向mpnt，即指向线性地址区间外部的那个新线性区。这样，prev仍然指向要删除的第一个线性区前面的那个线性区。

5. 如果线性地址区间的结束地址在一个线性区内部，就再次调用split_vma()把最后重叠的那个线性区划分成两个较小的区：一个区在线性地址区间内部，而另一个在区间外部（注6）：

```
last = find_vma(mm, end);
if (last && end > last->vm_start){
    if (split_vma(mm, last, start, end, 1))
        return -ENOMEM;
}
```

6. 更新mpnt的值，使它指向线性地址区间的第一个线性区。如果prev为NULL，即没有上述线性区，就从mm->mmap获得第一个线性区的地址：

```
mpnt = prev ? prev->vm_next : mm->mmap;
```

7. 调用detach_vmas_to_be_unmapped()从进程的线性地址空间中删除位于线性地址区间中的线性区。该函数本质上执行下面的代码：

```
vma = mpnt;
insertion_point = (prev ? &prev->vm_next : &mm->mmap);
do {
    rb_erase(&vma->vm_rb, &mm->mm_rb);
    mm->map_count--;
    tail_vma = vma;
    vma = vma->next;
} while (vma && vma->start < end);
*insertion_point = vma;
tail_vma->vm_next = NULL;
mm->map_cache = NULL;
```

要删除的线性区的描述符放在一个排好序的链表中，局部变量mpnt指向该链表的头（实际上，这个链表只是进程初始线性区链表的一部分）。

8. 获得mm->page_table_lock自旋锁。

9. 调用unmap_region()清除与线性地址区间对应的页表项并释放相应的页框（稍后讨论）：

```
unmap_region(mm, mpnt, prev, start, end);
```

10. 释放mm->page_table_lock自旋锁。

注6：如果线性地址区间正好包含在某个线性区内部，就必须用用个较小的新线性区取代该线性区。当发生这种情况时，在第4步和第5步把该线性区分成三个较小的线性区：删除中间的那个线性区，而保留第一个和最后一个线性区。

11. 释放在第7步建立链表时收集的线性区描述符：

```
do {
    struct vm_area_struct * next = mpnt->vm_next;
    unmap_vma(mm, mpnt);
    mpnt = next;
} while (mpnt != NULL);
```

对在链表中的所有线性区调用unmap_vma()函数，它本质上执行下述步骤：

a. 更新mm->total_vm和mm->locked_vm字段。

b. 执行内存描述符的mm->unmap_area方法。根据进程线性区的不同类型（参见前面“线性区的处理”一节）可以选择arch_unmap_area()或arch_unmap_area_topdown()中的一个来实现mm->unmap_area方法。如果必要，在两种情况下都要更新mm->free_area_cache字段。

c. 调用线性区的close方法（如果定义了的话）。

d. 如果线性区是匿名的，则函数把它从mm->anon_vma所指向的匿名线性区链表中删除。

e. 调用kmem_cache_free()释放线性区描述符。

12. 返回0（成功）。

split_vma()函数

split_vma()函数的功能是把与线性地址区间交叉的线性区划分成两个较小的区，一个在线性地址区间外部，另一个在区间的内部。该函数接收4个参数：内存描述符指针mm，线性区描述符指针vma（标识要被划分的线性区），表示区间与线性区之间交叉点的地址addr，以及表示区间与线性区之间交叉点在区间起始处还是结束处的标志new_below。函数执行下述基本步骤：

1. 调用kmem_cache_alloc()获得线性区描述符vm_area_struct，并把它的地址存在新的局部变量中，如果没有可用的空闲空间，就返回-ENOMEM。

2. 用vma描述符的字段值初始化新描述符的字段。

3. 如果标志new_below为0，说明线性地址区间的起始地址在vma线性区的内部，因此必须把新线性区放在vma线性区之后，所以函数把new->vm_start和vma->vm_end字段都赋值为addr。

4. 相反，如果new_below标志等于1，说明线性地址区间的结束地址在vma线性区的内部，因此必须把新线性区放在vma线性区的前面，所以函数把字段new->vm_end和vma->vm_start都赋值为addr。

5. 如果定义了新线性区的open方法，函数就执行它。

6. 把新线性区描述符链接到线性区链表mm->mmap和红－黑树mm->mm_rb。此外，函数还要根据线性区vma的最新大小对红－黑树进行调整。
7. 返回0（成功）。

unmap_region()函数

unmap_region()函数遍历线性区链表并释放它们的页框。该函数作用于5个参数：内存描述符指针mm，指向第一个被删除线性区描述符的指针vma，指向进程链表中vma前面的线性区的指针prev（参见do_munmap()执行步骤中心的第2步和第4步），以及两个地址start和end，它们界定被删除线性地址区间的范围。函数本质上执行下述步骤：

1. 调用lru_add_drain()（参见第十七章）。
2. 调用tlb_gather_mmu()函数初始化每CPU变量mmu_gathers。mmu_gathers的值依赖于体系结构：通常该变量应该存放成功更新进程页表项所需要的所有信息。在80x86体系结构中，tlb_gather_mmu()函数只是简单地把内存描述符指针mm的值赋给本地CPU的mmu_gathers变量。
3. 把mmu_gathers变量的地址保存在局部变量tlb中。
4. 调用unmap_vmas()扫描线性地址空间的所有页表项：如果只有一个有效CPU，函数就调用free_swap_and_cache()反复释放相应的页（参见第十七章）；否则，函数就把相应页描述符的指针保存在局部变量mmu_gathers中。
5. 调用free_pgtables(tlb,prev,start,end)回收在上一步已经清空的进程页表。
6. 调用tlb_finish_mmu(tlb,start,end)结束unmap_region()函数的工作，tlb_finish_mmu(tlb,start,end)依次执行下面的操作：
   a. 调用flush_tlb_mm()刷新TLB（参见第二章“处理硬件高速缓存和TLB”一节）。
   b. 在多处理器系统中，调用free_pages_and_swap_cache()释放页框，这些页框的指针已经集中存放在mmu_gather数据结构中了。该函数的说明见第十七章。

缺页异常处理程序

如前所述，Linux的缺页（Page Fault）异常处理程序必须区分以下两种情况：由编程错误所引起的异常，及由引用属于进程地址空间但还尚未分配物理页框的页所引起的异常。

线性区描述符可以让缺页异常处理程序非常有效地完成它的工作。do_page_fault()函数是80x86上的缺页中断服务程序，它把引起缺页的线性地址和当前进程的线性区相比较，从而能够根据和图9-4所示的方案选择适当的方法处理这个异常。

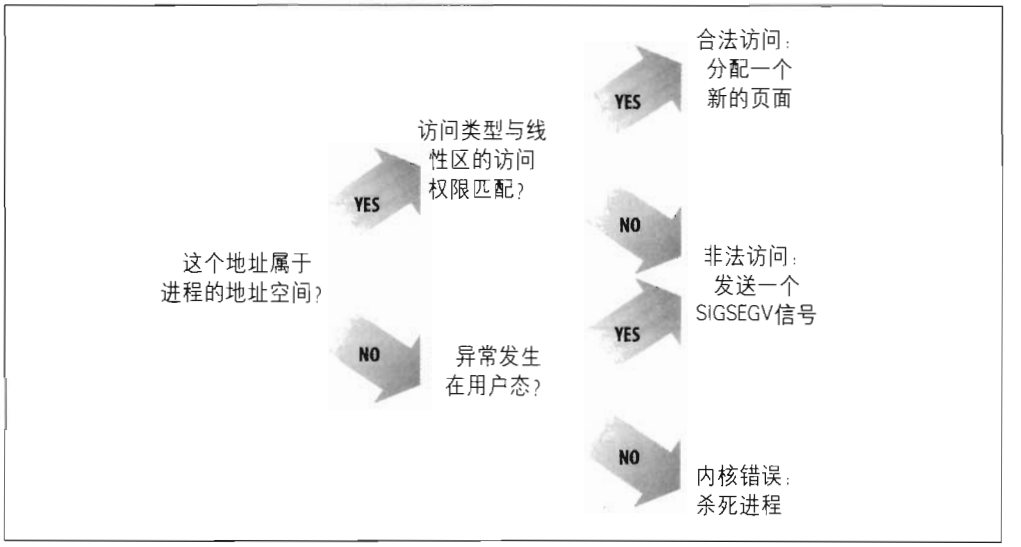


图 9-4：缺页异常处理程序的总体方案

在实际中，情况更复杂一些，因为缺页处理程序必须处理多种分得更细的特殊情况，它们不宜在总体方案中列出来，还必须区分许多种合理的访问。处理程序的详细流程图如图 9-5 所示。

标识符vmalloc_fault、good_area、bad_area和no_context是出现在do_page_fault()中的标记，它们有助于你理清流程图中的块与代码中特定行之间的关系。

do_page_fault()函数接收以下输入参数：

- pt_regs结构的地址regs，该结构包含当异常发生时的微处理器寄存器的值。
- 3位的error_code，当异常发生时由控制单元压入栈中（参见第四章中的“中断和异常的硬件处理”一节）。这些位有以下含义：
  - 如果第0位被清0，则异常由访问一个不存在的页所引起（页表项中的Present标志被清0）；否则，如果第0位被设置，则异常由无效的访问权限所引起。
  - 如果第1位被清0，则异常由读访问或者执行访问所引起；如果该位被设置，则异常由写访问所引起。
  - 如果第2位被清0，则异常发生在处理器处于内核态时；否则，异常发生在处理器处于用户态时。

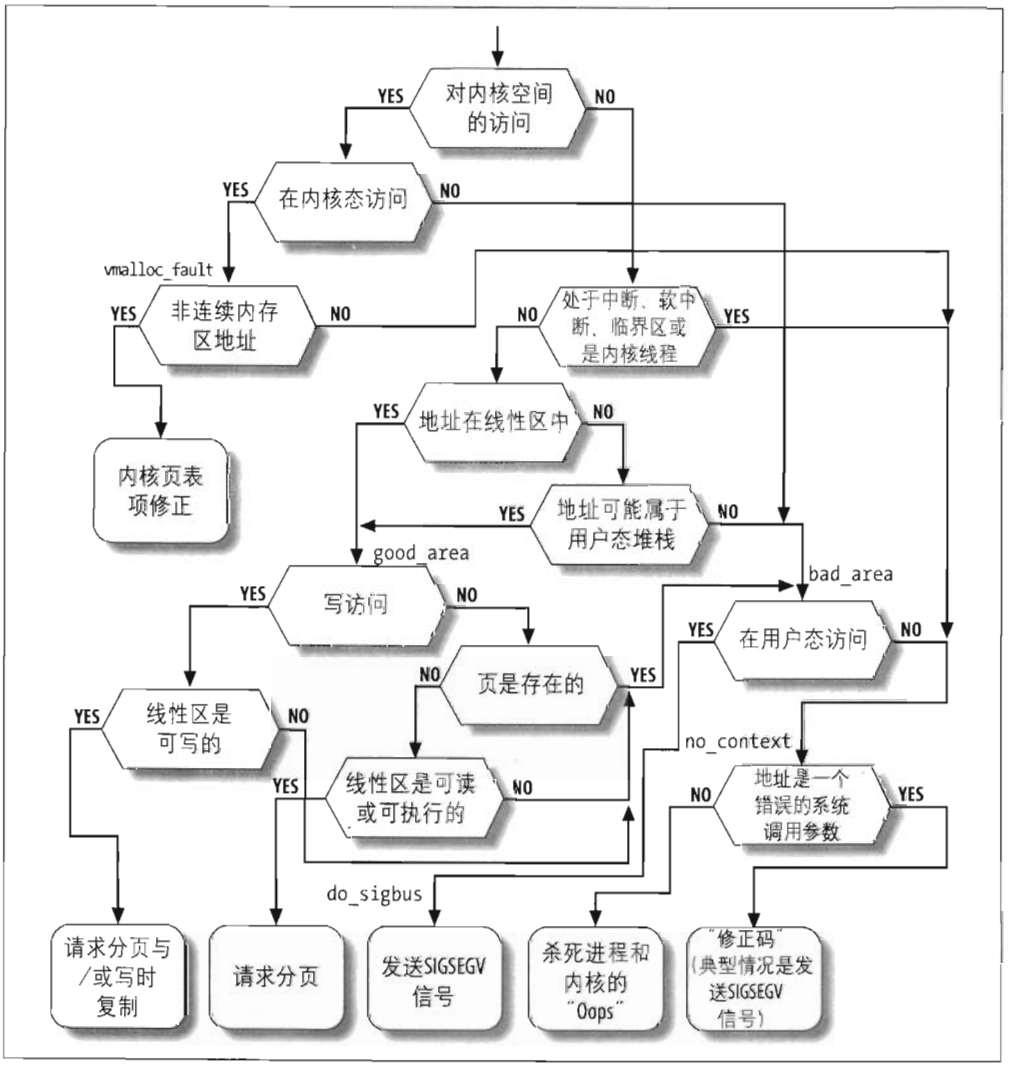


图 9-5：缺页处理程序流程图

do_page_fault()的第一步操作是读取引起缺页的线性地址。当异常发生时，CPU控制单元把这个值存放在 cr2 控制寄存器中：

```
asm("movl %%cr2,%0":"=r" (address));
if (regs->eflags & 0x00020200)
    local_irq_enable();
tsk = current;
```

这个线性地址保存在address局部变量中。如果缺页发生之前或CPU运行在虚拟8086

模式时就打开了本地中断，那么该函数还要确保本地中断打开，并把指向current进程描述符的指针保存在tsk局部变量中。

正如图9-5中的顶部所示，do_page_fault()首先检查引起缺页的线性地址是否属于第4个GB：

```
info.si_code = SEGV_MAPERR;
if (address >= TASK_SIZE ) {
    if  (!(error_code & 0x101))
        goto vmalloc_fault;
    goto bad_area_nosemaphore;
}
```

如果发生了由于内核试图访问不存在的页框引起的异常，就跳转去执行vmalloc_fault标记处的代码，该部分代码处理可能由于在内核态访问非连续内存区而引起的缺页，我们在稍后“处理非连续内存区访问”一节中对此进行说明。否则，就跳转去执行bad_area_nosemaphore标记处的代码，在稍后“处理地址空间以外的错误地址”一节中将对此进行说明。

接下来，缺页处理程序检查异常发生时是否内核正在执行一些关键例程或正在运行内核线程（回想一下，对内核线程而言，进程描述符的mm字段总为NULL）：

```
if (in_atomic(  ) || !tsk->mm)
   goto bad_area_nosemaphore;
```

如果缺页发生在下面任何一种情况下，则in_atomic()宏产生等于1的值：

- 内核正在执行中断处理程序或可延迟函数。
- 内核正在禁用内核抢占的情况下执行临界区代码（参见第五章“内核抢占”一节）。

如果缺页的确发生在中断处理程序、可延迟函数、临界区或内核线程中，do_page_fault()就不会试图把这个线性地址与current的线性区做比较。内核线程从来不使用小于TASK_SIZE的地址。同样，中断处理程序、可延迟函数和临界区代码也不应该使用小于TASK_SIZE的地址，因为这可能导致当前进程的阻塞。（参见本章稍后“处理地址空间以外的错误地址”一节中，对info局部变量的信息说明和对bad_area_nosemaphor标记处代码的说明。）

让我们假定缺页没有发生在中断处理程序、可延迟函数、临界区或者内核线程中。于是函数必须检查进程所拥有的线性区以决定引起缺页的线性地址是否包含在进程的地址空间中，为此必须获得进程的mmap_sem 读/写信号量：

```
if (!down_read_trylock(&tsk->mm->mmap_sem)) {
    if ((error_code & 4) == 0 &&
```

```
            !search_exception_table(regs->eip))
            goto bad_area_nosemaphore;
        down_read(&tsk->mm->mmap_sem);
    }
```

如果内核bug和硬件故障有可能被排除，那么当缺页发生时，当前进程就还没有为写而获得信号量mmap_sem。尽管如此，do_page_fault()还是想确定的确没有获得这个信号量，因为如果不是这样就会发生死锁。所以，函数使用down_read_trylock()而不是down_read()(参见第五章“读/写信号量”一节)。如果这个信号量被关闭而且缺页发生在内核态，do_page_fault()就要确定异常发生的时候，是否正在使用作为系统调用参数被传递给内核的线性地址（参见下一节“处理地址空间以外的错误地址”一节)。此时，因为每个系统调用服务例程都小心地避免在访问用户态地址空间以前为写而获得mmap_sem信号量，所以do_page_fault()确信mmap_sem信号量由另外一个进程占有了，从而do_page_fault()一直等待直到该信号量被释放。否则，如果缺页是由于内核bug或严重的硬件故障引起的，函数就跳转到bad_area_nosemaphore标记处。

我们假设已经为读而获得了mmap_sem信号量。现在，do_page_fault()开始搜索错误线性地址所在的线性区：

```
vma = find_vma(tsk->mm, address);
if (!vma)
    goto bad_area;
if (vma->vm_start <= address)
    goto good_area;
```

如果vma为NULL，说明address之后没有线性区，因此这个错误的地址肯定是无效的。另一方面，如果在address之后结束处的第一个线性区包含address，则函数跳到标记为good_area的代码处。

如果两个“if”条件都不满足，则函数已确定address没有包含在任何线性区中。可是，它还必须执行进一步的检查，由于这个错误地址可能是由push或pusha指令在进程的用户态堆栈上的操作所引起的。

我们稍微离题一点，解释一下栈是如何映射到线性区上的。每个向低地址扩展的栈所在的区，它的VM_GROWSDOWN标志被设置，这样，当vm_start字段的值可能被减小的时候，而vm_end字段的值保持不变。这种线性区的边界包括、但不严格限定用户态堆栈当前的大小。这种细微的差别主要基于以下原因：

- 线性区的大小是4KB的倍数（必须包含完整的页），而栈的大小却是任意的。
- 分配给一个线性区的页框在这个线性区被删除前永远不被释放。尤其是，一个栈所在线性区的vm_start字段的值只能减小，永远也不能增加。甚至进程执行一系列的pop指令时，这个线性区的大小仍然保持不变。

现在这一点就很清楚了，当进程填满分配给它的堆栈的最后一个页框后，进程如何引起一个“缺页”异常——push引用了这个线性区以外的一个地址（即引用一个不存在的页框）。注意，这种异常不是由程序错误引起的，因此它必须由缺页处理程序单独处理。

我们现在回到对do_page_fault()的描述，它检查上面所描述的情况：

```
if (!(vma->vm_flags & VM_GROWSDOWN))
    goto bad_area;
if (error_code & 4      /* User Mode */
    && address + 32 < regs->esp)
        goto bad_area;
if (expand_stack(vma, address))
    goto bad_area;
goto good_area;
```

如果线性区的VM_GROWSDOWN标志被设置，并且异常发生在用户态，函数就要检查address是否小于regs->esp栈指针（它应该只小于一点点）。因为几个与栈相关的汇编语言指令（如pusha）只有在访问内存之后才执行减esp寄存器的操作，所以允许进程有32字节的后备区间。如果这个地址足够高（在允许的范围内），则代码调用expand_stack()函数检查是否允许进程既扩展它的栈也扩展它的地址空间。如果一切都可以，就把vma的vm_start字段设为address，并返回0；否则，返回-ENOMEM。

注意：只要线性区的VM_GROWSDOWN标志被设置，但异常不是发生在用户态，上述代码就跳过容错检查。这些条件意味着内核正在访问用户态的栈，也意味着这段代码总是应当运行expand_stack()。

处理地址空间以外的错误地址

如果address不属于进程的地址空间，那么do_page_fault()函数继续执行bad_area标记处的语句。如果错误发生在用户态，则发送一个SIGSEGV信号给current进程（参见第十一章的“产生信号”一节）并结束函数：

```
bad_area:
up_read(&tsk->mm->mmap_sem);
bad_area_nosemaphore:
if (error_code & 4) {    /* User Mode */
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code | (address >= TASK_SIZE);
    tsk->thread.trap_no = 14;
    info.si_signo = SIGSEGV;
    info.si_errno = 0;
    info.si_addr = (void *) address;
    force_sig_info(SIGSEGV, &info, tsk);
    return;
}
```

force_sig_info()确信进程不忽略或阻塞SIGSEGV信号，并通过info局部变量传递附加信息的同时把该信号发送给用户态进程（参见第十一章“产生信号”一节）。info.si_code字段已被置为SEGV_MAPERR（如果异常是由于一个不存在的页框引起），或置为SEGV_ACCERR（如果异常是由于对现有页框的无效访问引起）。

如果异常发生在内核态（error_code的第2位被清0），仍然有两种可选的情况：

- 异常的引起是由于把某个线性地址作为系统调用的参数传递给内核。
- 异常是因一个真正的内核缺陷所引起的。

函数这样区分这两种可选的情况：

```
no_context:
if ((fixup = search_exception_table(regs->eip)) != 0) {
    regs->eip = fixup;
    return;
}
```

在第一种情况中，代码跳到一段“修正代码”处，这段代码的典型操作就是向当前进程发送SIGSEGV信号，或用一个适当的出错码终止系统调用处理程序（参见第十章中的“动态地址检查：修正代码”一节）。

在第二种情况中，函数把CPU寄存器和内核态堆栈的全部转储打印到控制台，并输出到一个系统消息缓冲区，然后调用函数do_exit()杀死当前进程（参见第二十章）。这就是所谓按所显示的消息命名的“内核漏洞（*Kernel oops*）”错误。这些输出值可由内核编程高手用于推测引发此错误的条件，进而发现并纠正错误。

处理地址空间内的错误地址

如果addr地址属于进程的地址空间，则do_page_fault()转到good_area标记处的语句执行：

```
good_area:
info.si_code = SEGV_ACCERR;
write = 0;
if (error_code & 2) { /* write access */
    if (!(vma->vm_flags & VM_WRITE))
        goto bad_area;
    write++;
} else                /* read access */
    if ((error_code & 1) || !(vma->vm_flags & (VM_READ | VM_EXEC)))
        goto bad_area;
```

如果异常由写访问引起，函数检查这个线性区是否可写。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。

如果异常由读或执行访问引起，函数检查这一页是否已经存在于RAM中。在存在的情况下，异常发生是由于进程试图访问用户态下的一个有特权的页框（页框的User/Supervisor标志被清除），因此函数跳到bad_area代码处（注7）。在不存在的情况下，函数还将检查这个线性区是否可读或可执行。

如果这个线性区的访问权限与引起异常的访问类型相匹配，则调用handle_mm_fault()函数分配一个新的页框：

```
survive:
ret = handle_mm_fault(tsk->mm, vma, address, write);
if (ret == VM_FAULT_MINOR || ret == VM_FAULT_MAJOR) {
    if (ret == VM_FAULT_MINOR) tsk->min_flt++; else tsk->maj_flt++;
    up_read(&tsk->mm->mmap_sem);
    return;
}
```

如果handle_mm_fault()函数成功地给进程分配一个页框，则返回VM-FAULT-MINOR或VM-FAULT-MAJOR。值VM-FAULT-MINOR表示在没有阻塞当前进程的情况下处理了缺页；这种缺页叫做次缺页（*minor fault*）。值VM-FAULT-MAJOR表示缺页迫使当前进程睡眠（很可能是由于当用磁盘上的数据填充所分配的页框时花费时间）；阻塞当前进程的缺页就叫做主缺页（*major fault*）。函数也返回VM-FAULT-OOM（没有足够的内存）或VM-FAULT-STGBOS（其他任何错误）。

如果handle_mm_fault()返回值VM_FAULT_SIGBUS，则向进程发送SIGBUS信号：

```
if (ret == VM_FAULT_SIGBUS) {
do_sigbus:
    up_read(&tsk->mm->mmap_sem);
    if (!(error_code & 4)) /*内核态*/
        goto no_context;
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code;
    tsk->thread.trap_no = 14;
    info.si_signo = SIGBUS;
    info.si_errno = 0;
    info.si_code = BUS_ADRERR;
    info.si_addr = (void *) address;
    force_sig_info(SIGBUS, &info, tsk);
}
```

如果handle_mm_fault()不分配新的页框，就返回值VM_FAULT_OOM，此时内核通常杀死当前进程。不过，如果当前进程是init进程，则只是把它放在运行队列的末尾并调用调度程序；一旦init恢复执行，则handle_mm_fault()又执行：

注7：　然而，这种情况从不会发生，因为内核不会把具有特权的页框赋给进程。

```
if (ret == VM_FAULT_OOM) {
  out_of_memory:
    up_read(&tsk->mm->mmap_sem);
    if (tsk->pid != 1) {
        if (error_code & 4) /* User Mode */
            do_exit(SIGKILL);
        goto no_context;
    }
    yield();
    down_read(&tsk->mm->mmap_sem);
    goto survive;
}
```

handle_mm_fault()函数作用于4个参数：

mm

指向异常发生时正在CPU上运行的进程的内存描述符。

vma

指向引起异常的线性地址所在线性区的描述符。

address

引起异常的线性地址。

write_access

如果tsk试图向address写，则置为1；如果tsk试图在address读或执行，则置为0。

这个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，相应的页表也可能还没有被分配，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(mm, address);
spin_lock(&mm->page_table_lock);
pud = pud_alloc(mm, pgd, address);
if (pud) {
    pmd = pmd_alloc(mm, pud, address);
    if (pmd) {
        pte = pte_alloc_map(mm, pmd, address);
        if (pte)
            return handle_pte_fault(mm, vma, address,
                                    write_access, pte, pmd);
    }
}
spin_unlock(&mm->page_table_lock);
return VM_FAULT_OOM;
```

pgd局部变量包含引用address的页全局目录项。如果需要的话，调用pud_alloc()和

pmd_alloc()函数分别分配一个新的页上级目录和页中间目录（注8）。然后，如果需要的话，调用pte_alloc_map()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地址所对应的页表项，并决定如何为进程分配一个新页框：

- 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框中，那么，内核分配一个新的页框并适当地初始化。这种技术称为请求调页（demand paging）。
- 如果被访问的页存在但是标记为只读，也就是说，它已经被存放在一个页框中，那么，内核分配一个新的页框，并把旧页框的数据拷贝到新页框来初始化它的内容。这种技术称为写时复制（*Copy On Write*，COW）。

请求调页

术语“请求调页”指的是一种动态内存分配技术，它把页框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RAM中时为止，由此引起一个缺页异常。

请求调页技术背后的动机是：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，有一部分地址也许永远不被进程使用。此外，程序的局部性原理（参见第二章中的“硬件高速缓存”一节）保证了在程序执行的每个阶段，真正引用的进程页只有一小部分，因此临时用不着的页所在的页框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框，直到程序结束才释放这些页框）来说，请求调页是首选的，因为它增加了系统中的空闲页框的平均数，从而更好地利用空闲内存。从另一个观点来看，在RAM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”异常必须由内核处理，这将浪费CPU的时钟周期。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”异常是一种稀有事件。

被访问的页可能不在主存中，其原因或者是进程从没访问过该页，或者是内核已经回收了相应的页框（参见第十七章）。

注8：在80x86微处理器中，这种分配永远不会发生，因为页上级目录总是包含在页全局目录中，并且页中间目录或者包含在页上级目录中（PAE未激活），或者与页上级目录一块被分配（PAE被激活）。

在这两种情况下，缺页处理程序必须为进程分配新的页框。不过，如何初始化这个页框取决于是哪一种页以及页以前是否被进程访问过。特殊情况下：

1. 或者这个页从未被进程访问到且没有映射磁盘文件，或者页映射了磁盘文件。内核能够识别这些情况，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. 页属于非线性磁盘文件的映射（参见第十六章“非线性内存映射”一节）。内核能够识别这种情况，因为Present标志被清0而且Dirty标志被置1，也就是说，pte_file宏返回1。
3. 进程已经访问过这个页，但是其内容被临时保存在磁盘上。内核能够识别这种情况，这是因为相应表项没被填充为0，但是Present和Dirty标志被清0。

因此，handle_pte_fault()函数通过检查address对应的页表项能够区分这三种情况：

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

我们将在第十六章和第十七章分别讨论第2和第3种情况。

在第1种情况下，当页从未被访问或页线性地映射磁盘文件时则调用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一个磁盘文件。该函数通过检查vma线性区描述符的nopage字段来确定这一点，如果页被映射到一个文件，nopage字段就指向一个函数，该函数把所缺的页从磁盘装入到RAM。因此，可能的情况是：

- vma->vm_ops->nopage字段不为NULL。在这种情况下，线性区映射了一个磁盘文件，nopage字段指向装入页的函数。这种情况将在第十六章的“内存映射的请求调页”一节和第十九章的“IPC共享内存”一节进行阐述。
- 或者vma->vm_ops字段为NULL，或者vma->vm_ops->nopage字段为NULL。在这种情况下，线性区没有映射磁盘文件，也就是说，它是一个匿名映射（*anonymous mapping*）。因此，do_no_page()调用do_anonymous_page()函数获得一个新的页框：

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

do_anonymous_page()函数（注9）分别处理写请求和读请求：

```
if (write_access) {
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
    page = alloc_page(GFP_HIGHUSER | _  _GFP_ZERO);
    spin_lock(&mm->page_table_lock);
    page_table = pte_offset_map(pmd, addr);
    mm->rss++;
    entry = maybe_mkwrite(pte_mkdirty(mk_pte(page,
                                        vma->vm_page_prot)), vma);
    lru_cache_add_active(page);
    SetPageReferenced(page);
    set_pte(page_table, entry);
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_MINOR;
}
```

pte_unmap宏的第一次执行释放一种临时内核映射，它映射了在调用handle_pte_fault()函数之前由pte_offset_map所建立页表项的高端内存物理地址（参见第二章“页表处理”一节中的表2-7）。pte_offset_map和pte_unmap这对宏获取和释放同一个临时内核映射。临时内核映射必须在调用alloc_page()之前释放，因为这个函数可能阻塞当前进程。

函数递增内存描述符的rss字段以记录分配给进程的页框总数。相应的页表项设置为页框的物理地址，页表框被标记为既脏又可写的（注10）。lru_cache_add_active()函数把新页框插入与交换相关的数据结构中，我们在第十七章对它进行说明。

相反，当处理读访问时，页的内容是无关紧要的，因为进程第一次对它访问。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0的新页框，由于我们也可以给它一个现有的称为零页（*zero page*）的页，这样可以进一步推迟页框的分配。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（长为4096字节的数组，并用0填充）。

因此用零页的物理地址设置页表项：

注9：为了简化对这个函数的说明，我们略过处理反映射的语句，有关反映射的内容见第十七章“反向映射”一节。

注10：如果调试程序试图往被跟踪进程只读线性区中的页中写数据，内核就不设置Read/Write标志。函数maybe_mkwrite()处理这种特殊情况。

```
entry = pte_wrprotect(mk_pte(virt_to_page(empty_zero_page),
                             vma->vm_page_prot));
set_pte(page_table, entry);
spin_unlock(&mm->page_table_lock);
return VM_FAULT_MINOR;
```

由于这个页被标记为不可写的，因此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操作。这种机制将在下一节中进行描述。

写时复制

第一代Unix系统实现了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框
- 为子进程的页分配页框
- 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

这种创建地址空间的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（参见第二十章）。

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（*Copy On Write*，COW）。这种思想相当简单：父进程和子进程共享页框而不是复制页框。然而，只要页框被共享，它们就不能被修改。无论父进程还是子进程何时试图写一个共享的页框，就产生一个异常，这时内核就把这个页复制到一个新的页框中并标记为可写。原来的页框仍然是写保护的：当其他进程试图写入时，内核检查写进程是否是这个页框的唯一属主，如果是，就把这个页框标记为对这个进程是可写的。

页描述符的_count字段用于跟踪共享相应页框的进程数目。只要进程释放一个页框或者在它上面执行写时复制，它的_count字段就减小；只有当_count变为-1时，这个页框才被释放（参见第八章“页描述符”一节）

现在我们讲述Linux怎样实现写时复制。当handle_pte_fault()确定缺页异常是由访问内存中现有的一个页而引起时，它执行以下指令：

```
if (pte_present(entry)) {
```

```
    if (write_access) {
        if (!pte_write(entry))
            return do_wp_page(mm, vma, address, pte, pmd, entry);
        entry = pte_mkdirty(entry);
    }
    entry = pte_mkyoung(entry);
    set_pte(pte, entry);
    flush_tlb_page(vma, address);
    pte_unmap(pte);
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_MINOR;
}
```

handle_pte_fault()函数是与体系结构无关的：它考虑任何违背页访问权限的可能。然而，在80x86体系结构上，如果页是存在的，那么，访问权限是写允许的而页框是写保护的（参见前面“处理地址空间内的错误地址”一节）。因此，总是要调用do_wp_page()函数。

do_wp_page()函数（注11）首先获取与缺页异常相关的页框描述符（缺页表项对应的页框）。接下来，函数确定页的复制是否真正必要。如果仅有一个进程拥有这个页，那么，写时复制就不必应用，且该进程应当自由地写该页。具体来说，函数读取页描述符的_count字段：如果它等于0（只有一个所有者），写时复制就不必进行。实际上，检查要稍微复杂些，因为当页插入到交换高速缓存（参见第十七章“交换高速缓存”一节）并且当设置了页描述符的PG_private标志时，_count字段也增加。不过，当写时复制不进行时，就把该页框标记为可写的，以免试图写时引起进一步的缺页异常：

```
set_pte(page_table, maybe_mkwrite(pte_mkyoung(pte_mkdirty(pte)),vma));
flush_tlb_page(vma, address);
pte_unmap(page_table);
spin_unlock(&mm->page_table_lock);
return VM_FAULT_MINOR;
```

如果两个或多个进程通过写时复制共享页框，那么函数就把旧页框（old_page）的内容复制到新分配的页框（new_page）中。为了避免竞争条件，在开始复制操作前调用get_page()把old_page的使用计数加1：

```
old_page = pte_page(pte);
pte_unmap(page_table);
get_page(old_page);
spin_unlock(&mm->page_table_lock);
if (old_page == virt_to_page(empty_zero_page))
    new_page = alloc_page(GFP_HIGHUSER | __GFP_ZERO);
```

注11：为了简化对这个函数的说明，我们略过处理反映射的语句，有关反映射的内容见第十七章“反向映射”一节。

```
} else {
    new_page = alloc_page(GFP_HIGHUSER);
    vfrom = kmap_atomic(old_page, KM_USER0)
    vto = kmap_atomic(new_page, KM_USER1);
    copy_page(vto, vfrom);
    kunmap_atomic(vfrom, KM_USER0);
    kunmap_atomic(vto, KM_USER0);
}
```

如果旧页框是零页，就在分配新的页框时（__GFP_ZERO标志）把它填充为0。否则，使用copy_page()宏复制页框的内容。不要求一定要对零页做特殊的处理，但是特殊处理确实能够提高系统的性能，因为它减少地址引用而保护了微处理器的硬件高速缓存。

因为页框的分配可能阻塞进程，因此，函数检查自从函数开始执行以来是否已经修改了页表项（pte和*page_table具有不同的值）。如果是，新的页框被释放，old_page的使用计数器被减少（取消以前的增加），函数结束。

如果所有的事情看起来进展顺利，那么，新页框的物理地址最终被写进页表项，且使相应的TLB寄存器无效：

```
spin_lock(&mm->page_table_lock);
entry = maybe_mkwrite(pte_mkdirty(mk_pte(new_page,
                                         vma->vm_page_prot)),vma);
set_pte(page_table, entry);
flush_tlb_page(vma, address);
lru_cache_add_active(new_page);
pte_unmap(page_table);
spin_unlock(&mm->page_table_lock);
```

lru_cache_add_active()函数把新页框插入到与交换相关的数据结构中；参见第十七章对其的描述。

最后，do_wp_page()把old_page的使用计数器减少两次。第一次的减少是取消复制页框内容之前进行的安全性增加，第二次的减少是反映当前进程不再拥有该页框这一事实。

处理非连续内存区访问

我们已经在第八章“非连续内存区管理”一节中看到，内核在更新非连续内存区对应的页表项时是非常懒惰的。事实上，vmalloc()和vfree()函数只把自己限制在更新主内核页表（即页全局目录init_mm.pgd和它的子页表）。

然而，一旦内核初始化阶段结束，任何进程或内核线程便都不直接使用主内核页表。因此，我们来考虑内核态进程对非连续内存区的第一次访问。当把线性地址转换为物理地址时，CPU的内存管理单元遇到空的页表项并产生一个缺页。但是，缺页异常处理程序

认识这种特殊情况，因为异常发生在内核态且产生缺页的线性地址大于TASK_SIZE。因此，do_page_fault()检查相应的主内核页表项：

```
vmalloc_fault:
asm("movl %%cr3,%0":"=r" (pgd_paddr));
pgd = pgd_index(address) + (pgd_t *) _ _va(pgd_paddr);
pgd_k = init_mm.pgd + pgd_index(address);
if (!pgd_present(*pgd_k))
    goto no_context;
pud = pud_offset(pgd, address);
pud_k = pud_offset(pgd_k, address);
if (!pud_present(*pud_k))
    goto no_context;
pmd = pmd_offset(pud, address);
pmd_k = pmd_offset(pud_k, address);
if (!pmd_present(*pmd_k))
    goto no_context;
set_pmd(pmd, *pmd_k);
pte_k = pte_offset_kernel(pmd_k, address);
if (!pte_present(*pte_k))
    goto no_context;
return;
```

把存放在cr3寄存器中的当前进程页全局目录的物理地址赋给局部变量pgd_paddr（注12），把与pgd_paddr相应的线性地址赋给局部变量pgd，并且把主内核页全局目录的线性地址赋给pgd_k局部变量。

如果产生缺页的线性地址所对应的主内核页全局目录项为空，则函数跳到标号为no_context的代码处（参见前面“处理地址空间以外的错误地址”一节）。否则，函数检查与错误线性地址相对应的主内核页上级目录项和主内核页中间目录项。如果它们中有一个为空，就再次跳转到no_context标记处。否则，就把主目录项复制到进程页中间目录的相应项中（注13）。随后对主页表项重复上述整个操作。

创建和删除进程的地址空间

除了“进程的地址空间”这一节所提到的进程获得一个新的线性区的六种典型的情况之

注12：内核不使用current->mm->pgd导出当前进程的页全局目录地址，因为这种缺页可能在任何时刻都发生，甚至在进程切换期间发生。

注13：你也许还记得在第二章“Linux中的分页”一节中的内容：如果PAE被激活，页上级目录项就不可能为空；如果没有激活PAE，设置页中间目录项的同时也就隐含设置了页上级目录项。

外，首先要指出的是fork()系统调用要求为子进程创建一个完整的新地址空间。相反，当进程结束时，内核撤消它的地址空间。这一节我们讨论Linux如何执行这两种操作。

创建进程的地址空间

我们在第三章的“clone()、fork()及vfork()系统调用”一节中已经提到，当创建一个新的进程时内核调用copy_mm()函数。这个函数通过建立新进程的所有页表和内存描述符来创建进程的地址空间。

通常，每个进程都有自己的地址空间，但是轻量级进程可以通过调用clone()函数（设置了CLONE_VM标志）来创建。这些轻量级进程共享同一地址空间，也就是说，允许它们对同一组页进行寻址。

按照前面讲述的写时复制方法，传统的进程继承父进程的地址空间，只要页是只读的，就依然共享它们。当其中的一个进程试图对某个页进行写时，这个页就被复制一份。一段时间之后，所创建的进程通常获得与父进程不一样的完全属于自己的地址空间。另一方面，轻量级的进程使用父进程的地址空间。Linux实现轻量级进程很简单，即不复制父进程地址空间。创建轻量级的进程比创建普通进程相应要快得多，而且只要父进程和子进程谨慎地协调它们的访问，就可以认为页的共享是有益的。

如果通过clone()系统调用已经创建了新进程，并且flag参数的CLONE_VM标志被设置，则copy_mm()函数把父进程（current）地址空间给子进程（tsk）：

```
if (clone_flags & CLONE_VM) {
    atomic_inc(&current->mm->mm_users);
    spin_unlock_wait(&current->mm->page_table_lock);
    tsk->mm = current->mm;
    tsk->active_mm = current->mm;
    return 0;
}
```

如果其他CPU持有进程页表自旋锁，就调用spin_unlock_wait()函数保证在释放锁之前，缺页处理程序不会结束。实际上，这个自旋锁除了保护页表以外，还必须禁止创建新的轻量级进程，因为它共享current->mm描述符。

如果没有设置CLONE_VM标志，copy_mm()函数就必须创建一个新的地址空间（在进程请求一个地址之前，即使在地址空间内没有分配内存）。这个函数分配一个新的内存描述符，把它的地址存放在新进程描述符tsk的mm字段中，并把current->mm的内容复制到tsk->mm中。然后改变新进程描述符的一些字段：

```
tsk->mm = kmem_cache_alloc(mm_cachep, SLAB_KERNEL);
memcpy(tsk->mm, current->mm, sizeof(*tsk->mm));
```

```
atomic_set(&tsk->mm->mm_users, 1);
atomic_set(&tsk->mm->mm_count, 1);
init_rwsem(&tsk->mm->mmap_sem);
tsk->mm->core_waiters = 0;
tsk->mm->page_table_lock = SPIN_LOCK_UNLOCKED;
tsk->mm->ioctx_list_lock = RW_LOCK_UNLOCKED;
tsk->mm->ioctx_list = NULL;
tsk->mm->default_kioctx = INIT_KIOCTX(tsk->mm->default_kioctx,
                                      *tsk->mm);
tsk->mm->free_area_cache = (TASK_SIZE/3+0xfff)&0xfffff000;
tsk->mm->pgd = pgd_alloc(tsk->mm);
tsk->mm->def_flags = 0;
```

回想一下，pgd_alloc()宏为新进程分配页全局目录。

然后调用依赖于体系结构的init_new_context()函数：对于80x86处理器，该函数检查当前进程是否拥有定制的局部描述符表，如果是，init_new_context()复制一份current的局部描述符表并把它插入tsk的地址空间。

最后，调用dup_mmap()函数既复制父进程的线性区，也复制父进程的页表。dup_mmap()函数把新内存描述符tsk->mm插入到内存描述符的全局链表中，然后，从current->mm->mmap所指向的线性区开始扫描父进程的线性区链表。它复制遇到的每个vm_area_struct线性区描述符，并把复制品插入到子进程的线性区链表和红－黑树中。

在插入一个新的线性区描述符之后，dup_mmap()立即调用copy_page_range()创建必要的页表来映射这个线性区所包含的一组页，并且初始化新页表的表项。尤其是，与私有的、可写的页（VM_SHARED标志关闭，VM_MAYWRITE标志打开）所对应的任一页框都标记为对父子进程是只读的，以便这种页框能用写时复制机制进行处理。

删除进程的地址空间

当进程结束时，内核调用exit_mm()函数释放进程的地址空间：

```
mm_release(tsk, tsk->mm);
if (!(mm = tsk->mm)) /* kernel thread ? */
    return;
down_read(&mm->mmap_sem);
```

mm_release()函数唤醒在tsk->vfork_done补充原语上睡眠的任一进程（参见第五章“补充原语”一节）。典型地，只有当现有进程通过vfork()系统调用被创建时，相应的等待队列才会为非空（参见第三章“clone()、fork()及vfork()系统调用”一节）。

如果正在被终止的进程不是内核线程，exit_mm()函数就必须释放内存描述符和所有相关的数据结构。首先，它检查mm->core_waiters标志是否被置位：如果是，进程就把

内存的所有内容卸载到一个转储文件中。为了避免转储文件的混乱，函数利用mm->core_done和mm->core_startup_done补充原语使共享同一个内存描述符mm的轻量级进程的执行串行化。

接下来，函数递增内存描述符的主使用计数器，重新设置进程描述符的mm字段，并使处理器处于懒惰TLB模式（参见第二章的“处理硬件高速缓存和TLB”一节）：

```
atomic_inc(&mm->mm_count);
spin_lock(tsk->alloc_lock);
tsk->mm = NULL;
up_read(&mm->map_sem);
enter_lazy_tlb(mm, current);
spin_unlock(tsk->alloc_lock);
mmput(mm);
```

最后，调用mmput()函数释放局部描述符表、线性区描述符和页表。不过，因为exit_mm()已经递增了主使用计数器，所以并不释放内存描述符本身。当要把正在被终止的进程从本地CPU撤消时，将由finish_task_switch()函数释放内存描述符（参见第七章“schedule()函数”一节）。

堆的管理

每个Unix进程都拥有一个特殊的线性区，这个线性区就是所谓的堆（*heap*），堆用于满足进程的动态内存请求。内存描述符的start_brk与brk字段分别限定了这个区的开始地址和结束地址。

进程可以使用下面的API来请求和释放动态内存：

`malloc(size)`

请求size个字节的动态内存。如果分配成功，就返回所分配内存单元第一个字节的线性地址。

`calloc(n,size)`

请求含有n个大小为size的元素的一个数组。如果分配成功，就把数组元素初始化为0，并返回第一个元素的线性地址。

`realloc(ptr,size)`

改变由前面的malloc()或calloc()分配的内存区字段的大小。

`free(addr)`

释放由malloc()或calloc()分配的起始地址为addr的线性区。

brk(addr)

直接修改堆的大小。addr参数指定current->mm->brk的新值，返回值是线性区新的结束地址（进程必须检查这个地址和所请求的地址值addr是否一致）。

sbrk(incr)

类似于brk()，不过其中的incr参数指定是增加还是减少以字节为单位的堆大小。

brk()函数和以上列出的函数有所不同，因为它是唯一以系统调用的方式实现的函数，而其他所有的函数都是使用brk()和mmap()系统调用实现的C语言库函数（注14）。

当用户态的进程调用brk()系统调用时，内核执行sys_brk(addr)函数。该函数首先验证addr参数是否位于进程代码所在的线性区。如果是，则立即返回，因为堆不能与进程代码所在的线性区重叠：

```
mm = current->mm;
down_write(&mm->mmap_sem);
if (addr < mm->end_code) {
out:
    up_write(&mm->mmap_sem);
    return mm->brk;
}
```

由于brk()系统调用作用于某一个线性区，它分配和释放完整的页。因此，该函数把addr的值调整为PAGE_SIZE的倍数，然后把调整的结果与内存描述符的brk字段的值进行比较：

```
newbrk = (addr + 0xfff) & 0xfffff000;
oldbrk = (mm->brk + 0xfff) & 0xfffff000;
if (oldbrk == newbrk) {
    mm->brk = addr;
    goto out;
}
```

如果进程请求缩小堆，则sys_brk()调用do_munmap()函数完成这项任务，然后返回：

```
if (addr <= mm->brk) {
    if (!do_munmap(mm, newbrk, oldbrk-newbrk))
        mm->brk = addr;
    goto out;
}
```

如果进程请求扩大堆，则sys_brk()首先检查是否允许进程这样做。如果进程企图分配在其限制范围之外的内存，函数并不多分配内存，只简单地返回mm->brk的原有值：

```
rlim = current->signal->rlim[RLIMIT_DATA].rlim_cur;
```

注14： realloc()库函数也可以使用mremap()系统调用。

```
if (rlim < RLIM_INFINITY && addr - mm->start_data > rlim)
    goto out;
```

然后，函数检查扩大后的堆是否和进程的其他线性区相重叠，如果是，不做任何事情就返回：

```
if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
    goto out;
```

如果一切都顺利，则调用do_brk()函数。如果它返回oldbrk，则分配成功且sys_brk()函数返回addr的值；否则，返回旧的mm->brk值：

```
if (do_brk(oldbrk, newbrk-oldbrk) == oldbrk)
    mm->brk = addr;
goto out;
```

do_brk()函数实际上是仅处理匿名线性区的do_mmap()的简化版。可以认为它的调用等价于：

```
do_mmap(NULL, oldbrk, newbrk-oldbrk, PROT_READ|PROT_WRITE|PROT_EXEC,
        MAP_FIXED|MAP_PRIVATE, 0)
```

当然，do_brk()比do_mmap()稍快，因为前者假定线性区不映射磁盘上的文件，从而避免了检查线性区对象的几个字段。



第十章

系统调用

操作系统为在用户态运行的进程与硬件设备（如CPU、磁盘、打印机等等）进行交互提供了一组接口。在应用程序和硬件之间设置一个额外层具有很多优点。首先，这使得编程更加容易，把用户从学习硬件设备的低级编程特性中解放出来。其次，这极大地提高了系统的安全性，因为内核在试图满足某个请求之前在接口级就可以检查这种请求的正确性。最后，更重要的是这些接口使得程序更具有可移植性，因为只要内核所提供的一组接口相同，那么在任一内核之上就可以正确地编译和执行程序。

Unix系统通过向内核发出系统调用（*system call*）实现了用户态进程和硬件设备之间的大部分接口。本章将详细讨论Linux内核是如何实现这些由用户态进程向内核发出的系统调用的。

POSIX API和系统调用

让我们先强调一下应用编程接口（API）与系统调用之不同。前者只是一个函数定义，说明了如何获得一个给定的服务；而后者是通过软中断向内核态发出一个明确的请求。

Unix系统给程序员提供了很多API的库函数。*libc*的标准C库所定义的一些API引用了封装例程（*wrapper routine*）（其唯一目的就是发布系统调用）。通常情况下，每个系统调用对应一个封装例程，而封装例程定义了应用程序使用的API。

反之则不然，顺便说一句，一个API没必要对应一个特定的系统调用。首先，API可能直接提供用户态的服务（例如一些抽象的数学函数，根本没必要使用系统调用）。其次，一个单独的API函数可能调用几个系统调用。此外，几个API函数可能调用封装了不同

功能的同一系统调用。例如，Linux的*libc*库函数实现了malloc()、calloc()和free()等POSIX API，这几个函数分配和释放所请求的内存，并都利用brk()系统调用来扩大或缩小进程的堆（heap）（参见第九章中的“堆的管理”一节）。

POSIX标准针对API而不针对系统调用。判断一个系统是否与POSIX兼容要看它是否提供了一组合适的应用程序接口，而不管对应的函数是如何实现的。事实上，一些非Uuix系统被认为是与POSIX兼容的，因为它们在用户态的库函数中提供了传统Unix能提供的所有服务。

从编程者的观点看，API和系统调用之间的差别是没有关系的：唯一相关的事情就是函数名、参数类型及返回代码的含义。然而，从内核设计者的观点看，这种差别确实有关系，因为系统调用属于内核，而用户态的库函数不属于内核。

大部分封装例程返回一个整数，其值的含义依赖于相应的系统调用。返回值-1通常表示内核不能满足进程的请求。系统调用处理程序的失败可能是由无效参数引起的，也可能是因为缺乏可用资源，或硬件出了问题等等。在*libc*库中定义的errno变量包含特定的出错码。

每个出错码都定义为一个常量宏（产生一个相应的正整数值）。POSIX标准指定了很多出错码的宏名。在基于80x86系统的Linux中，在一个名为*include/asm-i386/errno.h*的头文件中定义了这些宏。为了使各种Unix系统上的C程序具有可移植性，在标准的C库头文件*/usr/include/errno.h*中也包含了*include/asm-i386/errno.h*头文件。其他的系统有它们自己专门的头文件子目录。

系统调用处理程序及服务例程

当用户态的进程调用一个系统调用时，CPU切换到内核态并开始执行一个内核函数。正如我们在下一节将要看到的那样，在80x86体系结构中，可以用两种不同的方式调用Linux的系统调用。两种方式的最终结果都是跳转到所谓系统调用处理程序（*system call handler*）的汇编语言函数。

因为内核实现了很多不同的系统调用，因此进程必须传递一个名为系统调用号（*system call number*）的参数来识别所需的系统调用，eax寄存器就用作此目的。正如我们将在本章的“参数传递”一节所看到的，当调用一个系统调用时通常还要传递另外的参数。

所有的系统调用都返回一个整数值。这些返回值与封装例程返回值的约定是不同的。在内核中，正数或0表示系统调用成功结束，而负数表示一个出错条件。在后一种情况下，这个值就是存放在errno变量中必须返回给应用程序的负出错码。内核没有设置或使用errno变量，而封装例程从系统调用返回之后设置这个变量。

系统调用处理程序与其他异常处理程序的结构类似，执行下列操作：

- 在内核态栈保存大多数寄存器的内容（这个操作对所有的系统调用都是通用的，并用汇编语言编写）。
- 调用名为系统调用服务例程（*system call service routine*）的相应的C函数来处理系统调用。
- 退出系统调用处理程序：用保存在内核栈中的值加载寄存器，CPU从内核态切换回到用户态（所有的系统调用都要执行这一相同的操作，该操作用汇编语言代码实现）。

xyz()系统调用对应的服务例程的名字通常是sys_*xyz*()。不过也有一些例外。

图10-1显示了调用系统调用的应用程序、相应的封装例程、系统调用处理程序及系统调用服务例程之间的关系。箭头表示函数之间的执行流。占位符"SYSCALL"和"SYSEXIT"是真正的汇编语言指令，它们分别把CPU从用户态切换到内核态和从内核态切换到用户态。

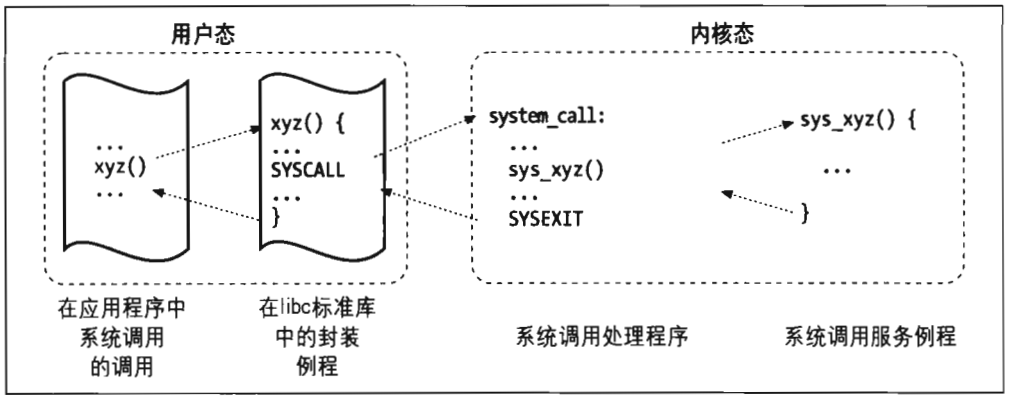


图10-1：调用一个系统调用

为了把系统调用号与相应的服务例程关联起来，内核利用了一个系统调用分派表（*dispatch table*）。这个表存放在sys_call_table数组中，有NR_syscalls个表项（在Linux 2.6.11内核中是289）：第*n*个表项包含系统调用号为*n*的服务例程的地址。

NR_syscalls宏只是对可实现的系统调用最大个数的静态限制，并不表示实际已实现的系统调用个数。实际上，分派表中的任意一个表项也可以包含sys_ni_syscall()函数的地址，这个函数是“未实现”系统调用的服务例程，它仅仅返回出错码-ENOSYS。

进入和退出系统调用

本地应用（注1）可以通过两种不同的方式调用系统调用：

- 执行 int $0x80 汇编语言指令。在Linux内核的老版本中，这是从用户态切换到内核态的唯一方式。
- 执行 sysenter 汇编语言指令。在Intel Pentium II微处理器芯片中引入了这条指令，现在Linux 2.6内核支持这条指令。

同样，内核可以通过两种方式从系统调用退出，从而使CPU切换回到用户态：

- 执行 iret 汇编语言指令。
- 执行 sysexit 汇编语言指令，它和 sysenter 指令同时在Intel Pentium II微处理器中引入。

但是，支持进入内核的两种不同方式并不像看起来那么简单，因为：

- 内核必须既支持只使用 int $0x80 指令的旧函数库，同时支持也可以使用sysenter指令的新函数库。
- 使用 sysenter 指令的标准库必须能处理仅支持 int $0x80 指令的旧内核。
- 内核和标准库必须既能运行在不包含sysenter 指令的旧处理器上，也能运行在包含它的新处理器上。

在本章稍后“通过sysenter指令发出系统调用”一节，我们将看到Linux内核是如何解决这些兼容性问题的。

通过 int $0x80 指令发出系统调用

调用系统调用的传统方法是使用汇编语言指令 int，在第四章“中断和异常的硬件处理”一节曾讨论过这条指令。

向量128（十六进制0x80）对应于内核入口点。在内核初始化期间调用的函数 trap_init()，用下面的方式建立对应于向量128的中断描述符表表项：

注1：就像我们在第二十章“执行域一节”中将要看到的，Linux可以执行其他操作系统所编译的程序。因此内核提供了一种进入系统调用的兼容模式：执行iBCS和Solaris/x86程序的用户态进程可以通过跳转到默认局部描述符表中的适当调用门进入内核。

```
set_system_gate(0x80, &system_call);
```

该调用把下列值存入这个门描述符的相应字段（参见第四章中的“中断门、陷阱门及系统门”一节）：

Segment Selector

内核代码段 __KERNEL_CS 的段选择符。

Offset

指向 system_call() 系统调用处理程序的指针。

Type

置为 15。表示这个异常是一个陷阱，相应的处理程序不禁止可屏蔽中断。

DPL（描述符特权级）

置为 3。这就允许用户态进程调用这个异常处理程序（参见第四章中的“中断和异常的硬件处理”一节）。

因此，当用户态进程发出 int $0x80 指令时，CPU 切换到内核态并开始从地址 system_call 处开始执行指令。

system_call() 函数

system_call() 函数首先把系统调用号和这个异常处理程序可以用到的所有 CPU 寄存器保存到相应的栈中，不包括由控制单元已自动保存的 eflags、cs、eip、ss 和 esp 寄存器（参见第四章中的“中断和异常的硬件处理”一节）。在第四章的“I/O 中断处理”一节中已经讨论的 SAVE_ALL 宏，也在 ds 和 es 中装入内核数据段的段选择符：

```
system_call:
  pushl %eax
  SAVE_ALL
  movl $0xfffffe000, %ebx /* or 0xfffff000 for 4-KB stacks */
  andl %esp, %ebx
```

随后，这个函数在 ebx 中存放当前进程的 thread_info 数据结构的地址，这是通过获得内核栈指针的值并把它取整到 4KB 或 8KB 的倍数而完成的（参见第三章中的“标识一个进程”一节）。

接下来，system_call() 函数检查 thread_info 结构 flag 字段的 TIF_SYSCALL_TRACE 和 TIF_SYSCALL_AUDIT 标志之一是否被设置为 1，也就是检查是否有某一调试程序正在跟踪执行程序对系统调用的调用。如果是这种情况，那么 system_call() 函数两次调用 do_syscall_trace() 函数：一次正好在这个系统调用服务例程执行之前，一次在其之后（稍后对其进行说明）。这个函数停止 current，并因此允许调试进程收集关于 current 的信息。

然后，对用户态进程传递来的系统调用号进行有效性检查。如果这个号大于或等于系统调用分派表中的表项数，系统调用处理程序就终止：

```
    cmpl $NR_syscalls, %eax
    jb nobadsys
    movl $(-ENOSYS), 24(%esp)
    jmp resume_userspace
nobadsys:
```

如果系统调用号无效，该函数就把-ENOSYS值存放在栈中曾保存eax寄存器的单元中（从当前栈顶开始偏移量为24的单元）。然后跳到resume_userspace（见下面）。这样，当进程恢复它在用户态的执行时，会在eax中发现一个负的返回码。

最后，调用与eax中所包含的系统调用号对应的特定服务例程：

```
call *sys_call_table(0, %eax, 4)
```

因为分派表中的每个表项占4个字节，因此首先把系统调用号乘以4，再加上sys_call_table分派表的起始地址，然后从这个地址单元获取指向服务例程的指针，内核就找到了要调用的服务例程。

从系统调用退出

当系统调用服务例程结束时，system_call()函数从eax获得它的返回值，并把这个返回值存放在曾保存用户态eax寄存器值的那个栈单元的位置上：

```
movl %eax, 24(%esp)
```

因此，用户态进程将在eax中找到系统调用的返回码。

然后，system_call()函数关闭本地中断并检查当前进程的thread_info结构中的标志：

```
cli
movl 8(%ebp), %ecx
testw $0xffff, %cx
je restore_all
```

flags字段在thread_info结构中的偏移量为8，掩码0xffff选择与表4-15中列出的所有标志（不包括TIF_POLLING_NRFLAG）相对应的位。如果所有的标志都没有被设置，函数就跳转到restore_all标记处，就像在第四章“从中断和异常返回”一节中所描述的：restore_all标记处的代码恢复保存在内核栈中的寄存器的值，并执行iret汇编语言指令以重新开始执行用户态进程（你可以参考如图4-6所示的流程图）。

只要有任何一种标志被设置，那么就要在返回用户态之前完成一些工作。如果TIF_SYSCALL_TRACE标志被设置，system_call()函数就第二次调用do_syscall_trace()

函数，然后跳转到resume_userspace标记处。否则，如果TIF_SYSCALL_TRACE标志没有被设置，函数就跳转到work_pending标记处。

就像在第四章“从中断和异常返回”一节中所描述的，在resume_userspace和work_pending标记处的代码检查重新调度请求、虚拟8086模式、挂起信号和单步执行，最终跳转到restore_all标记处以恢复用户态进程的执行。

通过sysenter指令发出系统调用

汇编语言指令int由于要执行几个一致性和安全性检查，所以速度较慢（这条指令的详细说明见第四章“中断和异常的硬件处理”一节）。

在Intel文档中被称为“快速系统调用”的sysenter指令，提供了一种从用户态到内核态的快速切换方法。

sysenter指令

汇编语言指令sysenter使用三种特殊的寄存器，它们必须装入下述信息（注2）：

SYSENTER_CS_MSR
: 内核代码段的段选择符

SYSENTER_EIP_MSR
: 内核入口点的线性地址

SYSENTER_ESP_MSR
: 内核堆栈指针

执行sysenter指令时，CPU控制单元：

1. 把SYSENTER_CS_MSR的内容拷贝到cs。
2. 把SYSENTER_EIP_MSR的内容拷贝到eip。
3. 把SYSENTER_ESP_MSR的内容拷贝到esp。
4. 把SYSENTER_CS_MSR加8的值装入ss。

因此，CPU切换到内核态并开始执行内核入口点的第一条指令。就像我们在第二章

注2： “MSR”是“Model-Specific Register”的缩写，表示仅在当前一些80 × 86微处理器中存在的某个寄存器。

"Linux GDT"一节中所见到的，内核堆栈段与内核数据段是一致的，而且在全局描述符表中，其描述符紧跟在内核代码段的描述符之后；所以，第4步把正确的段选择符装入了ss寄存器。

在内核初始化期间，一旦系统中的每个CPU执行函数enable_sep_cpu()，三个特定于模型的寄存器就由该函数初始化了。enable_sep_cpu()函数执行以下步骤：

1. 把内核代码（__KERNEL_CS）的段选择符写入SYSENTER_CS_MSR寄存器。
2. 把下面要说明的函数sysenter_entry()的线性地址写入SYSENTER_CS_EIP寄存器。
3. 计算本地TSS末端的线性地址，并把这个值写入SYSENTER_CS_ESP寄存器（注3）。

对SYSENTER_CS_ESP寄存器的设置有必要进行一些说明。系统调用开始的时候，内核堆栈是空的，因此esp寄存器应该指向4KB或8KB内存区域的末端，该内存区域包括内核堆栈和当前进程的描述符（见图3-2）。因为用户态的封装例程不知道这个内存区域的地址，所以它不能正确设置这个寄存器。另一方面，由于必须在切换到内核态之前设置该寄存器的值，因此，内核初始化这个寄存器以便为本地CPU的任务状态段编址。就像我们在__switch_to()函数的第3步所描述的（见第三章"执行进程切换"一节），在每次进程切换时，内核把当前进程的内核栈指针保存到本地TSS的esp0字段。这样，系统调用处理程序读esp寄存器，计算本地TSS的esp0字段的地址，然后把正确的内核堆栈指针装入esp寄存器。

vsyscall页

只要CPU和Linux内核都支持sysenter指令，标准库*libc*中的封装函数就可以使用它。

这个兼容性问题需要非常复杂的解决方案。本质上，在初始化阶段，sysenter_setup()函数建立一个称为*vsyscall*页的页框，其中包括一个小的EFL共享对象（也就是）一个很小的EFL动态链接库）。当进程发出execve()系统调用而开始执行一个EFL程序时，vsyscall页中的代码就会自动地被链接到进程的地址空间（参见第二十章"exec函数"一节）。vsyscall页中的代码使用最有用的指令发出系统调用。

函数sysenter_setup()为vsyscall页分配一个新页框，并把它的物理地址与FIX_VSYSCALL固定映射的线性地址相关联（参见第二章"固定映射的线性地址"一节）。然后，函数sysenter_setup()把预先定义好的一个或两个EFL共享对象拷贝到该页中：

注3：把本地TSS地址编码写入SYSENTER_ESP_MSR寄存器，是因为该寄存器应该指向一个实际的栈，该栈向低地址方向增长。实际上可以用任何值初始化SYSENTER_ESP_MSR寄存器，只要能够从这个值获得本地TSS的地址即可。

- 如果CPU不支持sysenter，sysenter_setup()函数建立一个包括下列代码的vsyscall页：

```
__kernel_vsyscall:
    int $0x80
    ret
```

- 否则，如果CPU的确支持sysenter，sysenter_setup()函数建立一个包括下列代码的vsyscall页：

```
__kernel_vsyscall:
  pushl %ecx
  pushl %edx
  pushl %ebp
  movl %esp, %ebp
  sysenter
```

当标准库中的封装例程必须调用系统调用时，都调用__kernel_vsyscall()函数，不管它的实现代码是什么。

最后一个兼容性问题是由于老版本的Linux内核不支持sysenter指令，在这种情况下，内核当然不建立vsyscall页，而且函数__kernel_vsyscall()不会被链接到用户态进程的地址空间。当新近的标准库识别出这种状况后，就简单地执行int $0x80指令来调用系统调用。

进入系统调用

当用sysenter指令发出系统调用时，依次执行下述步骤：

1. 标准库中的封装例程把系统调用号装入eax寄存器，并调用__kernel_vsyscall()函数。
2. 函数__kernel_vsyscall()把ebp、edx和ecx的内容保存到用户态堆栈中（系统调用处理程序将使用这些寄存器），把用户栈指针拷贝到ebp中，然后执行sysenter指令。
3. CPU从用户态切换到内核态，内核开始执行sysenter_entry()函数（由SYSENTER_EIP_MSR寄存器指向）。
4. sysenter_entry()汇编语言函数执行下述步骤：
   a. 建立内核堆栈指针：

```
movl -508(%esp), %esp
```

   开始时，esp寄存器指向本地TSS的第一个位置，本地TSS的大小为512字节。因此，sysenter指令把本地TSS中偏移量为4处的字段的内容（就是esp0

字段的内容）装入esp。就像前面我们已经说明的，esp0字段总是存放当前进程的内核堆栈指针。

b. 打开本地中断：

```
sti
```

c. 把用户数据段的段选择符、当前用户栈指针、 eflags寄存器、用户代码段的段选择符以及从系统调用退出时要执行的指令的地址保存到内核堆栈中：

```
pushl $(__USER_DS)
pushl %ebp
pushfl
pushl $(__USER_CS)
pushl $SYSENTER_RETURN
```

这些指令仿效汇编语言指令int所执行的一些操作（参见第四章“中断和异常的硬件处理”一节中对int描述的第5c步和第7步操作）。

d. 把原来由封装例程传递的寄存器的值恢复到ebp中：

```
movl (%ebp), %ebp
```

上面这条指令完成恢复的工作，因为__kernel_vsyscall()函数把ebp的原始值存入用户态堆栈中，并在随后把用户堆栈指针的当前值装入ebp中。

e. 通过执行一系列指令调用系统调用处理程序，这些指令与前面“通过int $0x80指令发出系统调用”一节所描述的在system_call标记处开始的指令是一样的。

退出系统调用

当系统调用服务例程结束时，sysenter_entry()函数本质上执行与system_call()函数相同的操作（见上一节）。首先，它从eax获得系统调用服务例程的返回码，并将返回码存入内核栈中保存用户态eax寄存器值的位置。然后，函数禁止本地中断，并检查current的thread_info结构中的标志。

如果有任何标志被设置，那么在返回到用户态之前还需要完成一些工作。为了避免代码复制（像在system_call()函数中所做的那样正确处理这种情况），函数跳转到resume_userspace或work_pending标记处（参见第四章图4-6所示的流程图）。最后，汇编语言指令iret从内核堆栈中去取5个参数(这些参数是在sysenter_entry()函数的第4c步被保存到内核堆栈中的，这样，CPU切换到用户态并开始执行SYSENTER_RETURN标记处的代码（见下面）。

如果sysenter_entry()函数确定标志都被清0，它就快速返回到用户态：

```
movl 40(%esp), %edx
movl 52(%esp), %ecx
xorl %ebp, %ebp
sti
sysexit
```

把在上一节由sysenter_entry()函数在第4c步保存的一对堆栈值加载到edx和ecx寄存器中：edx获得SYSENTER_RETURN标记处的地址，而ecx获得当前用户数据栈的指针。

sysexit指令

sysexit是与sysenter配对的汇编语言指令：它允许从内核态快速切换到用户态。执行这条指令时，CPU控制单元执行下述步骤：

1. 把SYSENTER_CS_MSR寄存器中的值加16所得到的结果加载到cs寄存器。
2. 把edx寄存器的内容拷贝到eip寄存器。
3. 把SYSENTER_CS_MSR寄存器中的值加24所得到的结果加载到ss寄存器。
4. 把ecx寄存器的内容拷贝到esp寄存器。

因为SYSENTER_CS_MSR寄存器加载的是内核代码的段选择符，cs寄存器加载的是用户代码的段选择符，而ss寄存器加载的是用户数据段的段选择符（参见第二章“LinuxLDT”一节）。

结果，CPU从内核态切换到用户态，并开始执行其地址存放在edx中的那条指令。

SYSENTER_RETURN的代码

SYSENTER_RETURN标记处的代码存放在vsyscall页中，当通过sysenter进入的系统调用被iret或sysexit指令终止时，该页框中的代码被执行。

该代码片段恢复保存在用户态堆栈中的ebp、edx和ecx寄存器的原始内容，并把控制权返回给标准库中的封装例程：

```
SYSENTER_RETURN:
  popl %ebp
  popl %edx
  popl %ecx
  ret
```

参数传递

与普通函数类似，系统调用通常也需要输入/输出参数，这些参数可能是实际的值（例如数值），也可能是用户态进程地址空间的变量，甚至是指向用户态函数的指针的数据结构地址（参见第十一章“与信号处理相关的系统调用”一节）。

因为system_call()和sysenter_entry()函数是Linux中所有系统调用的公共入口点，因此每个系统调用至少有一个参数，即通过eax寄存器传递来的系统调用号。例如，如果一个应用程序调用fork()封装例程，那么在执行int $0x80或sysenter汇编指令之前就把eax寄存器置为2（即__NR_fork）。因为这个寄存器的设置是由libc库中的封装例程进行的，因此程序员通常并不用关心系统调用号。

fork()系统调用并不需要其他的参数。不过，很多系统调用确实需要由应用程序明确地传递另外的参数。例如，mmap()系统调用可能需要多达6个额外参数（除了系统调用号以外）。

普通C函数的参数传递是通过把参数值写入活动的程序栈（用户态栈或者内核态栈）实现的。因为系统调用是一种横跨用户和内核两大陆地的特殊函数，所以既不能使用用户态栈也不能使用内核态栈。更确切地说，在发出系统调用之前，系统调用的参数被写入CPU寄存器，然后在调用系统调用服务例程之前，内核再把存放在CPU中的参数拷贝到内核态堆栈中，这是因为系统调用服务例程是普通的C函数。

为什么内核不直接把参数从用户态的栈拷贝到内核态的栈呢？首先，同时操作两个栈是比较复杂的。其次，寄存器的使用使得系统调用处理程序的结构与其他异常处理程序的结构类似。

然而，为了用寄存器传递参数，必须满足两个条件：

- 每个参数的长度不能超过寄存器的长度，即32位（注4）。
- 参数的个数不能超过6个（除了eax中传递的系统调用号），因为80x86处理器的寄存器的数量是有限的。

第一个条件总能成立，因为根据POSIX标准，不能存放在32位寄存器中的长参数必须通过指定它们的地址来传递。一个典型的例子就是settimeofday()系统调用，它必须读一个64位的结构。

注4：与往常一样，这里指的是80x86处理器的32位体系结构。本节的讨论并不适用于64位体系结构。

然而，确实存在多于6个参数的系统调用。在这样的情况下，用一个单独的寄存器指向进程地址空间中这些参数值所在的一个内存区。当然，编程者不用关心这个工作区。正如任何C调用一样，当调用封装例程时，参数被自动地保存在栈中。封装例程将找到合适的方式把参数传递给内核。

用于存放系统调用号和系统调用参数的寄存器是（以字母递增的顺序）：eax（存放系统调用号）、ebx、ecx、edx、esi、edi以及ebp。正如以前看到的一样，system_call()和sysenter_entry()使用SAVE_ALL宏把这些寄存器的值保存在内核态堆栈中。因此，当系统调用服务例程转到内核态堆栈时，就会找到system_call()或sysenter_entry()的返回地址，紧接着是存放在ebx中的参数（即系统调用的第一个参数），存放在ecx中的参数等等（参见第四章中的“为中断处理程序保存寄存器的值”一节）。这种栈结构与普通函数调用的栈结构完全相同，因此，系统调用服务例程很容易通过使用C语言结构来引用它的参数。

让我们来看一个例子。处理write()系统调用的sys_write()服务例程的声明如下：

```
int sys_write (unsigned int fd, const char * buf, unsigned int count)
```

C编译器产生一个汇编语言函数，该函数期望在栈的顶部找到fd、buf和count参数，而这些参数位于返回地址（就是用来分别存放ebx、ecx和edx寄存器的那些位置）的下面。

在少数情况下，即使系统调用不使用任何参数，相应的服务例程也需要知道在发出系统调用之前CPU寄存器的内容。例如，实现了fork()的do_fork()函数需要知道有关寄存器的值，以便在子进程的thread字段中复制它们（参见第三章的“thread字段”一节）。在这些情况下，类型为pt_regs的一个单独参数允许服务例程访问由SAVE_ALL宏保存在内核态堆栈中的值（参见第四章中的“do_IRQ()函数”一节）：

```
int sys_fork (struct pt_regs regs)
```

服务例程的返回值必须写入eax寄存器中。这是在执行“return n;”指令时由C编译程序自动完成的。

验证参数

在内核打算满足用户的请求之前，必须仔细地检查所有的系统调用参数。检查的类型既依赖于系统调用，也依赖于特定的参数。让我们再回到前面引入的write()系统调用：fd参数应该是描述一个特定文件的文件描述符，因此，sys_write()必须检查fd是否确实是以前已打开文件的一个文件描述符，是否允许进程向这个文件中写数据（参见第一章中的“文件操作的系统调用”一节）。如果这些条件中有一个不成立，那么这个处理程序必须返回一个负数，在这种情况下的出错码为-EBADF。

然而，有一种检查对所有的系统调用都是通用的。只要一个参数指定的是地址，那么内核必须检查它是否在这个进程的地址空间之内。有两种可能的方式来执行这种检查：

- 验证这个线性地址是否属于进程的地址空间，如果是，这个线性地址所在的线性区就具有正确的访问权限。
- 仅仅验证这个线性地址是否小于PAGE_OFFSET(即没有落在留给内核的线性地址区间内)。

早期的Linux内核执行第一种检查。但是这是非常费时的，因为它必须对系统调用中包含的每个地址参数都进行检查；此外，这通常没有什么意义，因为有错误的程序并不是很普遍。

因此，从Linux 2.2内核开始执行第二种检查。这是一种更高效的检查，因为不需要对进程的线性区描述符进行任何扫描。很显然，这是一种非常粗略的检查，验证线性地址小于PAGE_OFFSET是判断它的有效性的必要条件而不是充分条件。但是，因为其他的错误可以在随后捕获到，所以内核使用这种有限的检查没有任何风险。

因此，接着采用的方法是将真正的检查尽可能向后推迟，也就是说，推迟到分页单元将线性地址转换为物理地址时。我们将在本章稍后的“动态地址检查：修正代码”一节中讨论，缺页异常处理程序如何成功地检测到由用户态进程以参数传递的这些地址在内核态是无效的。

在这里，你可能想知道究竟为什么要进行这种粗略检查。事实上，这种粗略的检查是至关重要的，它确保了进程地址空间和内核地址空间都不被非法访问。我们在第二章中已经看到，RAM的映射是从PAGE_OFFSET开始的。这就意味着内核例程能对内存中现有的所有页进行寻址。因此，如果不进行这种粗略检查，用户态进程就可能把属于内核地址空间的一个地址作为参数来传递，然后还能对内存中现有的任何页进行读写而不引起缺页异常。

对系统调用所传递地址的检查是通过access_ok()宏实现的，它有两个分别为addr和size的参数。该宏检查addr到addr+size-1之间的地址区间，本质上等价于下面的C函数：

```
int access_ok(const void * addr, unsigned long size)
{
    unsigned long a = (unsigned long) addr;
    if (a + size < a ||
        a + size > current_thread_info()->addr_limit.seg)
        return 0;
    return 1;
}
```

该函数首先验证addr+size(要检查的最高地址)是否大于$2^{32}-1$；这是因为GNU C编译器（gcc）用32位数表示无符号长整型数和指针，这就等价于对溢出条件进行检查。该函数还检查addr+size是否超过crurrent的thread_info结构的addr_limit.seg字段中存放的值。通常情况下，普通进程这个字段的值是PAGE_OFFSET，内核线程是0xffffffff。可以通过get_fs和set_fs宏动态地改变addr_limit.seg字段的值；这就允许内核绕过由access_ok()执行的安全性检查，调用系统调用的服务例程，并直接把内核数据段的地址传递给它们。

函数verify_area()执行与access_ok()宏类似的检查，虽然它被认为是陈旧过时的，但在源代码中仍然被广泛使用。

访问进程地址空间

系统调用服务例程需要非常频繁地读写进程地址空间的数据。Linux包含的一组宏使这种访问更加容易。我们将描述其中的两个名为get_user()和put_user()的宏。第一个宏用来从一个地址读取1、2或4个连续字节，而第二个宏用来把这几种大小的内容写入一个地址中。

这两个函数都接收两个参数，一个要传送的值x和一个变量ptr。第二变量还决定有多少个字节要传送。因此，在get_user (x, ptr) 中，由ptr指向的变量大小使该函数展开为__get_user_1()、__get_user_2()或__get_user_4()汇编语言函数。让我们看一下其中一个，比如，__get_user_2()：

```
__get_user_2:
    addl $1, %eax
    jc bad_get_user
    movl $0xfffe000, %edx /* or 0xfffff000 for 4-KB stacks */
    andl %esp, %edx
    cmpl 24(%edx), %eax
    jae bad_get_user
2:  movzwl -1(%eax), %edx
    xorl %eax, %eax
    ret
bad_get_user:
    xorl %edx, %edx
    movl $-EFAULT, %eax
    ret
```

eax寄存器包含要读取的第一个字节的地址ptr。前6个指令所执行的检查事实上与access_ok()宏相同，即确保要读取的两个字节的地址小于4GB并小于current进程的addr_limit.seg字段(这个字段位于current的thread_info结构中偏移量为24处，出现在cmpl指令的第一个操作数中)。

如果这个地址有效，函数就执行 movzwl 指令，把要读的数据存到 edx 寄存器的两个低字节，而把两个高字节置为0，然后在 eax 中设置返回码0并终止。如果这个地址无效，函数清 edx，将 eax 置为 -EFAULT 并终止。

put_user(x, ptr)宏类似于前边讨论的 get_user，但它把值 x 写入以地址 ptr 为起始地址的进程地址空间。根据 x 的大小，它使用 __put_user_asm()宏（大小为 1、2 或 4 字节），或 __put_user_u64()宏（大小为 8 字节）。这两个宏如果成功地写入了值，那它们在 eax 寄存器中返回 0，否则返回 -EFAULT。

在表10-1中列出了内核态下用来访问进程地址空间的另外几个函数或宏。注意，许多函数或宏的名字前缀有两个下划线（__）。首部没有下划线的函数或宏要用额外的时间对所请求的线性地址区间进行有效性检查，而有下划线的则会跳过检查。当内核必须重复访问进程地址空间的同一块线性区时，比较高效的办法是开始时只对该地址检查一次，以后就不用再对该进程区进行检查了。

表 10-1：访问进程地址空间的函数和宏

函数	操作
get_user __get_user	从用户空间读一个整数（1、2 或 4 字节）
put_user __put_user	给用户空间写一个整数（1、2 或 4 字节）
copy_from_user __copy_from_user	从用户空间复制任意大小的块
copy_to_user __copy_to_user	把任意大小的块复制到用户空间
strncpy_from_user__strncpy_from_user	从用户空间复制一个以 null 结束的字符串
strlen_user strnlen_user	返回用户空间以 null 结束的字符串的长度
clear_user __clear_user	用 0 填充用户空间的一个内存区域

动态地址检查:修正代码

正如前面所看到的，access_ok()宏对系统调用以参数传递来的线性地址的有效性只进行粗略检查。该检查只保证用户态进程不会试图侵扰内核地址空间。但是，由参数传递的线性地址依然可能不属于进程地址空间。在这种情况下，当内核试图使用任何这种错误地址时，将会发生缺页异常。

在描述内核如何检测这种错误之前，我们先说明一下在内核态引起缺页异常的四种情况。这些情况必须由缺页异常处理程序来区分，因为不同情况采取的操作很不相同：

1. 内核试图访问属于进程地址空间的页，但是，或者是相应的页框不存在，或者是内核试图去写一个只读页。在这些情况下，处理程序必须分配和初始化一个新的页框（参见第九章“请求调页”和“写时复制”两节）。

2. 内核寻址到属于其地址空间的页，但是相应的页表项还没有被初始化（参见第九章“处理非连续内存区访问”一节）。在这种情况下，内核必须在当前进程页表中适当地建立一些表项。
3. 某一内核函数包含编程错误，当这个函数运行时就引起异常；或者，可能由于瞬时的硬件错误引起异常。当这种情况发生时，处理程序必须执行一个内核漏洞（参见第九章的“处理地址空间以外的错误地址”一节）。
4. 本章所讨论的一种情况：系统调用服务例程试图读写一个内存区，而该内存区的地址是通过系统调用参数传递来的，但却不属于进程的地址空间。

通过确定错误的线性地址是否属于进程所拥有的线性地址区间，缺页处理程序可以很容易地识别第一种情况。通过检查相应的主内核页表是否包含一个映射该地址的非空项也可以检测第二种情况。下面让我们解释一下缺页处理程序如何区分另外两种情况。

异常表

决定缺页的来源关键在于内核使用有限的范围访问进程的地址空间。只有前一节描述的少数函数和宏用来访问进程的地址空间；因此，如果异常是由一个无效的参数引起的，那么，引起异常的指令一定包含在其中的一个函数中或由展开的宏产生。这些对用户空间寻址的指令数量是非常少的。

因此，把访问进程地址空间的每条内核指令的地址放到一个叫异常表（*exception table*）的结构中并不用费太多功夫。如果我们成功地做到这点，其他的事情就很容易了。当在内核态发生缺页异常时，`do_page_fault()`处理程序检查异常表：如果表中包含产生异常的指令地址，那么这个错误就是由非法的系统调用参数引起的，否则，就是由某一更严重的bug引起的。

Linux定义了几个异常表。主要的异常表在建立内核程序映像时由C编译器自动生成。它存放在内核代码段的`__ex_table`节，其起始与终止地址由C编译器产生的两个符号`__start__ex_table`和`__stop__ex_table`来标识。

此外，每个动态装载的内核模块（参看附录二）都包含有自己的局部异常表。这个表是在建立模块映像时由C编译器自动产生的，当把模块插入到运行中的内核时把这个表装入内存。

每一个异常表的表项是一个`exception_table_entry`结构，它有两个字段：

`insn`

访问进程地址空间的指令的线性地址。

fixup

当存放在insn单元中的指令所触发的缺页异常发生时，fixup就是要调用的汇编语言代码的地址。

修正代码由几条汇编指令组成，用以解决由缺页异常所引起的问题。在后面我们将会看到，修正通常由插入的一个指令序列组成，这个指令序列强制服务例程返回一个出错码给用户态进程。这些指令通常在访问进程地址空间的同一函数或宏中定义；由C编译器把它们放置在内核代码段的一个叫作 .fixup的独立部分。

search_exception_tables()函数用来在所有异常表中查找一个指定地址：若这个地址在某一个表中，则返回指向相应exception_table_entry结构的指针；否则，返回NULL。因此，缺页处理程序do_page_fault()执行下列语句：

```
if ((fixup = search_exception_tables(regs->eip))) {
    regs->eip = fixup->fixup;
    return 1;
}
```

regs->eip字段包含异常发生时保存到内核态栈eip寄存器中的值。如果eip寄存器（指令指针）中的这个值在某个异常表中，do_page_fault()就把所保存的值替换为search_exception_tables()的返回地址。然后缺页处理程序终止，被中断的程序以修正代码的执行而恢复运行。

生成异常表和修正代码

GNU汇编程序（Assembler）伪指令 .section允许程序员指定可执行文件的哪部分包含紧接着要执行的代码。我们将在第二十章中看到，可执行文件包括一个代码段，这个代码段可能又依次被划分为节。因此，下面的汇编指令在异常表中加入一个表项；“a”属性指定必须把这一节与内核映像的剩余部分一块加载到内存中。

```
.section __ex_table, "a"
    .long faulty_instruction_address, fixup_code_address
.previous
```

.previous 伪指令强制汇编程序把紧接着的代码插入到遇到上一个.section伪指令时激活的节。

让我们再看一下前面论及过的__get_user_1()、__get_user_2()和__get_user_4()函数。访问进程地址空间的指令用1、2和3标记。

```
__get_user_1:
    [...]
```

```
1:  movzbl (%eax), %edx
    [...]
__get_user_2:
    [...]
2:  movzwl -1(%eax), %edx
    [...]
__get_user_4:
    [...]
3:  movl -3(%eax), %edx
    [...]
bad_get_user:
    xorl %edx, %edx
    movl $-EFAULT, %eax
    ret
.section __ex_table,"a"
    .long 1b, bad_get_user
    .long 2b, bad_get_user
    .long 3b, bad_get_user
.previous
```

每个异常表项由两个标号组成。第一个是一个数字标号，其前缀b表示标号是“向后的”；换句话说，标号出现在程序的前一行。修正代码对这三个函数是公用的，且被标记为bad_get_user。如果缺页异常是由标号1、2或3处的指令产生的，那么修正代码就执行。在bad_get_user处的修正代码给发出系统调用的进程只简单地返回一个出错码-EFAULT。

其他作用于用户态地址空间的内核函数也使用修正代码技术。再看下一个例子，strlen_user(string)宏。该宏或者返回系统调用中作为参数传递的以null结束的字符串string的长度，或在出错时返回0。这个宏本质上产生以下的汇编指令：

```
    movl $0, %eax
    movl $0x7fffffff, %ecx
    movl %ecx, %ebx
    movl string, %edi
0:  repne; scasb
    subl %ecx, %ebx
    movl %ebx, %eax
1:
.section .fixup,"ax"
2:  xorl %eax, %eax
    jmp 1b
.previous
.section _  _ex_table,"a"
    .long 0b, 2b
.previous
```

ecx和ebx寄存器的初始值设为0x7fffffff，表示用户态地址空间字符串的最大长度。repne; scasb汇编指令循环扫描由edi指向的字符串，在eax中查找值为0的字符（字

符串的结束标志\0字符)。因为每一次循环scasb都将ecx减1，所以eax中最后存放的是在字符串中所扫描过的字节总数（也就是字符串的长度)。

这个宏的修正代码被插入.fixup节。"ax"属性指定这一节必须加载到内存并包含可执行代码。如果缺页异常是由标号为0的指令引起，就执行修正代码，它只简单地把eax置为0——因此强制该宏返回一个出错码0而不是字符串长度——然后跳转到标号1，即宏之后的相应指令。

第二个.section指令在__ex_table中增加一个表项，内容包括repne，scasb指令的地址和相应的修正代码的地址。

内核封装例程

尽管系统调用主要由用户态进程使用，但也可以被内核线程调用，内核线程不能使用库函数。为了简化相应封装例程的声明，Linux定义了7个从_syscall0到_syscall6的一组宏。

每个宏名字中的数字0～6对应着系统调用所用的参数个数(系统调用号除外)。也可以用这些宏来声明没有包含在libc标准库中的封装例程（例如，因为Linux系统调用还未受到库的支持)。然而，不能用这些宏来为超过6个参数(系统调用号除外)的系统调用或产生非标准返回值的系统调用定义封装例程。

每个宏严格地需要2+2 × n 个参数，n 是系统调用的参数个数。前两个参数指明系统调用的返回值类型和名字；每一对附加参数指明相应的系统调用参数的类型和名字。因此，以fork()系统调用为例，其封装例程可以通过如下语句产生：

```
_syscall0(int,fork)
```

而write()系统调用的封装例程可以通过如下语句产生：

```
_syscall3(int,write,int,fd,const char *,buf,unsigned int,count)
```

在后一种情况下，可以把这个宏展开成如下的代码:

```
int write(int fd,const char * buf,unsigned int count)
{
    long __res;
    asm("int $0x80"
        : "=a" (__res)
        : "0" (__NR_write), "b" ((long)fd),
          "c" ((long)buf), "d" ((long)count));
    if ((unsigned long)__res >= (unsigned long)-129) {
        errno = -__res;
        __res = -1;
    }
```

```
    return (int) __res;
}
```

__NR_write宏来自_syscall3的第二个参数，它可以展开成write()的系统调用号。当编译前面的函数时，生成下面的汇编代码：

```
write:
     pushl %ebx              ;将ebx推入堆栈
     movl 8(%esp), %ebx      ; 将第一个参数放入ebx
     movl 12(%esp), %ecx     ; 将第二个参数放入ecx
     movl 16(%esp), %edx     ; 将第三个参数放入edx
     movl $4, %eax           ;将__NR_write 放入eax
     int $0x80               ; 调用系统调用
     cmpl $-125, %eax        ; 检测返回码
     jbe .L1                 ; 如无错则跳转
     negl %eax               ; 求eax的补码
     movl %eax, errno        ; 将结果放入errno
     movl $-1, %eax          ; 将eax置为-1
.L1: popl %ebx               ; 从堆栈弹出ebx
     ret                     ; 返回调用程序
```

注意write()函数的参数是如何在执行int $0x80指令前被装入到CPU寄存器中的。如果eax中的返回值在-1～-129之间，则必须被解释为出错码（内核假定在*include/generic /errno.h*中定义的最大出错码为129）。如果是这种情况，封装例程就在errno中存放-eax的值并返回值-1；否则，返回eax中的值。

第十一章



信号

信号在最早的Unix系统中即被引入，用于在用户态进程间通信。内核也用信号通知进程系统所发生的事件。信号已有30多年的历史，但只有很小的变化。

本章的第一部分详细考察Linux内核如何处理信号，然后，我们讨论几个允许进程交换信号的系统调用。

信号的作用

信号(signal)是很短的消息，可以被发送到一个进程或一组进程。发送给进程的唯一信息通常是一个数，以此来标识信号。在标准信号中，对参数、消息或者其他相随的信息没有给予关注。

名字前缀为SIG的一组宏用来标识信号。在前几章中，我们已经涉及到几个信号。例如，在第三章的“clone()、fork()及vfork()系统调用”一节中已提及到的SIGCHLD宏。在Linux中，这个宏扩展为值17，当某一子进程停止或终止时，SIGCHLD宏产生发送给父进程的信号标识符。SIGSEGV宏扩展为值11，在第九章的“缺页异常处理程序”一节中已提及到：当一个进程引用无效的内存时，SIGSEGV宏产生发送给进程的信号标识符。

使用信号的两个主要目的是：

- 让进程知道已经发生了一个特定的事件。
- 强迫进程执行它自己代码中的信号处理程序。

当然，这两个目的不是互斥的，因为进程经常通过执行一个特定的例程来对某一事件做出反应。

表11-1列出了基于80x86的Linux 2.6所处理的前31个信号（像SIGCHLD或SIGSTOP这样的一些信号是与体系结构相关的；此外，像SIGSTKFLT这样的一些信号只为特定的体系结构而定义）。缺省操作的含义将在下一节描述。

表 11-1：Linux/i386 中的前 31 个信号

编号	信号名称	缺省操作	解释	POSIX
1	SIGHUP	Terminate	挂起控制终端或进程	是
2	SIGINT	Terminate	来自键盘的中断	是
3	SIGQUIT	Dump	从键盘退出	是
4	SIGILL	Dump	非法指令	是
5	SIGTRAP	Dump	跟踪的断点	否
6	SIGABRT	Dump	异常结束	是
6	SIGIOT	Dump	等价于 SIGABRT	否
7	SIGBUS	Dump	总线错误	否
8	SIGFPE	Dump	浮点异常	是
9	SIGKILL	Terminate	强迫进程终止	是
10	SIGUSR1	Terminate	对进程可用	是
11	SIGSEGV	Dump	无效的内存引用	是
12	SIGUSR2	Terminate	对进程可用	是
13	SIGPIPE	Terminate	向无读者的管道写	是
14	SIGALRM	Terminate	实时定时器时钟	是
15	SIGTERM	Terminate	进程终止	是
16	SIGSTKFLT	Terminate	协处理器栈错误	否
17	SIGCHLD	Ignore	子进程停止、结束或在被跟踪时获得信号	是
18	SIGCONT	Continue	如果已停止则恢复执行	是
19	SIGSTOP	Stop	停止进程执行	是
20	SIGTSTP	Stop	从 tty 发出停止进程	是
21	SIGTTIN	Stop	后台进程请求输入	是
22	SIGTTOU	Stop	后台进程请求输出	是
23	SIGURG	Ignore	套接字上的紧急条件	否

表 11-1：Linux/i386 中的前 31 个信号（续）

编号	信号名称	缺省操作	解释	POSIX
24	SIGXCPU	Dump	超过 CPU 时限	否
25	SIGXFSZ	Dump	超过文件大小的限制	否
26	SIGVTALRM	Terminate	虚拟定时器时钟	否
27	SIGPROF	Terminate	概况定时器时钟	否
28	SIGWINCH	Ignore	窗口调整大小	否
29	SIGIO	Terminate	I/O 现在可能发生	否
29	SIGPOLL	Terminate	等价于 SIGIO	否
30	SIGPWR	Terminate	电源供给失效	否
31	SIGSYS	Dump	坏的系统调用	否
31	SIGUNUSED	Dump	等价于 SIGSYS	否

除了在这张表中描述的常规信号（*regular signal*）外，POSIX 标准还引入了一类新的信号，叫做实时信号（*real-time signal*）；在 Linux 中它们的编码范围为 32～64。它们与常规信号有很大的不同，因为它们必须排队以便发送的多个信号能被接收到。另一方面，同种类型的常规信号并不排队：如果一个常规信号被连续发送多次，那么，只有其中的一个发送到接收进程。尽管 Linux 内核并不使用实时信号，它还是通过几个特定的系统调用完全实现了 POSIX 标准。

许多系统调用允许程序员发送信号，并决定他们的进程如何响应所接收的信号。表 11-2 简洁地描述了这些系统调用，更详细的内容将在后面“与信号处理相关的系统调用”一节中描述。

表 11-2：与信号相关的最重要的系统调用

系统调用	说明
kill()	向线程组发送一个信号
tkill()	向进程发送一个信号
tgkill()	向一个特定线程组中的进程发送信号
sigaction()	改变与信号相关的操作
signal()	类似于 sigaction()
sigpending()	检查是否有挂起的信号
sigprocmask()	修改阻塞信号的集合
sigsuspend()	等待一个信号

表 11-2：与信号相关的最重要的系统调用（续）

系统调用	说明
rt_sigaction()	改变与实时信号相关的操作
rt_sigpending()	检查是否挂起实时信号
rt_sigprocmask()	修改阻塞的实时信号的集合
rt_sigqueueinfo()	向线程组发送一个实时信号
rt_sigsuspend()	等待一个实时信号
rt_sigtimedwait()	类似于 rt_sigsuspend()

信号的一个重要特点是它们可以随时被发送给状态经常不可预知的进程。发送给非运行进程的信号必须由内核保存，直到进程恢复执行。阻塞一个信号（后面描述）要求信号的传递拖延，直到随后解除阻塞，这使得信号产生一段时间之后才能对其传递这一问题变得更加严重。

因此，内核区分信号传递的两个不同阶段：

信号产生

内核更新目标进程的数据结构以表示一个新信号已被发送。

信号传递

内核强迫目标进程通过以下方式对信号做出反应：或改变目标进程的执行状态，或开始执行一个特定的信号处理程序，或两者都是。

每个所产生的信号至多被传递一次。信号是可消费资源：一旦它们已传递出去，进程描述符中有关这个信号的所有信息都被取消。

已经产生但还没有传递的信号称为挂起信号（*pending signal*）。任何时候，一个进程仅存在给定类型的一个挂起信号，同一进程同种类型的其他信号不被排队，只被简单地丢弃。但是，实时信号是不同的：同种类型的挂起信号可以有好几个。

一般来说，信号可以保留不可预知的挂起时间。必须考虑下列因素：

- 信号通常只被当前正运行的进程传递（即由 current 进程传递）。
- 给定类型的信号可以由进程选择性地阻塞（*blocked*）（参见“修改阻塞信号的集合”一节）。这种情况下，在取消阻塞前进程将不接收这个信号。
- 当进程执行一个信号处理程序的函数时，通常“屏蔽”相应的信号，即自动阻塞这个信号直到处理程序结束。因此，所处理的信号的另一次出现不能中断信号处理程序，所以，信号处理函数不必是可重入的。

尽管信号的表示比较直观，但内核的实现相当复杂。内核必须：

- 记住每个进程阻塞哪些信号。
- 当从内核态切换到用户态时，对任何一个进程都要检查是否有一个信号已到达。这几乎在每个定时中断时都发生（大约每毫秒发生一次）。
- 确定是否可以忽略信号。这发生在下列所有的条件都满足时：
  - 目标进程没有被另一个进程跟踪（进程描述符中ptrace字段的PT_PTRACED标志等于0）（注1）。
  - 信号没有被目标进程阻塞。
  - 信号被目标进程忽略（或者因为进程已显式地忽略了信号，或者因为进程没有改变信号的缺省操作且这个缺省操作就是“忽略”）。
- 处理这样的信号，即信号可能在进程运行期间的任一时刻请求把进程切换到一个信号处理函数，并在这个函数返回以后恢复原来执行的上下文。

此外，Linux必须考虑BSD和System V所采用的不同的信号语义，而且，还必须与相当麻烦的POSIX标准相兼容。

传递信号之前所执行的操作

进程以三种方式对一个信号做出应答：

1. 显式地忽略信号。
2. 执行与信号相关的缺省操作（参见表11-1）。由内核预定义的缺省操作取决于信号的类型，可以是下列类型之一：

   Terminate

   进程被终止（杀死）。

   Dump

   进程被终止（杀死），并且，如果可能，创建包含进程执行上下文的核心转储文件；这个文件可以用于调试。

   Ignore

   信号被忽略。

注1：如果一个进程正在被跟踪时接收到一个信号，内核就停止这个进程，并向跟踪进程发送一个SIGCHLD信号以通知它一下。跟踪进程又可以使用SIGCOUNT信号重新恢复被跟踪进程的执行。

Stop

进程被停止，即把进程置为TASK_STOPPED状态（参见第三章的“进程状态”一节）。

Continue

如果进程被停止（TASK_STOPPED），就把它置为TASK_RUNNING状态。

3. 通过调用相应的信号处理函数捕获信号。

注意，被对一个信号的阻塞和忽略是不同的：只要信号被阻塞，它就不被传递；只有在信号解除阻塞后才传递它。而一个被忽略的信号总是被传递，只是没有进一步的操作。

SIGKILL和SIGSTOP信号不可以被显式地忽略、捕获或阻塞，因此，通常必须执行它们的缺省操作。因此，SIGKILL和SIGSTOP允许具有适当特权的用户分别终止并停止任何进程（注2），不管程序执行时采取怎样的防御措施。

如果信号的传递会引起内核杀死一个进程，那么这个信号对该进程就是致命的。SIGKILL信号总是致命的；而且，缺省操作为Terminate的每个信号，以及不被进程捕获的信号对该进程也是致命的。注意，如果一个被进程所捕获的信号，其对应的信号处理函数终止了这个进程，那么这个信号就不是致命的，因为进程自己选择了终止，而不是被内核杀死。

POSIX信号和多线程应用

POSIX 1003.1标准对多线程应用的信号处理有一些严格的要求：

- 信号处理程序必须在多线程应用的所有线程之间共享；不过，每个线程必须有自己的挂起信号掩码和阻塞信号掩码。
- POSIX库函数kill()和sigqueue()（见稍后“与信号处理相关的系统调用”一节）必须向所有的多线程应用而不是某个特殊的线程发送信号。所有由内核产生的信号同样如此（如：SIGCHLD、SIGINT或SIGQUIT）。
- 每个发送给多线程应用的信号仅传送给一个线程，这个线程是由内核在从不会阻塞该信号的线程中随意选择出来的。
- 如果向多线程应用发送了一个致命的信号，那么内核将杀死该应用的所有线程，而不仅仅是杀死接收信号的那个线程。

注2：有两个例外：不可能给进程0（swapper）发送信号，而发送给进程1（init）的信号在捕获到它们之前也总被丢弃。因此，进程0永不死亡，而进程1只有当init程序终止时才死亡。

为了遵循POSIX标准，Linux内核2.6把多线程应用实现为一组属于同一个线程组的轻量级进程（参见第三章“进程、轻量级进程和线程”一节）。

在本章中，术语“线程组”指任意一种线程组，包括仅由单一（普通）进程构成的线程组。例如，当我们规定kill()能够向线程组发送信号时，我们的意思是这个系统调用也能够向普通进程发送信号。我们将使用术语“进程”表示普通进程或轻量级进程，即属于某个线程组的特定成员。

此外，如果一个挂起信号被发送给了某个特定进程，那么这个信号是私有的；如果被发送给了整个线程组，它就是共享的。

与信号相关的数据结构

对系统中的每个进程来说，内核必须跟踪什么信号当前正在挂起或被屏蔽，以及每个线程组是如何处理所有信号的。为了完成这些操作，内核使用几个从处理器描述符可存取的数据结构。最重要的数据结构如图11-1所示。

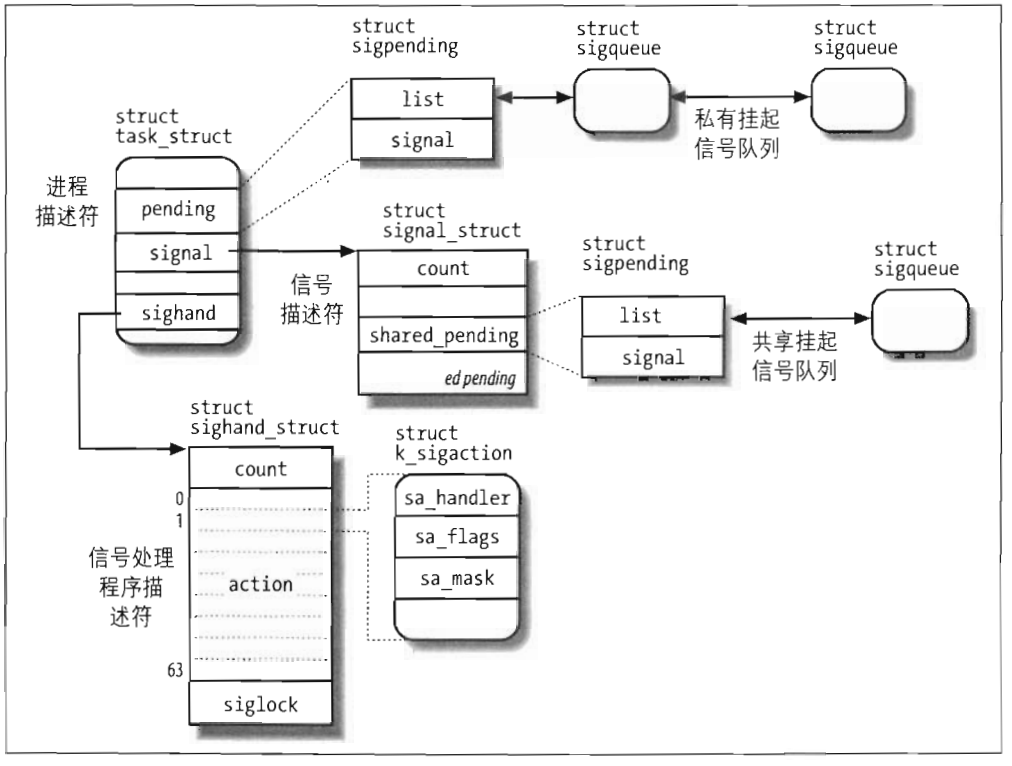


图11-1：与信号处理相关的最重要的数据结构

与信号处理相关的进程描述符中的字段如表 11-3 所示：

表 11-3：与信号处理相关的进程描述符中的字段

类型	名称	说明
struct signal_struct *	signal	指向进程的信号描述符的指针
struct sighand_struct *	sighand	指向进程的信号处理程序描述符的指针
sigset_t	blocked	被阻塞信号的掩码
sigset_t	real_blocked	被阻塞信号的临时掩码（由 rt_sigtimedwait()系统调用使用）
struct sigpending	pending	存放私有挂起信号的数据结构
unsigned long	sas_ss_sp	信号处理程序备用堆栈的地址
size_t	sas_ss_size	信号处理程序备用堆栈的大小
int (*) (void *)	notifier	指向一个函数的指针，设备驱动程序用这个函数阻塞进程的某些信号
void *	notifier_data	指向 notifier 函数（表中的前一个字段）可能使用的数据
sigset_t *	notifier_mask	设备驱动程序通过 notifier 函数所阻塞的信号的位掩码

blocked字段存放进程当前所屏蔽的信号。它是一个 sigset_t 位数组，每种信号类型对应一个元素：

```
typedef struct {
    unsigned long sig[2];
} sigset_t;
```

因为每个无符号长整数由 32 位组成，所以在 Linux 中可以声明的信号最大数是 64（_NSIG 宏表示这个值）。没有值为 0 的信号，因此，信号的编号对应于 sigset_t 类型变量中的相应位下标加 1。1～31 之间的编号对应于表 11-1 所列出的信号，而 32～64 之间的编号对应于实时信号。

信号描述符和信号处理程序描述符

进程描述符的 signal 字段指向信号描述符（*signal descriptor*）——一个 signal_struct类型的结构，用来跟踪共享挂起信号。实际上，信号描述符还包括与信号处理关系并不密切的一些字段，如：每进程的资源限制数组 rlim（见第三章“进程资源限制”一节）、分别用于存放进程的组领头进程和会话领头进程 PID 的字段 pgrp 和 session

(见第三章“进程间的关系”一节)。实际上，就像在第三章“clone()、fork()及vfork()系统调用”一节所提到的，信号描述符被属于同一线程组的所有进程共享，也就是被调用clone()系统调用（CLONE_THREAD标志置位）创建的所有进程共享，因此，对属于同一线程组的每个进程而言，信号描述符中的字段必须都是相同的。

信号描述符中与信号处理有关的字段如表11-4所示：

表11-4：信号描述符中与信号处理有关的字段

类型	名称	说明
atomic_t	count	信号描述符的使用计数器
atomic_t	live	线程组中活动进程的数量
wait_queue_head_t	wait_chldexit	在系统调用wait4()中睡眠的进程的等待队列
struct task_struct *	curr_target	接收信号的线程组中最后一个进程的描述符
struct sigpending	shared_pending	存放共享挂起信号的数据结构
int	group_exit_code	线程组的进程终止代码
struct task_struct *	group_exit_task	在杀死整个线程组的时候使用
int	notify_count	在杀死整个线程组的时候使用
int	group_stop_count	在停止整个线程组的时候使用
unsigned int	flags	在传递修改进程状态的信号时使用的标志

除了信号描述符以外，每个进程还引用一个信号处理程序描述符（*signad handler deseriplor*），它是一个sighand_struct类型的结构，用来描述每个信号必须怎样被线程组处理。它的字段在表11-5中说明。

表11-5：信号处理程序描述符的字段

类型	名称	说明
atomic_t	count	信号处理程序描述符的使用计数器
struct k_sigaction [64]	action	说明在所传递信号上执行操作的结构数组
spinlock_t	siglock	保护信号描述符和信号处理程序描述符的自旋锁

正如在第三章的“clone()、fork()及vfork()系统调用”一节中所提到的，在调用clone()系统调用时设置CLONE_SIGHAND标志，信号处理程序描述符就可以由几个进程共享。

描述符的count字段表示共享该结构的进程个数。在一个POSIX的多线程应用中，线程组中的所有轻量级进程都引用相同的信号描述符和信号处理程序描述符。

sigaction 数据结构

一些体系结构把特性赋给仅对内核可见的信号。因此，信号的特性存放在k_sigaction结构中，k_sigaction结构既包含对用户态进程所隐藏的特性，也包含大家熟悉的sigaction结构，该结构保存了用户态进程能看见的所有特性。实际上，在80x86平台上，信号的所有特性对用户态的进程都是可见的。因此，k_sigaction结构只不过简化为类型为sigaction的单个sa结构，该结构包含下列字段（注3）。

sa_handler

这个字段指定要执行操作的类型。它的值可以是指向信号处理程序的一个指针，SIG_DFL（即值0，指定执行缺省操作），或者SIG_IGN（即值1，指定忽略信号）。

sa_flags

这是一个标志集，指定必须怎样处理信号。其中的一些标志在表11-6中列出（注4）。

sa_mask

这是类型为sigset_t的变量，指定当运行信号处理程序时要屏蔽的信号。

表11-6：指定如何处理信号的一组标志

标志的名称	说明
SA_NOCLDSTOP	仅应用于SIGCHLD，当进程被停止时不向父进程发送SIGCHLD信号
SA_NOCLDWAIT	仅应用于SIGCHLD，当进程终止时不创建僵死状态
SA_SIGINFO	为信号处理程序提供附加信息（参见后面“改变信号的操作”一节）
SA_ONSTACK	为信号处理程序的执行使用一个备用栈（参见后面“捕获信号”一节）
SA_RESTART	自动地重新开始执行被中断的系统调用（参见后面“系统调用的重新执行”一节）

注3：用户态应用程序所使用的sigaction结构向signal()和sigaction()系统调用传递参数，它与内核所使用的结构稍有不同，虽然本质上它们存放相同的信息。

注4：由于历史原因，这些标志与irqaction描述符的标志一样都有相同的前缀“SA_”（参见第四章的表4-7），不过这两种标志集合之间没有关系。

表 11-6：指定如何处理信号的一组标志（续）

标志的名称	说明
SA_NODEFER, SA_NOMASK	当执行信号处理程序时不屏蔽信号
SA_RESETHAND, SA_ONESHOT	执行信号处理程序后重新设置缺省操作

挂起信号队列

如我们在本章前面的表11-2中所见到的，有几个系统调用能产生发送给整个线程组的信号，如kill()和rt_sigqueueinfo()，而其他的一些则产生发送给特定进程的信号，如tkill()和tgkill()。

因此，为了跟踪当前的挂起信号是什么，内核把两个挂起信号队列与每个进程相关联：

- 共享挂起信号队列，它位于信号描述符的shared_pending字段，存放整个线程组的挂起信号。
- 私有挂起信号队列，它位于进程描述符的pending字段，存放特定进程（轻量级进程）的挂起信号。

挂起信号队列由 sigpending 数据结构组成，它的定义如下：

```
struct sigpending {
    struct list_head list;
    sigset_t signal;
}
```

signal 字段是指定挂起信号的位掩码，而 list 字段是包含 sigqueue 数据结构的双向链表的头，sigqueue 的字段如表 11-7 所示。

表 11-7：sigqueue 数据结构的字段

类型	名称	说明
struct list_head	list	链接挂起信号队列的链表
spinlock_t *	lock	指向与挂起信号相应的信号处理程序描述符中 siglock 字段的指针
Int	flags	Sigqueue 数据结构的标志
siginfo_t	info	描述产生信号的事件
struct user_struct *	user	指向进程拥有者的每用户数据结构的指针（见第三章“clone()、 fork()及 vfork()系统调用”一节）

siginfo_t是一个128字节的数据结构，其中存放有关出现特定信号的信息。它包含下列字段：

si_signo
: 信号编号。

si_errno
: 引起信号产生的指令的出错码，或者如果没有错误则为0。

si_code
: 发送信号者的代码（参见表11-8）。

表11-8：最重要的信号发送者代码

代码名	发送者
SI_USER	kill()和raise()(参见后面“与信号处理相关的系统调用”一节)
SI_KERNEL	一般内核函数
SI_QUEUE	sigqueue()(参见后面“与信号处理相关的系统调用”一节)
SI_TIMER	定时器到期
SI_ASYNCIO	异步I/O完成
SI_TKILL	tkill()和tgkill()(参见后面“与信号处理相关的系统调用”一节)

_sifields
: 存放依赖于信号类型的信息的联合体。例如，相对于SIGKILL信号的出现，siginfo_t数据结构在这里记录发送者进程的PID和UID；相反，相对于SIGSEGV信号的出现，该数据结构存放某个内存地址，对该地址的访问导致信号产生。

在信号数据结构上的操作

内核使用几个函数和宏来处理信号。在下面的描述中，set 是指向sigset_t类型变量的一个指针，nsig是信号的编号，mask是无符号长整数的位掩码。

sigemptyset(set)和sigfillset(set)
: 把sigset_t类型的变量中的位分别置为0或1。

sigaddset(set,nsig)和sigdelset(set,nsig)
: 把nsig信号在sigset_t类型变量中对应的位分别置为1或0。实际上，sigaddset()简化为：

```
set->sig[(nsig - 1) / 32] |= 1UL << ((nsig - 1) % 32);
```

并且把 sigdelset()简化为：

```
set->sig[(nsig - 1) / 32] &= ~(1UL << ((nsig - 1) % 32));
```

sigaddsetmask(set,mask)和sigdelsetmask(set,mask)

把mask中的位在sigset_t类型变量中对应的所有位分别设置为1或0。它们仅用于编号为1~32之间的信号。对应的函数简化为：

```
set->sig[0] |= mask;
```

和

```
set->sig[0] &= ~mask;
```

sigismember(set,nsig)

返回nsig信号在sigset_t类型变量中对应位的值。实际上，这个函数简化为：

```
reture 1 & (set->sig[(nsig - 1) / 32] >> ((nsig - 1) % 32));
```

sigmask(nsig)

产生nsig信号的位索引。换句话说，如果内核需要设置、清除或测试一个特定信号在sigset_t类型变量中对应的位，那么就能通过这个宏得到合适的位。

sigandsets(d,s1,s2)、sigorsets(d,s1,s2)和signandsets(d,s1,s2)

在sigset_t类型的s1和s2变量之间分别执行逻辑“与”、逻辑“或”及逻辑“与非”。其结果保存在d指向的sigset_t类型的变量中。

sigtestsetmask(set,mask)

如果mask在sigset_t类型的变量中对应的任何一位被设置，就返回值1；否则返回0。这只用于编号为1～32的信号。

siginitset(set,mask)

用mask中的位初始化1～32之间的信号在sigset_t类型的变量中对应的低位，并把33～63之间信号的对应位清0。

siginitsetinv(set,mask)

用mask中位的补码初始化1～32之间的信号在sigset_t类型的变量中对应的低位，并把33～63之间信号的对应位置位。

signal_pending(p)

如果*p进程描述符所表示的进程有非阻塞的挂起信号，就返回值1（真），否则返回0（假）。该函数只是通过检查进程的TIF_SIGPENDING标志就可实现。

recalc_sigpending_tsk(t)和recalc_sigpending()

第一个函数检查是*t进程描述符所表示的进程有挂起的信号（通过检查t->pending->signa（字段），还是进程所属的线程组有挂起的信号（通过检查t-

>signal->shared_pending->signal字段)。然后函数把t->thread_info->flags的TIF_SIGPENDING标志置位。函数recalc_sigpending()等价于recalc_sigpending_tsk(current)。

rm_from_queue(mask,q)

从挂起信号队列q中删除与mask位掩码相对应的挂起信号。

flush_sigqueue(q)

从挂起信号队列q中删除所有的挂起信号。

flush_signals(t)

删除发送给*t进程描述符所表示的进程的所有信号。这是通过清除t->thread_info->flags中的TIF_SIGPENDING标志，并在t->pending和t->signal->shared_pending队列上两次调用flush_sigqueue()函数来实现的。

产生信号

很多内核函数都会产生信号：它们完成信号处理第一步的工作（在前面“信号的作用”一节中已经描述），即根据需要更新一个或多个进程的描述符。它们不直接执行第二步的信号传递操作，而是可能根据信号的类型和目标进程的状态唤醒一些进程，并促使这些进程接收信号。

当发送给进程一个信号时，这个信号可能来自内核，也可能来自另一个进程。内核通过对如表11-9所示的某个函数进行调用而产生信号。

表11-9：为进程产生信号的内核函数

函数名	说明
send_sig()	向单一进程发送信号
send_sig_info()	与send_sig()类似，只是还使用siginfo_t结构中的扩展信息
force_sig()	发送既不能被进程显式忽略，也不能被进程阻塞的信号
force_sig_info()	与force_sig()类似，只是还使用siginfo_t结构中的扩展信息
force_sig_specific()	与force_sig()类似，但优化了对SIGSTOP和SIGKILL信号的处理
sys_tkill()	tkill()的系统调用处理函数（参见后面“与信号处理相关的系统调用”一节）
sys_tgkill()	tgkill()的系统调用处理函数

表11-9中的所有函数在结束时都调用specific_send_sig_info()函数，下一节将对其进行描述。

当一个信号被发往整个线程组时，这个信号可能来自内核，也可能来自另外一个进程。内核对如表11-10所示的某个函数进行调用产生这类信号。

表11-10：为线程组产生信号的内核函数

函数名	说明
send_group_sig_info()	向某一个线程组发送信号，该线程组由它的一个成员进程的描述符来标识
kill_pg()	向一个进程组中的所有线程组发送信号（参见第一章“进程管理”一节
kill_pg_info()	与kill_pg()类似，只是还使用siginfo_t结构中的扩展信息
kill_proc()	向某一个线程组发送信号，该线程组由它的一个成员进程的PID来标识
kill_proc_info()	与kill_proc()类似，只是还使用siginfo_t结构中的扩展信息
sys_kill()	kill()的系统调用处理函数（参见后面“与信号处理相关的系统调用”一节）
sys_rt_sigqueueinfo()	rt_sigqueueinfo()的系统调用处理函数

表11-10中的所有函数在结束时都调用group_send_sig_info()函数，将在后面的“group-stnd-sig-info()函数”一节中对其进行描述。

specific_send_sig_info()函数

specific_send_sig_info()函数向指定进程发送信号，它作用于三个参数：

sig

信号编号。

info

或者是siginfo_t表的地址，或者是三个特殊值中的一个：0意味着信号是由用户态进程发送的，1意味着是由内核发送的，2意味着是由内核发送的SIGSTOP或SIGKILL信号。

t

指向目标进程描述符的指针。

必须在关本地中断和已经获得t->sighand->siglock自旋锁的情况下调用specific_send_sig_info()函数。函数执行下面的步骤：

1. 检查进程是否忽略信号，如果是就返回0（不产生信号）。当下面的三个忽略信号的条件全部满足时，信号就被忽略：
   - 进程没有被跟踪（t->ptrace中的PT_PTRACED标志被清0）
   - 信号没有被阻塞（sigismember(&t->blocked, sig)返回0）
   - 或者显式地忽略信号（t->sighand->action[sig-1]的sa_handler字段等于SIG_IGN），或者隐含地忽略信号（sa_handler字段等于SIG_DFL而且信号是SIGCONT、SIGCHLD、SIGWINCH或SIGURG）
2. 检查信号是否是非实时的(sig<32),而且是否在进程的私有挂起信号队列上已经有另外一个相同的挂起信号（sigismember(&t->pending.signal,sig)返回1），如果是，就什么都不需要做，因此返回0。
3. 调用send_signal(sig, info, t, &t->pending)把信号添加到进程的挂起信号集合中，在下一节将详细描述这个函数。
4. 如果send_signal()成功地结束，而且信号不被阻塞（sigismember(&t->blocked,sig)返回0），就调用signal_wake_up()函数通知进程有新的挂起信号。随后，该函数执行下述步骤：
   a. 把t->thread_info->flags中的TIF_SIGPENDING标志置位。
   b. 如果进程处于TASK_INTERRUPTIBLE或TASK_STOPPED状态，而且信号是SIGKILL，就调用try_to_wake_up()（参见第七章“try_to_wake_up()函数”一节）唤醒进程。
   c. 如果try_to_wake_up()返回0，那么说明进程已经是可运行的：这种情况下，它检查进程是否已经在另外一个CPU上运行，如果是就向那个CPU发送一个处理器间中断，以强制当前进程的重新调度（参见第四章“处理器间中断的处理”一节）。因为在从调度函数返回时，每个进程都检查是否存在挂起信号，因此，处理器间中断保证了目标进程能很快注意到新的挂起信号。
5. 返回1（已经成功地产生信号）。

send_signal()函数

send_signal()函数在挂起信号队列中插入一个新元素，它接收信号编号sig、siginfo_t数据结构的地址info（或一个特殊编码，见上一节对specific_send_sig_info()的描述）、目标进程描述符的地址t以及挂起信号队列的地址signals作为它的参数。

函数执行下面的步骤：

1. 如果info的值是2，这个信号就是SIGKILL或SIGSTOP，而且已经由内核通过force_sig_specific()函数产生：在这种情况下，函数跳转到第9步，内核立即强制执行与这些信号相关的操作，因此函数不用把信号添加到挂起信号队列中。

2. 如果进程拥有者的挂起信号的数量(t->user->sigpending)小于当前进程的资源限制（t->signal->rlim[RLIMIT_SIGPENDING].rlim_cur），函数就为新出现的信号分配sigqueue数据结构。

```
q = kmem_cache_alloc(sigqueue_cachep, GFP_ATOMIC);
```

3. 如果进程拥有者的挂起信号的数量太多，或者上一步的内存分配失败，就跳转到第9步。

4. 递增拥有者挂起信号的数量（t->user->sigpending）和t->user所指向的每用户数据结构的引用计数器。

5. 在挂起信号队列signals中增加sigqueue数据结构。

```
list_add_tail(&q->list, &signals->list);
```

6. 在sigqueue数据结构中填充表siginfo_t。

```
if ((unsigned long)info == 0) {
    q->info.si_signo = sig;
    q->info.si_errno = 0;
    q->info.si_code = SI_USER;
    q->info._sifields._kill._pid = current->pid;
    q->info._sifields._kill._uid = current->uid;
} else if ((unsigned long)info == 1) {
    q->info.si_signo = sig;
    q->info.si_errno = 0;
    q->info.si_code = SI_KERNEL;
    q->info._sifields._kill._pid = 0;
    q->info._sifields._kill._uid = 0;
} else
   copy_siginfo(&q->info, info);
```

copy_siginfo()函数复制由调用者传递的siginfo_t表。

7. 把队列位掩码中与信号相应的位置1：

```
sigaddset(&signals->signal, sig);
```

8. 返回0：说明信号已被成功地追加到挂起信号队列中。

9. 此时，不再向信号挂起队列中增加元素，因为已经有太多的挂起信号，或已经没有可以分给sigqueue数据结构的空闲空间，或者信号已经由内核强制立即发送。如果信号是实时的，并已经通过内核函数发送给队列排队，则send_signal()函数返回错误代码-EAGAIN:

```
if (sig>=32 && info && (unsigned long) info != 1 &&
    info->si_code != SI_USER)
   return -EAGAIN;
```

10. 设置队列的位掩码中与信号相关的位：

```
sigaddset(&signals->signal, sig);
```

11. 返回0：即使信号没有被追加到队列中，挂起信号掩码中相应的位也被设置。

即使在挂起队列中没有空间存放相应的挂起信号，让目标进程能接收信号也是至关重要的。假设一个进程正在消耗过多内存的情形。内核必须保证即使没有空闲内存，kill()系统调用也能够成功执行，否则，系统管理员就没有机会通过终止有害进程来恢复系统。

group_send_sig_info()函数

group_send_sig_info()函数向整个线程组发送信号。它作用于三个参数：信号编号sig、siginfo_t表的地址info（可选的值为0、1或2，如前面“specific_send_sig_info()函数“一节中所描述的）以及进程描述符的地址p。

该函数主要执行下面的步骤：

1. 检查参数sig是否正确：

```
if (sig < 0 || sig > 64)
   return -EINVAL;
```

2. 如果信号是由用户态进程发送的，则该函数确定是否允许这个操作。下列条件中至少有一个成立时信号才能被传递：

   - 发送进程的拥有者具有适当的权能（这通常意味着通过系统管理员发布信号；参见第二十章）。
   - 信号为SIGCONT且目标进程与发送进程处于同一个注册会话中。
   - 两个进程属于同一个用户。

   如果不允许用户态进程发送信号，函数就返回值-EPERM。

3. 如果参数sig的值为0，则函数不产生任何信号，立即返回：

```
if (!sig || !p->sighand)
    return 0;
```

因为0是无效的信号编码，用于让发送进程检查它是否有向目标线程组发送信号所必需的特权。如果目标进程正在被杀死（通过检查它的信号处理程序描述符是否已经被释放来获知），那么函数也返回。

4. 获取p->sighand->siglock自旋锁并关闭本地中断。

5. 调用handle_stop_signal()函数，该函数检查信号的某些类型，这些类型可能使目标线程组的其他挂起信号无效。handle_stop_signal()函数执行下面的步骤：

   a. 如果线程组正在被杀死（信号描述符的flags字段的SIGNAL_GROUP_EXIT标志被设置），则函数返回。

   b. 如果sig是SIGSTOP、SIGTSTP、SIGTTIN或SIGTTOU信号，就调用rm_from_queue()函数从共享挂起信号队列p->signal->shared_pending和线程组所有成员的私有信号队列中删除SIGCONT信号。

   c. 如果sig是SIGCONT信号，就调用rm_from_queue()函数从共享挂起信号队列p->signal->shared_pending中删除所有的SIGSTOP、SIGTSTP、SIGTTIN和SIGTTOU信号，然后从属于线程组的进程的私有挂起信号队列中删除上述信号，并唤醒进程：

```
rm_from_queue(0x003c0000, &p->signal->shared_pending);
t = p;
do {
    rm_from_queue(0x003c0000, &t->pending);
    try_to_wake_up(t, TASK_STOPPED, 0);
    t = next_thread(t);
} while (t != p);
```

   掩码0x003c0000选择以上四种停止信号。宏next_thread每次循环都返回线程组中不同轻量级进程的描述符地址（见第三章“进程间的关系”一节）（注5）。

6. 检查线程组是否忽略信号，如果是就返回0值（成功）。如果在前面“信号的作用”一节中所提到的忽略信号的三个条件都满足（也可参见前面“specific-send-sig.info()函数”一节中的第1步），就忽略信号。

7. 检查信号是否是非实时的，并且在线程组的共享挂起信号队列中已经有另外一个相同的信号，如果是，就什么都不需要做，因此返回0值（成功）。

```
if (sig<32 && sigismember(&p->signal->shared_pending.signal,sig))
    return 0;
```

8. 调用send_signal()函数把信号添加到共享挂起信号队列中（参见前面“send_signal()函数”一节）。如果send_signal()返回非0的错误代码，则函数终止并返回相同的值。

注5：实际代码比这里所给出的代码片段要复杂得多，因为handle_stop_signal()还要考虑SIGCONT信号被捕获的特殊情况，以及当线程组的所有进程都正在被停止时由SIGCONT信号引起竞争条件的特殊情况。

9. 调用__group_complete_signal()函数唤醒线程组中的一个轻量级进程（见下面）。
10. 释放p->sighand->siglock自旋锁并打开本地中断。
11. 返回0（成功）。

函数__group_complete_signal()扫描线程组中的进程，查找能接收新信号的进程。满足下述所有条件的进程可能被选中：

- 进程不阻塞信号。
- 进程的状态不是EXIT_ZOMBIE、EXIT_DEAD、TASK_TRACED或TASK_STOPPED（作为一种异常情况，如果信号是SIGKILL，那么进程可能处于TASK_TRACED或者TASK_STOPPED状态）。
- 进程没有正在被杀死，即它的PF_EXITING标志没有置位。
- 进程或者当前正在CPU上运行，或者它的TIF_SIGPENDING标志还没有设置。（实际上，唤醒一个有挂起信号的进程是毫无意义的：通常，唤醒操作已经由设置了TIF_SIGPENDING标志的内核控制路径执行；另一方面，如果进程正在执行，则应该向它通报有新的挂起信号。）

一个线程组可能有很多满足上述条件的进程，函数按照下面的规则选择其中的一个进程：

- 如果p标识的进程（由group_send_sig_info()的参数传递的描述符地址）满足所有的优先准则，并因此而能接收信号，函数就选择该进程。
- 否则，函数通过扫描线程组的成员搜索一个适当的进程，搜索从接收线程组最后一个信号的进程（p->signal->curr_target）开始。

如果函数__group_complete_signal()成功地找到一个适当的进程，就开始向被选中的进程传递信号。首先，函数检查信号是否是致命的，如果是，通过向线程组中的所有轻量级进程发送SIGKILL信号杀死整个线程组。否则，函数调用signal_wake_up()函数通知被选中的进程：有新的挂起信号到来（见前面“specific_send_sig_info()函数”一节的第4步）。

传递信号

我们假定内核已注意到一个信号的到来，并调用前面所介绍的函数为接收此信号的进程准备描述符。但万一这个进程在那一刻并不在CPU上运行，内核就延迟传递信号的任务。我们现在转向另一个主题，即为确保进程的挂起信号得到处理内核所执行的操作。

我们在第四章“从中断和异常返回”一节中提到，内核在允许进程恢复用户态下的执行之前，检查进程TIF_SIGPENDING标志的值。每当内核处理完一个中断或异常时，就检查是否存在挂起信号。

为了处理非阻塞的挂起信号，内核调用do_signal()函数，它接收两个参数：

regs
: 栈区的地址，当前进程在用户态下寄存器的内容存放在这个栈中。

oldset
: 变量的地址，假设函数把阻塞信号的位掩码数组存放在这个变量中。如果没有必要保存位掩码数组，则它为NULL。

对于do_signal()函数，我们将重点说明信号传递的一般机制。它的实现代码很累赘，这是因为要对竞争条件和其他特殊情况（如：冻结系统、产生内存信息转储、停止和杀死整个线程组等等）进行详细处理，因此我们将悄然地略过这些细节。

就像已经提到过的，通常只是在CPU要返回到用户态时才调用do_signal()函数。因此，如果中断处理程序调用do_signal()，则该函数立刻返回：

```
if ((regs->xcs & 3) != 3)
    return 1;
```

如果oldset参数为NULL，函数就用current->blocked字段的地址对它初始化：

```
if (!oldset)
    oldset = &current->blocked;
```

do_signal()函数的核心由重复调用dequeue_signal()函数的循环组成，直到在私有挂起信号队列和共享挂起信号队列中都没有非阻塞的挂起信号时，循环才结束。dequeue_signal()的返回码存放在signr局部变量中。如果值为0，意味着所有挂起的信号已全部被处理，并且do_signal()可以结束。只要返回一个非0值，就意味着挂起的信号正等待被处理，并且do_signal()处理了当前信号后又调用了dequeue_signal()。

dequeue_signal()函数首先考虑私有挂起信号队列中的所有信号，并从最低编号的挂起信号开始。然后考虑共享队列中的信号。它更新数据结构以表示信号不再是挂起的，并返回它的编号。这就涉及清current->pending.signal或current->signal->shared_pending.signal中对应的位，并调用recalc_sigpending()更新TIF_SIGPENDING标志的值。

让我们来看do_signal()函数如何处理每一个挂起的信号，其编号由dequeue_signal()返回。首先，它检查current接收进程是否正受到其他一些进程的监控；在肯定的情况

下，do_signal()调用do_notify_parent_cldstop()和schedule()让监控进程知道进程的信号处理。

然后，do_signal()把要处理信号的k_sigaction数据结构的地址赋给局部变量ka：

```
ka = &current->sig->action[signr-1];
```

根据ka的内容可以执行三种操作：忽略信号、执行缺省操作或执行信号处理程序。

如果显式忽略被传递的信号，那么do_signal()函数仅仅继续执行循环，并由此考虑另一个挂起信号：

```
if (ka->sa.sa_handler == SIG_IGN)
    continue;
```

在下面两节，我们将说明如何执行缺省操作和信号处理程序。

执行信号的缺省操作

如果ka->sa.sa_handler等于SIG_DFL，do_signal()就必须执行信号的缺省操作。唯一的例外是当接收进程是init时，在这种情况下，正如前面“传递信号之前所执行的操作”一节中所描述的那样，这个信号被丢弃：

```
if (current->pid == 1)
   continue;
```

如果接收进程是其他进程，对缺省操作是Ignore的信号进行处理也很简单：

```
if (signr==SIGCONT || signr==SIGCHLD ||
        signr==SIGWINCH || signr==SIGURG)
   continue;
```

缺省操作是Stop的信号可能停止线程组中的所有进程。为此，do_signal()把进程的状态都置为TASK_STOPPED，并在随后调用schedule()函数（参见第七章“schedule()函数”一节）。

```
if (signr==SIGSTOP || signr==SIGTSTP ||
        signr==SIGTTIN || signr==SIGTTOU) {
    if (signr != SIGSTOP &&
            is_orphaned_pgrp(current->signal->pgrp))
        continue;
    do_signal_stop(signr);
}
```

SIGSTOP与其他信号的差异比较微妙：SIGSTOP总是停止线程组，而其他信号只停止不在“孤儿进程组”中的线程组。POSIX标准规定，只要进程组中有一个进程有父进程，

尽管父进程处于不同的进程组中但在同一个会话中，那么这个进程组就不是孤儿。因此，如果父进程死亡，但启动该进程的用户仍登录在线，那么该进程组就不是一个孤儿。

do_signal_stop()函数检查current是否是线程组中第一个被停止的进程，如果是，它激活"组停止"：本质上，该函数把一个正数值赋给信号描述符中的group_stop_count字段，并唤醒线程组中的所有进程。所有这样的进程都检查该字段以确认正在进行"组停止"操作，然后把进程的状态置为TASK_STOPPED，并调用schedule()。如果线程组领头进程的父进程没有设置SIGCHLD的SA_NOCLDSTOP标志，那么do_signal_stop()函数还要向它发送SIGCHLD信号。

缺省操作为Dump的信号可以在进程的工作目录中创建一个"转储"文件，这个文件列出进程地址空间和CPU寄存器的全部内容。do_signal()创建了转储文件后，就杀死这个线程组。剩余18个信号的缺省操作是Terminate，它仅仅是杀死线程组。为了杀死整个线程组，函数调用do_group_exit()执行彻底的"组退出"过程（参见第三章的"进程终止"一节）。

捕获信号

如果信号有一个专门的处理程序，do_signal()就函数必须强迫该处理程序执行。这是通过调用handle_signal()进行的：

```
handle_signal(signr, &info, &ka, oldset, regs);
if (ka->sa.sa_flags & SA_ONESHOT)
    ka->sa.sa_handler = SIG_DFL;
return 1;
```

如果所接收信号的SA_ONESHOT标志被置位，就必须重新设置它的缺省操作，以便同一信号的再次出现不会再次触发这一信号处理程序的执行。注意do_signal()在处理了一个单独的信号后怎样返回。直到下一次调用do_signal()时才考虑其他挂起的信号。这种方式确保了实时信号将以适当的顺序得到处理。

执行一个信号处理程序是件相当复杂的任务，因为在用户态和内核态之间切换时需要谨慎地处理栈中的内容。我们将正确地解释这里所承担的任务。

信号处理程序是用户态进程所定义的函数，并包含在用户态的代码段中。handle_signal()函数运行在内核态，而信号处理程序运行在用户态，这就意味着在当前进程恢复"正常"执行之前，它必须首先执行用户态的信号处理程序。此外，当内核打算恢复进程的正常执行时，内核态堆栈不再包含被中断程序的硬件上下文，因为每当从内核态向用户态转换时，内核态堆栈都被清空。

而另外一个复杂性是因为信号处理程序可以调用系统调用，在这种情况下，执行了系统调用的服务例程以后，控制权必须返回到信号处理程序而不是到被中断程序的正常代码流。

Linux所采用的解决方法是把保存在内核态堆栈中的硬件上下文拷贝到当前进程的用户态堆栈中。用户态堆栈也以这样的方式被修改，即当信号处理程序终止时，自动调用sigreturn()系统调用把这个硬件上下文拷贝回到内核态堆栈中，并恢复用户态堆栈中原来的内容。

图11-2说明了有关捕获一个信号的函数的执行流。一个非阻塞的信号发送给一个进程。当中断或异常发生时，进程切换到内核态。正要返回到用户态前，内核执行do_signal()函数，这个函数又依次处理信号（通过调用handle_signal()）和建立用户态堆栈（通过调用setup_frame()或setup_rt_frame()）。当进程又切换到用户态时，因为信号处理程序的起始地址被强制放进程序计数器中，因此开始执行信号处理程序。当处理程序终止时，setup_frame()或setup_rt_frame()函数放在用户态堆栈中的返回代码就被执行。这个代码调用sigreturn()或rt_sigreturn()系统调用，相应的服务例程把正常程序的用户态堆栈硬件上下文拷贝到内核态堆栈，并把用户态堆栈恢复到它原来的状态（通过调用restore_sigcontext()）。当这个系统调用结束时，普通进程就因此能恢复自己的执行。

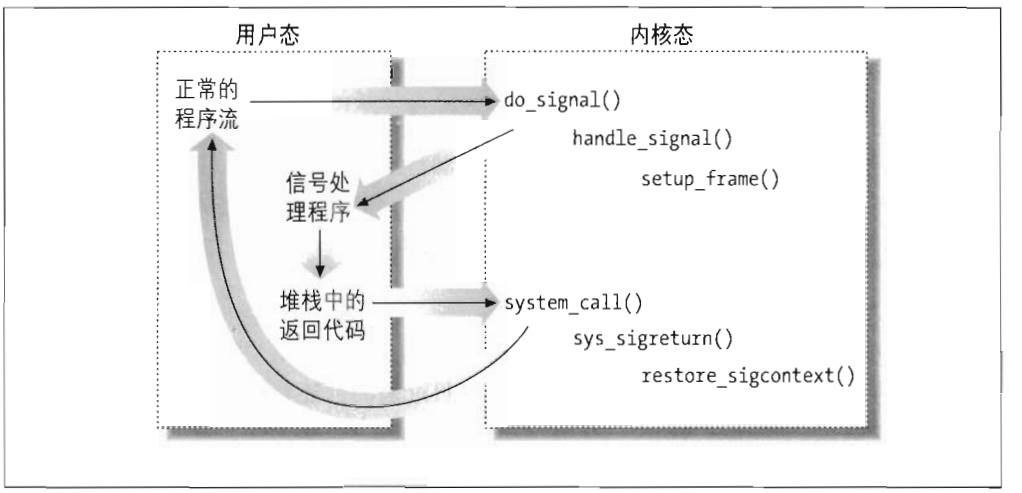


图11-2：捕获一个信号

现在我们详细考察如何实施这种方案。

建立帧

为了适当地建立进程的用户态堆栈，handle_signal()函数或者调用setup_frame()(对不需要siginfo_t表的信号；参见本章后面“与信号处理相关的系统调用”一节)，或者调用setup_rt_frame()(对需要siginfo_t表的信号)。为了在这两个函数之间进行选择，内核检查与信号相关的sigaction表sa_flags字段的SA_SIGINFO标志值。

setup_frame()函数接收四个参数，它们具有下列含义：

sig
: 信号编号

ka
: 与信号相关的k_sigaction表的地址

oldset
: 阻塞信号的位掩码数组的地址

regs
: 用户态寄存器的内容保存在内核态堆栈区的地址

setup_frame()函数把一个叫做帧（*frame*）的数据结构推进用户态堆栈中，这个帧含有处理信号所需要的信息，并确保正确返回到handle_signal()函数。一个帧就是包含下列字段的sigframe表（见图11-3)：

pretcode
: 信号处理函数的返回地址，它指向__kernel_sigreturn标记处的代码（稍后列出)。

sig
: 信号编号，这是信号处理程序所需的参数。

sc
: 类型为sigcontext的结构，它包含正好切换到内核态前用户态进程的硬件上下文(这种信息是从current的内核态堆栈中拷贝过来的)，还包含进程被阻塞的常规信号的位数组。

fpstate
: 类型为_fpstate的结构，可以用来存放用户态进程的浮点寄存器内容（参见第三章的“保存和加载FPU、MMX及XMM寄存器”一节)。

extramask
: 被阻塞的实时信号的位数组。

retcode

发出sigreturn()系统调用的8字节代码。在Linux的早期版本中，这段程序有效地执行从信号处理程序返回的功能，但在Linux 2.6中，它仅被当做一个标记（signature）来使用，以便调式程序能识别出信号栈帧。

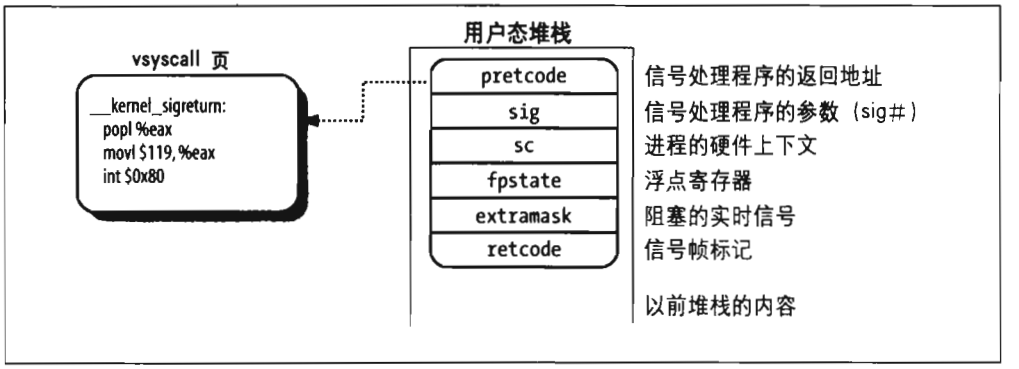


图11-3：在用户态堆栈中的帧

setup_frame()函数首先调用get_sigframe()计算帧的第一个内存单元，这个内存单元通常是在用户态堆栈中（注6），因此函数返回值：

```
(regs->esp - sizeof(struct sigframe)) & 0xfffffff8
```

因为栈朝低地址方向延伸，所以通过把当前栈顶的地址减去它的大小，使其结果与8的倍数对齐，就获得了帧的起始地址。

然后用access_ok宏对返回地址进行验证。如果地址有效，setup_frame()就反复调用__put_user()填充帧的所有字段。帧的pretcode字段初始化为&__kernel_sigreturn，一些粘合代码的地址放在vsyscall页中（参见第十章“通过sysenter指令发出系统调用”一节）。

一旦完成了这个操作，就修改内核态堆栈的regs区，这就保证了当current恢复它在用户态的执行时，控制权将传递给信号处理程序。

注6：Linux允许进程通过调用signalstack()系统调用来为它们的信号处理程序指定一个预备的栈，这种特点也是X/Open标准所要求的。当一个预备的栈存在时，get_sigframe()函数就返回这个栈中的一个地址。我们在此不进一步讨论这种情况，因为它从概念上非常类似于常规信号的处理。

```
regs->esp = (unsigned long) frame;
regs->eip = (unsigned long) ka->sa.sa_handler;
regs->eax = (unsigned long) sig;
regs->edx = regs->ecx = 0;
regs->xds = regs->xes = regs->xss = __USER_DS;
regs->xcs = __USER_CS;
```

setup_frame()函数把保存在内核态堆栈的段寄存器内容重新设置成它们的缺省值以后才结束。现在，信号处理程序所需的信息就在用户态堆栈的顶部。

setup_rt_frame()函数与setup_frame()非常相似，但它把用户态堆栈存放在一个扩展的帧中（保存在rt_sigframe数据结构中），这个帧也包含了与信号相关的siginfo_t表的内容。此外，该函数设置pretcode字段以使它指向vsyscall页中的__kernel_rt_sigreturn代码。

检查信号标志

建立了用户态堆栈以后，handle_signal()函数检查与信号相关的标志值。如果信号没有设置SA_NODEFER标志，在sigaction表中sa_mask字段对应的信号就必须在信号处理程序执行期间被阻塞：

```
if (!(ka->sa.sa_flags & SA_NODEFER)) {
    spin_lock_irq(&current->sighand->siglock);
    sigorsets(&current->blocked, &current->blocked, &ka->sa.sa_mask);
    sigaddset(&current->blocked, sig);
    recalc_sigpending(current);
    spin_unlock_irq(&current->sighand->siglock);
}
```

如前所述，recalc_sigpending()函数检查进程是否有非阻塞的挂起信号，并因此而设置它的TIF_SIGPENDING标志。

然后，handle_signal()返回到do_signal()，do_signal()也立即返回。

开始执行信号处理程序

do_signal()返回时，当前进程恢复它在用户态的执行。由于如前所述setup_frame()的准备，eip寄存器指向信号处理程序的第一条指令，而esp指向已推进用户态堆栈顶的帧的第一个内存单元。因此，信号处理程序被执行。

终止信号处理程序

信号处理程序结束时，返回栈顶地址，该地址指向帧的pretcode字段所引用的vsyscall页中的代码：

```
__kernel_sigreturn:
  popl %eax
  movl $__NR_sigreturn, %eax
  int $0x80
```

因此，信号编号（即帧的sig字段）被从栈中丢弃，然后调用sigreturn()系统调用。

sys_sigreturn()函数计算类型为pt_regs的数据结构regs的地址，其中pt_regs包含用户态进程的硬件上下文（参见第十章“参数传递”一节）。从存放在esp字段中的值，由此而导出并检查帧在用户态堆栈内的地址：

```
frame = (struct sigframe *)(regs.esp - 8);
if (verify_area(VERIFY_READ, frame, sizeof(*frame)) {
    force_sig(SIGSEGV, current);
    return 0;
}
```

然后，函数把调用信号处理程序前所阻塞的信号的位数组从帧的sc字段拷贝到current的blocked字段。结果，为信号处理函数的执行而屏蔽的所有信号解除阻塞。然后调用recalc_sigpending()函数。

此时，sys_sigreturn()函数必须把来自帧的sc字段的进程硬件上下文拷贝到内核态堆栈中，并从用户态堆栈中删除帧，这两个任务是通过调用restore_sigcontext()函数完成的。

像rt_sigqueueinfo()这样的系统调用需要与信号相关的siginfo_t表，如果信号是这种系统调用发送的，则其实现机制非常相似。扩展帧的pretcode字段指向vsyscall页面中的__kernel_rt_sigreturn代码，它依次调用rt_sigreturn()系统调用，其相应的sys_rt_sigreturn()服务例程把来自扩展帧的进程硬件上下文拷贝到内核态堆栈，并通过从用户态堆栈删除扩展帧以恢复用户态堆栈原来的内容。

系统调用的重新执行

内核并不总是能立即满足系统调用发出的请求，在这种情况发生时，把发出系统调用的进程置为TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE状态。

如果进程处于TASK_INTERRUPTIBLE状态，并且某个进程向它发送了一个信号，那么，内核不完成系统调用就把进程置成TASK_RUNNING状态（参看第四章的“从中断和异常返回”一节）。当切换回用户态时信号被传递给进程。当这种情况发生时，系统调用服务例程没有完成它的工作，但返回EINTR、ERESTARTNOHAND、ERESTART_RESTARTBLOCK、ERESTARTSYS或ERESTARTNOINTR错误码。

实际上，这种情况下用户态进程获得的唯一错误码是EINTR，这个错误码表示系统调用还没有执行完(应用程序的编写者可以测试这个错误码并决定是否重新发出系统调用)。内核内部使用剩余的错误码来指定信号处理程序结束后是否自动重新执行系统调用。

表11-11列出了与未完成的系统调用相关的出错码及这些出错码对信号三种可能的操作产生的影响。在表项中出现的几个术语的含义如下：

Terminate

不会自动重新执行系统调用；进程在int $0x80或sysenter指令紧接着的那条指令处将恢复它在用户态的执行，这时eax寄存器包含的值为-EINTR。

Reexecute

内核强迫用户态进程把系统调用号重新装入eax寄存器，并重新执行int $0x80指令或sysenter指令。进程意识不到这种重新执行，因此出错码也不传递给进程。

Depends

只有被传递信号的SA_RESTART标志被设置，才重新执行系统调用；否则，系统调用以-EINTR出错码结束。

表 11-11：系统调用的重新执行

错误码及其对系统调用执行的影响

信号			ERESTARTNOHAND	
操作	EINTR	ERESTARTSYS	ERESTART_RESTARTBLOCK[a]	ERESTARTNOINTR
Default	Terminate	Reexecute	Reexecute	Reexecute
Ignore	Terminate	Reexecute	Reexecute	Reexecute
Catch	Terminate	Depends	Terminate	Reexecute

a. 在允许重新开始执行系统调用的机制中，错误代码ERESTARTNOHAND和ERESTART_RESTARTBLOCK是有所不同的（见下面）。

当传递信号时，内核在试图重新执行一个系统调用前必须确定进程确实发出过这个系统调用。这就是regs硬件上下文的orig_eax字段起重要作用之处。让我们回顾一下中断或异常处理程序开始时是如何初始化这个字段的：

中断

这个字段包含的值为与中断相关的IRQ号减去256（参看第四章的“为中断处理程序保存寄存器的值”一节）。

0x80异常（或者sysenter）

这个字段包含系统调用号（参看第十章的“进入和退出系统调用”一节）。

其他异常

这个字段包含的值为-1(参看第四章的“为异常处理程序保存寄存器的值”一节)。

因此，orig_eax字段中的非负数意味着信号已经唤醒了在系统调用上睡眠的TASK_INTERRUPTIBLE进程。服务例程认识到系统调用曾被中断，并返回前面提到的某个错误码。

重新执行被未捕获信号中断的系统调用

如果信号被显式地忽略，或者如果它的缺省操作已被强制执行，do_signal()就分析系统调用的出错码，并如表11-11中所说明的那样决定是否重新自动执行未完成的系统调用。如果必须重新开始执行系统调用，那么do_signal()就修改regs硬件上下文，以便在进程返回到用户态时，eip指向int $0x80指令或sysenter指令，且eax包含系统调用号：

```
if (regs->orig_eax >= 0) {
    if (regs->eax == -ERESTARTNOHAND || regs->eax == -ERESTARTSYS ||
          regs->eax == -ERESTARTNOINTR) {
        regs->eax = regs->orig_eax;
        regs->eip -= 2;
    }
    if (regs->eax == -ERESTART_RESTARTBLOCK) {
        regs->eax = __NR_restart_syscall;
        regs->eip -= 2;
    }
}
```

把系统调用服务例程的返回代码赋给regs->eax字段(参见第十章“进入和退出系统调用”一节)。注意，int $0x80和sysreturn的长度都是两个字节，因此该函数从eip中减去2，使eip指向引起系统调用的指令。

ERESTART_RESTARTBLOCK错误代码是特殊的，因为eax寄存器中存放了restart_syscall()的系统调用号，因此，用户态进程不会重新执行被信号中断的同一个系统调用。这个错误代码仅用于与时间相关的系统调用，当重新执行这些系统调用时，应该调整它们的用户态参数。一个典型的例子是nanosleep()系统调用(参见第六章“动态定时器应用之一：nanosleep()系统调用”一节)：假设进程为了暂停执行20ms而调用了nanosleep()，而在10ms后出现了一个信号。如果像通常那样重新执行该系统调用(不调整其用户态参数)，那么总的时间延迟会超过30ms。

可以采用另一种方式，nanosleep()系统调用的服务例程把重新执行时所使用的特定服务例程的地址赋给current的thread_info结构中的restart_block字段，并在被中断时返回-ERESTART_RESTARTBLOCK。sys_restart_syscall()服务例程只执行特

定的nanosleep()的服务例程，考虑到原始系统调用的调用到重新执行之间有时间间隔，该服务例程调整这种延迟。

为所捕获的信号重新执行系统调用

如果信号被捕获，那么handle_signal()分析出错码，也可能分析sigaction表的SA_RESTART标志来决定是否必须重新执行未完成的系统调用：

```
if (regs->orig_eax >= 0) {
    switch (regs->eax) {
        case -ERESTART_RESTARTBLOCK:
        case -ERESTARTNOHAND:
            regs->eax = -EINTR;
            break;
        case -ERESTARTSYS:
            if (!(ka->sa.sa_flags & SA_RESTART)) {
                regs->eax = -EINTR;
                break;
            }
        /* fallthrough */
        case -ERESTARTNOINTR:
            regs->eax = regs->orig_eax;
            regs->eip -= 2;
    }
}
```

如果系统调用必须被重新开始执行，handle_signal()就与do_signal()完全一样地继续执行；否则，它向用户态进程返回一个出错码-EINTR。

与信号处理相关的系统调用

正如本章已提到的，在用户态运行的进程可以发送和接收信号。这意味着必须定义一组系统调用来完成这些操作。遗憾的是，由于历史的原因，已经存在几个具有相同功能的系统调用，因此，其中一些系统调用从未被调用。例如：系统调用sys_sigaction()和sys_rt_sigaction()几乎是相同的，因此C库中封装函数sigaction()调用sys_rt_sigaction()而不是sys_sigaction()。下面几节我们将描述其中一些最重要的系统调用。

kill()系统调用

一般用kill(pid,sig)系统调用向普通进程或多线程应用发送信号，其相应的服务例程是sys_kill()函数。整数参数pid的几个含义取决于它的值：

pid > 0

把sig信号发送到其PID等于pid的进程所属的线程组。

pid = 0

把sig信号发送到与调用进程同组的进程的所有线程组。

pid = -1

把信号发送到所有进程，除了*swapper*（PID 0）、*init*（PID 1）和current以外。

pid < -1

把信号发送到进程组 -pid中进程的所有线程组。

sys_kill()函数为信号建立最小的siginfo_t表，然后调用kill_something_info()函数：

```
info.si_signo = sig;
info.si_errno = 0;
info.si_code = SI_USER;
info._sifields._kill._pid = current->tgid;
info._sifields._kill._uid = current->uid;
return kill_something_info(sig, &info, pid);
```

kill_something_info还依次调用kill_proc_info()(通过group_send_sig_info()向一个单独的线程组发送信号)，或者调用kill_pg_info()(扫描目标进程组的所有进程，并为目标进程组中的每个进程调用send_sig_info())，或者为系统中的所有进程反复调用group_send_sig_info()(如果pid等于-1)。

kill()系统调用能发送任何信号，即使编号在32～64之间的实时信号。然而，我们在前面“产生信号”一节已看到，kill()系统调用不能确保把一个新的元素加入到目标进程的挂起信号队列，因此，挂起信号的多个实例可能被丢失。实时信号应当通过rt_sigqueueinfo()系统调用进行发送（参见后面“实时信号的系统调用”一节）。

System V和BSD Unix各种版本还有一个killpg()系统调用，它能显式地向一组进程发送信号。在Linux中，这个函数是作为一个库函数来实现的，其实现利用了kill()系统调用。另外一个变体是raise()，向当前进程（即正在执行该函数的进程）发送信号，该函数在Linux中是作为库函数来实现的。

tkill()和tgkill()系统调用

tkill()和tgkill()系统调用向线程组中的指定进程发送信号。所有遵循POSIX标准的pthread库的pthread_kill()函数，都是调用这两个函数中的任意一个向指定的轻量级进程发送信号。

tkill()系统调用需要两个参数：信号接收进程的pid PID和信号编号sig。sys_tkill()服务例程为siginfo表赋值、获取进程描述符地址、进行许可性检查

(如前面“group_send_sig_info()函数”一节第2步所执行的操作)，并调用specific_send_sig_info()发送信号。

tgkill()系统调用和tkill()有所不同，tgkill()还需要第三个参数：信号接收进程所在线程组的线程组ID(tgid)。sys_tgkill()服务例程执行的操作与sys_tkill()完全一样，不过还要检查信号接收进程是否确实属于线程组tgid。这个附加的检查解决了在向一个正在被杀死的进程发送消息时出现的竞争条件的问题：如果另外一个多线程应用正以足够快的速度创建轻量级进程，信号就可能被传递给一个错误的进程。因为线程组ID在多线程应用的整个生存期中是不会改变的，所以系统调用tgkill()解决了这个问题。

改变信号的操作

sigaction(sig,act,oact)系统调用允许用户为信号指定一个操作。当然，如果没有自定义的信号操作，那么内核执行与传递的信号相关的缺省操作。

相应的sys_sigaction()服务例程作用于两个参数：sig信号编号和类型为old_sigaction的act表(表示新的操作)。第三个可选的输出参数oact可以用来获得与信号相关的以前的操作。(old_sigaction数据结构包括与sigaction结构相同的字段，只是字段的顺序不同，在前面“与信号相关的数据结构”一节对sigaction结构进行过说明)。

这个函数首先检查act地址的有效性。然后用*act相应的字段填充类型为k_sigaction的new_ka局部变量的sa_handler、sa_flags和sa_mask字段：

```
__get_user(new_ka.sa.sa_handler, &act->sa_handler);
__get_user(new_ka.sa.sa_flags, &act->sa_flags);
__get_user(mask, &act->sa_mask);
siginitset(&new_ka.sa.sa_mask, mask);
```

函数还调用do_sigaction()把新的new_ka表拷贝到current->sig->action的在sig-1位置的表项中(信号的编号大于在数组中的位置，因为没有0信号)：

```
k = &current->sig->action[sig-1];
if (act) {
    *k = *act;
    sigdelsetmask(&k->sa.sa_mask, sigmask(SIGKILL) | sigmask(SIGSTOP));
    if (k->sa.sa_handler == SIG_IGN || (k->sa.sa_handler == SIG_DFL &&
     (sig==SIGCONT || sig==SIGCHLD || sig==SIGWINCH || sig==SIGURG))) {
        rm_from_queue(sigmask(sig), &current->signal->shared_pending);
        t = current;
        do {
            rm_from_queue(sigmask(sig), &current->pending);
            recalc_sigpending_tsk(t);
            t = next_thread(t);
```

```
            } while (t != current);
        }
    }
```

POSIX标准规定，当缺省操作是“Ignore”时，把信号操作设置成SIG_IGN或SIG_DFL将引起同类型的的任一挂起信号被丢弃。此外还要注意，对信号处理程序来说，不论请求屏蔽的信号是什么，SIGKILL和SIGSTOP从不被屏蔽。

sigaction()系统调用还允许用户初始化表sigaction的sa_flags字段。在表11-6（本章前面）中，我们列出了这个字段的可能取值及其相关的含义。

原来的System V Unix变体提供了signal()系统调用，它仍由编程人员广泛使用。新近的C库调用rt_sigaction()实现了signal()。不过，Linux依然支持原来的C库，并提供了sys_signal()服务例程：

```
new_sa.sa.sa_handler = handler;
new_sa.sa.sa_flags = SA_ONESHOT | SA_NOMASK;
ret = do_sigaction(sig, &new_sa, &old_sa);
return ret ? ret : (unsigned long)old_sa.sa.sa_handler;
```

检查挂起的阻塞信号

sigpending()系统调用允许进程检查挂起的阻塞信号的集合，也就是说，检查信号被阻塞时已产生的那些信号。相应的服务例程sys_sigpending()只作用于一个参数set，即用户变量的地址，必须将位数组拷贝到这个变量中：

```
sigorsets(&pending, &current->pending.signal,
                    &current->signal->shared_pending.signal);
sigandsets(&pending, &current->blocked, &pending);
copy_to_user(set, &pending, 4);
```

修改阻塞信号的集合

sigprocmask()系统调用允许进程修改阻塞信号的集合。这个系统调用只应用于常规信号（非实时信号）。相应的sys_sigprocmask()服务例程作用于三个参数：

oset

进程地址空间的一个指针，指向存放以前位掩码的一个位数组。

set

进程地址空间的一个指针，指向包含新位掩码的位数组。

how

一个标志，可以有下列的一个值：

SIG_BLOCK

*set 位掩码数组，指定必须加到阻塞信号的位掩码数组中的信号。

SIG_UNBLOCK

*set 位掩码数组，指定必须从阻塞信号的位掩码数组中删除的信号。

SIG_SETMASK

*set 位掩码数组，指定阻塞信号新的位掩码数组。

sys_sigprocmask()调用 copy_from_user()把 set 参数所指向的值拷贝到局部变量 new_set 中，并把 current 标准阻塞信号的位掩码数组拷贝到 old_set 局部变量中。然后根据 how 标志来指定这两个变量的值：

```
if (copy_from_user(&new_set, set, sizeof(*set)))
    return -EFAULT;
new_set &= ~(sigmask(SIGKILL)|sigmask(SIGSTOP));
old_set = current->blocked.sig[0];
if (how == SIG_BLOCK)
    sigaddsetmask(&current->blocked, new_set);
else if (how == SIG_UNBLOCK)
    sigdelsetmask(&current->blocked, new_set);
else if (how == SIG_SETMASK)
    current->blocked.sig[0] = new_set;
else
    return -EINVAL;
recalc_sigpending(current);
if (oset && copy_to_user(oset, &old_set, sizeof(*oset)))
        return -EFAULT;
return 0;
```

挂起进程

sigsuspend()系统调用把进程置为 TASK_INTERRUPTIBLE 状态，当然这是把 mask 参数指向的位掩码数组所指定的标准信号阻塞以后设置的。只有当一个非忽略、非阻塞的信号发送到进程以后，进程才被唤醒。

相应的 sys_sigsuspend()服务例程执行下列这些语句：

```
mask &= ~(sigmask(SIGKILL) | sigmask(SIGSTOP));
saveset = current->blocked;
siginitset(&current->blocked, mask);
recalc_sigpending(current);
regs->eax = -EINTR;
while (1) {
    current->state = TASK_INTERRUPTIBLE;
    schedule( );
    if (do_signal(regs, &saveset))
```

```
        return -EINTR;
    }
```

schedule()函数选择另一个进程运行。当发出sigsuspend()系统调用的进程又开始执行时，sys_sigsuspend()调用do_signal()函数来传递唤醒了该进程的信号。如果do_signal()的返回值为1，则不忽略这个信号。因此，这个系统调用返回-EINTR出错码后终止。

sigsuspend()系统调用可能看似多余，因为sigprocmask()和sleep()的组合执行显然能产生同样的效果。但这并不正确：这是因为进程可能在任何时候交错执行，你必须意识到调用一个系统调用执行操作A，紧接着又调用另一个系统调用执行操作B，并不等于调用一个单独的系统调用执行操作A，然后执行操作B。

在这种特殊情况中，sigprocmask()可以在调用sleep()之前解除对所传递信号的阻塞。如果这种情况发生，进程就可以一直停留在TASK_INTERRUPTIBLE状态，等待已被传递的信号。另一方面，在解除阻塞之后、schedule()调用之前，因为其他进程在这个时间间隔内无法获得CPU，因此，sigsuspend()系统调用不允许信号被发送。

实时信号的系统调用

因为前面所提到的系统调用只应用到标准信号，因此，必须引入另外的系统调用来允许用户态进程处理实时信号。

实时信号的几个系统调用（rt_sigaction()、rt_sigpending()、rt_sigprocmask()及rt_sigsuspend()）与前面描述的类似，因此不再进一步讨论。出于同样的理由，我们也不进一步讨论处理实时信号队列的两个系统调用：

rt_sigqueueinfo()
: 发送一个实时信号以便把它加入到目标进程的共享挂起信号队列中。一般通过标准库函数sigqueue()调用rt_sigqueueinfo()。

rt_sigtimedwait()
: 把阻塞的挂起信号从队列中删除而不传递它，并向调用者返回信号编号；如果没有阻塞的信号在挂起，就把当前进程挂起一个固定的时间间隔。一般通过标准库函数sigwaitinfo()和sigtimedwait()调用rt_sigtimedwait()。

第十二章



虚拟文件系统

Linux成功的关键因素之一是它具有与其他操作系统和谐共存的能力。你能够透明地安装具有其他操作系统文件格式的磁盘或分区，这些操作系统如Windows、其他版本的Unix，甚至像Amiga那样的市场占有率很低的系统。通过所谓的虚拟文件系统概念，Linux使用与其他Unix变体相同的方式设法支持多种文件系统类型。

虚拟文件系统所隐含的思想是把表示很多不同种类文件系统的共同信息放入内核；其中有一个字段或函数来支持Linux所支持的所有实际文件系统所提供的任何操作。对所调用的每个读、写或其他函数，内核都能把它们替换成支持本地Linux文件系统、NTFS文件系统，或者文件所在的任何其他文件系统的实际函数。

本章讨论Linux虚拟文件系统的设计目标、结构及其实现。集中讨论五个Unix标准文件类型中的三个文件类型，即普通文件、目录文件和符号链接文件。设备文件将在第十三章中进行介绍，而管道文件会在第十九章中进行讨论。为了进一步说明实际文件系统如何工作，将在第十八章中对第二扩展文件系统（Second Extended Filesystem）进行讨论（几乎所有的Linux系统都使用了Ext2）。

虚拟文件系统（VFS）的作用

虚拟文件系统（*Virtual Filesystem*）也可以称之为虚拟文件系统转换（Virtual Filesystem Switch，VFS），是一个内核软件层，用来处理与Unix标准文件系统相关的所有系统调用。其健壮性表现在能为各种文件系统提供一个通用的接口。

例如，假设一个用户输入以下shell命令：

```
$ cp /floppy/TEST /tmp/test
```

其中 */floppy* 是 MS-DOS 磁盘的一个安装点，而 */tmp* 是一个标准的第二扩展文件系统（second Extended Filesystom，Ext2）的目录。正如图 12-1 (a)所示，VFS 是用户的应用程序与文件系统实现之间的抽象层。因此，*cp* 程序并不需要知道*/floppy/TEST*和*/tmp/test* 是什么文件系统类型。相反，*cp* 程序直接与 VFS 交互，这是通过 Unix 程序设计人员都熟悉的普通系统调用来进行的(参见第一章中的“文件操作的系统调用“一节）。*cp* 的执行代码如图 12-1(b)所示。

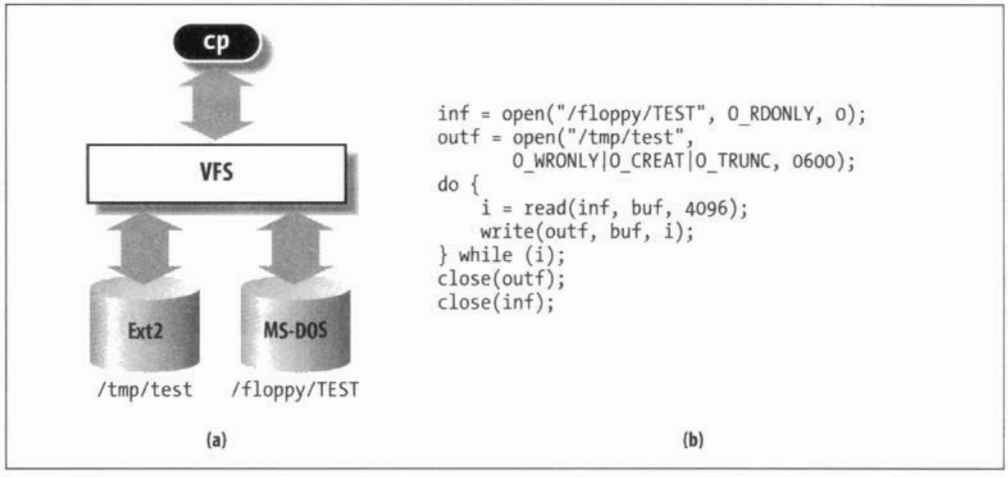


图 12-1：VFS 在一个简单的文件复制操作中的作用

VFS 支持的文件系统可以划分为三种主要类型：

磁盘文件系统

这些文件系统管理在本地磁盘分区中可用的存储空间或者其他可以起到磁盘作用的设备（比如说一个 USB 闪存）。VFS 支持的基于磁盘的某些著名文件系统还有：

- Linux使用的文件系统，如广泛使用的第二扩展文件系统（Ext2），新近的第三扩展文件系统(Third Extended Filesystem,Ext3)及Reiser文件系统(ReiserFS)（注 1）。
- Unix 家族的文件系统，如sysv 文件系统(System V、Coherent、Xenix)、UFS (BSD、Solaris、NEXTSTEP)，MINIX 文件系统及 VERITAS VxFS (SCO UnixWare)。

注 1：　尽管这些文件系统诞生于 Linux，但已经移植到其他几个操作系统中。

- 微软公司的文件系统，如MS-DOS、VFAT（Windows 95及随后的版本）及NTFS（Windows NT以及随后的版本）。
- ISO9660 CD-ROM文件系统（以前的High Sierra文件系统）和通用磁盘格式(UDF)的DVD文件系统。
- 其他有专利权的文件系统，如HPFS（IBM公司的OS/2）、HFS（苹果公司的Macintosh）、AFFS（Amiga公司的快速文件系统）以及ADFS（Acorn公司的磁盘文件归档系统）。
- 起源于非Linux系统的其他日志文件系统，如IBM的JFS和SGI的XFS。

网络文件系统

这些文件系统允许轻易地访问属于其他网络计算机的文件系统所包含的文件。虚拟文件系统所支持的一些著名的网络文件系统有：NFS、Coda、AFS (Andrew文件系统)、CIFS（用于Microsoft Windows的通用网络文件系统）以及NCP (Novell公司的NetWare Core Protocol)。

特殊文件系统

这些文件系统不管理本地或者远程磁盘空间。/proc文件系统是特殊文件系统的一个典型范例（参见稍后“特殊文件系统“一节）。

由于篇幅所限，本书只详细描述Ext2和Ext3文件系统（参见第十八章），对其他文件系统将不做介绍。

在第一章“Unix文件系统概述“一节中曾提到，Unix的目录建立了一棵根目录为“/“的树。根目录包含在根文件系统（*root filesystem*）中，在Linux中这个根文件系统通常就是Ext2或Ext3类型。其他所有的文件系统都可以被“安装“在根文件系统的子目录中（注2）。

基于磁盘的文件系统通常存放在硬件块设备中，如硬盘、软盘或者CD-ROM。Linux VFS的一个有用特点是能够处理如*/dev/loop0*这样的虚拟块设备，这种设备可以用来安装普通文件所在的文件系统。作为一种可能的应用，用户可以保护自己的私有文件系统，这可以通过把自己文件系统的加密版本存放在一个普通文件中来实现。

注2：当一个文件系统被安装在某一个目录上时，在父文件系统中的目录内容不再是可访问的，因为任何路径（包括安装点），都将引用已安装的文件系统。但是，当被安装文件系统卸载时，原目录的内容又可再现。这种令人惊讶的Unix文件系统特点可以由系统管理员用来隐藏文件，他们只需把一个文件系统安装在要隐藏文件的目录中即可。

第一个虚拟文件系统包含在1986年由Sun公司发布的SunOS操作系统中。从那时起，多数Unix文件系统都包含VFS。然而，Linux的VFS支持最广泛的文件系统。

通用文件模型

VFS所隐含的主要思想在于引入了一个通用的文件模型（*common file model*），这个模型能够表示所有支持的文件系统。该模型严格反映传统Unix文件系统提供的文件模型。这并不奇怪，因为Linux希望以最小的额外开销运行它的本地文件系统。不过，要实现每个具体的文件系统，必须将其物理组织结构转换为虚拟文件系统的通用文件模型。

例如，在通用文件模型中，每个目录被看作一个文件，可以包含若干文件和其他的子目录。但是，存在几个非Unix的基于磁盘的文件系统，它们利用文件分配表(File Allocation Table，FAT)存放每个文件在目录树中的位置，在这些文件系统中，存放的是目录而不是文件。为了符合VFS的通用文件模型，对上述基于FAT的文件系统的实现，Linux必须在必要时能够快速建立对应于目录的文件。这样的文件只作为内核内存的对象而存在。

从本质上说，Linux内核不能对一个特定的函数进行硬编码来执行诸如`read()`或`ioctl()`这样的操作，而是对每个操作都必须使用一个指针，指向要访问的具体文件系统的适当函数。

为了进一步说明这一概念，参见图12-1，其中显示了内核如何把`read()`转换为专对MS-DOS文件系统的一个调用。应用程序对`read()`的调用引起内核调用相应的`sys_read()`服务例程，这与其他系统调用完全类似。我们在本章后面会看到，文件在内核内存中是由一个`file`数据结构来表示的。这种数据结构中包含一个称为`f_op`的字段，该字段中包含一个指向专对MS-DOS文件的函数指针，当然还包括读文件的函数。`sys_read()`查找到指向该函数的指针，并调用它。这样一来，应用程序的read()就被转化为相对间接的调用：

```
file->f_op->read(...);
```

与之类似，`write()`操作也会引发一个与输出文件相关的Ext2写函数的执行。简而言之，内核负责把一组合适的指针分配给与每个打开文件相关的`file`变量，然后负责调用针对每个具体文件系统的函数（由`f_op`字段指向）。

你可以把通用文件模型看作是面向对象的，在这里，对象是一个软件结构，其中既定义了数据结构也定义了其上的操作方法。出于效率的考虑，Linux的编码并未采用面向对象的程序设计语言（比如C++）。因此对象作为普通的C数据结构来实现，数据结构中指向函数的字段就对应于对象的方法。

通用文件模型由下列对象类型组成：

超级块对象（superblock object）

存放已安装文件系统的有关信息。对基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件系统控制块（*filesystem control block*）。

索引节点对象（inode object）

存放关于具体文件的一般信息。对基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件控制块（*file control block*）。每个索引节点对象都有一个索引节点号，这个节点号唯一地标识文件系统中的文件。

文件对象(file object)

存放打开文件与进程之间进行交互的有关信息。这类信息仅当进程访问文件期间存在于内核内存中。

目录项对象(dentry object)

存放目录项（也就是文件的特定名称）与对应文件进行链接的有关信息。每个磁盘文件系统都以自己特有的方式将该类信息存在磁盘上。

如图12-2所示是一个简单的示例，说明进程怎样与文件进行交互。三个不同进程已经打开同一个文件，其中两个进程使用同一个硬链接。在这种情况下，其中的每个进程都使用自己的文件对象，但只需要两个目录项对象，每个硬链接对应一个目录项对象。这两个目录项对象指向同一个索引节点对象，该索引节点对象标识超级块对象，以及随后的普通磁盘文件。

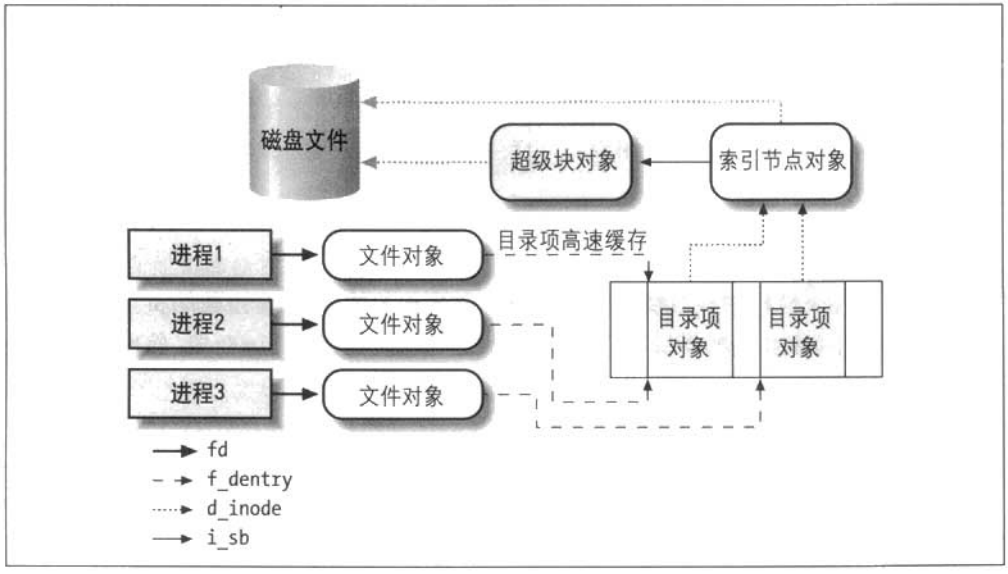


图 12-2：进程与 VFS 对象之间的交互

VFS除了能为所有文件系统的实现提供一个通用接口外，还具有另一个与系统性能相关的重要作用。最近最常使用的目录项对象被放在所谓目录项高速缓存（*dentry cache*）的磁盘高速缓存中，以加速从文件路径名到最后一个路径分量的索引节点的转换过程。

一般说来，磁盘高速缓存（*disk cache*）属于软件机制，它允许内核将原本存在磁盘上的某些信息保存在RAM中，以便对这些数据的进一步访问能快速进行，而不必慢速访问磁盘本身。

注意，磁盘高速缓存不同于硬件高速缓存或内存高速缓存，后两者都与磁盘或其他设备无关。硬件高速缓存是一个快速静态RAM，它加快了直接对慢速动态RAM的请求（参见第二章中的“硬件高速缓存”一节）。内存高速缓存是一种软件机制，引入它是为了绕过内核内存分配器（参见第八章中的“slab分配器”一节）。

除了目录项高速缓存和索引结点高速缓存之外，Linux还使用其他磁盘高速缓存。其中最重要的一种就是所谓的页高速缓存，我们将在第十五章中进行详细介绍。

VFS所处理的系统调用

表12-1列出了VFS的系统调用，这些系统调用涉及文件系统、普通文件、目录文件以及符号链接文件。另外还有少数几个由VFS处理的其他系统调用，诸如ioperm()、ioctl()、pipe()和mknod()，涉及设备文件和管道文件，这些将在后续章节中讨论。最后一组由VFS处理的系统调用，诸如socket()、connect()和bind()属于套接字系统调用，并用于实现网络功能。与表12-1列出的系统调用对应的一些内核服务例程，我们会在本章或第十八章中陆续进行讨论。

表12-1：由VFS处理的一些系统调用

系统调用名	说明
mount() umount() umount2()	安装/卸载文件系统
sysfs()	获取文件系统信息
statfs() fstatfs() statfs64()fstatfs64() ustat()	获取文件系统统计信息
chroot() pivot_root()	更改根目录
chdir() fchdir() getcwd()	对当前目录进行操作
mkdir() rmdir()	创建/删除目录
getdents() getdents64() readdir() link() unlink() rename() lookup_dcookie()	对目录项进行操作
readlink() symlink()	对软链接进行操作

表12-1：由VFS处理的一些系统调用（续）

系统调用名	说明
chown() fchown() lchown() chown16() fchown16() lchown16()	更改文件所有者性
chmod() fchmod() utime()	更改文件属性
stat() fstat() lstat() access() oldstat() oldfstat() oldlstat() stat64() lstat64() fstat64()	读取文件状态
open() close() creat() umask()	打开 / 关闭 / 创建文件
dup() dup2() fcntl() fcntl64()	对文件描述符进行操作
select() poll()	等待一组文件描述符上发生的事件
truncate() ftruncate() truncate64() ftruncate64()	更改文件长度
lseek() _llseek()	更改文件指针
read() write() readv() writev() sendfile() sendfile64() readahead()	进行文件 I/O 操作
io_setup() io_submit() io_getevents() io_cancel() io_destroy()	异步 I/O（允许多个读和写请求）
pread64() pwrite64()	搜索并访问文件
mmap() mmap2() munmap() madvise() mincore() remap_file_pages()	处理文件内存映射
fdatasync() fsync() sync() msync()	同步文件数据
flock()	处理文件锁
setxattr() lsetxattr() fsetxattr() getxattr() lgetxattr() fgetxattr() listxattr() llistxattr() flistxattr() removexattr() lremovexattr() fremovexattr()	处理文件扩展属性

前面我们已经提到，VFS是应用程序和具体文件系统之间的一层。不过，在某些情况下，一个文件操作可能由VFS本身去执行，无需调用低层函数。例如，当某个进程关闭一个打开的文件时，并不需要涉及磁盘上的相应文件，因此VFS只需释放对应的文件对象。类似地，当系统调用lseek()修改一个文件指针，而这个文件指针是打开文件与进程交互所涉及的一个属性时，VFS就只需修改对应的文件对象，而不必访问磁盘上的文件，因此，无需调用具体文件系统的函数。从某种意义上说，可以把VFS看成“通用“文件系统，它在必要时依赖某种具体文件系统。

VFS 的数据结构

每个VFS对象都存放在一个适当的数据结构中，其中包括对象的属性和指向对象方法表的指针。内核可以动态地修改对象的方法，因此可以为对象建立专用的行为。下面几节详细介绍 VFS 的对象及其内在关系。

超级块对象

超级块对象由 super_block 结构组成，表 12-2 列举了其中的字段。

表12-2：超级块对象的字段

类型	字段	说明
struct list_head	s_list	指向超级块链表的指针
dev_t	s_dev	设备标识符
unsigned long	s_blocksize	以字节为单位的块大小
unsigned long	s_old_blocksize	基本块设备驱动程序中提到的以字节为单位的块大小
unsigned char	s_blocksize_bits	以位为单位的块大小
unsigned char	s_dirt	修改（脏）标志
unsigned long long	s_maxbytes	文件的最长长度
struct file_system_type *	s_type	文件系统类型
struct super_operations *	s_op	超级块方法
struct dquot_operations *	dq_op	磁盘限额处理方法
struct quotactl_ops *	s_qcop	磁盘限额管理方法
struct export_operations	s_export_op	网络文件系统使用的输出操作
unsigned long	s_flags	安装标志
unsigned long	s_magic	文件系统的魔数
struct dentry *	s_root	文件系统根目录的目录项对象
struct rw_semaphore	s_umount	卸载所用的信号量
struct semaphore	s_lock	超极块信号量
int	s_count	引用计数器
int	s_syncing	表示对超级块的索引节点进行同步的标志
int	s_need_sync_fs	对超级块的已安装文件系统进行同步的标志

表12-2：超级块对象的字段（续）

类型	字段	说明
atomic_t	s_active	次级引用计数器
void *	s_security	指向超级块安全数据结构的指针
struct xattr_handler **	s_xattr	指向超级块扩展属性结构的指针
struct list_head	s_inodes	所有索引节点的链表
struct list_head	s_dirty	改进型索引节点的链表
struct list_head	s_io	等待被写入磁盘的索引节点的链表
struct hlist_head	s_anon	用于处理远程网络文件系统的匿名目录项的链表
struct list_head	s_files	文件对象的链表
struct block_device *	s_bdev	指向块设备驱动程序描述符的指针
struct list_head	s_instances	用于给定文件系统类型的超级块对象链表的指针（参见后面的“文件系统类型注册“一节）
struct quota_info	s_dquot	磁盘限额的描述符
int	s_frozen	冻结文件系统时使用的标志（强制置于一致状态）
wait_queue_head_t	s_wait_unfrozen	进程挂起的等待队列，直到文件系统被解冻
char[]	s_id	包含超级块的块设备名称
void *	s_fs_info	指向特定文件系统的超级块信息的指针
struct semaphore	s_vfs_rename_sem	当VFS通过目录重命名文件时使用的信号量
u32	s_time_gran	时间戳的粒度（纳秒级）

所有超级块对象都以双向循环链表的形式链接在一起。链表中第一个元素用super_blocks变量来表示，而超级块对象的s_list字段存放指向链表相邻元素的指针。sb_lock自旋锁保护链表免受多处理器系统上的同时访问。

s_fs_info字段指向属于具体文件系统的超级块信息，例如，我们在第十八章将会看到，假如超级块对象指的是Ext2文件系统，该字段就指向ext2_sb_info数据结构，该结构包括磁盘分配位掩码和其他与VFS的通用文件模型无关的数据。

通常，为了效率起见，由s_fs_info字段所指向的数据被复制到内存。任何基于磁盘的文件系统都需要访问和更改自己的磁盘分配位图，以便分配或释放磁盘块。VFS允许这些文件系统直接对内存超级块的s_fs_info字段进行操作，而无需访问磁盘。

但是，这种方法带来一个新问题：有可能VFS超级块最终不再与磁盘上相应的超级块同步。因此，有必要引入一个s_dirt标志来表示该超级块是否是脏的——那磁盘上的数据是否必须要更新。缺乏同步还会导致产生我们熟悉的一个问题：当一台机器的电源突然断开而用户来不及正常关闭系统时，就会出现文件系统崩溃。我们将会在第十五章的“把脏页写入磁盘“一节中看到，Linux是通过周期性地将所有“脏“的超级块写回磁盘来减少该问题带来的危害。

与超级块关联的方法就是所谓的超级块操作。这些操作是由数据结构super_operations来描述的，该结构的起始地址存放在超级块的s_op字段中。

每个具体的文件系统都可以定义自己的超级块操作。当VFS需要调用其中一个操作时，比如说read_inode()，它执行下列操作：

```
sb->s_op->read_inode(inode);
```

这里sb存放所涉及超级块对象的地址。super_operations表的read_inode字段存放这一函数的地址，因此，这一函数被直接调用。

让我们简要描述一下超级块操作，其中实现了一些高级操作，比如删除文件或安装磁盘。下面这些操作按照它们在super_operation表中出现的顺序来排列：

alloc_inode(sb)
: 为索引节点对象分配空间，包括具体文件系统的数据所需要的空间。

destroy_inode(inode)
: 撤消索引节点对象，包括具体文件系统的数据。

read_inode(inode)
: 用磁盘上的数据填充以参数传递过来的索引节点对象的字段；索引节点对象的i_ino字段标识从磁盘上要读取的具体文件系统的索引节点。

dirty_inode(inode)
: 当索引节点标记为修改（脏）时调用。像ReiserFS和Ext3这样的文件系统用它来更新磁盘上的文件系统日志。

write_inode(inode,flag)
: 用通过传递参数指定的索引节点对象的内容更新一个文件系统的索引节点。索引节点对象的i_ino字段标识所涉及磁盘上文件系统的索引节点。flag参数表示I/O操作是否应当同步。

put_inode(inode)

释放索引节点时调用（减少该节点引用计数器值）以执行具体文件系统操作。

drop_inode(inode)

在即将撤消索引节点时调用——也就是说，当最后一个用户释放该索引节点时；实现该方法的文件系统通常使用 generic_drop_inode()函数。该函数从 VFS 数据结构中移走对索引节点的每一个引用，如果索引节点不再出现在任何目录中，则调用超级块方法 delete_inode 将它从文件系统中删除。

delete_inode(inode)

在必须撤消索引节点时调用。删除内存中的 VFS 索引节点和磁盘上的文件数据及元数据。

put_super(super)

释放通过传递的参数指定的超级块对象（因为相应的文件系统被卸载）。

write_super(super)

用指定对象的内容更新文件系统的超级块。

sync_fs(sb, wait)

在清除文件系统来更新磁盘上的具体文件系统数据结构时调用(由日志文件系统使用)。

write_super_lockfs(super)

阻塞对文件系统的修改并用指定对象的内容更新超级块。当文件系统被冻结时调用该方法，例如，由逻辑卷管理器驱动程序（LVM）调用。

unlockfs(super)

取消由 write_super_lockfs()超级块方法实现的对文件系统更新的阻塞。

statfs(super, buf)

将文件系统的统计信息返回，填写在 buf 缓冲区中。

remount_fs(super, flags, data)

用新的选项重新安装文件系统（当某个安装选项必须被修改时被调用）。

clear_inode(inode)

当撤消磁盘索引节点执行具体文件系统操作时调用。

umount_begin(super)

中断一个安装操作，因为相应的卸载操作已经开始（只在网络文件系统中使用）。

show_options(seq_file, vfsmount)

用来显示特定文件系统的选项。

quota_read(super, type, data, size, offset)

限额系统使用该方法从文件中读取数据，该文件详细说明了所在文件系统的限制（注3）。

quota_write(super, type, data, size, offset)

限额系统使用该方法将数据写入文件中，该文件详细说明了所在文件系统的限制。

前述的方法对所有可能的文件系统类型均是可用的。但是，只有其中的一个子集应用到每个具体的文件系统；未实现的方法对应的字段置为NULL。注意，系统没有定义get_super方法来读超级块，那么，内核如何能够调用一个对象的方法而从磁盘读出该对象？我们将在描述文件系统类型的另一个对象中找到等价的get_sb方法（参见后面的“文件系统类型注册“一节）。

索引节点对象

文件系统处理文件所需要的所有信息都放在一个名为索引节点的数据结构中。文件名可以随时更改，但是索引节点对文件是唯一的，并且随文件的存在而存在。内存中的索引节点对象由一个inode数据结构组成，其字段如表12-3所示。

表12-3：索引节点对象的字段

类型	字段	说明
Struct hlist_node	i_hash	用于散列链表的指针
struct list_head	i_list	用于描述索引节点当前状态的链表的指针
struct list_head	i_sb_list	用于超级块的索引节点链表的指针
struct list_head	i_dentry	引用索引节点的目录项对象链表的头
unsigned long	i_ino	索引节点号
atomic_t	i_count	引用计数器
umode_t	i_mode	文件类型与访问权限
unsigned int	i_nlink	硬链接数目
uid_t	i_uid	所有者标识符

注3：限额系统（*quota system*）为每个用户和（或）组限制了它们在给定文件系统上所能使用的空间大小（参见quotactl()系统调用）。

表 12-3：索引节点对象的字段（续）

类型	字段	说明
gid_t	i_gid	组标识符
dev_t	i_rdev	实设备标识符
loff_t	i_size	文件的字节数
struct timespec	i_atime	上次访问文件的时间
struct timespec	i_mtime	上次写文件的时间
struct timespec	i_ctime	上次修改索引节点的时间
unsigned int	i_blkbits	块的位数
unsigned long	i_blksize	块的字节数
unsigned long	i_version	版本号（每次使用后自动递增）
unsigned long	i_blocks	文件的块数
unsigned short	i_bytes	文件中最后一个块的字节数
unsigned char	i_sock	如果文件是一个套接字则为非零
spinlock_t	i_lock	保护索引节点一些字段的自旋锁
struct semaphore	i_sem	索引节点信号量
struct rw_semaphore	i_alloc_sem	在直接 I/O 文件操作中避免出现竞争条件的读 / 写信号量
struct inode_operations *	i_op	索引节点的操作
struct file_operations *	i_fop	缺省文件操作
struct super_block *	i_sb	指向超级块对象的指针
struct file_lock *	i_flock	指向文件锁链表的指针
struct address_space *	i_mapping	指向address_space对象的指针（参见第十五章）
struct address_space	i_data	文件的 address_space 对象
struct dquot * []	i_dquot	索引节点磁盘限额
struct list_head	i_devices	用于具体的字符或块设备索引节点链表的指针（参见第十三章）
struct pipe_inode_info *	i_pipe	如果文件是一个管道则使用它（参见第十九章）
struct block_device *	i_bdev	指向块设备驱动程序的指针
struct cdev * i_cdev		指向字符设备驱动程序的指针
int	i_cindex	拥有一组次设备号的设备文件的索引

表 12-3：索引节点对象的字段（续）

类型	字段	说明
_ _u32	i_generation	索引节点版本号（由某些文件系统使用）
unsigned long	i_dnotify_mask	目录通知事件的位掩码
struct dnotify_struct *	i_dnotify	用于目录通知
unsigned long	i_state	索引节点的状态标志
unsigned long	dirtied_when	索引节点的弄脏时间（以节拍为单位）
unsigned int	i_flags	文件系统的安装标志
atomic_t	i_writecount	用于写进程的引用计数器
void *	i_security	指向索引节点安全结构的指针
void *	u.generic_ip	指向私有数据的指针
seqcount_t	i_size_seqcount	SMP 系统为 i_size 字段获取一致值时使用的顺序计数器

每个索引节点对象都会复制磁盘索引节点包含的一些数据，比如分配给文件的磁盘块数。如果 i_state 字段的值等于 I_DIRTY_SYNC、I_DIRTY_DATASYNC 或 I_DIRTY_PAGES，该索引节点就是“脏“的，也就是说，对应的磁盘索引节点必须被更新。I_DIRTY 宏可以用来立即检查这三个标志的值（详细内容参见后面）。i_state 字段的其他值有 I_LOCK（涉及的索引节点对象处于 I/O 传送中）、I_FREEING（索引节点对象正在被释放）、I_CLEAR（索引节点对象的内容不再有意义）以及 I_NEW（索引节点对象已经分配但还没有用从磁盘索引节点读取来的数据填充）。

每个索引节点对象总是出现在下列双向循环链表的某个链表中（所有情况下，指向相邻元素的指针存放在 i_list 字段中）：

- 有效未使用的索引节点链表，典型的如那些镜像有效的磁盘索引节点，且当前未被任何进程使用。这些索引节点不为脏，且它们的 i_count 字段置为 0。链表中的首元素和尾元素是由变量 inode_unused 的 next 字段和 prev 字段分别指向的。这个链表用作磁盘高速缓存。
- 正在使用的索引节点链表，也就是那些镜像有效的磁盘索引节点，且当前被某些进程使用。这些索引节点不为脏，但它们的 i_count 字段为正数。链表中的首元素和尾元素是由变量 inode_in_use 引用的。
- 脏索引节点的链表。链表中的首元素和尾元素是由相应超级块对象的 s_dirty 字段引用的。

这些链表都是通过适当的索引节点对象的i_list字段链接在一起的。

此外，每个索引节点对象也包含在每文件系统（*per-filesystem*）的双向循环链表中，链表的头存放在超级块对象的s_inodes字段中；索引节点对象的i_sb_list字段存放了指向链表相邻元素的指针。

最后，索引节点对象也存放在一个称为inode_hashtable的散列表中。散列表加快了对索引节点对象的搜索，前提是系统内核要知道索引节点号及文件所在文件系统对应的超级块对象的地址。由于散列技术可能引发冲突，所以索引节点对象包含一个i_hash字段，该字段中包含向前和向后的两个指针，分别指向散列到同一地址的前一个索引节点和后一个索引节点；该字段因此创建了由这些索引节点组成的一个双向链表。

与索引节点对象关联的方法也叫索引节点操作。它们由inode_operations结构来描述，该结构的地址存放在i_op字段中。以下是索引节点的操作，以它们在inode_operations表中出现的次序来排列：

`create(dir, dentry, mode, nameidata)`

在某一目录下，为与目录项对象相关的普通文件创建一个新的磁盘索引节点。

`lookup(dir, dentry, nameidata)`

为包含在一个目录项对象中的文件名对应的索引节点查找目录。

`link(old_dentry, dir, new_dentry)`

创建一个新的名为new_dentry的硬链接，它指向dir目录下名为old_dentry的文件。

`unlink(dir, dentry)`

从一个目录中删除目录项对象所指定文件的硬链接。

`symlink(dir, dentry, symname)`

在某个目录下，为与目录项对象相关的符号链接创建一个新的索引节点。

`mkdir(dir, dentry, mode)`

在某个目录下，为与目录项对象相关的目录创建一个新的索引节点。

`rmdir(dir, dentry)`

从一个目录删除子目录，子目录的名称包含在目录项对象中。

`mknod(dir, dentry, mode, rdev)`

在某个目录中，为与目录项对象相关的特定文件创建一个新的磁盘索引节点。其中参数mode和rdev分别表示文件的类型和设备的主次设备号。

rename(old_dir, old_dentry, new_dir, new_dentry)

将old_dir目录下由old_entry标识的文件移到new_dir目录下。新文件名包含在new_dentry指向的目录项对象中。

readlink(dentry, buffer, buflen)

将目录项所指定的符号链接中对应的文件路径名拷贝到buffer所指定的用户态内存区。

follow_link(inode, nameidata)

解析索引节点对象所指定的符号链接；如果该符号链接是一个相对路径名，则从第二个参数所指定的目录开始进行查找。

put_link(dentry, nameidata)

释放由follow_link方法分配的用于解析符号链接的所有临时数据结构。

truncate(inode)

修改与索引节点相关的文件长度。在调用该方法之前，必须将inode对象的i_size字段设置为需要的新长度值。

permission(inode, mask, nameidata)

检查是否允许对与索引节点所指的文件进行指定模式的访问。

setattr(dentry, iattr)

在触及索引节点属性后通知一个“修改事件“。

getattr(mnt, dentry, kstat)

由一些文件系统用于读取索引节点属性。

setxattr(dentry, name, value, size, flags)

为索引节点设置“扩展属性“(扩展属性存放在任何索引节点之外的磁盘块中)。

getxattr(dentry, name, buffer, size)

获取索引节点的扩展属性。

listxattr(dentry, buffer, size)

获取扩展属性名称的整个链表。

removexattr(dentry, name)

删除索引节点的扩展属性。

上述列举的方法对所有可能的索引节点和文件系统类型都是可用的。不过，只有其中的一个子集应用到某一特定的索引节点和文件系统；未实现的方法对应的字段被置为NULL。

文件对象

文件对象描述进程怎样与一个打开的文件进行交互。文件对象是在文件被打开时创建的，由一个 file结构组成，其中包含的字段如表 12-4 所示。注意，文件对象在磁盘上没有对应的映像，因此 file结构中没有设置“脏“字段来表示文件对象是否已被修改。

表12-4：文件对象的字段

类型	字段	说明
struct list_head	f_list	用于通用文件对象链表的指针
struct dentry *	f_dentry	与文件相关的目录项对象
struct vfsmount *	f_vfsmnt	含有该文件的已安装文件系统
struct file_operations *	f_op	指向文件操作表的指针
atomic_t	f_count	文件对象的引用计数器
unsigned int	f_flags	当打开文件时所指定的标志
mode_t	f_mode	进程的访问模式
int	f_error	网络写操作的错误码
loff_t	f_pos	当前的文件位移量（文件指针）
struct fown_struct	f_owner	通过信号进行 I/O 事件通知的数据
unsigned int	f_uid	用户的 UID
unsigned int	f_gid	用户的 GID
struct file_ra_state	f_ra	文件预读状态（参见第十六章）
size_t	f_maxcount	一次单一的操作所能读或写的最大字节数（当前设置为 2^{31}-1）
unsigned long	f_version	版本号，每次使用后自动递增
void *	f_security	指向文件对象的安全结构的指针
void *	private_data	指向特定文件系统或设备驱动程序所需的数据的指针
struct list_head	f_ep_links	文件的事件轮询等待者链表的头
spinlock_t	f_ep_lock	保护 f_ep_links 链表的自旋锁
struct address_space *	f_mapping	指向文件地址空间对象的指针（参见第十五章）

存放在文件对象中的主要信息是文件指针，即文件中当前的位置，下一个操作将在该位置发生。由于几个进程可能同时访问同一文件，因此文件指针必须存放在文件对象而不是索引节点对象中。

文件对象通过一个名为filp的slab高速缓存分配，filp描述符地址存放在filp_cachep变量中。由于分配的文件对象数目是有限的，因此files_stat变量在其max_files字段中指定了可分配文件对象的最大数目，也就是系统可同时访问的最大文件数（注4）。

“在使用“文件对象包含在由具体文件系统的超级块所确立的几个链表中。每个超级块对象把文件对象链表的头存放在s_files字段中；因此，属于不同文件系统的文件对象就包含在不同的链表中。链表中分别指向前一个元素和后一个元素的指针都存放在文件对象的f_list字段中。files_lock自旋锁保护超级块的s_files链表免受多处理器系统上的同时访问。

文件对象的f_count字段是一个引用计数器：它记录使用文件对象的进程数（记住，以CLONE_FILES标志创建的轻量级进程共享打开文件表，因此它们可以使用相同的文件对象）。当内核本身使用该文件对象时也要增加计数器的值——例如，把对象插入链表中或发出dup()系统调用时。

当VFS代表进程必须打开一个文件时，它调用get_empty_filp()函数来分配一个新的文件对象。该函数调用kmem_cache_alloc()从filp高速缓存中获得一个空闲的文件对象，然后初始化这个对象的字段，如下所示：

```
memset(f, 0, sizeof(*f));
INIT_LIST_HEAD(&f->f_ep_links);
spin_lock_init(&f->f_ep_lock);
atomic_set(&f->f_count, 1);
f->f_uid = current->fsuid;
f->f_gid = current->fsgid;
f->f_owner.lock = RW_LOCK_UNLOCKED;
INIT_LIST_HEAD(&f->f_list);
f->f_maxcount = INT_MAX;
```

正如在“通用文件模型“一节中讨论过的那样，每个文件系统都有其自己的文件操作集合，执行诸如读写文件这样的操作。当内核将一个索引节点从磁盘装入内存时，就会把指向这些文件操作的指针存放在file_operations结构中，而该结构的地址存放在该索引节点对象的i_fop字段中。当进程打开这个文件时，VFS就用存放在索引节点中的这个地址初始化新文件对象的f_op字段，使得对文件操作的后续调用能够使用这些函数。如果需要，VFS随后也可以通过在f_op字段存放一个新值而修改文件操作的集合。

注4：内核初始化期间，files_init()函数把max_files字段设置为可用RAM大小的1/10（以千字节为单位）。不过，系统管理员可以通过写*/proc/sys/fs/file-max*文件来修改这个值。而且，即使max_files文件对象已经被分配，超级用户也总是可以获得一个文件对象。

下面的列表描述了文件的操作，以它们在 file_operations 表中出现的次序来排列：

llseek(file, offset, origin)

更新文件指针。

read(file, buf, count, offset)

从文件的 *offset 处开始读出 count 个字节；然后增加 *offset 的值（一般与文件指针对应）。

aio_read(req, buf, len, pos)

启动一个异步 I/O 操作，从文件的 pos 处开始读出 len 个字节的数据并将它们放入 buf 中（引入它是为了支持 io_submit()系统调用）。

write(file, buf, count, offset)

从文件的 *offset 处开始写入 count 个字节，然后增加 *offset 的值（一般与文件指针对应）。

aio_write(req, buf, len, pos)

启动一个异步 I/O 操作，从 buf 中取 len 个字节写入文件 pos 处。

readdir(dir, dirent, filldir)

返回一个目录的下一个目录项，返回值存入参数 dirent；参数 filldir 存放一个辅助函数的地址，该函数可以提取目录项的各个字段。

poll(file, poll_table)

检查是否在一个文件上有操作发生，如果没有则睡眠，直到该文件上有操作发生。

ioctl(inode, file, cmd, arg)

向一个基本硬件设备发送命令。该方法只适用于设备文件。

unlocked_ioctl(file, cmd, arg)

与 ioctl 方法类似，但是它不用获得大内核锁（参见第五章的“大内核锁“一节）。我们认为所有的设备驱动程序和文件系统都将使用这个新方法而不是 ioctl 方法。

compat_ioctl(file, cmd, arg)

64 位的内核使用该方法执行 32 位的系统调用 ioctl()。

mmap(file, vma)

执行文件的内存映射，并将映射放入进程的地址空间（参见第十六章的“内存映射“一节）。

open(inode, file)

通过创建一个新的文件对象而打开一个文件，并把它链接到相应的索引节点对象（参见本章后面的“open()系统调用“一节）。

flush(file)

当打开文件的引用被关闭时调用该方法。该方法的实际用途取决于文件系统。

release(inode, file)

释放文件对象。当打开文件的最后一个引用被关闭时（即文件对象f_count字段的值变为0时）调用该方法。

fsync(file, dentry, flag)

将文件所缓存的全部数据写入磁盘。

aio_fsync(req, flag)

启动一次异步I/O刷新操作。

fasync(fd, file, on)

通过信号来启用或禁止I/O事件通告。

lock(file, cmd, file_lock)

对file文件申请一个锁（参见本章后面的“文件加锁“一节）。

readv(file, vector, count, offset)

从文件中读字节，并把结果放入vector描述的缓冲区中；缓冲区的个数由count指定。

writev(file, vector, count, offset)

把vector描述的缓冲区中的字节写入文件；缓冲区的个数由count指定。

sendfile(in_file, offset, count, file_send_actor, out_file)

把数据从in_file传送到out_file（引入它是为了支持sendfile()系统调用）。

sendpage(file, page, offset, size, pointer, fill)

把数据从文件传送到页高速缓存的页；这个低层方法由sendfile()和用于套接字的网络代码使用。

get_unmapped_area(file, addr, len, offset, flags)

获得一个未用的地址范围来映射文件。

check_flags(flags)

当设置文件的状态标志（F_SETFL命令）时，fcntl()系统调用的服务例程调用该方法执行附加的检查。当前只适用于NFS网络文件系统。

dir_notify(file, arg)

当建立一个目录更改通告（F_NOTIFY命令）时，由fcntl()系统调用的服务例程调用该方法。当前只适用于CIFS（Common Internet File system，公用互联网文件系统）网络文件系统。

`flock(file, flag, lock)`

用于定制`flock()`系统调用的行为。官方Linux文件系统不使用该方法。

以上描述的方法对所有可能的文件类型都是可用的。不过，对于一个具体的文件类型，只使用其中的一个子集；那些未实现的方法对应的字段被置为NULL。

目录项对象

在“通用文件模型“一节中我们曾提到，VFS把每个目录看作由若干子目录和文件组成的一个普通文件。在第十八章我们将会讨论如何在具体的文件系统上实现目录。然而，一旦目录项被读入内存，VFS就把它转换成基于`dentry`结构的一个目录项对象，该结构的字段如表12-5所示。对于进程查找的路径名中的每个分量，内核都为其创建一个目录项对象；目录项对象将每个分量与其对应的索引节点相联系。例如，在查找路径名*/tmp/test*时，内核为根目录“/“创建一个目录项对象，为根目录下的*tmp*项创建一个第二级目录项对象，为*/tmp*目录下的*test*项创建一个第三级目录项对象。

请注意，目录项对象在磁盘上并没有对应的映像，因此在`dentry`结构中不包含指出该对象已被修改的字段。目录项对象存放在名为`dentry_cache`的slab分配器高速缓存中。因此，目录项对象的创建和删除是通过调用`kmem_cache_alloc()`和`kmem_cache_free()`实现的。

表12-5：目录项对象的字段

类型	字段	说明
`atomic_t`	`d_count`	目录项对象引用计数器
`unsigned int`	`d_flag`	目录项高速缓存标志
`spinlock_t`	`d_lock`	保护目录项对象的自旋锁
`struct inode *`	`d_inode`	与文件名关联的索引节点
`struct dentry *`	`d_parent`	父目录的目录项对象
`struct qstr`	`d_name`	文件名
`struct list_head`	`d_lru`	用于未使用目录项链表的指针
`struct list_head`	`d_child`	对目录而言，用于同一父目录中的目录项链表的指针
`struct list_head`	`d_subdirs`	对目录而言，子目录项链表的头
`struct list_head`	`d_alias`	用于与同一索引节点（别名）相关的目录项链表的指针

表12-5：目录项对象的字段

类型	字段	说明
unsigned long	d_time	由d_revalidate方法使用
struct dentry_operations*	d_op	目录项方法
struct super_block *	d_sb	文件的超级块对象
void *	d_fsdata	依赖于文件系统的数据
struct rcu_head	d_rcu	回收目录项对象时，由RCU描述符使用（参见第五章的“读－拷贝－更新（RCU）”一节）
struct dcookie_struct *	d_cookie	指向内核配置文件使用的数据结构的指针
struct hlist_node	d_hash	指向散列表表项链表的指针
int	d_mounted	对目录而言，用于记录安装该目录项的文件系统数的计数器
unsigned char[]	d_iname	存放短文件名的空间

每个目录项对象可以处于以下四种状态之一：

空闲状态（free）

处于该状态的目录项对象不包括有效的信息，且还没有被VFS使用。对应的内存区由slab分配器进行处理。

未使用状态（unused）

处于该状态的目录项对象当前还没有被内核使用。该对象的引用计数器d_count的值为0，但其d_inode字段仍然指向关联的索引节点。该目录项对象包含有效的信息，但为了在必要时回收内存，它的内容可能被丢弃。

正在使用状态（in use）

处于该状态的目录项对象当前正在被内核使用。该对象的引用计数器d_count的值为正数，其d_inode字段指向关联的索引节点对象。该目录项对象包含有效的信息，并且不能被丢弃。

负状态（negative）

与目录项关联的索引节点不复存在，那是因为相应的磁盘索引节点已被删除，或者因为目录项对象是通过解析一个不存在文件的路径名创建的。目录项对象的d_inode字段被置为NULL，但该对象仍然被保存在目录项高速缓存中，以便后续对同一文件目录名的查找操作能够快速完成。术语“负状态“容易使人误解，因为根本不涉及任何负值。

与目录项对象关联的方法称为目录项操作。这些方法由dentry_operations结构加以描述，该结构的地址存放在目录项对象的d_op字段中。尽管一些文件系统定义了它们自己的目录项方法，但是这些字段通常为NULL，而VFS使用缺省函数代替这些方法。以下按照其在dentry_operations表中出现的顺序来列举一些方法。

d_revalidate(dentry, nameidata)

在把目录项对象转换为一个文件路径名之前，判定该目录项对象是否仍然有效。缺省的VFS函数什么也不做，而网络文件系统可以指定自己的函数。

d_hash(dentry, name)

生成一个散列值；这是用于目录项散列表的、特定于具体文件系统的散列函数。参数dentry标识包含路径分量的目录。参数name指向一个结构，该结构包含要查找的路径名分量以及由散列函数生成的散列值。

d_compare(dir, name1, name2)

比较两个文件名。name1应该属于dir所指的目录。缺省的VFS函数是常用的字符串匹配函数。不过，每个文件系统可用自己的方式实现这一方法。例如，MS-DOS文件系统不区分大写和小写字母。

d_delete(dentry)

当对目录项对象的最后一个引用被删除（d_count变为“0“）时，调用该方法。缺省的VFS函数什么也不做。

d_release(dentry)

当要释放一个目录项对象时（放入slab分配器），调用该方法。缺省的VFS函数什么也不做。

d_iput(dentry, ino)

当一个目录项对象变为“负“状态（即丢弃它的索引节点）时，调用该方法。缺省的VFS函数调用iput()释放索引节点对象。

目录项高速缓存

由于从磁盘读入一个目录项并构造相应的目录项对象需要花费大量的时间，所以，在完成对目录项对象的操作后，可能后面还要使用它，因此仍在内存中保留它有重要的意义。例如，我们经常需要编辑文件，随后编译它，或者编辑并打印它，或者复制它并编辑这个拷贝，在诸如此类的情况中，同一个文件需要被反复访问。

为了最大限度地提高处理这些目录项对象的效率，Linux使用目录项高速缓存，它由两种类型的数据结构组成：

- 一个处于正在使用、未使用或负状态的目录项对象的集合。
- 一个散列表，从中能够快速获取与给定的文件名和目录名对应的目录项对象。同样，如果访问的对象不在目录项高速缓存中，则散列函数返回一个空值。

目录项高速缓存的作用还相当于索引节点高速缓存（*inode cache*）的控制器。在内核内存中，并不丢弃与未用目录项相关的索引节点，这是由于目录项高速缓存仍在使用它们。因此，这些索引节点对象保存在RAM中，并能够借助相应的目录项快速引用它们。

所有“未使用“目录项对象都存放在一个“最近最少使用（Least Recently used，LRU）“的双向链表中，该链表按照插入的时间排序。换句话说，最后释放的目录项对象放在链表的首部，所以最近最少使用的目录项对象总是靠近链表的尾部。一旦目录项高速缓存的空间开始变小，内核就从链表的尾部删除元素，使得最近最常使用的对象得以保留。LRU链表的首元素和尾元素的地址存放在list_head类型的dentry_unused变量的next字段和prev字段中。目录项对象的d_lru字段包含指向链表中相邻目录项的指针。

每个“正在使用“的目录项对象都被插入一个双向链表中，该链表由相应索引节点对象的i_dentry字段所指向（由于每个索引节点可能与若干硬链接关联，所以需要一个链表）。目录项对象的d_alias字段存放链表中相邻元素的地址。这两个字段的类型都是struct list_head。

当指向相应文件的最后一个硬链接被删除后，一个“正在使用“的目录项对象可能变成“负“状态。在这种情况下，该目录项对象被移到“未使用“目录项对象组成的LRU链表中。每当内核缩减目录项高速缓存时，“负“状态目录项对象就朝着LRU链表的尾部移动，这样一来，这些对象就逐渐被释放（参见第十七章中的“回收可压缩磁盘高速缓存的页“一节）。

散列表是由dentry_hashtable数组实现的。数组中的每个元素是一个指向链表的指针，这种链表就是把具有相同散列表值的目录项进行散列而形成的。该数组的长度取决于系统已安装RAM的数量；缺省值是每兆字节RAM包含256个元素。目录项对象的d_hash字段包含指向具有相同散列值的链表中的相邻元素。散列函数产生的值是由目录的目录项对象及文件名计算出来的。

dcache_lock自旋锁保护目录项高速缓存数据结构免受多处理器系统上的同时访问。d_lookup()函数在散列表中查找给定的父目录项对象和文件名；为了避免发生竞争，使用顺序锁（seqlock）（参见第五章中的“顺序锁“一节）。__d_lookup()函数与之类似，不过它假定不会发生竞争，因此不使用顺序锁。

与进程相关的文件

在第一章的“Unix文件系统概述“一节中我们曾提到，每个进程都有它自己当前的工作目录和它自己的根目录。这仅仅是内核用来表示进程与文件系统相互作用所必须维护的数据中的两个例子。类型为fs_struc的整个数据结构就用于此目的（参见表12-6），且每个进程描述符的fs字段就指向进程的fs_struc结构。

表12-6：fs_struct结构中的字段

类型	字段	说明
atomic_t	count	共享这个表的进程个数
rwlock_t	lock	用于表中字段的读/写自旋锁
Int	umask	当打开文件设置文件权限时所使用的位掩码
struct dentry *	root	根目录的目录项
struct dentry *	pwd	当前工作目录的目录项
struct dentry *	altroot	模拟根目录的目录项（在80x86结构上始终为NULL）
struct vfsmount *	rootmnt	根目录所安装的文件系统对象
struct vfsmount *	pwdmnt	当前工作目录所安装的文件系统对象
struct vfsmount *	altrootmnt	模拟根目录所安装的文件系统对象（在80x86结构上始终为NULL）

第二个表表示进程当前打开的文件，表的地址存放于进程描述符的files字段。该表的类型为files_struct结构，它的各个字段如表12-7所示。

表12-7：files_struct结构中的字段

类型	字段	说明
atomic_t	count	共享该表的进程数目
rwlock_t	file_lock	用于表中字段的读/写自旋锁
Int	max_fds	文件对象的当前最大数目
Int	max_fdset	文件描述符的当前最大数目
Int	next_fd	所分配的最大文件描述符加1
struct file **	fd	指向文件对象指针数组的指针
fd_set *	close_on_exec	指向执行exec()时需要关闭的文件描述符的指针

表12-7：files_struct结构中的字段（续）

类型	字段	说明
fd_set *	open_fds	指向打开文件描述符的指针
fd_set	close_on_exec_init	执行exec()时需要关闭的文件描述符的初始集合
fd_set	open_fds_init	文件描述符的初始集合
struct file *[]	fd_array	文件对象指针的初始化数组

fd字段指向文件对象的指针数组。该数组的长度存放在max_fds字段中。通常，fd字段指向files_struct结构的fd_array字段，该字段包括32个文件对象指针。如果进程打开的文件数目多于32，内核就分配一个新的、更大的文件指针数组，并将其地址存放在fd字段中，内核同时也更新max_fds字段的值。

对于在fd数组中有元素的每个文件来说，数组的索引就是文件描述符(*file descriptor*)。通常，数组的第一个元素（索引为0）是进程的标准输入文件，数组的第二个元素（索引为1）是进程的标准输出文件，数组的第三个元素（索引为2）是进程的标准错误文件(参见图12-3)。Unix进程将文件描述符作为主文件标识符。请注意，借助于dup()、dup2()和fcntl()系统调用，两个文件描述符可以指向同一个打开的文件，也就是说，数组的两个元素可能指向同一个文件对象。当用户使用shell结构（如2>&1）将标准错误文件重定向到标准输出文件上时，用户总能看到这一点。

进程不能使用多于NR_OPEN（通常为1 048 576)个文件描述符。内核也在进程描述符的signal->rlim[RLIMIT_NOFILE]结构上强制动态限制文件描述符的最大数；这个值通常为1024，但是如果进程具有超级用户特权，就可以增大这个值。

open_fds字段最初包含open_fds_init字段的地址，open_fds_init字段表示当前已打开文件的文件描述符的位图。max_fdset字段存放位图中的位数。由于fd_set数据结构有1024位，所以通常不需要扩大位图的大小。不过，如果确有必要的话，内核仍能动态增加位图的大小，这非常类似于文件对象的数组的情形。

当内核开始使用一个文件对象时，内核提供fget()函数以供调用。这个函数接收文件描述符fd作为参数，返回在current->files->fd[fd]中的地址，即对应文件对象的地址，如果没有任何文件与fd对应，则返回NULL。在第一种情况下，fget()使文件对象引用计数器f_count的值增1。

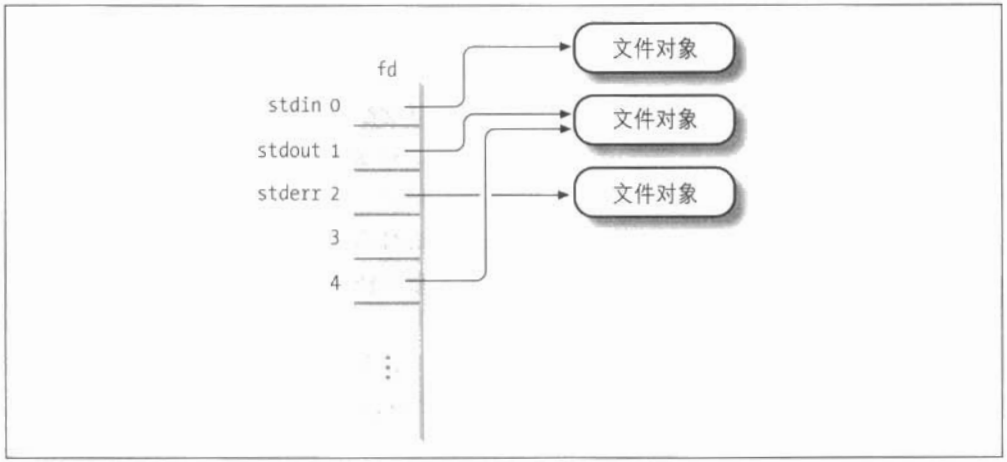


图 12-3：fd 数组

当内核控制路径完成对文件对象的使用时，调用内核提供的 fput() 函数。该函数将文件对象的地址作为参数，并减少文件对象引用计数器 f_count 的值。另外，如果这个字段变为0，该函数就调用文件操作的 release 方法（如果已定义），减少索引节点对象的 i_write count 字段的值（如果该文件是可写的），将文件对象从超级块链表中移走，释放文件对象给slab分配器，最后减少相关的文件系统描述符的目录项对象的引用计数器的值（参见“文件系统安装”一节）。

fget_light() 和 fget_light() 函数是 fget() 和 fput() 的快速版本：内核要使用它们，前提是能够安全地假设当前进程已经拥有文件对象，即进程先前已经增加了文件对象引用计数器的值。例如，它们由接收一个文件描述符作为参数的系统调用服务例程使用，这是由于先前的 open() 系统调用已经增加了文件对象引用计数器的值。

文件系统类型

Linux 内核支持很多不同的文件系统类型。在下面的内容中，我们介绍一些特殊的文件系统类型，它们在 Linux 内核的内部设计中具有非常重要的作用。

接下来，我们将讨论文件系统注册 —— 也就是通常在系统初始化期间并且在使用文件系统类型之前必须执行的基本操作。一旦文件系统被注册，其特定的函数对内核就是可用的，因此文件系统类型可以安装在系统的目录树上。

特殊文件系统

当网络和磁盘文件系统能够使用户处理存放在内核之外的信息时，特殊文件系统可以为系统程序员和管理员提供一种容易的方式来操作内核的数据结构并实现操作系统的特殊特征。表12-8列出了Linux中所用的最常用的特殊文件系统；对于其中的每个文件系统，表中给出了它的安装点和简短描述。

注意，有几个文件系统没有固定的安装点（表中的关键词“任意”）。这些文件系统可以由用户自由地安装和使用。此外，一些特殊文件系统根本没有安装点（表中的关键词“无”）。它们不是用于与用户交互，但是内核可以用它们来很容易地重新使用VFS层的某些代码；例如，我们在第十九章会看到，有了pipefs特殊文件系统，就可以把管道和FIFO文件以相同的方式对待。

表12-8：最常用的特殊文件系统

名字	安装点	说明
bdev	无	块设备（参见第十三章）
binfmt_misc	任意	其他可执行格式（参见第二十章）
devpts	/dev/pts	伪终端支持（开放组织的Unix98标准）
eventpollfs	无	由有效事件轮询机制使用
futexfs	无	由futex（快速用户空间加锁）机制使用
pipefs	无	管道 （参见第十九章）
proc	/proc	对内核数据结构的常规访问点
rootfs	无	为启动阶段提供一个空的根目录
shm	无	IPC共享线性区（参见第十九章）
mqueue	任意	实现POSIX消息队列时使用（参见第十九章）
sockfs	无	套接字
sysfs	/sys	对系统数据的常规访问点（参见第十三章）
tmpfs	任意	临时文件（如果不被交换出去就保持在RAM中）
usbfs	/proc/bus/usb	USB设备

特殊文件系统不限于物理块设备。然而，内核给每个安装的特殊文件系统分配一个虚拟的块设备，让其主设备号为0而次设备号具有任意值（每个特殊文件系统有不同的值）。set_anon_super()函数用于初始化特殊文件系统的超级块；该函数本质上获得一个未使用的次设备号dev，然后用主设备号0和次设备号dev设置新超级块的s_dev字段。而另一个kill_anon_super()函数移走特殊文件系统的超级块。unnamed_dev_idr变

量包含指向一个辅助结构（记录当前在用的次设备号）的指针。尽管有些内核设计者不喜欢虚拟块设备标识符，但是这些标识符有助于内核以统一的方式处理特殊文件系统和普通文件系统。

我们在后面“安装普通文件系统“一节会看到内核如何定义和初始化一个特殊文件系统的实际例子。

文件系统类型注册

通常，用户在为自己的系统编译内核时可以把Linux配置为能够识别所有需要的文件系统。但是，文件系统的源代码实际上要么包含在内核映像中，要么作为一个模块被动态装入（参见附录二）。VFS必须对代码目前已在内核中的所有文件系统的类型进行跟踪。这就是通过进行文件系统类型注册来实现的。

每个注册的文件系统都用一个类型为file_system_type的对象来表示，该对象的所有字段在表12-9中列出。

表12-9：file_system_type对象的字段

类型	字段	说明
const char *	name	文件系统名
int	fs_flags	文件系统类型标志
struct super_block * (*)()	get_sb	读超级块的方法
void (*)()	kill_sb	删除超级块的方法
struct module *	owner	指向实现文件系统的模块的指针(参见附录二)
struct file_system_type *	next	指向文件系统类型链表中下一个元素的指针
struct list_head	fs_supers	具有相同文件系统类型的超级块对象链表的头

所有文件系统类型的对象都插入到一个单向链表中。由变量file_systems指向链表的第一个元素，而结构中的next字段指向链表的下一个元素。file_systems_lock读/写自旋锁保护整个链表免受同时访问。

fs_supers字段表示给定类型的已安装文件系统所对应的超级块链表的头（第一个伪元素）。链表元素的向后和向前链接存放在超级块对象的s_instances字段中。

get_sb字段指向依赖于文件系统类型的函数，该函数分配一个新的超级块对象并初始化它（如果需要，可读磁盘）。而kill_sb字段指向删除超级块的函数。

fs_flags字段存放几个标志，如表12-10所示。

表12-10：文件系统类型的标志

名字	说明
FS_REQUIRES_DEV	这种类型的任何文件系统必须位于物理磁盘设备上
FS_BINARY_MOUNTDATA	文件系统使用的二进制安装数据
FS_REVAL_DOT	始终在目录项高速缓存中使“.”和“..”路径重新生效（针对网络文件系统）
FS_ODD_RENAME	“重命名”操作就是“移动”操作（针对网络文件系统）

在系统初始化期间，调用register_filesystem()函数来注册编译时指定的每个文件系统；该函数把相应的file_system_type对象插入到文件系统类型的链表中。

当实现了文件系统的模块被装入时，也要调用register_filesystem()函数。在这种情况下，当该模块被卸载时，对应的文件系统也可以被注销（调用unregister_filesystem()函数）。

get_fs_type()函数（接收文件系统名作为它的参数）扫描已注册的文件系统链表以查找文件系统类型的name字段，并返回指向相应的file_system_type对象（如果存在）的指针。

文件系统处理

就像每个传统的Unix系统一样，Linux也使用系统的根文件系统（*system's root filesystem*）：它由内核在引导阶段直接安装，并拥有系统初始化脚本以及最基本的系统程序。

其他文件系统要么由初始化脚本安装，要么由用户直接安装在已安装文件系统的目录上。作为一个目录树，每个文件系统都拥有自己的根目录（*root directory*）。安装文件系统的这个目录称之为安装点（*mount point*）。已安装文件系统属于安装点目录的一个子文件系统。例如，*/proc*虚拟文件系统是系统的根文件系统的孩子（且系统的根文件系统是*/proc*的父亲）。已安装文件系统的根目录隐藏了父文件系统的安装点目录原来的内容，而且父文件系统的整个子树位于安装点之下（注5）。

注5：文件系统的根目录有可能不同于进程的根目录：正如我们在前面“与文件相关的进程”一节所见，进程的根目录是与“/”路径对应的目录。缺省情况下，进程的根目录与系统的根文件系统的根目录一致（更准确地说是与进程的命名空间中的根文件系统的根目录一致，这一点将在下一节描述），但是可以通过调用chroot()系统调用改变进程的根目录。

命名空间

在传统的Unix系统中，只有一个已安装文件系统树：从系统的根文件系统开始，每个进程通过指定合适的路径名可以访问已安装文件系统中的任何文件。从这个方面考虑，Linux 2.6更加的精确：每个进程可拥有自己的已安装文件系统树——叫做进程的命名空间（*namespace*）。

通常大多数进程共享同一个命名空间，即位于系统的根文件系统且被*init*进程使用的已安装文件系统树。不过，如果clone()系统调用以CLONE_NEWNS标志创建一个新进程，那么进程将获取一个新的命名空间（参见第三章的“clone()、fork()及vfork()系统调用”一节）。这个新的命名空间随后由子进程继承（如果父进程没有以CLONE_NEWNS标志创建这些子进程）。

当进程安装或卸载一个文件系统时，仅修改它的命名空间。因此，所做的修改对共享同一命名空间的所有进程都是可见的，并且也只对它们可见。进程甚至可通过使用Linux特有的pivot_root()系统调用来改变它的命名空间的根文件系统。

进程的命名空间由进程描述符的namespace字段指向的namespace结构描述。该结构的字段如表12-11所示。

表12-11：namespace结构中的字段

类型	字段	说明
atomic_t	count	引用计数器（共享命名空间的进程数）
struct vfsmount *	root	命名空间根目录的已安装文件系统描述符
struct list_head	list	所有已安装文件系统描述符链表的头
struct rw_semaphore	sem	保护这个结构的读/写信号量

list字段是双向循环链表的头，该表聚集了属于命名空间的所有已安装文件系统。root字段表示已安装文件系统，它是这个命名空间的已安装文件系统树的根。正如我们在下一节将看到的，已安装文件系统由vfsmount结构描述。

文件系统安装

在大多数传统的类Unix内核中，每个文件系统只能安装一次。假定存放在*/dev/fd0*软磁盘上的Ext2文件系统通过如下命令安装在*/flp*：

```
mount -t ext2 /dev/fd0 /flp
```

在用umount命令卸载该文件系统前，所有其他作用于*/dev/fd0*的安装命令都会失败。

然而，Linux 有所不同：同一个文件系统被安装多次是可能的。当然，如果一个文件系统被安装了 *n* 次，那么它的根目录就可通过 *n* 个安装点来访问。尽管同一文件系统可以通过不同的安装点来访问，但是文件系统的的确确是唯一的。因此，不管一个文件系统被安装了多少次，都仅有一个超级块对象。

安装的文件系统形成一个层次：一个文件系统的安装点可能成为第二个文件系统的目录，而第二个文件系统又安装在第三个文件系统之上，等等（注 6）。

把多个安装堆叠在一个单独的安装点上也是可能的。尽管已经使用先前安装下的文件和目录的进程可以继续使用，但在同一安装点上的新安装隐藏前一个安装的文件系统。当最顶层的安装被删除时，下一层的安装再一次变为可见的。

你可以想像，跟踪已安装的文件系统很快会变为一场恶梦。对于每个安装操作，内核必须在内存中保存安装点和安装标志，以及要安装文件系统与其他已安装文件系统之间的关系。这样的信息保存在已安装文件系统描述符中；每个描述符是一个具有 vfsmount 类型的数据结构，其字段如表 12-12 所示。

表12-12：vfsmount 数据结构中的字段

类型	字段	说明
struct list_head	mnt_hash	用于散列表链表的指针
struct vfsmount *	mnt_parent	指向父文件系统，这个文件系统安装在其上
struct dentry *	mnt_mountpoint	指向这个文件系统安装点目录的 dentry
struct dentry *	mnt_root	指向这个文件系统根目录的 dentry
struct super_block *	mnt_sb	指向这个文件系统的超级块对象
struct list_head	mnt_mounts	包含所有文件系统描述符链表的头（相对于这个文件系统）
struct list_head	mnt_child	用于已安装文件系统链表 mnt_mounts 的指针
atomic_t	mnt_count	引用计数器（增加该值以禁止文件系统被卸载）
Int	mnt_flags	标志

注 6：令人非常惊讶的是，一个文件系统的安装点可能就是这同一文件系统中的一个目录，假定这个文件系统以前已经安装。例如：

```
mount-t ext2/dev/fd0/flp;touch/flp/foo
mkdir/flp/mnt;mount-t ext2/dev/fd0/flp/mnt
```

现在，软盘文件系统上的空 *foo* 文件就可以通过 */flp/foo* 和 */flp/mnt/foo* 来访问。

表12-12：vfsmount 数据结构中的字段（续）

类型	字段	说明
Int	mnt_expiry_mark	如果文件系统标记为到期，那么就设置该标志为true（如果设置了该标志，并且没有任何人使用它，那么就可以自动卸载这个文件系统）
char *	mnt_devname	设备文件名
struct list_head	mnt_list	已安装文件系统描述符的namespace链表的指针
struct list_head	mnt_fslink	具体文件系统到期链表的指针
struct namespace *	mnt_namespace	指向安装了文件系统的进程命名空间的指针

vfsmount 数据结构保存在几个双向循环链表中：

- 由父文件系统vfsmount描述符的地址和安装点目录的目录项对象的地址索引的散列表。散列表存放在 mount_hashtable 数组中，其大小取决于系统中 RAM 的容量。表中每一项是具有同一散列值的所有描述符形成的双向循环链表的头。描述符的 mnt_hash 字段包含指向链表中相邻元素的指针。
- 对于每一个命名空间，所有属于此命名空间的已安装的文件系统描述符形成了一个双向循环链表。namespace结构的 list 字段存放链表的头，vfsmount 描述符的 mnt_list 字段包含链表中指向相邻元素的指针。
- 对于每一个已安装的文件系统，所有已安装的子文件系统形成了一个双向循环链表。每个链表的头存放在已安装的文件系统描述符的 mnt_mounts 字段；此外，描述符的 mnt_child 字段存放指向链表中相邻元素的指针。

vfsmount_lock 自旋锁保护已安装文件系统对象的链表免受同时访问。

描述符的mnt_flags字段存放几个标志的值，用以指定如何处理已安装文件系统中的某些种类的文件。这些标志可通过 mount 命令的选项进行设置，其标志如表 12-13 所示。

表12-13：已安装文件系统中的标志

名字	说明
MNT_NOSUID	在已安装文件系统中禁止 setuid 和 setgid 标志
MNT_NODEV	在已安装文件系统中禁止访问设备文件
MNT_NOEXEC	在已安装文件系统中不允许程序执行

下列函数处理已安装文件系统描述符：

alloc_vfsmnt(name)

分配和初始化一个已安装文件系统描述符。

free_vfsmnt(mnt)

释放由mnt指向的已安装文件系统描述符。

lookup_mnt(mnt, dentry)

在散列表中查找一个描述符并返回它的地址。

安装普通文件系统

我们现在描述安装一个文件系统时内核所要执行的操作。我们首先考虑一个文件系统将被安装在一个已安装文件系统之上的情形（在这里我们把这种新文件系统看作“普通的”）。

mount()系统调用被用来安装一个普通文件系统；它的服务例程sys_mount()作用于以下参数：

- 文件系统所在的设备文件的路径名，或者如果不需要的话就为NULL（例如，当要安装的文件系统是基于网络时）
- 文件系统被安装其上的某个目录的目录路径名（安装点）
- 文件系统的类型，必须是已注册文件系统的名字
- 安装标志（所允许的值如表12-14所示）
- 指向一个与文件系统相关的数据结构的指针（也许为NULL）

表12-14：mount()系统调用使用的安装标志

宏	说明
MS_RDONLY	文件只能被读
MS_NOSUID	禁止setuid和setgid标志
MS_NODEV	禁止访问设备文件
MS_NOEXEC	不允许程序执行
MS_SYNCHRONOUS	文件和目录上的写操作是即时的
MS_REMOUNT	重新安装改变了安装标志的文件系统
MS_MANDLOCK	允许强制加锁
MS_DIRSYNC	目录上的写操作是即时的
MS_NOATIME	不更新文件访问时间

表12-14：mount()系统调用使用的安装标志（续）

宏	说明
MS_NODIRATIME	不更新目录访问时间
MS_BIND	创建一个“绑定安装“，这就使得一个文件或目录在系统目录树的另外一个点上可以看得见（mount命令的__bind选项）
MS_MOVE	自动把一个已安装文件系统移动到另一个安装点（mount命令的__move选项）
MS_REC	为目录子树递归地创建“绑定安装“
MS_VERBOSE	在安装出错时产生内核消息

sys_mount()函数把参数的值拷贝到临时内核缓冲区，获取大内核锁，并调用do_mount()函数。一旦do_mount()返回，则这个服务例程释放大内核锁并释放临时内核缓冲区。

do_mount()函数通过执行下列操作处理真正的安装操作：

1. 如果安装标志MS_NOSUID、MS_NODEV或MS_NOEXEC中任一个被设置，则清除它们，并在已安装文件系统对象中设置相应的标志（MNT_NOSUID、MNT_NODEV、MNT_NOEXEC）。
2. 调用path_lookup()查找安装点的路径名；该函数把路径名查找的结果存放在nameidata类型的局部变量nd中（参见后面的“路径名查找”一节）。
3. 检查安装标志以决定必须做什么。尤其是：
   a. 如果MS_REMOUNT标志被指定，其目的通常是改变超级块对象s_flags字段的安装标志，以及已安装文件系统对象mnt_flags字段的安装文件系统标志。do_remount()函数执行这些改变。
   b. 否则，检查MS_BIND标志。如果它被指定，则用户要求在在系统目录树的另一个安装点上的文件或目录能够可见。
   c. 否则，检查MS_MOVE标志。如果它被指定，则用户要求改变已安装文件系统的安装点。do_move_mount()函数原子地完成这一任务。
   d. 否则，调用do_new_mount()。这是最普通的情况。当用户要求安装一个特殊文件系统或存放在磁盘分区中的普通文件系统时，触发该函数。它调用do_kern_mount()函数，给它传递的参数为文件系统类型、安装标志以及块设备名。do_kern_mount()处理实际的安装操作并返回一个新安装文件系统描述符的地址（如下描述）。然后，do_new_mount()调用do_add_mount()，后者本质上执行下列操作：

(1) 获得当前进程的写信号量 namespace->sem，因为函数要更改 namespace 结构。

(2) do_kern_mount()函数可能让当前进程睡眠；同时，另一个进程可能在完全相同的安装点上安装文件系统或者甚至更改根文件系统(current->namespace->root)。验证在该安装点上最近安装的文件系统是否仍指向当前的 namespace；如果不是，则释放读/写信号量并返回一个错误码。

(3) 如果要安装的文件系统已经被安装在由系统调用的参数所指定的安装点上，或该安装点是一个符号链接，则释放读/写信号量并返回一个错误码。

(4) 初始化由do_kern_mount()分配的新安装文件系统对象的mnt_flags字段的标志。

(5) 调用 graft_tree()把新安装的文件系统对象插入到 namespace 链表、散列表及父文件系统的子链表中。

(6) 释放 namespace->sem 读/写信号量并返回。

4. 调用path_release()终止安装点的路径名查找(参见后面的“路径名查找“一节)并返回 0。

do_kern_mount() 函数

安装操作的核心是do_kern_mount()函数，它检查文件系统类型标志以决定安装操作是如何完成的。该函数接收下列参数：

fstype
: 要安装的文件系统的类型名。

flags
: 安装标志（参见表 12-14）。

name
: 存放文件系统的块设备的路径名（或特殊文件系统的类型名）。

data
: 指向传递给文件系统的 read_super 方法的附加数据的指针。

本质上，该函数通过执行下列操作实现实际的安装操作：

1. 调用get_fs_type()在文件系统类型链表中搜索并确定存放在fstype参数中的名字的位置；返回局部变量 type 中对应 file_system_type 描述符的地址。

2. 调用alloc_vfsmnt()分配一个新的已安装文件系统的描述符，并将它的地址存放在mnt局部变量中。
3. 调用依赖于文件系统的type->get_sb()函数分配，并初始化一个新的超级块（参见下面）。
4. 用新超级块对象的地址初始化mnt->mnt_sb字段。
5. 将mnt->mnt_root字段初始化为与文件系统根目录对应的目录项对象的地址，并增加该目录项对象的引用计数器值。
6. 用mnt中的值初始化mnt->mnt_parent字段（对于普通文件系统，当graft_tree()把已安装文件系统的描述符插入到合适的链表中时，要把mnt_parent字段置为合适的值；参见do_mount()的第3d5步）。
7. 用current->namespace中的值初始化mnt->mnt_namespace字段。
8. 释放超级块对象的读/写信号量s_umount（在第3步中分配对象时获得）。
9. 返回已安装文件系统对象的地址mnt。

分配超级块对象

文件系统对象的get_sb方法通常是由单行函数实现的。例如，在Ext2文件系统中该方法的实现如下：

```
struct super_block * ext2_get_sb(struct file_system_type *type,
                        int flags, const char *dev_name, void *data)
{
    return get_sb_bdev(type, flags, dev_name, data, ext2_fill_super);
}
```

get_sb_bdev()VFS函数分配并初始化一个新的适合于磁盘文件系统的超级块；它接收ext2_fill_super()函数的地址，该函数从Ext2磁盘分区读取磁盘超级块。

为了分配适合于特殊文件系统的超级块，VFS也提供get_sb_pseudo()函数（对于没有安装点的特殊文件系统，例如pipefs）、get_sb_single()函数（对于具有唯一安装点的特殊文件系统，例如*sysfs*）以及get_sb_nodev()函数（对于可以安装多次的特殊文件系统，例如*tmpfs*；参见下面）。

get_sb_bdev()执行的最重要的操作如下：

1. 调用open_bdev_excl()打开设备文件名为dev_name的块设备（参见第十三章的“字符设备驱动程序“一节）。
2. 调用sget()搜索文件系统的超级块对象链表（type->fs_supers，参见前面的“文

件系统类型注册”一节)。如果找到一个与块设备相关的超级块，则返回它的地址。否则，分配并初始化一个新的超级块对象，把它插入到文件系统链表和超级块全局链表中，并返回其地址。

3. 如果不是新的超级块（它不是上一步分配的，因为文件系统已经被安装），则跳到第6步。
4. 把参数flags中的值拷贝到超级块的s_flags字段，并将s_id、s_old_blocksize以及s_blocksize字段设置为块设备的合适值。
5. 调用依赖文件系统的函数(该函数作为传递给get_sb_bdev()的最后一个参数)访问磁盘上的超级块信息，并填充新超级块对象的其他字段。
6. 返回新超级块对象的地址。

安装根文件系统

安装根文件系统是系统初始化的关键部分。这是一个相当复杂的过程，因为Linux内核允许根文件系统存放在很多不同的地方，比如硬盘分区、软盘、通过NFS共享的远程文件系统，甚至保存在*ramdisk*中（RAM中的虚拟块设备）。

为了使叙述变得简单，让我们假定根文件系统存放在硬盘分区（毕竟这是最常见的情况)。当系统启动时，内核就要在变量ROOT_DEV中寻找包含根文件系统的磁盘主设备号(参见附录一)。当编译内核时，或者向最初的启动装入程序传递一个合适的“*root*”选项时，根文件系统可以被指定为*/dev*目录下的一个设备文件。类似地，根文件系统的安装标志存放在root_mountflags变量中。用户可以指定这些标志，或者通过对已编译的内核映像使用*rdev*外部程序，或者向最初的启动装入程序传递一个合适的*rootflags*选项来达到（参见附录一)。

安装根文件系统分两个阶段，如下所示：

1. 内核安装特殊*rootfs*文件系统，该文件系统仅提供一个作为初始安装点的空目录。
2. 内核在空目录上安装实际根文件系统。

为什么内核不怕麻烦，要在安装实际根文件系统之前安装*rootfs*文件系统呢？我们知道，*rootfs*文件系统允许内核容易地改变实际根文件系统。事实上，在某些情况下，内核逐个地安装和卸载几个根文件系统。例如，一个发布版的初始启动光盘可能把具有一组最小驱动程序的内核装入RAM中，内核把存放在ramdisk中的一个最小的文件系统作为根安装。接下来，在这个初始根文件系统中的程序探测系统的硬件（例如，它们判断硬盘是否是EIDE、SCSI等等)，装入所有必需的内核模块，并从物理块设备重新安装根文件系统。

阶段 1：安装 rootfs 文件系统

第一阶段是由 init_rootfs()和 init_mount_tree()函数完成的，它们在系统初始化过程中执行。

init_rootfs()函数注册特殊文件系统类型 *rootfs*：

```
struct file_system_type rootfs_fs_type = {
    .name = "rootfs";
    .get_sb = rootfs_get_sb;
    .kill_sb = kill_litter_super;
};
register_filesystem(&rootfs_fs_type);
```

init_mount_tree()函数执行如下操作：

1. 调用 do_kern_mount()函数，把字符串“rootfs”作为文件系统类型参数传递给它，并把该函数返回的新安装文件系统描述符的地址保存在mnt局部变量中。正如前一节所介绍的，do_kern_mount()最终调用 *rootfs* 文件系统的 get_sb方法，也即 rootfs_get_sb()函数：

   ```
   struct superblock *rootfs_get_sb(struct file_system_type *fs_type,
                           int flags, const char *dev_name, void *data)
   {
       return get_sb_nodev(fs_type, flags|MS_NOUSER, data,
                           ramfs_fill_super);
   }
   ```

   get_sb_nodev()函数执行如下步骤：

   a. 调用sget()函数分配新的超级块，传递set_anon_super()函数的地址作为参数（参见前面的“特殊文件系统“一节）。接下来，用合适的方式设置超级块的s_dev字段：主设备号为0，次设备号不同于其他已安装的特殊文件系统的次设备号。

   b. 将 flags 参数的值拷贝到超级块的 s_flags 字段中。

   c. 调用 ramfs_fill_super()函数分配索引节点对象和对应的目录项对象，并填充超级块字段值。由于 rootfs 是一种特殊文件系统，没有磁盘超级块，因此只需执行两个超级块操作。

   d. 返回新超级块的地址。

2. 为进程0的命名空间分配一个namespace对象，并将它插入到由do_kern_mount()函数返回的已安装文件系统描述符中：

   ```
   namespace = kmalloc(sizeof(*namespace), GFP_KERNEL);
   list_add(&mnt->mnt_list, &namespace->list);
   ```

```
namespace->root = mnt;
mnt->mnt_namespace = init_task.namespace = namespace;
```

3. 将系统中其他每个进程的namespace字段设置为namespace对象的地址；同时初始化引用计数器namespace->count（缺省情况下，所有的进程共享同一个初始namespace）。

4. 将进程0的根目录和当前工作目录设置为根文件系统。

阶段2：安装实际根文件系统

根文件系统安装操作的第二阶段是由内核在系统初始化即将结束时进行的。根据内核被编译时所选择的选项，和内核装入程序所传递的启动选项，可以有几种方法安装实际根文件系统。为了简单起见，我们只考虑磁盘文件系统的情况，它的设备文件名已通过"*root*"启动参数传递给内核。同时我们假定除了*rootfs*文件系统外，没有使用其他初始特殊文件系统。

prepare_namespace()函数执行如下操作：

1. 把root_device_name变量置为从启动参数"*root*"中获取的设备文件名。同样，把ROOT_DEV变量置为同一设备文件的主设备号和次设备号。

2. 调用mount_root()函数，依次执行如下操作：

   a. 调用sys_mknod()(mknod()系统调用的服务例程）在*rootfs*初始根文件系统中创建设备文件*/dev/root*，其主、次设备号与存放在ROOT_DEV中的一样。

   b. 分配一个缓冲区并用文件系统类型名链表填充它。该链表要么通过启动参数"*rootfstype*"传送给内核，要么通过扫描文件系统类型单向链表中的元素建立。

   c. 扫描上一步建立的文件系统类型名链表。对每个名字，调用sys_mount()试图在根设备上安装给定的文件系统类型。由于每个特定于文件系统的方法使用不同的魔数，因此，对get_sb()的调用大都会失败，但有一个例外，那就是用根设备上实际使用过的文件系统的函数来填充超级块的那个调用，该文件系统被安装在*rootfs*文件系统的*/root*目录上。

   d. 调用sys_chdir（"/root"）改变进程的当前目录。

3. 移动*rootfs*文件系统根目录上的已安装文件系统的安装点。

```
sys_mount(".", "/", NULL, MS_MOVE, NULL);
sys_chroot(".");
```

注意，*rootfs*特殊文件系统没有被卸载：它只是隐藏在基于磁盘的根文件系统下了。

卸载文件系统

umount()系统调用用来卸载一个文件系统。相应的sys_umount()服务例程作用于两个参数：文件名（多是安装点目录或是块设备文件名）和一组标志。该函数执行下列操作：

1. 调用path_lookup()查找安装点路径名；该函数把返回的查找操作结果存放在nameidata类型的局部变量nd中（参见下一节）。
2. 如果查找的最终目录不是文件系统的安装点，则设置retval返回码为-EINVAL并跳到第6步。这种检查是通过验证nd->mnt->mnt_root（它包含由nd.dentry指向的目录项对象地址）进行的。
3. 如果要卸载的文件系统还没有安装在命名空间中，则设置retval返回码为-EINVAL并跳到第6步（回想一下，某些特殊文件系统没有安装点）。这种检查是通过在nd->mnt上调用check_mnt()函数进行的。
4. 如果用户不具有卸载文件系统的特权，则设置retval返回码为-EPERM并跳到第6步。
5. 调用do_umount()，传递给它的参数为nd.mnt（已安装文件系统对象）和flags（一组标志）。该函数执行下列操作：
   a. 从已安装文件系统对象的mnt_sb字段检索超级块对象sb的地址。
   b. 如果用户要求强制卸载操作，则调用umount_begin超级块操作中断任何正在进行的安装操作。
   c. 如果要卸载的文件系统是根文件系统，且用户并不要求真正地把它卸载下来，则调用do_remount_sb()重新安装根文件系统为只读并终止。
   d. 为进行写操作而获取当前进程的namespace->sem读/写信号量和vfsmount_lock自旋锁。
   e. 如果已安装文件系统不包含任何子安装文件系统的安装点，或者用户要求强制卸载文件系统，则调用umount_tree()卸载文件系统（及其所有子文件系统）。
   f. 释放vfsmount_lock自旋锁和当前进程的namespace->sem读/写信号量。
6. 减少相应文件系统根目录的目录项对象和已安装文件系统描述符的引用计数器值；这些计数器值由path_lookup()增加。
7. 返回retval的值。

路径名查找

当进程必须识别一个文件时，就把它的文件路径名传递给某个VFS系统调用，如open()、mkdir()、rename()或stat()。本节我们要说明VFS如何实现路径名查找，也就是说如何从文件路径名导出相应的索引节点。

执行这一任务的标准过程就是分析路径名并把它拆分成一个文件名序列。除了最后一个文件名以外，所有的文件名都必定是目录。

如果路径名的第一个字符是“/”，那么这个路径名是绝对路径，因此从current->fs->root（进程的根目录）所标识的目录开始搜索。否则，路径名是相对路径，因此从current->fs->pwd（进程的当前目录）所标识的目录开始搜索。

在对初始目录的索引节点进行处理的过程中，代码要检查与第一个名字匹配的目录项，以获得相应的索引节点。然后，从磁盘读出包含那个索引节点的目录文件，并检查与第二个名字匹配的目录项，以获得相应的索引节点。对于包含在路径中的每个名字，这个过程反复执行。

目录项高速缓存极大地加速了这一过程，因为它把最近最常使用的目录项对象保留在内存中。正如我们以前看到的，每个这样的对象使特定目录中的一个文件名与它相应的索引节点相联系。因此在很多情况下，路径名的分析可以避免从磁盘读取中间目录。

但是，事情并不像看起来那么简单，因为必须考虑如下的Unix和VFS文件系统的特点：

- 对每个目录的访问权必须进行检查，以验证是否允许进程读取这一目录的内容。
- 文件名可能是与任意一个路径名对应的符号链接；在这种情况下，分析必须扩展到那个路径名的所有分量。
- 符号链接可能导致循环引用；内核必须考虑这个可能性，并能在出现这种情况时将循环终止。
- 文件名可能是一个已安装文件系统的安装点。这种情况必须检测到，这样，查找操作必须延伸到新的文件系统。
- 路径名查找应该在发出系统调用的进程的命名空间中完成。由具有不同命名空间的两个进程使用的相同路径名，可能指定了不同的文件。

路径名查找是由path_lookup()函数执行的，它接收三个参数：

name
: 指向要解析的文件路径名的指针。

flags

标志的值，表示将会怎样访问查找的文件。在后面的表12-16中列出了所允许的标志。

nd

nameidata数据结构的地址，这个结构存放了查找操作的结果，其字段如表12-15所示。

当path_lookup()返回时，nd指向的nameidata结构用与路径名查找操作有关的数据来填充。

表12-15：nameidata数据结构的字段

类型	字段	说明
struct dentry *	dentry	目录项对象的地址
struct vfs_mount *	mnt	已安装文件系统对象的地址
struct qstr	last	路径名的最后一个分量（当LOOKUP_PARENT标志被设置时使用）
unsigned int	flags	查找标志
int	last_type	路径名最后一个分量的类型（当LOOKUP_PARENT标志被设置时使用）
unsigned int	depth	符号链接嵌套的当前级别（参见下面）；它必须小于6
char[ ] *	saved_names	与嵌套的符号链接关联的路径名数组
union	intent	单个成员联合体，指定如何访问文件

dentry和mnt字段分别指向所解析的最后一个路径分量的目录项对象和已安装文件系统对象。这两个字段“描述“由给定路径名表示的文件。

由于path_lookup()函数返回的nameidata结构中的目录项对象和已安装文件系统对象代表了查找操作的结果，因此在path_lookup()的调用者完成使用查找结果之前，这两个对象都不能被释放。因此，path_lookup()增加两个对象引用计数器的值。如果调用者想释放这些对象，则调用path_release()函数，传递给它的参数为nameidata结构的地址。

flags字段存放查找操作中使用的某些标志的值；它们在表12-16中列出。这些标志中的大部分可由调用者在path_lookup()的flags参数中进行设置。

表12-16：查找操作的标志

宏	说明
LOOKUP_FOLLOW	如果最后一个分量是符号链接，则解释（追踪）它
LOOKUP_DIRECTORY	最后一个分量必须是目录
LOOKUP_CONTINUE	在路径名中还有文件名要检查
LOOKUP_PARENT	查找最后一个分量名所在的目录
LOOKUP_NOALT	不考虑模拟根目录（在80x86体系结构中没有用）
LOOKUP_OPEN	试图打开一个文件
LOOKUP_CREATE	试图创建一个文件（如果不存在）
LOOKUP_ACCESS	试图为一个文件检查用户的权限

path_lookup()函数执行下列步骤：

1. 如下初始化nd参数的某些字段：
   a. 把last_type字段置为LAST_ROOT（如果路径名是一个“/”或“/”序列，那么这是必需的；参见后面的“父路径名查找“一节）。
   b. 把flags字段置为参数flags的值。
   c. 把depth字段置为0。
2. 为进行读操作而获取当前进程的current->fs->lock读/写信号量。
3. 如果路径名的第一个字符是“/“，那么查找操作必须从当前根目录开始：获取相应已安装文件对象(current->fs->rootmnt)和目录项对象(current->fs->root)的地址，增加引用计数器的值，并把它们的地址分别存放在nd->mnt和nd->dentry中。
4. 否则，如果路径名的第一个字符不是“/“，则查找操作必须从当前工作目录开始：获得相应已安装文件系统对象(current->fs->pwdmnt)和目录项对象(current->fs->pwd)的地址，增加引用计数器的值，并把它们的地址分别存放在nd->mnt和nd->dentry中。
5. 释放当前进程的current->fs->lock读/写信号量。
6. 把当前进程描述符中的total_link_count字段置为0（参见后面的“符号链接的查找”一节）。
7. 调用link_path_walk()函数处理正在进行的查找操作：

```
return link_path_walk(name, nd);
```

我们现在准备描述路径名查找操作的核心，也就是link_path_walk()函数。它接收的参数为要解析的路径名指针name和nameidata数据结构的地址nd。

为了简单起见，我们首先描述当LOOKUP_PARENT未被设置且路径名不包含符号链接时，link_path_walk()做些什么(标准路径名查找)。接下来，我们讨论LOOKUP_PARENT被设置的情况：这种类型的查找在创建、删除或更名一个目录项时是需要的，也就是在父目录名查找过程中是需要的。最后，我们阐明该函数如何解析符号链接。

标准路径名查找

当LOOKUP_PARENT标志被清零时，link_path_walk()执行下列步骤：

1. 用nd->flags初始化lookup_flags局部变量。
2. 跳过路径名第一个分量前的任何斜杠（/）。
3. 如果剩余的路径名为空，则返回0。在nameidata数据结构中，dentry和mnt字段指向原路径名最后一个所解析分量对应的对象。
4. 如果nd描述符中的depth字段的值为正，则把lookup_flags局部变量置为LOOKUP_FOLLOW标志（参见“符号链接的查找”一节）。
5. 执行一个循环，把name参数中传递的路径名分解为分量（中间的“/”被当作文件名分隔符对待)；对于每个找到的分量，该函数：
   a. 从nd->dentry->d_inode检索最近一个所解析分量的索引节点对象的地址(在第一次循环中，索引节点指向开始路径名查找的目录)。
   b. 检查存放到索引节点中的最近那个所解析分量的许可权是否允许执行(在Unix中，只有目录是可执行的，它才可以被遍历)。如果索引节点有自定义的permission方法，则执行它；否则，执行exec_permission_lite()函数，该函数检查存放在索引节点i_mode字段的访问模式和运行进程的特权。在两种情况中，如果最近所解析分量不允许执行，那么link_path_walk()跳出循环并返回一个错误码。
   c. 考虑要解析的下一个分量。从它的名字，函数为目录项高速缓存散列表计算一个32位的散列值。
   d. 如果“/”终止了要解析的分量名，则跳过“/”之后的任何尾部“/”。
   e. 如果要解析的分量是原路径名中的最后一个分量，则跳到第6步。
   f. 如果分量名是一个“.“(单个圆点)，则继续下一个分量（“.“指的是当前目录，因此，这个点在目录内没有什么效果)。
   g. 如果分量名是“..“(两个圆点)，则尝试回到父目录：

(1) 如果最近解析的目录是进程的根目录（nd->dentry等于current->fs->root，而nd->mnt等于current->fs->rootmnt），那么再向上追踪是不允许的：在最近解析的分量上调用follow_mount()（见下面），继续下一个分量。

(2) 如果最近解析的目录是nd->mnt文件系统的根目录(nd->dentry等于nd->mnt->mnt_root)，并且这个文件系统也没有被安装在其他文件系统之上（nd->mnt等于nd->mnt->mnt_parent），那么nd->mnt文件系统通常（注7）就是命名空间的根文件系统：在这种情况下，再向上追踪是不可能的，因此在最近解析的分量上调用follow_mount()（参见下面），继续下一个分量。

(3) 如果最近解析的目录是nd->mnt文件系统的根目录，而这个文件系统被安装在其他文件系统之上，那么就需要文件系统交换。因此，把nd->dentry置为nd->mnt->mnt_mountpoint，且把nd->mnt置为nd->mnt->mnt_parent，然后重新开始第5g步（回想一下，几个文件系统可以安装在同一个安装点上）。

(4) 如果最近解析的目录不是已安装文件系统的根目录，那么必须回到父目录：把nd->dentry置为nd->dentry->d_parent，在父目录上调用follow_mount()，继续下一个分量。

follow_mount()函数检查nd->dentry是否是某文件系统的安装点（nd->dentry->d_mounted的值大于0）；如果是，则调用lookup_mnt()搜索目录项高速缓存中已安装文件系统的根目录，并把nd->dentry和nd->mnt更新为相应已安装文件系统的对象地址；然后重复整个操作（几个文件系统可以安装在同一个安装点上）。从本质上说，由于进程可能从某个文件系统的目录开始路径名的查找，而该目录被另一个安装在其父目录上的文件系统所隐藏，那么当需要回到父目录时，则调用follow_mount()函数。

h. 分量名既不是“.”，也不是“..”，因此函数必须在目录项高速缓存中查找它。如果低级文件系统有一个自定义的d_hash目录项方法，则调用它来修改已在第5c步计算出的散列值。

i. 把nd->flags字段中LOOKUP_CONTINUE标志对应的位置位，这表示还有下一个分量要分析。

j. 调用do_lookup()，得到与给定的父目录(nd->dentry)和文件名（要解析的路径名分量）相关的目录项对象。该函数本质上首先调用__d_lookup()在目

注7： 这种情况还可能发生在解除网络文件系统与命名空间的目录树的连接时。

录项高速缓存中搜索分量的目录项对象。如果没有找到这样的目录项对象，则调用real_lookup()。而real_lookup()执行索引节点的lookup方法从磁盘读取目录，创建一个新的目录项对象并把它插入到目录项高速缓存中，然后创建一个新的索引节点对象并把它插入到索引节点高速缓存中（注8）。在这一步结束时，next局部变量中的dentry和mnt字段将分别指向这次循环要解析的分量名的目录项对象和已安装文件系统对象。

k. 调用follow_mount()函数检查刚解析的分量（next.dentry）是否指向某个文件系统安装点的一个目录（next.dentry->d_mounted值大于0）。follow_mount()更新next.dentry和next.mnt的值，以使它们指向由这个路径名分量所表示的目录上安装的最上层文件系统的目录项对象和已安装文件系统对象（参见第5g步）。

l. 检查刚解析的分量是否指向一个符号链接（next.dentry->d_inode具有一个自定义的follow_link方法）。我们将在后面的“符号链接的查找“一节中描述。

m. 检查刚解析的分量是否指向一个目录（next.dentry->d_inode具有一个自定义的lookup方法）。如果没有，返回一个错误码-ENOTDIR，因为这个分量位于原路径名的中间。

n. 把nd->dentry和nd->mnt分别置为next.dentry和next.mnt，然后继续路径名的下一个分量。

6. 现在，除了最后一个分量，原路径名的所有分量都被解析。清除nd->flags中的LOOKUP_CONTINUE标志。

7. 如果路径名尾部有一个“/”，则把lookup_flags局部变量中LOOKUP_FOLLOW和LOOKUP_DIRECTORY标志对应的位置位，以强制由后面的函数来解释最后一个作为目录名的分量。

8. 检查lookup_flags变量中LOOKUP_PARENT标志的值。下面假定这个标志被置为0，并把相反的情况推迟到下一节介绍。

9. 如果最后一个分量名是“.”（单个圆点），则终止执行并返回值0（无错误）。在nd指向的nameidata数据结构中，dentry和mnt字段指向路径名中倒数第二个分量对应的对象（任何分量“.”在路径名中没有效果）。

注8：在少数情况下，函数real_lookup()可能发现所请求的索引节点已经在索引节点高速缓存中。路径名分量是最后一个路径名而且不是指向一个目录、与路径名相应的文件有几个硬链接，并且最近通过与这个路径名中被使用过的硬链接不同的硬链接访问过相应的文件。

10. 如果最后一个分量名是“..”(两个圆点)，则尝试回到父目录：

    a. 如果最后解析的目录是进程的根目录(nd->dentry等于current->fs->root，nd->mnt 等于 current->fs->rootmnt)，则在倒数第二个分量上调用follow_mount()，终止执行并返回值0（无错误)。nd->dentry和nd->mnt指向路径名的倒数第二个分量对应的对象，也就是进程的根目录。

    b. 如果最后解析的目录是 nd->mnt 文件系统的根目录（nd->dentry 等于 nd->mnt->mnt_root)，并且该文件系统没有被安装在另一个文件系统之上（nd->mnt 等于 nd->mnt->mnt_parent)，那么再向上搜索是不可能的，因此在倒数第二个分量上调用 follow_mount()，终止执行并返回值 0（无错误)。

    c. 如果最后解析的目录是nd->mnt文件系统的根目录，并且该文件系统被安装在其他文件系统之上，那么把 nd->dentry 和 nd->mnt 分别置为 nd->mnt->mnt_mountpoint 和 nd->mnt->mnt_parent，然后重新执行第 10 步。

    d. 如果最后解析的目录不是已安装文件系统的根目录，则把 nd->dentry 置为nd->dentry->d_parent，在父目录上调用follow_mount()，终止执行并返回值0（无错误)。nd->dentry和nd->mnt指向前一个分量（即路径名倒数第二个分量）对应的对象。

11. 路径名的最后分量名既不是“.”也不是“..”，因此，必须在高速缓存中查找它。如果低级文件系统有自定义的d_hash目录项方法，则该函数调用它来修改在第5c步已经计算出的散列值。

12. 调用do_lookup()，得到与父目录和文件名相关的目录项对象（参见第5j步)。在这一步结束时，next局部变量存放的是指向最后分量名对应的目录项和已安装文件系统描述符的指针。

13. 调用follow_mount()检查最后一个分量名是否是某个文件系统的一个安装点，如果是，则把next局部变量更新为最上层已安装文件系统根目录对应的目录项对象和已安装文件系统对象的地址。

14. 检查在 lookup_flags 中是否设置了 LOOKUP_FOLLOW 标志，且索引节点对象next.dentry->d_inode是否有一个自定义的follow_link方法。如果是，分量就是一个必须进行解释的符号链接，这将在后面的“符号链接的查找“一节描述。

15. 要解析的分量不是一个符号链接或符号链接不该被解释。把 nd->mnt 和 nd->dentry 字段分别置为 next.mnt 和 next.dentry 的值。最后的目录项对象就是整个查找操作的结果。

16. 检查nd->dentry->d_inode是否为NULL。这发生在没有索引节点与目录项对象关联时，通常是因为路径名指向一个不存在的文件。在这种情况下，返回一个错误码-ENOENT。

17. 路径名的最后一个分量有一个关联的索引节点。如果在 lookup_flags 中设置了 LOOKUP_DIRECTORY 标志，则检查索引节点是否有一个自定义的 lookup 方法，也就是说它是一个目录。如果没有，则返回一个错误码 -ENOTDIR。

18. 返回值 0（无错误）。nd->dentry 和 nd->mnt 指向路径名的最后分量。

父路径名查找

在很多情况下，查找操作的真正目的并不是路径名的最后一个分量，而是最后一个分量的前一个分量。例如，当文件被创建时，最后一个分量表示还不存在的文件的文件名，而路径名中的其余路径指定新链接必须插入的目录。因此，查找操作应当取回最后分量的前一个分量的目录项对象。另举一个例子，把路径名 */foo/bar* 表示的文件 *bar* 拆分出来就包含从目录 *foo* 中移去 *bar*。因此，内核真正的兴趣在于访问文件目录 *foo* 而不是 *bar*。

当查找操作必须解析的是包含路径名最后一个分量的目录而不是最后一个分量本身时，使用 LOOKUP_PARENT 标志。

当 LOOKUP_PARENT 标志被设置时，link_path_walk() 函数也在 nameidata 数据结构中建立 last 和 last_type 字段。last 字段存放路径名中的最后一个分量名。last_type 字段标识最后一个分量的类型；可以把它置为如表 12-17 所示的值之一。

表 12-17：在 nameidata 数据结构中 last_type 字段的值

值	说明
LAST_NORM	最后一个分量是普通文件名
LAST_ROOT	最后一个分量是“/”（也就是整个路径名为“/”）
LAST_DOT	最后一个分量是“.”
LAST_DOTDOT	最后一个分量是“..”
LAST_BIND	最后一个分量是链接到特殊文件系统的符号链接

当整个路径名的查找操作开始时，LAST_ROOT 标志是由 path_lookup() 设置的缺省值（参见“路径名查找”一节开始部分的描述）。如果路径名正好是“/”，则内核不改变 last_type 字段的初始值。

last_type 字段的其他值在 LOOKUP_PARENT 标志置位时由 link_path_walk() 设置；在这种情况下，函数执行前一节描述的步骤，直到第 8 步。不过，从第 8 步往后，路径名中最后一个分量的查找操作是不同的：

1. 把 nd->last 置为最后一个分量名。

2. 把 nd->last_type 初始化为 LAST_NORM。
3. 如果最后一个分量名为“.”（一个圆点），则把 nd->last_type 置为 LAST_DOT。
4. 如果最后一个分量名为“..”（两个圆点），则把nd->last_type置为LAST_DOTDOT。
5. 通过返回值 0（无错误）终止。

你可以看到，最后一个分量根本就没有被解释。因此，当函数终止时，nameidata数据结构的 dentry 和 mnt 字段指向最后一个分量所在目录对应的对象。

符号链接的查找

回想一下，符号链接是一个普通文件，其中存放的是另一个文件的路径名。路径名可以包含符号链接，且必须由内核来解析。

例如，如果 */foo/bar* 是指向（包含路径名）*../dir* 的一个符号链接，那么，*/foo/bar/file* 路径名必须由内核解析为对 */dir/file* 文件的引用。在这个例子中，内核必须执行两个不同的查找操作。第一个操作解析 */foo/bar*：当内核发现 *bar* 是一个符号链接名时，就必须提取它的内容并把它解释为另一个路径名。第二个路径名操作从第一个操作所达到的目录开始，继续到符号链接路径名的最后一个分量被解析。接下来，原来的查找操作从第二个操作所达到的目录项恢复，且有了原目录名中紧随符号链接的分量。

对于更复杂的情景，含有符号链接的路径名可能包含其他的符号链接。你可能认为解析这类符号链接的内核代码是相当难理解的，但并非如此；代码实际上是相当简单的，因为它是递归的。

然而，难以驾驭的递归本质上是危险的。例如，假定一个符号链接指向自己。当然，解析含有这样符号链接的路径名可能导致无休止的递归调用流，这又依次引发内核栈的溢出。当前进程的描述符中的 link_count 字段用来避免这种问题：每次递归执行前增加这个字段的值，执行之后减少其值。如果该字段的值达到 6，整个循环操作就以错误码结束。因此，符号链接嵌套的层数不超过 5。

此外，当前进程的描述符中的total_link_count字段记录在原查找操作中有多少符号链接（甚至非嵌套的）被跟踪。如果这个计数器的值到40，则查找操作中止。没有这个计数器，怀有恶意的用户就可能创建一个病态的路径名，让其中包含很多连续的符号链接，使内核在无休止的查找操作中冻结。

这就是代码基本工作的方式：一旦 link_path_walk()函数检索到与路径名分量相关的目录项对象，就检查相应的索引节点对象是否有自定义的 follow_link 方法（参见“标准路径名查找“一节中的第 51 步和第 14 步）。如果是，索引节点就是一个符号链接，在原路径名的查找操作进行之前就必须先对这个符号链接进行解释。

在这种情况下，link_path_walk()函数调用do_follow_link()，前者传递给后者的参数为符号链接目录项对象的地址dentry和nameidata数据结构的地址nd。do_follow_link()依次执行下列步骤：

1. 检查current->link_count小于5；否则，返回错误码-ELOOP。
2. 检查current->total_link_count小于40；否则，返回错误码-ELOOP。
3. 如果当前进程需要，则调用cond_resched()进行进程交换（设置当前进程描述符thread_info中的TIF_NEED_RESCHED标志）。
4. 递增current->link_count、current->total_link_count和nd->depth的值。
5. 更新与要解析的符号链接关联的索引节点的访问时间。
6. 调用与具体文件系统相关的函数来实现follow_link方法，给它传递的参数为dentry和nd。它读取存放在符号链接索引节点中的路径名，并把这个路径名保存在nd->saved_names数组的合适项中。
7. 调用_ _vfs_follow_link()函数，给它传递的参数为地址nd和nd->saved_names数组中（参见下面）路径名的地址。
8. 如果定义了索引节点对象的put_link方法，就执行它，释放由follow_link方法分配的临时数据结构。
9. 减少current->link_count和nd->depth字段的值。
10. 返回由__vfs_follow_link()函数返回的错误码（0表示无错误）。

__vfs_follow_link()函数本质上依次执行下列操作：

1. 检查符号链接路径名的第一个字符是否是“/“：在这种情况下，已经找到一个绝对路径名，因此没有必要在内存中保留前一个路径的任何信息。如果是，对nameidata数据结构调用path_release()，因此释放由前一个查找步骤产生的对象；然后，设置nameidata数据结构的dentry和mnt字段，以使它们指向当前进程的根目录。
2. 调用link_path_walk()解析符号链的路径名，传递给它的参数为路径名和nd。
3. 返回从link_path_walk()取回的值。

当do_follow_link()最后终止时，它把局部变量next的dentry字段设置为目录项对象的地址，而这个地址由符号链接传递给原先就执行的link_path_walk()。link_path_walk()函数然后进行下一步。

VFS 系统调用的实现

为了简短起见，我们不打算对表 12-1 中列出的所有 VFS 系统调用的实现进行讨论。不过，概略叙述几个系统调用的实现还是有用的，这里仅仅说明VFS的数据结构怎样互相作用。

让我们重新考虑一下在本章开始所提到的例子，用户发出了一条 shell 命令：把 */floppy/TEST* 中的 MS-DOS 文件拷贝到 */tmp/test* 中的 Ext2 文件中。命令 shell 调用一个外部程序（如 *cp*），我们假定 *cp* 执行下列代码片段：

```
inf = open("/floppy/TEST ", O_RDONLY, 0);
outf = open("/tmp/test ", O_WRONLY | O_CREAT | O_TRUNC, 0600);
do {
    len = read(inf, buf, 4096);
    write(outf, buf, len);
} while (len);
close(outf);
close(inf);
```

实际上，真正的 *cp* 程序的代码要更复杂些，因为它还必须检查由每个系统调用返回的可能的出错码。在我们的例子中，我们只把注意力集中在拷贝操作的“正常“行为上。

open()系统调用

open()系统调用的服务例程为 sys_open()函数，该函数接收的参数为：要打开文件的路径名 filename、访问模式的一些标志 flags，以及如果该文件被创建所需要的许可权位掩码 mode。如果该系统调用成功，就返回一个文件描述符，也就是指向文件对象的指针数组 current->files->fd 中分配给新文件的索引；否则，返回 −1。

在我们的例子中，open()被调用两次；第一次是为读（O_RDONLY标志）而打开 */floppy/TEST*，第二次是为写（O_WRONLY 标志）而打开 */tmp/test*。如果 */tmp/test* 不存在，则该文件被创建（O_CREAT标志），文件主对该文件具有独占的读写访问权限（在第三个参数中的八进制数 0600）。

相反，如果该文件已经存在，则从头开始重写它（O_TRUNC 标志）。表 12-18 列出了 open()系统调用的所有标志。

表 12-18：open()系统调用的标志

标志名	说明
O_RDONLY	为读而打开
O_WRONLY	为写而打开

表 12-18：open()系统调用的标志（续）

标志名	说明
O_RDWR	为读和写而打开
O_CREAT	如果文件不存在，就创建它
O_EXCL	对于O_CREAT标志，如果文件已经存在，则失败
O_NOCTTY	从不把文件看作控制终端
O_TRUNC	截断文件（删除所有现有的内容）
O_APPEND	总是在文件末尾写
O_NONBLOCK	没有系统调用在文件上阻塞
O_NDELAY	与O_NONBLOCK相同
O_SYNC	同步写（阻塞，直到物理写终止）
FASYNC	通过信号发出I/O事件通告
O_DIRECT	直接I/O传送（无内核缓冲）
O_LARGEFILE	大型文件（长度大于2GB）
O_DIRECTORY	如果文件不是一个目录，则失败
O_NOFOLLOW	不解释路径名中尾部的符号链接
O_NOATIME	不更新索引节点的上次访问时间

下面来描述一下sys_open()函数的操作。它执行如下操作：

1. 调用getname()从进程地址空间读取该文件的路径名。
2. 调用get_unused_fd()在current->files->fd中查找一个空的位置。相应的索引（新文件描述符）存放在fd局部变量中。
3. 调用filp_open()函数，传递给它的参数为路径名、访问模式标志以及许可权位掩码。这个函数依次执行下列步骤：
   a. 把访问模式标志拷贝到namei_flags标志中，但是，用特殊的格式对访问模式标志O_RDONLY、O_WRONLY和O_RDWR进行编码：如果文件访问需要读特权，那么只设置namei_flags标志的下标为0的位（最低位）；类似地，如果文件访问需要写特权，就只设置下标为1的位。注意，不可能在open()系统调用中不指定文件访问的读或写特权；不过，这种情况在涉及符号链接的路径名查找中则是有意义的。
   b. 调用open_namei()，传递给它的参数为路径名、修改的访问模式标志以及局部nameidata数据结构的地址。该函数以下列方式执行查找操作：

- 如果访问模式标志中没有设置O_CREAT，则不设置LOOKUP_PARENT标志而设置LOOKUP_OPEN标志后开始查找操作。此外，只有O_NOFOLLOW被清零，才设置LOOKUP_FOLLOW标志，而只有设置了O_DIRECTORY标志，才设置LOOKUP_DIRECTORY标志。
- 如果在访问模式标志中设置了O_CREAT，则以LOOKUP_PARENT、LOOKUP_OPEN和LOOKUP_CREATE标志的设置开始查找操作。一旦path_lookup()函数成功返回，则检查请求的文件是否已存在。如果不存在，则调用父索引节点的create方法分配一个新的磁盘索引节点。

open_namei()函数也在查找操作确定的文件上执行几个安全检查。例如，该函数检查与已找到的目录项对象关联的索引节点是否存在、它是否是一个普通文件，以及是否允许当前进程根据访问模式标志访问它。如果文件也是为写打开的，则该函数检查文件是否被其他进程加锁。

c. 调用dentry_open()函数，传递给它的参数为访问模式标志、目录项对象的地址以及由查找操作确定的已安装文件系统对象。该函数依次执行下列操作：

(1) 分配一个新的文件对象。

(2) 根据传递给open()系统调用的访问模式标志初始化文件对象的f_flags和f_mode字段。

(3) 根据作为参数传递来的目录项对象的地址和已安装文件系统对象的地址初始化文件对象的f_fentry和f_vfsmnt字段。

(4) 把f_op字段设置为相应索引节点对象i_fop字段的内容。这就为进一步的文件操作建立起所有的方法。

(5) 把文件对象插入到文件系统超级块的s_files字段所指向的打开文件的链表。

(6) 如果文件操作的open方法被定义，则调用它。

(7) 调用file_ra_state_init()初始化预读的数据结构（参见第十六章）。

(8) 如果O_DIRECT标志被设置，则检查直接I/O操作是否可以作用于文件（参见第十六章）。

(9) 返回文件对象的地址。

d. 返回文件对象的地址。

4. 把current->files->fd[fd]置为由dentry_open()返回的文件对象的地址。

5. 返回fd。

read()和write()系统调用

让我们再回到*cp*例子的代码。open()系统调用返回两个文件描述符，分别存放在inf和outf变量中。然后，程序开始循环。在每次循环中，*/floppy/TEST*文件的一部分被拷贝到本地缓冲区（read()系统调用）中，然后，这个本地缓冲区中的数据又被拷贝到*/tmp/test*文件（write()系统调用）。

read()和write()系统调用非常相似。它们都需要三个参数：一个文件描述符fd、一个内存区的地址buf（该缓冲区包含要传送的数据），以及一个数count（指定应该传送多少字节）。当然，read()把数据从文件传送到缓冲区，而write()执行相反的操作。两个系统调用都返回所成功传送的字节数，或者发送一个错误条件的信号并返回-1。

返回值小于count并不意味着发生了错误。即使请求的字节没有都被传送，也总是允许内核终止系统调用，因此用户应用程序必须检查返回值并重新发出系统调用（如果必要）。在以下几种典型情况下返回小的值：当从管道或终端设备读取时，当读到文件的末尾时，或者当系统调用被信号中断时。文件结束条件（EOF）很容易从read()的空返回值中判断出来。这个条件不会与因信号引起的异常终止混淆在一起，因为如果读取数据之前read()被一个信号中断，则发生一个错误。

读或写操作总是发生在由当前文件指针所指定的文件偏移处（文件对象的f_pos字段）。两个系统调用都通过把所传送的字节数加到文件指针上而更新文件指针。

简而言之，sys_read()（read()的服务例程）和sys_write()（write()的服务例程）几乎都执行相同的步骤：

1. 调用fget_light()从fd获取相应文件对象的地址file（参见前面的“与进程相关的文件“一节）。
2. 如果file->f_mode中的标志不允许所请求的访问（读或写操作），则返回一个错误码-EBADF。
3. 如果文件对象没有read()或aio_read()（write()或aio_write()）文件操作，则返回一个错误码-EINVAL。
4. 调用access_ok()粗略地检查buf和count参数（参见第十章的“验证参数“一节）。
5. 调用rw_verify_area()对要访问的文件部分检查是否有冲突的强制锁。如果有，则返回一个错误码，如果该锁已经被F_SETLKW命令请求，那么就挂起当前进程（参见本章后面的“文件加锁“一节）。
6. 调用file->f_op->read或file->f_op->write方法（如果已定义）来传送数据；否则，调用file->f_op->aio_read或file->f_op->aio_write方法。所有这些

方法（在第十六章讨论）都返回实际传送的字节数。另一方面的作用是，文件指针被适当地更新。

7. 调用 fput_light()释放文件对象。

8. 返回实际传送的字节数。

close()系统调用

在我们例子的代码中，循环结束发生在 read()系统调用返回 0 时，也就是说，发生在 */floppy/TEST* 中的所有字节被拷贝到 */tmp/test* 中时。然后，程序关闭打开的文件，这是因为拷贝操作已经完成。

close()系统调用接收的参数为要关闭文件的文件描述符 fd。sys_close()服务例程执行下列操作：

1. 获得存放在 current->files->fd[fd]中的文件对象的地址；如果它为 NULL，则返回一个出错码。

2. 把 current->files->fd[fd]置为 NULL。释放文件描述符 fd，这是通过清除 current->files 中的 open_fds 和 close_on_exec 字段的相应位来进行的（参见第二十章中有关关闭执行标志的内容）。

3. 调用 filp_close()，该函数执行下列操作：

   a. 调用文件操作的 flush 方法（如果已定义）。

   b. 释放文件上的任何强制锁（参见下一节）。

   c. 调用 fput()释放文件对象。

4. 返回 0 或一个出错码。出错码可由 flush 方法或文件中的前一个写操作错误产生。

文件加锁

当一个文件可以被多个进程访问时，就会出现同步问题。如果两个进程试图对文件的同一位置进行写会出现什么情况？或者，如果一个进程从文件的某个位置进行读而另一个进程正在对同一位置进行写会出现什么情况？

在传统的 Unix 系统中，对文件同一位置的同时访问会产生不可预料的结果。但是，Unix 系统提供了一种允许进程对一个文件区进行加锁的机制，以使同时访问可以很容易地被避免。

POSIX标准规定了基于fcntl()系统调用的文件加锁机制。这样就有可能对文件的任意一部分（甚至一个字节）加锁或对整个文件（包含以后要追加的数据）加锁。因为进程可以选择仅仅对文件的一部分加锁，因此，它也可以在文件的不同部分保持多个锁。

这种锁并不把不知道加锁的其他进程关在外面。与用于保护代码中临界区的信号量类似，可以认为这种锁起“劝告“的作用，因为只有在访问文件之前其他进程合作检查锁的存在时，锁才起作用。因此，POSIX的锁被称为劝告锁（*advisory lock*）。

传统的BSD变体通过flock()系统调用来实现劝告锁。这个调用不允许进程对文件的一个区字段进行加锁，而只能对整个文件进行加锁。传统的System V变体提供了lockf()库函数，它仅仅是fcntl()的一个接口。

更重要的是，System V Release 3引入了强制加锁（*mandatory locking*）：内核检查open()、read()和write()系统调用的每次调用都不违背在所访问文件上的强制锁。因此，强制锁甚至在非合作的进程之间也被强制加上（注9）。

不管进程是使用劝告锁还是强制锁，它们都可以使用共享读锁和独占写锁。在文件的某个区字段上，可以有任意多个进程进行读，但在同一时刻只能有一个进程进行写。此外，当其他进程对同一个文件都拥有自己的读锁时，就不可能获得一个写锁，反之亦然。

Linux文件加锁

Linux支持所有的文件加锁方式：劝告锁和强制锁，以及fcntl()、flock()和lockf()系统调用。不过，lockf()系统调用仅仅是一个标准的库函数。

flock()系统调用不管MS_MANDLOCK安装标志如何设置，只产生劝告锁。这是任何类Unix操作系统所期望的系统调用行为。在Linux中，增加了一种特殊的flock()强制锁，以允许对专有的网络文件系统的实现提供适当的支持。这就是所谓的共享模式强制锁：当这个锁被设置时，其他任何进程都不能打开与锁访问模式冲突的文件。不鼓励本地Unix应用程序中使用这个特征，因为这样加锁的源代码是不可移植的。

在Linux中还引入了另一种基于fcntl()的强制锁，叫做租借锁（*lease*）。当一个进程试图打开由租借锁保护的文件时，它照样被阻塞。然而，拥有锁的进程接收到一个信号。

注9：很奇怪，即使一个进程拥有某个文件上的强制锁，其他某个进程仍然会解除链接（或删除）这个文件。这种令人困惑的情况是可能发生的，因为当一个进程删除文件硬链接时，它不修改其内容，而只是修改它的父目录的内容。

一旦该进程得到通知，它应当首先更新文件，以使文件的内容保持一致，然后释放锁。如果拥有者不在预定的时间间隔（可以通过在*/proc/sys/fs/lease-break-time*文件中写入秒数来进行调整，通常为45s）内这么做，则租借锁由内核自动删除，且允许阻塞的进程继续执行。

进程可以采用以下两种方式获得或释放一个文件劝告锁：

- 发出flock()系统调用。传递给它的两个参数为文件描述符fd和指定锁操作的命令。该锁应用于整个文件。
- 使用fcntl()系统调用。传递给它的三个参数为文件描述符fd、指定锁操作的命令以及指向flock结构的指针（参见表12-20）。flock结构中的几个字段允许进程指定要加锁的文件部分。因此进程可以在同一文件的不同部分保持几个锁。

fcntl()和flock()系统调用可以在同一文件上同时使用，但是通过fcntl()加锁的文件看起来与通过flock()加锁的文件不一样，反之亦然。这样当应用程序使用一种依赖于某个库的锁，而该库同时使用另一种类型的锁时，可以避免发生死锁。

处理强制文件锁要更复杂些。步骤如下：

1. 安装文件系统时强制锁是必需的，可使用mount命令的-o mand选项在mount()系统调用中设置MS_MANDLOCK标志。缺省操作是不使用强制锁。
2. 通过设置文件的set-group位（SGID）和清除group-execute许可权位将它们标记为强制锁的候选者。因为当group-execute位为0时，set-group位也没有任何意义，因此内核将这种合并解释成使用强制锁而不是劝告锁。
3. 使用fcntl()系统调用（参见下面）获得或释放一个文件锁。

处理租借锁比处理强制锁要容易得多：调用具有F_SETLEASE或F_GETLEASE命令的系统调用fcntl()就足够了。使用另一个带有F_SETSIG命令的fcntl()系统调用可以改变传送给租借锁进程拥有者的信号类型。

当维护所有可以修改文件内容的系统调用时，除了read()和write()系统调用中的检查以外，内核还需要考虑强制锁的存在性。例如，如果文件中存在任何强制锁，那么带有O_TRUNC标志的open()系统调用就会失效。

下一节描述内核使用的主要数据结构，它们用于处理由flock()(FL_FLOCK锁)和fcntl()系统调用（FL_POSIX锁）实现的文件锁。

文件锁的数据结构

Linux 中所有类型的锁都是由相同的 file_lock 数据结构描述的，它的字段如表 12-19 所示。

表12-19：file_lock数据结构的字段

类型	字段	说明
struct file_lock *	fl_next	与索引节点关联的锁链表中的下一个元素
struct list_head	fl_link	用于活动或阻塞链表的指针
struct list_head	fl_block	用于锁的等待者链表的指针
struct files_struct *	fl_owner	文件所有者的 files_struct
unsigned int	fl_pid	进程拥有者的 PID
wait_queue_head_t	fl_wait	阻塞进程的等待队列
struct file *	fl_file	指向文件对象的指针
unsigned char	fl_flags	锁标志
unsigned char	fl_type	锁类型
loff_t	fl_start	被锁区字段的起始偏移量
loff_t	fl_end	被锁区字段的末尾偏移量
struct fasync_struct *	fl_fasync	用于租借锁中断通知
unsigned long	fl_break_time	租借结束前的剩余时间
struct file_lock_operations *	fl_ops	指向文件锁操作的指针
struct lock_manager_operations *	fl_mops	指向锁管理操作的指针
Union	fl_u	具体文件系统的信息

指向磁盘上同一文件的所有 lock_file 结构都被收集在一个单向链表中，其第一个元素由索引节点对象的 i_flock 字段所指向。file_lock 结构的 fl_next 字段指向链表中的下一个元素。

当发出阻塞系统调用的进程请求一个独占锁而同一文件也存在共享锁时，该请求不能立即得到满足，并且进程必须被挂起。因此该进程被插入到由阻塞锁 file_lock 结构的 fl_wait 字段指向的等待队列中。使用两个链表区分已满足的锁请求（活动锁）和那些不能立刻得到满足的锁请求（阻塞锁）。

所有的活动锁被链接在“全局文件锁链表”中，该表的首元素被存放在 file_lock_list 变量中。类似地，所有的阻塞锁被链接在“阻塞链表“中，该表的首元素被存放在

blocked_list变量中。使用fl_link字段可把lock_file结构插入到上述任何一个链表中。

最后的一项要点是，内核必须跟踪所有与给定活动锁（“blocker”）关联的阻塞锁（“waiters”）：这就是为什么要使用链表根据给定的blocker把所有的waiter链接在一起的原因。blocker的fl_block字段是链表的伪首部，而waiter的fl_block字段存放了指向链表中相邻元素的指针。

FL_FLOCK锁

FL_LOCK锁总是与一个文件对象相关联，因此由一个打开该文件的进程（或共享同一打开文件的子进程）来维护。当一个锁被请求或允许时，内核就把进程保持在同一文件对象上的任何其他锁都替换掉。这只发生在进程想把一个已经拥有的读锁改变为一个写锁，或把一个写锁改变为一个读锁时。此外，当fput()函数正在释放一个文件对象时，对这个文件对象加的所有FL_LOCK锁都被撤销。不过，也有可能由其他进程对这同一文件（索引节点）设置了其他FL_LOCK读锁，它们依然是有效的。

flock()系统调用允许进程在打开文件上申请或删除劝告锁。它作用于两个参数：要加锁文件的文件描述符fd和指定锁操作的参数cmd。如果cmd参数为LOCK_SH，则请求一个共享的读锁；为LOCK_EX，则请求一个互斥的写锁；为LOCK_UN，则释放一个锁（注10）。

如果请求不能立即得到满足，系统调用通常阻塞当前进程，例如，如果进程请求一个独占锁而其他某个进程已获得了该锁。不过，如果LOCK_NB标记与LOCK_SH或LOCK_EX操作进行“或“，则这个系统调用不阻塞；换句话说，如果不能立即获得该锁，则该系统调用就返回一个错误码。

当sys_flock()服务例程被调用时，则执行下列步骤：

1. 检查fd是否是一个有效的文件描述符；如果不是，就返回一个错误码。否则，获得相应文件对象filp的地址。
2. 检查进程在打开文件上是否有读和/或写权限；如果没有，就返回一个错误码。
3. 获得一个新的file_lock对象锁并用适当的锁操作初始化它：根据参数cmd的值设置fl_type字段，把fl_file字段设为文件对象filp的地址，fl_flags字段设为FL_FLOCK，fl_pid字段设为current->tgid，并把fl_end字段设为-1，这表示对整个文件（而不是文件的一部分）加锁的事实。

注10：实际上，flock()系统调用还可以通过指定LOCK_MAND命令建立共享模式强制锁。不过，对此我们不做更多的讨论。

4. 如果参数 cmd 不包含 LOCK_NB 位，则把 FL_SLEEP 标志加入 fl_flags 字段。
5. 如果文件具有一个 flock 文件操作，则调用它，传递给它的参数为文件对象指针 filp、一个标志（F_SETLKW 或 F_SETLK，取决于 LOCK_NB 位的值）以及新的 file_lock 对象锁的地址。
6. 否则，如果没有定义flock文件操作(通常情况下)，则调用flock_lock_file_wait()试图执行请求的锁操作。传递给它的两个参数为：文件对象指针 filp和在第 3 步创建的新的 file_lock 对象的地址 lock。
7. 如果上一步中还没有把 file_lock 描述符插入活动或阻塞链表中，则释放它。
8. 返回 0（成功）。

flock_lock_file_wait()函数执行下列循环操作：

1. 调用 flock_lock_file()，传递给它的参数为文件对象指针 filp 和新的 file_lock 对象锁的地址 lock。这个函数依次执行下列操作：
   a. 搜索filp->f_dentry->d_inode->i_flock指向的链表。如果在同一文件对象中找到 FL_FLOCK 锁，则检查它的类型（LOCK_SH 或 LOCK_EX）：如果该锁的类型与新锁相同，则返回 0（什么也没有做）。否则，从索引节点锁链表和全局文件锁链表中删除这个 file_lock 元素，唤醒 fl-block 链表中在该锁的等待队列上睡眠的所有进程，并释放 file_lock 结构。
   b. 如果进程正在执行开锁(LOCK_UN)，则什么事情都不需要做：该锁已不存在或已被释放，因此返回 0。
   c. 如果已经找到同一个文件对象的FL_FLOCK锁——表明进程想把一个已经拥有的读锁改变为一个写锁（反之亦然），那么调用 cond_resched()给予其他更高优先级进程（特别是先前在原文件锁上阻塞的任何进程）一个运行的机会。
   d. 再次搜索索引节点锁链表以验证现有的FL_FLOCK锁并不与所请求的锁冲突。在索引节点链表中，肯定没有 FL_FLOCK 写锁，此外，如果进程正在请求一个写锁，那么根本就没有 FL_FLOCK 锁。
   e. 如果不存在冲突锁，则把新的file_lock结构插入索引节点锁链表和全局文件锁链表中，然后返回 0（成功）。
   f. 发现一个冲突锁：如果 fl_flags 字段中 FL_SLEEP 对应的标志位置位，则把新锁（waiter 锁）插入到 blocker 锁循环链表和全局阻塞链表中。
   g. 返回一个错误码 -EAGAIN。
2. 检查 flock_lock_file()的返回码：

a. 如果返回码为0（没有冲突迹象），则返回0（成功）。

b. 不相容的情况。如果fl_flags字段中的FL_SLEEP标志被清除，就释放file_lock锁描述符，并返回一个错误码-EAGAIN。

c. 否则，不相容但进程能够睡眠的情况：调用wait_event_interruptible()把当前进程插入到lock->fl_wait等待队列中并挂起它。当进程被唤醒时（正好在释放blocker锁后），跳转到第1步再次执行这个操作。

FL_POSIX锁

FL_POSIX锁总是与一个进程和一个索引节点相关联。当进程死亡或一个文件描述符被关闭时（即使该进程对同一文件打开了两次或复制了一个文件描述符），这种锁会被自动地释放。此外，FL_POSIX锁绝不会被子进程通过fork()继承。

当使用fcntl()系统调用对文件加锁时，该系统调用作用于三个参数：要加锁文件的文件描述符fd、指向锁操作的参数cmd，以及指向存放在用户态进程地址空间中的flock数据结构（注11）的指针fl。flock结构中的字段如表12-20所示。

表12-20：flock数据结构中的字段

类型	字段	说明
short	l_type	F_RDLOCK（请求一个共享锁），F_WRLOCK（请求一个独占锁），F_UNLOCK（释放锁）
short	l_whence	SEEK_SET（从文件的开始处），SEEK_CURRENT（从当前文件指针处），SEEK_END（从文件末尾处）
off_t	l_start	与l_whence的值相关的加锁区域的初始偏移量
off_t	l_len	加锁区域的长度（0表示该区域包含所有可能写过当前文件末尾的区域）
pid_t	l_pid	拥有者的PID

sys_fcntl()服务例程执行的操作取决于在cmd参数中所设置的标志值：

F_GETLK

确定由flock结构描述的锁是否与另一个进程已获得的某个FL_POSIX锁互相冲突。在冲突的情况下，用现有锁的有关信息重写flock结构。

注11：Linux还定义了flock64结构，它的offset和length字段使用64位长整数。下面我们主要讨论flock数据结构，这些内容对flock64同样有效。

F_SETLK

设置由flock结构描述的锁。如果不能获得该锁，则这个系统调用返回一个错误码。

F_SETLKW

设置由flock结构描述的锁。如果不能获得该锁，则这个系统调用阻塞，也就是说，调用进程进入睡眠状态直到该锁可用时为止。

F_GETLK64，F_SETLK64，F_SETLKW64

与前面描述的几个标志相同，但是使用的是flock64结构而不是flock结构。

sys_fcntl()服务例程首先获取与参数fd对应的文件对象，然后调用fcntl_getlk()或fcntl_setlk()函数（这取决于传递的参数：F_GETLK表示前一个函数，F_SETLK或F_SETLKW表示后一个函数）。我们仅仅考虑第二种情况。

fcntl_setlk()函数作用于三个参数：指向文件对象的指针filp、cmd命令（F_SETLK或F_SETLKW），以及指向flock数据结构的指针。该函数执行下列操作：

1. 读取局部变量中的参数fl所指向的flock结构。
2. 检查这个锁是否应该是一个强制锁，且文件是否有一个共享内存映射（参见第十六章的“内存映射“一节）。在肯定的情况下，该函数拒绝创建锁并返回-EAGAIN出错码，说明文件正在被另一个进程访问。
3. 根据用户flock结构的内容和存放在文件索引节点中的文件大小，初始化一个新的file_lock结构。
4. 如果命令cmd为F_SETLKW，则该函数把file_lock结构的fl_flags字段设为FL_SLEEP标志对应的位置位。
5. 如果flock结构中的l_type字段为F_RDLCK，则检查是否允许进程从文件读取；类似地，如果l_type为F_WRLCK，则检查是否允许进程写入文件。如果都不是，则返回一个出错码。
6. 调用文件操作的lock方法（如果已定义）。对于磁盘文件系统，通常不定义该方法。
7. 调用__posix_lock_file()函数，传递给它的参数为文件的索引节点对象地址以及file_lock对象地址。该函数依次执行下列操作：
   a. 对于索引节点的锁链表中的每个FL_POSIX锁，调用posix_locks_conflict()。该函数检查这个锁是否与所请求的锁互相冲突。从本质上说，在索引节点的链表中，必定没有用于同一区的FL_POSIX写锁，并且，如果进程正在请求一个写锁，那么同一个区字段也可能根本没有FL_POSIX锁。但是，同一个进程所拥有的锁从不会冲突；这就允许进程改变它已经拥有的锁的特性。

b. 如果找到一个冲突锁，则检查是否以F_SETLKW标志调用fcntl()。如果是，当前进程应当被挂起：在这种情况下，调用posix_locks_deadlock()来检查在等待FL_POSIX锁的进程之间没有产生死锁条件，然后把新锁（waiter锁）插入到冲突锁（blocker锁）blocker链表和阻塞链表中，最后返回一个出错码。否则，如果以F_SETLK标志调用fcntl()，则返回一个出错码。

c. 只要索引节点的锁链表中不包含冲突的锁，就检查把文件区重叠起来的当前进程的所有FL_POSIX锁，当前进程想按需要对文件区中相邻的区字段进行锁定、组合及拆分。例如，如果进程为某个文件区请求一个写锁，而这个文件区落在一个较宽的读锁区字段内，那么，以前的读锁就会被拆分为两部分，这两部分覆盖非重叠区域，而中间区域由新的写锁进行保护。在重叠的情况下，新锁总是代替旧锁。

d. 把新的file_lock结构插入到全局锁链表和索引节点链表中。

e. 返回值0（成功）。

8. 检查__posix_lock_file()的返回码：

a. 如果返回码为0（没有冲突迹象），则返回0（成功）。

b. 不相容的情况。如果fl_flags字段的FL_SLEEP标志被清除，就释放新的file_lock描述符，并返回一个错误码-EAGAIN。

c. 否则，如果不相容但进程能够睡眠时，调用wait_event_interruptible()把当前进程插入到lock->fl_wait等待队列中并挂起它。当进程被唤醒时（正好在释放blocker锁后），跳转到第7步再次执行这个操作。

第十三章



I/O 体系结构和设备驱动程序

上一章的虚拟文件系统在某种意义上依靠低层函数以进行适合于每个设备的读、写或其他操作。前一章还包括对不同文件系统如何处理这些操作的简单讨论。本章我们将看一下内核如何在实际的设备上调用这些操作。

在“I/O 体系结构”一节，我们简单考察一下 80x86 的 I/O 体系结构。在“设备驱动程序模型”一节，我们介绍 Linux 设备驱动程序模型。接下来，在“设备文件”一节，我们说明 VFS 如何把叫做“设备文件”的特殊文件与每个不同的硬件设备相对应，从而使应用程序可以用相同的方式使用所有的设备。在“设备驱动程序”一节，我们介绍一些常用的设备驱动程序特性。最后，在“字符设备驱动程序”一节，我们说明 Linux 字符设备驱动程序的整体组织结构。我们将在下一章中讨论块设备驱动程序。

有兴趣自己开发设备驱动程序的读者最好参考由 O'Reilly 出版，Jonathan Corbet Alessandro Rubini 和 Greg Kroah-Hartman 编写的《Linux Device Drivers》（第三版）一书。

I/O 体系结构

为了确保计算机能够正常工作，必须提供数据通路，让信息在连接到个人计算机的 CPU、RAM 和 I/O 设备之间流动。这些数据通路总称为总线，担当计算机内部主通信通道的作用。

所有计算机都拥有一条系统总线，它连接大部分内部硬件设备。一种典型的系统总线是 PCI（*Peripheral Component Interconnect*）总线。目前使用其他类型的总线也很多，例

如ISA、EISA、MCA、SCSI和USB。典型的情况是，一台计算机包括几种不同类型的总线，它们通过被称作“桥”的硬件设备连接在一起。两条高速总线用于在内存芯片上来回传送数据：前端总线将CPU连接到RAM控制器上，而后端总线将CPU直接连接到外部硬件的高速缓存上。主机上的桥将系统总线和前端总线连接在一起。

任何I/O设备有且仅能连接一条总线。总线的类型影响I/O设备的内部设计，也影响着内核如何处理设备。本节我们将讨论所有PC体系结构共有的功能性特点，而不具体介绍特定总线类型的技术细节。

CPU和I/O设备之间的数据通路通常称为I/O总线。80x86微处理器使用16位的地址总线对I/O设备进行寻址，使用8位、16位或32位的数据总线传输数据。每个I/O设备依次连接到I/O总线上，这种连接使用了包含3个元素的硬件组织层次：I/O端口、接口和设备控制器。图13-1显示了I/O体系结构的这些成分。

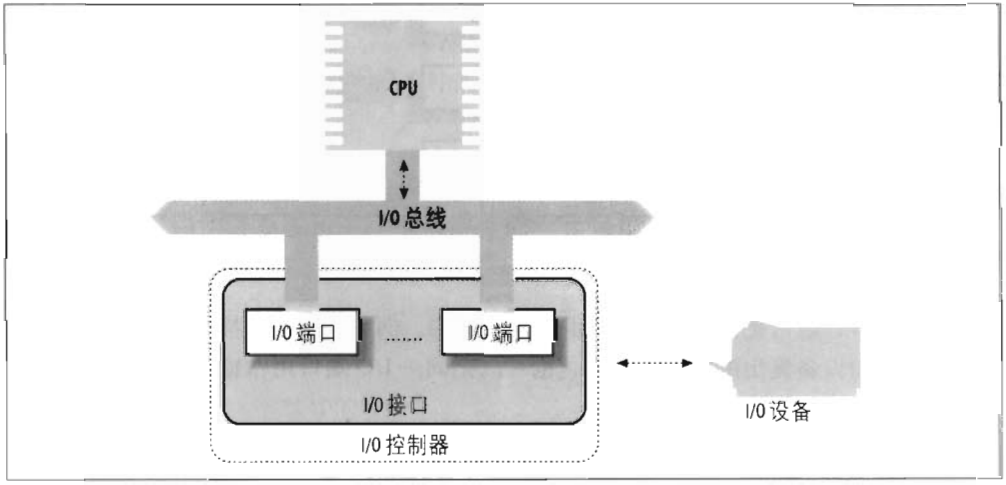


图13-1：PC的I/O体系结构

I/O 端口

每个连接到I/O总线上的设备都有自己的I/O地址集，通常称为I/O端口（I/O port）。在IBM PC体系结构中，I/O地址空间一共提供了65536个8位的I/O端口。可以把两个连续的8位端口看成一个16位端口，但是这必须从偶数地址开始。同理，也可以把两个连续的16位端口看成一个32位端口，但是这必须是从4的整数倍地址开始。有四条专用的汇编语言指令可以允许CPU对I/O端口进行读写，它们是in、ins、out和outs。在执行其中的一条指令时，CPU使用地址总线选择所请求的I/O端口，使用数据总线在CPU寄存器和端口之间传送数据。

I/O端口还可以被映射到物理地址空间。因此，处理器和I/O设备之间的通信就可以使用对内存直接进行操作的汇编语言指令（例如，mov、and、or等等）。现代的硬件设备更倾向于映射的I/O，因为这样处理的速度较快，并可以和DMA结合起来。

系统设计者的主要目的是对I/O编程提供统一的方法，但又不牺牲性能。为了达到这个目的，每个设备的I/O端口都被组织成如图13-2所示的一组专用寄存器。CPU把要发送给设备的命令写入设备控制寄存器（*device control register*），并从设备状态寄存器（*device status register*）中读出表示设备内部状态的值。CPU还可以通过读取设备输入寄存器（*device input register*）的内容从设备取得数据，也可以通过向设备输出寄存器（*device output register*）中写入字节而把数据输出到设备。

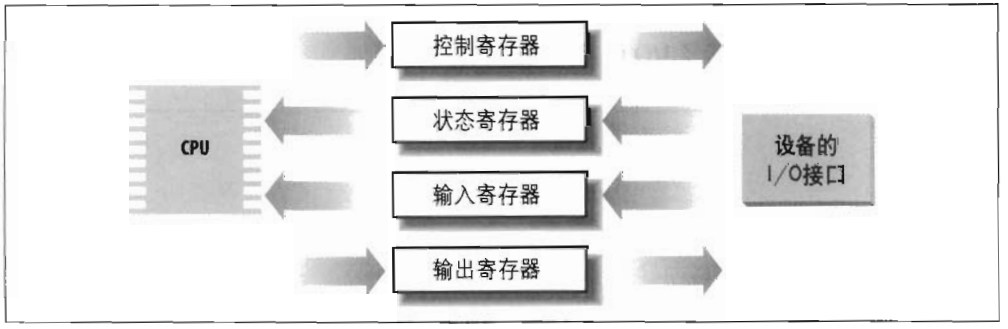


图13-2：专用I/O端口

为了降低成本，通常把同一I/O端口用于不同目的。例如，某些位描述设备的状态，而其他位指定向设备发出的命令。同理，也可以把同一I/O端口用作输入寄存器或输出寄存器。

访问I/O端口

in、out、ins和outs汇编语言指令都可以访问I/O端口。内核中包含了以下辅助函数来简化这种访问：

inb()，inw()，inl()

分别从I/O端口读取1、2或4个连续字节。后缀“b”、“w”、“l”分别代表一个字节（8位）、一个字（16位）以及一个长整型（32位）。

inb_p()，inw_p()，inl_p()

分别从I/O端口读取1、2或4个连续字节，然后执行一条“哑元（dummy，即空指令）”指令使CPU暂停。

`outb()`, `outw()`, `outl()`

分别向一个I/O端口写入1、2或4个连续字节。

`outb_p()`, `outw_p()`, `outl_p()`

分别向一个I/O端口写入1、2或4个连续字节，然后执行一条“哑元”指令使CPU暂停。

`insb()`, `insw()`, `insl()`

分别从I/O端口读取以1、2或4个字节为一组的连续字节序列。字节序列的长度由该函数的参数给出。

`outsb()`, `outsw()`, `outsl()`

分别向I/O端口写入以1、2或4个字节为一组的连续字节序列。

虽然访问I/O端口非常简单，但是检测哪些I/O端口已经分配给I/O设备可能就不这么简单了，对基于ISA总线的系统来说更是如此。通常，I/O设备驱动程序为了探测硬件设备，需要盲目地向某一I/O端口写入数据；但是，如果其他硬件设备已经使用了这个端口，那么系统就会崩溃。为了防止这种情况的发生，内核必须使用“资源”来记录分配给每个硬件设备的I/O端口。

资源（*resource*）表示某个实体的一部分，这部分被互斥地分配给设备驱动程序。在我们的情况中，一个资源表示I/O端口地址的一个范围。每个资源对应的信息存放在`resource`数据结构中，其字段如表13-1所示。所有的同种资源都插入到一个树型数据结构中；例如，表示I/O端口地址范围的所有资源都包含在一个根节点为ioport_resource的树中。

表13-1：resource数据结构中的字段

类型	字段	说明
`const char *`	`name`	资源拥有者的描述
`unsigned long`	`start`	资源范围的开始
`unsigned long`	`end`	资源范围的结束
`unsigned long`	`flags`	各种标志
`struct resource *`	`parent`	指向资源树中父亲的指针
`struct resource *`	`sibling`	指向资源树中兄弟的指针
`struct resource *`	`child`	指向资源树中第一个孩子的指针

节点的孩子被收集在一个链表中，其第一个元素由`child`指向。`sibling`字段指向链表中的下一个节点。

为什么使用树？例如，考虑一下IDE硬盘接口所使用的I/O端口地址——比如说从0xf000到0xf00f。然后，start字段为0xf000且end字段为0xf00f的这样一个资源包含在树中，控制器的常规名字存放在name字段中。但是，IDE设备驱动程序需要记住另外的信息，也就是IDE链（IDE chain）的主盘（master disk）使用0xf000～0xf007的子范围，从盘（slave disk）使用0xf008～0xf00f的子范围。为了做到这点，设备驱动程序把两个子范围对应的孩子插入到0xf000～0xf00f的整个范围对应的资源下。一般来说，树中的每个节点肯定相当于父节点对应范围的一个子范围。I/O端口资源树(ioport_resource)的根节点跨越了整个I/O地址空间（从端口0～65535）。

任何设备驱动程序都可以使用下面三个函数，传递给它们的参数为资源树的根节点和要插入的新资源数据结构的地址：

request_resource()
: 把一个给定范围分配给一个I/O设备。

allocate_resource()
: 在资源树中寻找一个给定大小和排列方式的可用范围；若存在，就将这个范围分配给一个I/O设备（主要由PCI设备驱动程序使用，这种驱动程序可以配置成使用任意的端口号和主板上的内存地址对其进行配置）。

release_resource()
: 释放以前分配给I/O设备的给定范围。

内核也为以上应用于I/O端口的函数定义了一些快捷函数：request_region()分配I/O端口的给定范围，release_region()释放以前分配给I/O端口的范围。当前分配给I/O设备的所有I/O地址的树都可以从*/proc/ioports*文件中获得。

I/O接口

I/O接口（*I/O interface*）是处于一组I/O端口和对应的设备控制器之间的一种硬件电路。它起翻译器的作用，即把I/O端口中的值转换成设备所需要的命令和数据。在相反的方向上，它检测设备状态的变化，并对起状态寄存器作用的I/O端口进行相应的更新。还可以通过一条IRQ线把这种电路连接到可编程中断控制器上，以使它代表相应的设备发出中断请求。

有两种类型的接口：

专用I/O接口
: 专门用于一个特定的硬件设备。在一些情况下，设备控制器与这种I/O接口处于同

一块卡中（注 1）。连接到专用 I/O 接口上的设备可以是内部设备（位于 PC 机箱内部的设备），也可以是外部设备（位于 PC 机箱外部的设备）。

通用 I/O 接口

用来连接多个不同的硬件设备。连接到通用 I/O 接口上的设备通常都是外部设备。

专用 I/O 接口

专用 I/O 接口的种类很多，因此目前已装在 PC 上设备的种类也很多，我们无法一一列出，在此只列出一些最通用的接口：

键盘接口

连接到一个键盘控制器上，这个控制器包含一个专用微处理器。这个微处理器对按下的组合键进行译码，产生一个中断并把相应的键盘扫描码写入输入寄存器。

图形接口

和图形卡中对应的控制器封装在一起，图形卡有自己的帧缓冲区，还有一个专用处理器以及存放在只读存储器（ROM）芯片中的一些代码。帧缓冲区是显卡上固化的存储器，其中存放的是当前屏幕内容的图形描述。

磁盘接口

由一条电缆连接到磁盘控制器，通常磁盘控制器与磁盘放在一起。例如，IDE 接口由一条40线的带形电缆连接到智能磁盘控制器上，在磁盘本身就可以找到这个控制器。

总线鼠标接口

由一条电缆把接口和控制器连接在一起，控制器就包含在鼠标中。

网络接口

与网卡中的相应控制器封装在一起，用以接收或发送网络报文。虽然广泛采用的网络标准很多，但还是以太网(IEEE 802.3)最为通用。

通用 I/O 接口

现代 PC 都包含连接很多外部设备的几个通用 I/O 接口。最常用的接口有：

并口

传统上用于连接打印机，它还可以用来连接可移动磁盘、扫描仪、备份设备、其他计算机等等。数据的传送以每次 1 字节（8 位）为单位进行。

注 1：每块卡都要插入PC的一个可用空闲总线插槽中。如果一块卡通过一条外部电缆连接到一个外部设备上，那么在 PC 后面的面板中就有一个对应的连接器。

串口

与并口类似，但数据的传送是逐位进行的。串口包括一个通用异步收发器（UART）芯片，它可以把要发送的字节信息拆分成位序列，也可以把接收到的位流重新组装成字节信息。由于串口本质上速度低于并口，因此主要用于连接那些不需要高速操作的外部设备，如调制解调器、鼠标以及打印机。

PCMCIA 接口

大多数便携式计算机都包含这种接口。在不重新启动系统的情况下，这种形状类似于信用卡的外部设备可以被插入插槽或从插槽中拔走。最常用的PCMCIA设备是硬盘、调制解调器、网卡和扩展RAM。

SCSI（小型计算机系统接口）接口

是把PC主总线连接到次总线（称为SCSI总线）的电路。SCSI-2总线允许一共8个PC和外部设备（硬盘、扫描仪、CR-ROM刻录机等等）连接在一起。如果有附加接口，宽带SCSI-2和新的SCSI-3接口可以允许你连接多达16个以上的设备。SCSI标准是通过SCSI总线连接设备的通信协议。

通用串行总线（USB）

高速运转的通用I/O接口，可用于连接外部设备，代替传统的并口、串口以及SCSI接口。

设备控制器

复杂的设备可能需要一个设备控制器（*device controller*）来驱动。从本质上说，控制器起两个重要作用：

- 对从I/O接口接收到的高级命令进行解释，并通过向设备发送适当的电信号序列强制设备执行特定的操作。
- 对从设备接收到的电信号进行转换和适当地解释，并修改（通过I/O接口）状态寄存器的值。

典型的设备控制器是磁盘控制器，它从微处理器（通过I/O接口）接收诸如“写这个数据块”之类的高级命令，并将其转换成诸如“把磁头定位在正确的磁道上”和“把数据写入这个磁道”之类的低级磁盘操作。现代的磁盘控制器相当复杂，因为它们可以把磁盘数据快速保存到内存的高速缓存中，还可以根据实际磁盘的几何结构重新安排CPU的高级请求，使其最优化。

比较简单的设备没有设备控制器，可编程中断控制器（参见第四章中的“中断和异常”一节）和可编程间隔定时器（参见第六章中的“可编程间隔定时器(PIT)一节）”就是这样的设备。

很多硬件设备都有自己的存储器，通常称之为I/O共享存储器。例如，所有比较新的图形卡在帧缓冲区中都有几MB的RAM，用它来存放要在屏幕上显示的屏幕映像。我们将在本章的“访问I/O共享存储器”一节中讨论I/O共享存储器。

设备驱动程序模型

Linux内核的早期版本为设备驱动程序的开发者提供微不足道的基本功能：分配动态内存，保留I/O地址范围或中断请求（IRQ），激活一个中断服务例程来响应设备的中断。事实上，在更老的硬件设备上编程棘手而困难重重，还有即使两种不同的硬件设备连在同一条总线上，但二者也很少有共同点。因此，试图为这种硬件设备的驱动程序开发者提供一种统一的模型是难以做到的。

现在的情形大不一样。诸如PCI这样的总线类型对硬件设备的内部设计提出了强烈的要求；因此，新的硬件设备即使类型不同但也有相似的功能。对这种设备的驱动程序应当特别关注：

- 电源管理（控制设备电源线上不同的电压级别）
- 即插即用（配置设备时透明的资源分配）
- 热插拔（系统运行时支持设备的插入和移走）

系统中所有硬件设备由内核全权负责电源管理。例如，在以电池供电的计算机进入“待机”状态时，内核应立刻强制每个硬件设备（硬盘、显卡、声卡、网卡、总线控制器等等）处于低功率状态。因此，每个能够响应“待机”状态的设备驱动程序必须包含一个回调函数，它能够使得硬件设备处于低功率状态。而且，硬件设备必须按准确的顺序进入“待机”状态，否则一些设备可能会处于错误的电源状态。例如，内核必须首先将硬盘置于“待机”状态，然后才是它们的磁盘控制器，因为若按照相反的顺序执行，磁盘控制器就不能向硬盘发送命令。

为了实现这些操作，Linux 2.6提供了一些数据结构和辅助函数，它们为系统中所有的总线、设备以及设备驱动程序提供了一个统一的视图；这个框架被称为设备驱动程序模型。

sysfs文件系统

*sysfs*文件系统是一种特殊的文件系统，被安装于*/sys*目录下的*/proc*文件系统相似。*/proc*文件系统是首次被设计成允许用户态应用程序访问内核内部数据结构的一种文件系统。*/sysfs*文件系统本质上与*/proc*有相同的目的，但是它还提供关于内核数据结构的附加信

息；此外，*/sysfs*的组织结构比*/proc*更有条理。或许，在不远的将来，*/proc*和*/sysfs*将会继续共存。

*sysfs*文件系统的目标是要展现设备驱动程序模型组件间的层次关系。该文件系统的相应高层目录是：

block

块设备，它们独立于所连接的总线。

devices

所有被内核识别的硬件设备，依照连接它们的总线对其进行组织。

bus

系统中用于连接设备的总线。

drivers

在内核中注册的设备驱动程序。

class

系统中设备的类型（声卡、网卡、显卡等等）；同一类可能包含由不同总线连接的设备，于是由不同的驱动程序驱动。

power

处理一些硬件设备电源状态的文件。

firmware

处理一些硬件设备的固件的文件。

*sysfs*文件系统中所表示的设备驱动程序模型组件之间的关系就像目录和文件之间符号链接的关系一样。例如，文件*/sys/block/sda/device*可以是一个符号链接，指向在*/sys/devices/pci0000:00*（表示连接到PCI总线的SCSI控制器）中嵌入的一个子目录。此外，文件*/sys/block/sda/device/block*是到目录*/sys/block/sda*的一个符号链接，这表明这个PCI设备是SCSI磁盘的控制器。

*sysfs*文件系统中普通文件的主要作用是表示驱动程序和设备的属性。例如，位于目录*/sys/block/hda*下的*dev*文件含有第一个IDE链主磁盘的主设备号和次设备号。

kobject

设备驱动程序模型的核心数据结构是一个普通的数据结构，叫做*kobject*，它与*sysfs*文件系统自然地绑定在一起：每个*kobject*对应于*sysfs*文件系统中的一个目录。

kobject 被嵌入一个叫做“容器”的更大对象中，容器描述设备驱动程序模型中的组件（注 2）。容器的典型例子有总线、设备以及驱动程序的描述符；例如，第一个 IDE 磁盘的第一个分区描述符对应于 */sys/block/hda/hda1* 目录。

将一个 kobject 嵌入容器中允许内核：

- 为容器保持一个引用计数器。
- 维持容器的层次列表或组（例如，与块设备相关的 *sysfs* 目录为每个磁盘分区包含一个不同的子目录）。
- 为容器的属性提供一种用户态查看的视图。

kobject、kset 和 subsystem

每个 kobject 由 kobject 数据结构描述，其各字段如表 13-2 所示。

表 13-2：kobject 数据结构中的字段

类型	字段	说明
char *	k_name	指向含有容器名称的字符串
char []	name	含有容器名称的字符串，如果它不超过 20 个字节
struct k_ref	kref	容器的引用计数器
struct list_head	entry	用于 kobject 所插入的链表的指针
struct kobject *	parent	指向父 kobject（如果存在的话）
struct kset *	kset	指向包含的 kset
struct kobj_type *	ktype	指向 kobject 的类型描述符
struct dentry *	dentry	指向与 kobject 相对应的 *sysfs* 文件的 dentry 数据结构

ktype 字段指向 kobj_type 对象，该对象描述了 kobject 的“类型”——本质上，它描述的是包括 kobject 的容器的类型。kobj_type 数据结构包括三个字段：release 方法（当 kobject 被释放时执行），指向 *sysfs* 操作表的 sysfs_ops 指针以及 *sysfs* 文件系统的缺省属性链表。

kref 字段是一个 k_ref 类型的结构，它仅包括一个 refcount 字段。顾名思义，这个字段就是 kobject 的引用计数器，但它也可以作为 kobject 容器的引用计数器。kobject_get()

注 2：kobject 主要用于实现设备驱动程序模型，但是为了使用 kobjeckt，还要继续努力改变其他的一些内核部件，如模块子系统。

和kobject_put()函数分别用于增加和减少引用计数器的值；如果该计数器的值等于0，就会释放kobject使用的资源，并且执行kobject的类型描述符kobj_type对象的release方法。该方法用于释放容器本身，通常只有在动态地分配kobject容器时才定义该方法。

通过*kset*数据结构可将kobjects组织成一棵层次树。kset是同类型kobject结构的一个集合体——也就是说，相关的kobject包含在同类型的容器中。kset数据结构的字段如表13-3所示。

表13-3：kset数据结构中的字段

类型	字段	说明
struct subsystem *	subsys	指向subsystem描述符
struct kobj_type *	ktype	指向kset的kobject类型描述符
struct list_head	list	包含在kset中的kobject链表的首部
struct kobject	kobj	嵌入的kobject结构（见下文）
struct kset_hotplug_ops *	hotplug_ops	指向用于处理kobject结构的过滤和热插拔操作的回调函数表

list字段表示包含在kset中的kobject结构的双向循环链表的首部。ktype字段是指向kset中的kobj_type描述符的指针，该描述符被kset中所有的kobject结构共享。

kobj字段是嵌入在kset数据结构中的kobject；而位于kset中的kobject，其parent字段指向这个内嵌的kobject结构。因此，一个kset就是kobject集合体，但是它依赖于层次树中用于引用计数和连接的更高层kobject。这种设计编码效率很高，并可获得最大的灵活性。例如，分别用于增加和减少kset引用计数器值的kset_get()函数和kset_put()函数，只需简单地调用内嵌的kobject结构中的kobject_get()函数和kobject_put()函数；因为kset的引用计数器只不过是内嵌在kset中的类型为kobject的kobj的引用计数器。而且，由于有了内嵌的kobject结构，kset数据结构可以嵌入到“容器”对象中，非常类似于嵌入的kobject数据结构。最后，kset可以作为其他kset的一个成员：它足以将内嵌的kobject插入到更高层次的kset中。

还存在所谓*subsystem*的kset集合。一个subsystem可以包括不同类型的kset，用包含两个字段的subsystem数据结构来描述：

kset

内嵌的kset结构，用于存放subsystem中的kset。

rwsem

读写信号量，保护递归地包含于subsystem中的所有kset和kobject。

subsystem 数据结构甚至也可以嵌入到一个更大的“容器”对象中；因此，容器的引用计数器也是内嵌 subsystem 的引用计数器 —— 也就是嵌入在 subsystem 中的 kset 所嵌的 kobject 的引用计数器。subsys_get()和 subsys_put()函数分别用于增加和减少这个引用计数器的值。

图 13-3 显示了设备驱动程序模型层次的一个例子。*bus* 子系统包括一个 *pci* 子系统，*pci* 子系统又依次包含驱动程序的一个 kset。这个 kset 包含一个串口 kobject（具有唯一 *new-id* 属性的串口对应的设备驱动器程序）。

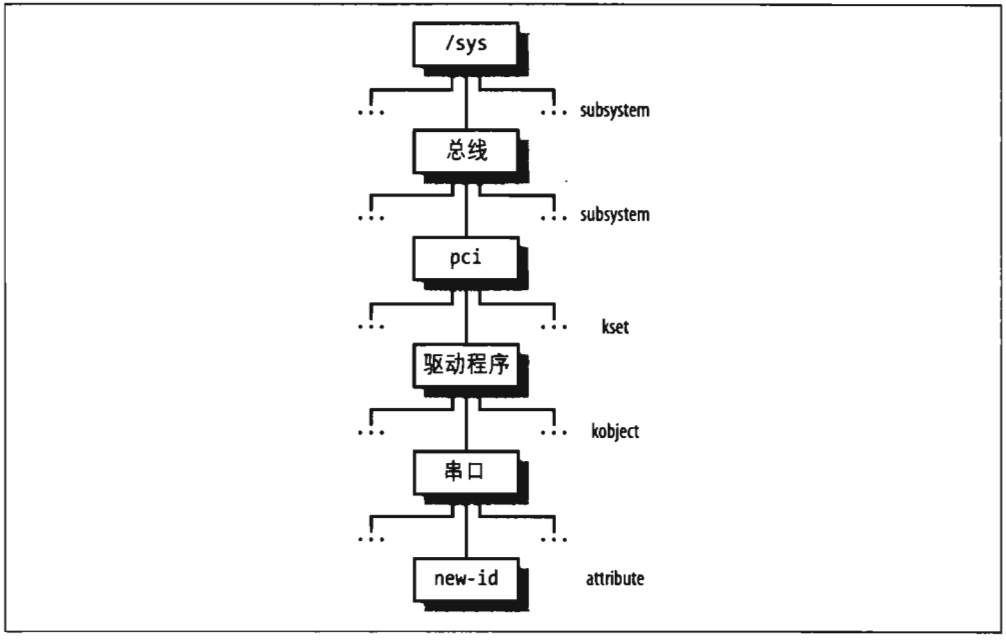


图 13-3：设备驱动程序层次模型的一个实例

注册 kobject、kset 和 subsystem

一般来说，如果想让 kobject、kset 或 subsystem 出现在 *sysfs* 子树中，就必须首先注册它们。与 kobject 对应的目录总是出现在其父 kobject 的目录中。例如，位于同一个 kset 中的 kobject 的目录出现在 kset 本身的目录中。因此，*sysfs* 子树的结构就描述了各种已注册的 kobject 之间以及各种容器对象之间的层次关系。

通常，*sysfs* 文件系统的上层目录肯定是已注册的 subsystem。

kobject_register()函数用于初始化 kobject，并且将其相应的目录增加到 *sysfs* 文件系统中。在调用此函数之前，调用程序应该先设置 kobject 结构中的 kset 字段，使它指向其

父kset（如果有的话）。kobject_unregister()函数则将kobject的目录从*sysfs*文件系统中移走。为了更易于内核开发者进行开发，Linux也提供了kset_register()和kset_unregister()函数，以及subsystem_register() subsystem_unregister()函数，但本质上它们是围绕kobject_register()和kobject_unregister()的封装函数。

如前所述，许多kobject目录都包括称作属性（*attribute*）的普通文件。sysfs_create_file()函数接收kobject的地址和属性描述符作为它的参数，并在合适的目录中创建特殊文件。*sysfs*文件系统中所描述的对象间的其他关系可以通过符号链接的方式来建立：sysfs_create_link()函数为目录中与其他kobject相关联的特定kobject创建一个符号链接。

设备驱动程序模型的组件

设备驱动程序模型建立在几个基本数据结构之上，这些结构描述了总线、设备、设备驱动器等等。让我们来考察一下它们。

设备

设备驱动程序模型中的每个设备是由一个device对象来描述的，其字段如表13-4所示。

表13-4：device对象中的字段

类型	字段	说明
struct list_head	node	指向兄弟设备链表的指针
struct list_head	bus_list	指向连于同一类型总线上的设备链表的指针
struct list_head	driver_list	指向设备驱动程序链表的指针
struct list_head	children	子设备链表的首部
struct device *	parent	指向父设备的指针
struct kobject	kobj	内嵌的kobject结构
char []	bus_id	连接到总线上设备的位置
struct bus_type *	bus	指向所连接总线的指针
struct device_driver *	driver	指向控制设备驱动程序的指针
void *	driver_data	指向驱动程序私有数据的指针
void *	platform_data	指向遗留设备驱动程序的私有数据的指针
struct dev_pm_info	power	电源管理信息

表 13-4：device对象中的字段（续）

类型	字段	说明
unsigned long	detach_state	卸载设备驱动程序时电源进入的状态
unsigned long long *	dma_mask	指向设备的DMA屏蔽字的指针［参见稍后的“直接内存方向（DMA）”一节］
unsigned long long	coherent_dma_mask	设备的一致性DMA的屏蔽字
struct list_head	dma_pools	聚集的DMA缓冲池链表的首部
struct dma_coherent_mem *	dma_mem	指向设备所使用的一致性DMA存储器描述符的指针［参见稍后的“直接内存访问（DMA）”一节］
void (*)(struct device *)	release	释放设备描述符的回调函数

device对象全部收集在devices_subsys子系统中，该子系统对应的目录为*/sys/devices*（参见前面的“kobject”一节）。设备是按照层次关系组织的：一个设备是某个“孩子”的“父亲”，其条件为子设备离开父设备无法正常工作。例如，在基于PCI总线的计算机上，位于PCI总线和USB总线之间的桥就是连接在USB总线上的所有设备的父设备。device对象的parent字段是指向其父设备描述符的指针，children字段是子设备链表的首部，而node字段存放指向children链表中相邻元素的指针。device对象中内嵌的kobject间的亲子关系也反映了设备的层次关系；因此，*/sys/devices*下的目录结构与硬件设备的物理组织是匹配的。

每个设备驱动程序都保持一个device对象链表，其中链接了所有可被管理的设备；device对象的driver_list字段存放指向相邻对象的指针，而driver字段指向设备驱动程序的描述符。此外，对于任何总线类型来说，都有一个链表存放连接到该类型总线上的所有设备；device对象的bus_list字段存放指向相邻对象的指针，而bus字段指向总线类型描述符。

引用计数器记录device对象的使用情况，它包含在kobject类型的kobj结构中，通过调用get_device()和put_device()函数分别增加和减少该计数器的值。

device_register()函数的功能是往设备驱动程序模型中插入一个新的device对象，并自动地在*/sys/devices*目录下为其创建一个新的目录。相反地，device_unregister()函数的功能是从设备驱动程序模型中移走一个设备。

通常，device对象被静态地嵌入到一个更大的描述符中。例如，PCI设备是由数据结构pci_dev描述；该数据结构的dev字段就是一个device对象，而其他字段则是PCI总

线所特有的。在PCI内核层上，当注册或注销设备时就会分别执行device_register()函数和device_unregister()函数。

驱动程序

设备驱动程序模型中的每个驱动程序都可由device_driver对象描述，其各字段如表13-5所示。

表13-5：device_driver对象中的字段

类型	字段	说明
char *	name	设备驱动程序的名称
struct bus_type *	bus	指向总线描述符的指针，总线连接所支持的设备
struct semaphore	unload_sem	禁止卸载设备驱动程序的信号量；当引用计数器的值为0时释放该信号量
struct kobject	kobj	内嵌的kobject结构
struct list_head	devices	驱动程序支持的所有设备组成的链表的首部
struct module *	owner	标识实现设备驱动程序的模块，如果有的话（参见附录二）
int (*)(struct device *)	probe	探测设备的方法（检验设备驱动程序是否可以控制该设备）
int (*)(struct device *)	remove	移走设备时所调用的方法
void (*)(struct device *)	shutdown	设备断电（关闭）时所调用的方法
int (*)(struct device *, unsigned long, unsigned long)	suspend	设备置于低功率状态时所调用的方法
int (*)(struct device *, unsigned long)	resume	设备恢复正常状态时所调用的方法

device_driver对象包括四个方法，它们用于处理热插拔、即插即用和电源管理。当总线设备驱动程序发现一个可能由它处理的设备时就会调用probe方法；相应的函数将会探测该硬件，从而对该设备进行更进一步的检查。当移走一个可热插拔的设备时驱动程序会调用remove方法；而驱动程序本身被卸载时，它所处理的每个设备也会调用remove方法。当内核必须改变设备的供电状态时，设备会调用shutdown、suspend和resume三个方法。

内嵌在描述符中的kobject类型的kobj所包含的引用计数器用于记录device_driver对

象的使用情况。通过调用 get_driver()函数和 put_driver()函数可分别增加和减少该计数器的值。

driver_register()函数的功能是往设备驱动程序模型中插入一个新的device_driver对象，并自动地在*sysfs*文件系统下为其创建一个新的目录。相反，driver_unregister()函数的功能则是从设备驱动程序模型中移走一个设备驱动对象。

通常，device_driver对象静态地被嵌入到一个更大的描述符中。例如，PCI设备驱动程序是由数据结构pci_driver描述的；该数据结构的driver字段是一个 device_driver对象，而其他字段则是PCI 总线所特有的。

总线

内核所支持的每一种总线类型都由一个 bus_type 对象描述，其各字段如表 13-6 所示。

表 13-6：bus_type 对象中的字段

类型	字段	说明
char *	name	总线类型的名称
struct subsystem	subsys	与总线类型相关的 kobject 子系统
struct kset	drivers	驱动程序的 kobject 集合
struct kset	devices	设备的 kobject 集合
struct bus_attribute *	bus_attrs	指向对象的指针，该对象包含总线属性和用于导出此属性到 *sysfs* 文件系统的方法
struct device_attribute *	dev_attrs	指向对象的指针，该对象包含设备属性和用于导出此属性到 *sysfs* 文件系统的方法
struct driver_attribute *	drv_ attrs	指向对象的指针，该对象包含设备驱动程序属性和用于导出此属性到 *sysfs* 文件系统的方法
int (*)(struct device *, struct device_driver *)	match	检验给定的设备驱动程序是否支持特定设备的方法
int (*)(struct device *, char **, int, char *, int)	hotplug	注册设备时调用的方法
int (*)(struct device *, unsigned long)	suspend	保存硬件设备的上下文状态并改变设备供电状态的方法
int (*)(struct device *)	resume	改变供电状态和恢复硬件设备上下文的方法

每个bus_type类型的对象都包含一个内嵌的子系统；存放于bus_subsys变量中的子系统把嵌入在bus_type对象中的所有子系统都集合在一起。bus_subsys子系统与目录*/sys/bus*是对应的；因此，例如，有一个*/sys/bus/pci*目录，它与PCI总线类型相对应。每种总线的子系统通常包括两个kset，它们是*drivers*和*devices*（分别对应于bus_type对象中的drivers和devices字段）。

名为drivers的kset包含描述符device_driver，它描述与该总线类型相关的所有设备驱动程序，而名为devices的kset包含描述符device，它描述给定总线类型上连接的所有设备。因为设备的kobject目录已经出现在*/sys/devices*下的*sysfs*文件系统中，所以每种总线子系统的devices目录存放了指向*/sys/devices*下目录的符号链接。bus_for_each_drv()和bus_for_each_dev()函数分别用于循环扫描drivers和devices链表中的所有元素。

当内核检查一个给定的设备否可以由给定的驱动程序处理时，就会执行match方法。对于连接设备的总线而言，即使其上每个设备的标识符都拥有一个特定的格式，实现match方法的函数通常也很简单，因为它只需要在所支持标识符的驱动程序表中搜索设备的描述符。在设备驱动程序模型中注册某个设备时会执行hotplug方法；实现函数应该通过环境变量把总线的具体信息传递给用户态程序，以通告一个新的可用设备（参见后面的“注册设备驱动程序”一节）。最后，当特定类型总线上的设备必须改变其供电状态时，就会执行suspend和resume方法。

类

每个类是由一个class对象描述的。所有的类对象都属于与*/sys/class*目录相对应的class_subsys子系统。此外，每个类对象还包括一个内嵌的子系统；因此，例如有一个*/sys/class/input*目录，它就与设备驱动程序模型的*input*类相对应。

每个类对象包括一个class_device描述符链表，其中每个描述符描述了一个属于该类的单独逻辑设备。class_device结构中包含一个dev字段，它指向一个设备描述符，因此一个逻辑设备总是对应于设备驱动程序模型中的一个给定的设备。然而，可以存在多个class_device描述符对应同一个设备。事实上，一个硬件设备可能包括几个不同的子设备，每个子设备都需要一个不同的用户态接口。例如，声卡就是一个硬件设备，它通常包括一个DSP（digital singnal processor，数字信号处理器）、一个混音器、一个游戏端口接口等等；每个子设备需要一个属于自己的用户态接口，因此*sysfs*文件系统中都有与它们相对应的目录。

同一类中的设备驱动程序可以对用户态应用程序提供相同的功能；例如，声卡上的所有设备驱动程序都提供一个可以向DSP中写入声音样本的方法。

设备驱动程序模型中的类本质上是要提供一个标准的方法，从而为向用户态应用程序导出逻辑设备的接口。每个class_device描述符中内嵌一个kobject，这是一个名为*dev*的属性（特殊文件）。该属性存放设备文件的主设备号和次设备号，通过它们可以访问相应的逻辑设备（参见下一节）。

设备文件

正如在第一章中所提到的那样，类Unix操作系统都是基于文件概念的，文件是由字节序列而构成的信息载体。根据这一点，可以把I/O设备当作设备文件（*device file*）这种所谓的特殊文件来处理；因此，与磁盘上的普通文件进行交互所用的同一系统调用可直接用于I/O设备。例如，用同一write()系统调用既可以向普通文件中写入数据，也可以通过向*/dev/lp0*设备文件中写入数据从而把数据发往打印机。

根据设备驱动程序的基本特性，设备文件可以分为两种：块和字符。这两种硬件设备之间的差异并不容易划分，但我们至少可以假定以下的差异：

- 块设备的数据可以被随机访问，而且从人类用户的观点看，传送任何数据块所需的时间都是较少且大致相同的。块设备的典型例子是硬盘、软盘、CD-ROM驱动器及DVD播放器。
- 字符设备的数据或者不可以被随机访问（考虑声卡这样的例子），或者可以被随机访问，但是访问随机数据所需的时间很大程度上依赖于数据在设备内的位置（考虑磁带驱动器这样的例子）。

网卡是这种模式的一种明显的例外，因为网卡是不直接与设备文件相对应的硬件设备。

自从Unix操作系统早期版本以来，设备文件就一直在使用。设备文件是存放在文件系统中的实际文件。然而，它的索引节点并不包含指向磁盘上数据块（文件的数据）的指针，因为它们是空的。相反，索引节点必须包含硬件设备的一个标识符，它对应字符或块设备文件。

传统上，设备标识符由设备文件的类型（字符或块）和一对参数组成。第一个参数称为主设备号（*major number*），它标识了设备的类型。通常，具有相同主设备号和类型的所有设备文件共享相同的文件操作集合，因为它们是由同一个设备驱动程序处理的。第二个参数称为次设备号（*minor number*），它标识了主设备号相同的设备组中的一个特定设备。例如，由相同的磁盘控制器管理的一组磁盘具有相同的主设备号和不同的次设备号。

mknod()系统调用用来创建设备文件。其参数有设备文件名、设备类型、主设备号及次

设备号。设备文件通常包含在*/dev*目录中。表13-7显示了一些设备文件的属性。注意字符设备和块设备有独立的编号，因此，块设备(3,0)不同于字符设备(3,0)。

表13-7：设备文件的例子

设备名	类型	主设备号	次设备号	说明
/dev/fd0	块设备	2	0	软盘
/dev/hda	块设备	3	0	第一个IDE磁盘
/dev/hda2	块设备	3	2	第一个IDE磁盘上的第二个主分区
/dev/hdb	块设备	3	64	第二个IDE磁盘
/dev/hdb3	块设备	3	67	第二个IDE磁盘上的第三个主分区
/dev/ttyp0	字符设备	3	0	终端
/dev/console	字符设备	5	1	控制台
/dev/lp1	字符设备	6	1	并口打印机
/dev/ttyS0	字符设备	4	64	第一个串口
/dev/rtc	字符设备	10	135	实时时钟
/dev/null	字符设备	1	3	空设备（黑洞）

设备文件通常与硬件设备（如硬盘*/dev/hda*），或硬件设备的某一物理或逻辑分区（如磁盘分区*/dev/hda2*）相对应。但在某些情况下，设备文件不会和任何实际的硬件对应，而是表示一个虚拟的逻辑设备。例如，*/dev/null*就是一个和“黑洞”对应的设备文件，所有写入这个文件的数据都被简单地丢弃，因此，该文件看起来总为空。

就内核所关心的内容而言，设备文件名是无关紧要的。如果你建立了一个名为*/tmp/disk*的设备文件，类型为“块”，主设备号是3，次设备号是0，那么这个设备文件就和表13-7中的*/dev/hda*等价。另一方面，对某些应用程序来说，设备文件名可能就很有意义。例如，通信程序可能假设第一个串口和*/dev/ttyS0*设备文件对应。但是，通常可以把大部分应用程序设定为随意地与指定的设备文件进行交互。

设备文件的用户态处理

传统的Unix系统中（以及Linux的早期版本中），设备文件的主设备号和次设备号都是8位长。因此，最多只能有65536个块设备文件和65536个字符设备文件。你可能认为这些已经足够了，但遗憾的是它们并不够用。

真正的问题是设备文件被分配一次且永远保存在*/dev*目录中；因此，系统中的每个逻辑设备都应该有一个与其相对应的、明确定义了设备号的设备文件。*Documentation/devices.txt*

文件存放了官方注册的已分配设备号和*/dev*目录节点；*include /linux/major.h*文件也可能包含设备的主设备号对应的宏。

不幸的是，如今各种不同的硬件设备数量惊人，几乎分配了所有的设备号。官方注册的设备号对于一般的Linux系统还能胜任；然而，它却不能很好地适用于大规模的系统。此外，高端系统可能使用数百或数千的同类型磁盘，因而8位的次设备号是远远不够的。例如，注册表为16个SCSI磁盘保留了设备号，而每个SCSI磁盘拥有15个分区；如果一个高端系统拥有多于16个的SCSI磁盘，那么必须改变原先主设备号和次设备号的标准分配 —— 这是一个非常繁琐的工作，它需要改变内核源代码并且使得系统难以维护。

为了解决上述问题，Linux 2.6已经增加了设备号的编码大小：目前主设备号的编码为12位，次设备号的编码为20位。通常把这两个参数合并成一个32位的`dev_t`变量；`MAJOR`宏和`MINOR`宏可以从`dev_t`中分别提取主设备号和次设备号，而`MKDEV`宏可以把主设备号和次设备号合并成一个`dev_t`值。为了实现向后兼容，内核仍然可以正确地处理设备号编码为16位的老式设备文件。

官方注册表不能静态地分配这些附加的可用设备号，只有在处理设备号的特殊要求时才允许使用。事实上，对分配设备号和创建设备文件来说，如今更倾向的做法是高度动态地处理设备文件。

动态分配设备号

每个设备驱动程序在注册阶段都会指定它将要处理的设备号范围（参见后面的“注册设备驱动程序”一节）。然而，驱动程序可以只指定设备号的分配范围，无需指定精确的值：在这种情形下，内核会分配一个合适的设备号范围给驱动程序。

因此，新的硬件设备驱动程序不再需要从官方注册表中分配的一个设备号；它们可以仅仅使用当前系统中空闲的设备号。

然而，在这种情形下，就不能永久性地创建设备文件；它只在设备驱动程序初始化一个主设备号和次设备号时才创建。因此，这就需要有一个标准的方法将每个驱动程序所使用的设备号输出到用户态应用程序中。正如我们在前面“设备驱动程序模型的组件”一节所看到的，设备驱动程序模型提供了一个非常好的解决办法：把主设备号和次设备号存放在*/sys/class*子目录下的*dev*属性中。

动态创建设备文件

Linux内核可以动态地创建设备文件：它无需把每一个可能想到的硬件设备的设备文件都填充到*/dev*目录下，因为设备文件可以按照需要来创建。由于设备驱动程序模型的存

在，Linux 2.6内核提供了一个非常简单的方法来处理这个问题。系统中必须安装一组称为*udev*工具集的用户态程序。当系统启动时，*/dev*目录是清空的，这时udev程序将扫描*/sys/class*子目录来寻找*dev*文件。对每一个这样的文件（主设备号和次设备号的组合表示一个内核所支持的逻辑设备文件），*udev*程序都会在*/dev*目录下为它创建一个相应的设备文件。*udev*程序也会根据配置文件为其分配一个文件名并创建一个符号链接，该方法类似于Unix设备文件的传统命名模式。最后，*/dev*目录里只存放了系统中内核所支持的所有设备的设备文件，而没有任何其他的文件。

通常在系统初始化后才创建设备文件。它要么发生在加载设备驱动程序（系统尚未支持该设备）所在的模块时，要么发生在一个热拔插的设备（如USB外围设备）加入系统中时。*udev*工具集可以自动地创建相应的设备文件，因为设备驱动程序模型支持设备的热插拔。当发现一个新的设备时，内核会产生一个新的进程来执行用户态shell脚本文件*/sbin/hotplug*（注3），并将新设备上的有用信息作为环境变量传递给shell脚本。用户态脚本文件读取配置文件信息并关注完成新设备初始化所必需的任何操作。如果安装了*udev*工具集，脚本文件也会在*/dev*目录下创建适当的设备文件。

设备文件的VFS处理

虽然设备文件也在系统的目录树中，但是它们和普通文件以及目录文件有根本的不同。当进程访问普通文件时，它会通过文件系统访问磁盘分区中的一些数据块；而在进程访问设备文件时，它只要驱动硬件设备就可以了。例如，进程可以访问一个设备文件以从连接到计算机的温度计读取房间的温度。为应用程序隐藏设备文件与普通文件之间的差异正是VFS的责任。

为了做到这点，VFS在设备文件打开时改变其缺省文件操作；因此，可以把设备文件的每个系统调用都转换成与设备相关的函数的调用，而不是对主文件系统相应函数的调用。与设备相关的函数对硬件设备进行操作以完成进程所请求的操作（注4）。

让我们假定进程在设备文件（块或字符类型）上执行open()系统调用。这个系统调用所执行的操作已经在第十二章“open()系统调用”一节进行了描述。从本质上说，相应的服务例程解析到设备文件的路径名，并建立相应的索引节点对象、目录项对象和文件对象。

注3：可以通过写*/proc/sys/kernel/hotplug*文件改变在发生热插拔事件时所调用的用户态程序的路径名。

注4：注意，根据第十二章中的“路径名查找”一节中介绍的命名解析机制，指向设备文件的符号链接与设备文件的作用相同。

通过适当的文件系统函数（通常为 ext2_read_inode()或 ext3_read_inode()；参见第十八章）读取磁盘上的相应索引节点来对索引节点对象进行初始化。当这个函数确定磁盘索引节点与设备文件对应时，则调用 init_special_inode()，该函数把索引节点对象的 i_rdev 字段初始化为设备文件的主设备号和次设备号，而把索引节点对象的 i_fop字段设置为def_blk_fops或者def_chr_fops文件操作表的地址（根据设备文件的类型）。因此，open()系统调用的服务例程也调用 dentry_open()函数，后者分配一个新的文件对象并把其 f_op 字段设置为 i_fop 中存放的地址，即再一次指向 def_blk_fops或def_chr_fops的地址。正是这两个表的引入，才使得在设备文件上所发出的任何系统调用都将激活设备驱动程序的函数而不是基本文件系统的函数。

设备驱动程序

设备驱动程序是内核例程的集合，它使得硬件设备响应控制设备的编程接口，而该接口是一组规范的 VFS 函数集（*open*，*read*，*lseek*，*ioctl* 等等）。这些函数的实际实现由设备驱动程序全权负责。由于每个设备都有一个唯一的 I/O 控制器，因此就有唯一的命令和唯一的状态信息，所以大部分 I/O 设备都有自己的驱动程序。

设备驱动程序的种类有很多。它们在对用户态应用程序提供支持的级别上有很大的不同，也对来自硬件设备的数据采集有不同的缓冲策略。这些选择极大地影响了设备驱动程序的内部结构，我们将在“直接内存访问（DMA）”和“字符设备的缓冲策略”两节进行讨论。

设备驱动程序并不仅仅由实现设备文件操作的函数组成。在使用设备驱动程序之前，有几个活动是肯定要发生的。我们将在下面几节考察它们。

注册设备驱动程序

我们知道在设备文件上发出的每个系统调用都由内核转化为对相应设备驱动程序的对应函数的调用。为了完成这个操作，设备驱动程序必须注册自己。换句话说，注册一个设备驱动程序意味着分配一个新的device_driver描述符，将其插入到设备驱动程序模型的数据结构中（参见“设备驱动程序模型的组件”一节），并把它与对应的设备文件（可能是多个设备文件）连接起来。如果设备文件对应的驱动程序以前没有注册，则对该设备文件的访问会返回错误码 -ENODEV。

如果设备驱动程序被静态地编译进内核，则它的注册在内核初始化阶段进行。相反，如果驱动程序是作为一个内核模块来编译的（参见附录二），则它的注册在模块装入时进行。在后一种情况下，设备驱动程序也可以在模块卸载时注销自己。

例如，我们考虑一个通用的PCI设备。为了能正确地对其进行处理，其设备驱动程序必须分配一个pci_driver类型的描述符，PCI内核层使用该描述符来处理设备。初始化描述符的一些字段后，设备驱动程序就会调用pci_register_driver()函数。事实上，pci_driver描述符包括一个内嵌的device_driver描述符（参见前面的“设备驱动程序模型的组件”一节）；pci_register_driver()函数仅仅初始化内嵌的驱动程序描述符中的字段，然后调用driver_register()函数把驱动程序插入设备驱动程序模型的数据结构中。

注册设备驱动程序时，内核会寻找可能由该驱动程序处理但还尚未获得支持的硬件设备。为了做到这点，内核主要依靠相关的总线类型描述符bus_type的match方法，以及device_driver对象的probe方法。如果探测到可被驱动程序处理的硬件设备，内核会分配一个设备对象，然后调用device_register()函数把设备插入设备驱动程序模型中。

初始化设备驱动程序

对设备驱动程序进行注册和初始化是两件不同的事。设备驱动程序应当尽快被注册，以便用户态应用程序能通过相应的设备文件使用它。相反，设备驱动程序在最后可能的时刻才被初始化。事实上，初始化驱动程序意味着分配宝贵的系统资源，这些资源因此就对其他驱动程序不可用了。

我们已经在第四章“I/O中断处理”一节看到一个例子：把IRQ分配给设备通常是自动进行的，这正好发生在使用设备之前，因为多个设备可能共享同一条IRQ线。其他可以在最后时刻被分配的资源是用于DMA传送缓冲区的页框和DMA通道本身（用于像软盘驱动器那样的老式非PCI设备）。

为了确保资源在需要时能够获得，在获得后不再被请求，设备驱动程序通常采用下列模式：

- 引用计数器记录当前访问设备文件的进程数。在设备文件的open方法中计数器被增加，在release方法中被减少（注5）。
- open方法在增加引用计数器的值之前先检查它。如果计数器为0，则设备驱动程序必须分配资源并激活硬件设备上的中断和DMA。
- release方法在减少使用计数器的值之后检查它。如果计数器为0，说明已经没有进程使用这个硬件设备。如果是这样，该方法将禁止I/O控制器上的中断和DMA，然后释放所分配的资源。

注5：更确切地说，引用计数器记录引用设备文件的文件对象的个数，因为子进程可能共享文件对象。

监控I/O操作

I/O操作的持续时间通常是不可预知的。这可能和机械装置的情况有关（对于要传送的数据块来说是磁头的当前位置），和实际的随机事件有关（数据包什么时候到达网卡），还和人为因素有关（用户在键盘上按下一个键或者发现打印机夹纸了）。在任何情况下，启动I/O操作的设备驱动程序都必须依靠一种监控技术在I/O操作终止或超时时发出信号。

在终止操作的情况下，设备驱动程序读取I/O接口状态寄存器的内容来确定I/O操作是否成功执行。在超时的情况下，驱动程序知道一定出了问题，因为完成操作所允许的最大时间间隔已经用完，但什么也没做。

监控I/O操作结束的两种可用技术分别称为轮询模式（*polling mode*）和中断模式（*interrupt mode*）。

轮询模式

CPU依照这种技术重复检查（轮询）设备的状态寄存器，直到寄存器的值表明I/O操作已经完成为止。我们已经在第五章的“自旋锁”一节中提到一种基于轮询的技术：当处理器试图获得一个繁忙的自旋锁时，它就重复地查询变量的值，直到该值变成0为止。但是，应用到I/O操作中的轮询技术更加巧妙，这是因为驱动程序还必须记住检查可能的超时。下面是轮询的一个简单例子：

```
for (;;) {
    if (read_status(device) & DEVICE_END_OPERATION) break;
    if (--count == 0) break;
}
```

在进入循环之前，count变量已被初始化，每次循环都对count的值减1，因此就可以使用这个变量实现一种粗略的超时机制。另外，更精确的超时机制可以通过这样的方法实现：在每次循环时读取节拍计数器jiffies的值（请参看第六章中的“更新时间和日期”一节），并将它与开始等待循环之前读取的原值进行比较。

如果完成I/O操作需要的时间相对较多，比如说毫秒级，那么这种模式就变得低效，因为CPU花费宝贵的机器周期去等待I/O操作的完成。在这种情况下，在每次轮询操作之后，可以通过把schedule()的调用插入到循环内部来自愿放弃CPU。

中断模式

如果I/O控制器能够通过IRQ线发出I/O操作结束的信号，那么中断模式才能被使用。

我们现在通过一个简单的例子说明中断模式如何工作。假定我们想实现一个简单的输入

字符设备的驱动程序。当用户在相应的设备文件上发出 read()系统调用时，一条输入命令被发往设备的控制寄存器。在一个不可预知的长时间间隔后，设备把一个字节的数据放进输入寄存器。设备驱动程序然后将这个字节作为 read()系统调用的结果返回。

这是一个用中断模式实现驱动程序的典型例子。实质上，驱动程序包含两个函数：

1. 实现文件对象 read 方法的 foo_read()函数。
2. 处理中断的 foo_interrupt()函数。

只要用户读设备文件，foo_read()函数就被触发：

```
ssize_t foo_read(struct file *filp, char *buf, size_t count,
                 loff_t *ppos)
{
    foo_dev_t * foo_dev = filp->private_data;
    if (down_interruptible(&foo_dev->sem)
        return -ERESTARTSYS;
    foo_dev->intr = 0;
    outb(DEV_FOO_READ, DEV_FOO_CONTROL_PORT);
    wait_event_interruptible(foo_dev->wait, (foo_dev->intr=  =1));
    if (put_user(foo_dev->data, buf))
        return -EFAULT;
    up(&foo_dev->sem);
    return 1;
}
```

设备驱动程序依赖类型为foo_dev_t的自定义描述符，它包含信号量sem（保护硬件设备免受并发访问）、等待队列wait、标志 intr（当设备发出一个中断时设置）及单个字节缓冲区 data（由中断处理程序写入且由 read方法读取）。一般而言，所有使用中断的I/O驱动程序都依赖中断处理程序及 read 和write方法均访问的数据结构。foo_dev_t描述符的地址通常存放在设备文件的文件对象的 private_data 字段中或一个全局变量中。

foo_read()函数的主要操作如下：

1. 获取 foo_dev->sem信号量，因此确保没有其他进程访问该设备。
2. 清 intr标志。
3. 对I/O设备发出读命令。
4. 执行 wait_event_interruptible以挂起进程，直到 intr标志变为1。这个宏已在第三章“等待队列”一节描述过。

一定时间后，我们的设备发出中断信号以通知I/O操作已经完成，数据已经放在适当的 DEV_FOO_DATA_PORT数据端口。中断处理程序置 intr标志并唤醒进程。当调度程序决定重新执行这个进程时，foo_read()的第二部分被执行，步骤如下：

1. 把准备在 foo_dev->data 变量中的字符拷贝到用户地址空间。
2. 释放 foo_dev->sem 信号量后终止。

为了简单起见，我们没有包含任何超时控制。一般来说，超时控制是通过静态或动态定时器实现的（参见第六章）；定时器必须设置为启动I/O操作后正确的时间，并在操作结束时删除。

让我们来看一下 foo_interrupt()函数的代码：

```
void foo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    foo->data = inb(DEV_FOO_DATA_PORT);
    foo->intr = 1;
    wake_up_interruptible(&foo->wait);
    return 1;
}
```

中断处理程序从设备的输入寄存器中读字符，并把它存放在foo全局变量指向的驱动程序描述符foo_dev_t的data字段中。然后设置intr标志，并调用wake_up_interruptible()函数唤醒在 foo->wait 等待队列上阻塞的进程。

注意，三个参数中没有一个被中断处理程序使用，这是相当普遍的情况。

访问 I/O 共享存储器

根据设备和总线的类型，PC体系结构里的I/O共享存储器可以被映射到不同的物理地址范围。主要有：

对于连接到 ISA 总线上的大多数设备

I/O 共享存储器通常被映射到 0xa0000～0xfffff 的 16 位物理地址范围；这就在 640 KB 和 1 MB 之间留出了一段空间，就是我们在第二章的“物理内存布局”一节中所介绍的那个“空洞”。

对于连接到 PCI 总线上的设备

I/O 共享存储器被映射到接近 4 GB 的 32 位物理地址范围。这种类型的设备更加容易处理。

几年以前，Intel引入了图形加速端口（AGP）标准，该标准是适合于高性能图形卡的PCI的增强。这种卡除了有自己的I/O共享存储器外，还能够通过图形地址再映像表（GART）这个特殊的硬件电路直接对主板的RAM部分进行寻址。GART电路能够使AGP卡比老式的PCI卡具有更高的数据传输速率。然而，从内核的观点看，物理存储器位于何处根本没有什么关系，GART映射的存储器与其他种类I/O共享存储器的处理方式完全一样。

设备驱动程序如何访问一个I/O共享存储器单元？让我们从比较简单的PC体系结构开始入手，之后再扩展到其他体系结构。

不要忘了内核程序作用于线性地址，因此I/O共享存储器单元必须表示成大于PAGE_OFFSET的地址。在后面的讨论中，我们假设PAGE_OFFSET等于0xc0000000，也就是说，内核线性地址是在第4个GB。

设备驱动程序必须把I/O共享存储器单元的物理地址转换成内核空间的线性地址。在PC体系结构中，这可以简单地把32位的物理地址和0xc0000000常量进行或运算得到。例如，假设内核需要把物理地址为0x000b0fe4的I/O单元的值存放在t1中，把物理地址为0xfc000000的I/O单元的值存放在t2中。你可能认为使用下面的表达式就可以完成这项工作：

```
t1 = *((unsigned char *)(0xc00b0fe4));
t2 = *((unsigned char *)(0xfc000000));
```

在初始化阶段，内核已经把可用的RAM物理地址映射到线性地址空间第4个GB的开始部分。因此，分页单元把出现在第一个语句中的线性地址0xc00b0fe4映射回到原来的I/O物理地址0x000b0fe4，这正好落在从640KB到1MB的这段“ISA洞”中（请参看第二章的“Linux中的分页”一节）。这工作得很好。

但是，对于第二个语句来说，这里有一个问题，因为其I/O物理地址超过了系统RAM的最大物理地址。因此，线性地址0xfc000000就不需要与物理地址0xfc000000相对应。在这种情况下，为了在内核页表中包括对这个I/O物理地址进行映射的线性地址，必须对页表进行修改。这可以通过调用ioremap()或ioremap_nocache()函数来实现。第一个函数与vmalloc()函数类似，都调用get_vm_area()为所请求的I/O共享存储器区的大小建立一个新的vm_struct描述符（请参看第八章中的“非连续内存器区的描述符”一节）。然后，这两个函数适当地更新常规内核页表中的对应页表项。ioremap_nocache()不同于ioremap()，因为前者在适当地引用再映射的线性地址时还使硬件高速缓存内容失效。

因此，第二个语句的正确形式应该为：

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = *((unsigned char *)(io_mem + 0x100000));
```

第一条语句建立一个2MB的新的线性地址区间，该区间映射了从0xfb000000开始的物理地址；第二条语句读取地址为0xfc000000的内存单元。设备驱动程序以后要取消这种映射，就必须使用iounmap()函数。

在其他体系结构（PC之外的体系结构）上，简单地间接引用物理内存单元的线性地址并不能正确访问I/O共享存储器。因此，Linux定义了下列依赖于体系结构的函数，当访问I/O共享存储器时来使用它们：

`readb()`, `readw()`, `readl()`

分别从一个I/O共享存储器单元读取1、2或者4个字节

`writeb()`, `writew()`, `writel()`

分别向一个I/O共享存储器单元写入1、2或者4个字节

`memcpy_fromio()`, `memcpy_toio()`

把一个数据块从一个I/O共享存储器单元拷贝到动态内存中，另一个函数正好相反

`memset_io()`

用一个固定的值填充一个I/O共享存储器区域

因此，对于0xfc000000 I/O单元的访问推荐使用这样的方法：

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = readb(io_mem + 0x100000);
```

正是由于这些函数，就可以隐藏不同平台访问I/O共享存储器所用方法的差异。

直接内存访问（DMA）

在最初的PC体系结构中，CPU是系统中唯一的总线主控器，也就是说，为了提取和存储RAM存储单元的值，CPU是唯一可以驱动地址/数据总线的硬件设备。随着更多诸如PCI这样的现代总线体系结构的出现，如果提供合适的电路，每一个外围设备都可以充当总线主控器。因此，现在所有的PC都包含一个辅助的DMA电路，它可以用来控制在RAM和I/O设备之间数据的传送。DMA一旦被CPU激活，就可以自行传送数据；当数据传送完成之后，DMA发出一个中断请求。当CPU和DMA同时访问同一内存单元时，所产生的冲突由一个名为内存仲裁器（参见第五章中的“原子操作”一节）的硬件电路来解决。

使用DMA最多的是磁盘驱动器和其他需要一次传送大量字节的设备。因为DMA的设置时间相当长，所以在传送数量很少的数据时直接使用CPU效率更高。

原来的ISA总线所使用的DMA电路非常复杂，难于对其进行编程，并且限于物理内存的低16MB。PCI和SCSI总线所使用的最新DMA电路依靠总线中的专用硬件电路，这就简化了设备驱动程序开发人员的开发工作。

同步 DMA 和异步 DMA

设备驱动程序可以采用两种方式使用DMA，分别是同步DMA和异步DMA。第一种方式，数据的传送是由进程触发的；而第二种方式，数据的传送是由硬件设备触发的。

采用同步DMA传送的例子如声卡，它可以播放电影音乐。用户态应用程序将声音数据（称为样本）写入一个与声卡的数字信号处理器（DSP）相对应的设备文件中。声卡的驱动程序把写入的这些样本收集在内核缓冲区中。同时，驱动程序命令声卡把这些样本从内核缓冲区拷贝到预先定时的DSP中。当声卡完成数据传送时，就会引发一个中断，然后驱动程序会检查内核缓冲区是否还有要播放的样本；如果没有，驱动程序就再启动一次DMA数据传送。

采用异步DMA传送的例子如网卡，它可以从一个LAN中接收帧（数据包）。网卡将接收到的帧存储在自己的I/O共享存储器中，然后引发一个中断。其驱动程序确认该中断后，命令网卡将接收到的帧从I/O共享存储器拷贝到内核缓冲区。当数据传送完成后，网卡会引发新的中断，然后驱动程序将这个新帧通知给上层内核层。

DMA 传送的辅助函数

当为使用DMA传送方式的设备设计驱动程序时，开发者编写的代码应该与体系结构和总线（就DMA传送方式来说）二者都不相关。由于内核提供了丰富的DMA辅助函数，因而现在上述目标是可以实现的。这些辅助函数隐藏了不同硬件体系结构的DMA实现机制的差异。

这是DMA辅助函数的两个子集：老式的子集为PCI设备提供了与体系结构无关的函数；新的子集则保证了与总线和体系结构两者都无关。我们现在将介绍其中的一些函数，同时指出DMA的一些硬件特性。

总线地址

DMA的每次数据传送（至少）需要一个内存缓冲区，它包含硬件设备要读出或写入的数据。一般而言，启动一次数据传送前，设备驱动程序必须确保DMA电路可以直接访问RAM内存单元。

到现在为止，我们已区分了三类存储器地址：逻辑地址、线性地址以及物理地址，前两个在CPU内部使用，最后一个是CPU从物理上驱动数据总线所用的存储器地址。但是，还有第四种存储器地址，称为总线地址（*bus address*），它是除CPU之外的硬件设备驱动数据总线时所用的存储器地址。

从根本上说，内核为什么应该关心总线地址呢？这是因为在DMA操作中，数据传送不

需要CPU的参与；I/O设备和DMA电路直接驱动数据总线。因此，当内核开始DMA操作时，必须把所涉及的内存缓冲区总线地址或写入DMA适当的I/O端口，或写入I/O设备适当的I/O端口。

在80x86体系结构中，总线地址与物理地址是一致的。然而，其他的体系结构例如Sun公司的SPARC和HP的Alpha都包括一个所谓的I/O存储器管理单元(IO-MMU)的硬件电路，它类似于微处理器的分页单元，将物理地址映射为总线地址。使用DMA的所有I/O驱动程序在启动一次数据传送前必须设置好IO-MMU。

不同的总线具有不同的总线地址大小。例如，ISA的总线地址是24位长，因此，在80x86体系结构中，可以在物理内存的低16 MB中完成DMA传送——这就是为什么DMA使用的内存缓冲区分配在ZONE_DMA内存区中（设置了GFP_DMA标志）。原来的PCI标准定义了32位的总线地址；但是，一些PCI硬件设备最初是为ISA总线而设计的，因此它们仍然访问不了物理地址0x00ffffff以上的RAM内存单元。新的PCI-X标准采用64位的总线地址并允许DMA电路可以直接寻址更高的内存。

在Linux中，数据类型dma_addr_t代表一个通用的总线地址。在80x86体系结构中，dma_addr_t对应一个32位长的整数，除非内核支持PAE［参见第二章的“物理地址扩展（PAE）分页机制”一节］，在这种情形下，dma_addr_t代表一个64位的整数。

pci_set_dma_mask()和dma_set_mask()两个辅助函数用于检查总线是否可以接收给定大小的总线地址（mask），如果可以，则通知总线层给定的外围设备将使用该大小的总线地址。

高速缓存的一致性

系统体系结构没有必要在硬件级为硬件高速缓存与DMA电路之间提供一个一致性协议，因此，执行DMA映射操作时，DMA辅助函数必须考虑硬件高速缓存。为了弄清楚这是为什么，假设设备驱动程序把一些数据填充到内存缓冲区中，然后立刻命令硬件设备利用DMA传送方式读取该数据。如果DMA访问这些物理RAM内存单元，而相应的硬件高速缓存行的内容还没有写入RAM中，那么硬件设备所读取的值就是内存缓冲区中的旧值。

设备驱动程序开发人员可以采用两种方法来处理DMA缓冲区，他们分别使用两类不同的辅助函数来完成。用Linux的术语来说，开发人员在下面两种DMA映射类型中进行选择：

一致性DMA映射

使用这种映射方式时，内核必须保证内存与硬件设备间高速缓存一致性不是什么问

题；也就是说CPU在RAM内存单元上所执行的每个写操作对硬件设备而言都是立即可见的，反过来也一样。这种映射方式也称为“同步的”或“一致的”。

流式 DMA 映射

使用这种映射方式时，设备驱动程序必须了解高速缓存一致性问题，这可以使用适当的同步辅助函数来解决。这种映射方式也称为“异步的”或“非一致性的”。

在80x86体系结构中使用DMA时，从不存在高速缓存一致性根本不是什么问题，因为硬件设备驱动程序本身会“窥探”所访问的硬件高速缓存。因此，80x86体系结构中为硬件设备所设计的驱动程序会从前述的两种DMA映射方式中选择一个：它们二者在本质上是等价的。另一方面，在诸如MIPS、SPARC以及PowerPC的一些模型等许多其他的体系结构中，硬件设备通常不窥探硬件高速缓存，因而就会产生高速缓存一致性问题。总的来说，为与体系结构无关的驱动程序选择一个合适的DMA映射方式是很重要的。

一般来说，如果CPU和DMA处理器以不可预知的方式去访问一个缓冲区，那么必须强制使用一致性DMA映射方式（例如，SCSI适配器的command数据结构的缓冲区）。其他情形下，流式DMA映射方式更可取，因为在一些体系结构中处理一致性DMA映射是很麻烦的，并且可能导致更低的系统性能。

一致性 DMA 映射的辅助函数

通常，设备驱动程序在初始化阶段会分配内存缓冲区并建立一致性DMA映射；在卸载时释放映射和缓冲区。为了分配内存缓冲区和建立一致性DMA映射，内核提供了依赖体系结构的pci_alloc_consistent()和dma_alloc_coherent()两个函数。它们均返回新缓冲区的线性地址和总线地址。在80x86体系结构中，它们返回新缓冲区的线性地址和物理地址。为了释放映射和缓冲区，内核提供了pci_free_consistent()和dma_free_coherent()两个函数。

流式 DMA 映射的辅助函数

流式DMA映射的内存缓冲区通常在数据传送之前被映射，在传送之后被取消映射。也有可能在几次DMA传送过程中保持相同的映射，但是在这种情况下，设备驱动程序开发人员必须知道位于内存和外围设备之间的硬件高速缓存。

为了启动一次流式DMA数据传送，驱动程序必须首先利用分区页框分配器（参见第八章的“分区页框分配器”一节）或通用内存分配器（参见第八章的“通用对象”一节）来动态地分配内存缓冲区。然后，驱动程序调用pci_map_single()函数或者dma_map_single()函数建立流式DMA映射，这两个函数接收缓冲区的线性地址作为其参数并返回相应的总线地址。为了释放该映射，驱动程序调用相应的pci_unmap_single()函数或dma_unmap_single()函数。

为了避免高速缓存一致性问题，驱动程序在开始从RAM到设备的DMA数据传送之前，如果有必要，应该调用pci_dma_sync_single_for_device()函数或dma_sync_single_for_device()函数刷新与DMA缓冲区对应的高速缓存行。同样地，从设备到RAM的一次DMA数据传送完成之前设备驱动程序是不可以访问内存缓冲区的；相反，如果有必要，在读缓冲区之前，驱动程序应该调用pci_dma_sync_single_for_cpu()函数或dma_sync_single_for_cpu()函数使相应的硬件高速缓存行无效。在80x86体系结构中，上述函数几乎不做任何事情，因为硬件高速缓存和DMA之间的一致性是由硬件来维护的。

即使是高端内存的缓冲区（参见第八章的“高端内存页框的内核映射”一节）也可以用于DMA传送；开发人员使用pci_map_page()或dma_map_page()函数，给其传递的参数为缓冲区所在页的描述符地址和页中缓冲区的偏移地址。相应地，为了释放高端内存缓冲区的映射，开发人员使用pci_unmap_page()或dma_unmap_page()函数。

内核支持的级别

Linux内核并不完全支持所有可能存在的I/O设备。一般来说，事实上有三种可能的方式支持硬件设备：

根本不支持

应用程序使用适当的in和out汇编语言指令直接与设备的I/O端口进行交互。

最小支持

内核不识别硬件设备，但能识别它的I/O接口。用户程序把I/O接口视为能够读写字符流的顺序设备。

扩展支持

内核识别硬件设备，并处理I/O接口本身。事实上，这种设备可能就没有对应的设备文件。

第一种方式与内核设备驱动程序毫无关系，最常见的例子是X Window系统对图形显示的传统处理方式。这种方法效率很高，尽管它限制了X服务器使用I/O设备产生的硬件中断。为了让X服务器访问所请求的I/O端口，这种方法还需要做一些其他努力。正如第三章的“任务状态段”一节中所介绍的那样，iopl()和ioperm()系统调用给进程授权访问I/O端口。只有具有root权限的用户才可以调用这两个系统调用。但是通过设置可执行文件的*setuid*标志，普通用户也可以使用这些程序（参见第二十章中的“进程的信任状和权能”一节）。

新近的Linux版本支持几种广泛使用的图形卡。*/dev/fb*设备文件为图形卡的帧缓冲区提供了一种抽象，并允许应用软件无需知道图形接口的I/O端口的任何事情就可以访问它。

此外，内核提供了直接绘制基本架构（Direct Rendering Infrastructure，DRI），DRI允许应用软件充分挖掘3D加速图形卡的硬件特性。不管怎样，传统的“自己动手配置”X Window系统服务器还依然被广泛采用。

最小支持方法是用来处理连接到通用I/O接口上的外部硬件设备的。内核通过提供设备文件（由此而提供一个设备驱动程序）来处理I/O接口；应用程序通过读写设备文件来处理外部硬件设备。

最小支持优于扩展支持，因为它保持内核尽可能小。但是，在基于PC的通用I/O接口之中，只有串口和并口的处理使用了这种方法。因此，诸如X服务器之类的应用程序可以直接控制串口鼠标，而串口调制解调器通常都需要一个诸如Minicom、Seyon或PPP（点对点协议）守护进程之类的通信程序。

最小支持的应用范围是有限的，因为当外部设备必须频繁地与内核内部数据结构进行交互时不能使用这种方法。例如，考虑一个连到通用I/O接口上的可移动硬盘。应用程序不能和所有的内核数据结构进程交互，也不能与识别磁盘所需要的函数和装载文件系统所需要的函数进行交互，因此，这种情况下就必须使用扩展支持。

一般情况下，直接连接到I/O总线上的任何硬件设备（如内置硬盘）都要根据扩展支持方法进行处理：内核必须为每个这样的设备提供一个设备驱动程序。通用串行总线（USB）、笔记本电脑上的PCMCIA接口或者SCSI接口——简而言之，除串口和并口之外的所有通用I/O接口之上连接的外部设备都需要扩展支持。

值得注意的是，与标准文件相关的系统调用，如`open()`、`read()`和`write()`，并不总让应用程序完全控制底层硬件设备。事实上，VFS的“最小公分母（lowest-common-denominator）”方法没有包含某些设备所需的特殊命令，或不让应用程序检查设备是否处于某一特殊的内部状态。

已引入的`ioctl()`系统调用可以满足这样的需要。这个系统调用除了设备文件的文件描述符和另一个表示请求的32位参数之外，还可以接收任意多个额外的参数。例如，特殊的`ioctl()`请求可以用来获得CD-ROM的音量或者弹出CD-ROM介质。应用程序可以用这类`ioctl()`请求提供一个CD播放器的用户接口。

字符设备驱动程序

处理字符设备相对比较容易，因为通常并不需要复杂的缓冲策略，也不涉及磁盘高速缓存。当然，字符设备在它们的需求方面有所不同：有些必须实现复杂的通信协议以驱动硬件设备，而有些仅仅需要从硬件设备的一对I/O端口读几个值。例如，多端口串口卡设备（一个硬件设备提供多个串口）的驱动程序比总线鼠标的设备驱动程序要复杂得多。

另一方面，块设备驱动程序本身就比字符设备驱动程序复杂得多。事实上，应用程序可以反复地要求读或写同一个数据块。此外，访问这些设备通常是很慢的。这些特性对磁盘驱动程序的结构产生了深刻的影响。然而，就如我们将在下一章看到的，内核提供了诸如页面高速缓存和块I/O子系统这些高级组件去处理驱动程序。在本章剩下的部分中我们把注意力集中于字符设备驱动程序。

字符设备驱动程序是由一个cdev结构描述的，其字段如表13-8所示。

表13-8：cdev结构中的字段

类型	字段	说明
struct kobject	kobj	内嵌的kobject
struct module *	owner	指向实现驱动程序模块（如果有的话）的指针
struct file_operations *	ops	指向设备驱动程序文件操作表的指针
struct list_head	list	与字符设备文件对应的索引节点链表的头
dev_t	dev	给设备驱动程序所分配的初始主设备号和次设备号
unsigned int	count	给设备驱动程序所分配的设备号范围的大小

list字段是双向循环链表的首部，该链表用于收集相同字符设备驱动程序所对应的字符设备文件的索引节点。可能很多设备文件具有相同的设备号，并对应于相同的字符设备。此外，一个设备驱动程序对应的设备号可以是一个范围，而不仅仅是一个号；设备号位于同一范围内的所有设备文件均由同一个字符设备驱动程序处理。设备号范围的大小存放在count字段中。

cdev_alloc()函数的功能是动态地分配cdev描述符，并初始化内嵌的kobject数据结构，因此在引用计数器的值变为0时会自动释放该描述符。

cdev_add()函数的功能是在设备驱动程序模型中注册一个cdev描述符。它初始化cdev描述符中的dev和count字段，然后调用kobj_map()函数。kobj_map()则依次建立设备驱动程序模型的数据结构，把设备号范围复制到设备驱动程序的描述符中。

设备驱动程序模型为字符设备定义了一个*kobject*映射域，该映射域由一个kobj_map类型的描述符描述，并由全局变量cdev_map引用。kobj_map描述符包括一个散列表，它有255个表项，并由0～255范围的主设备号进行索引。散列表存放probe类型的对象，每个对象都拥有一个已注册的主设备号和次设备号，其中各字段如表13-9所示。

表13-9：probe对象中的字段

类型	字段	说明
struct probe *	next	散列冲突链表中的下一个元素
dev_t	dev	设备号范围的初始设备号（主、次设备号）
unsigned long	range	设备号范围的大小
struct module *	owner	如果有的话，指向实现设备驱动程序模块的指针
struct kobject *(*) (dev_t, int *, void *)	get	探测谁拥有这个设备号范围
int (*)(dev_t, void *)	lock	增加设备号范围内拥有者的引用计数器
void *	data	设备号范围内拥有者的私有数据

调用kobj_map()函数时，把指定的设备号范围加入到散列表中。相应的probe对象的data字段指向设备驱动程序的cdev描述符。执行get和lock方法时把data字段的值传递给它们。在这种情况下，get方法通过一个简捷函数实现，其返回值为cdev描述符中内嵌的kobject数据结构的地址；相反，lock方法本质上用于增加内嵌的kobject数据结构的引用计数器的值。

kobj_lookup()函数接收kobject映射域和设备号作为输入参数；它搜索散列表，如果找到，则返回该设备号所在范围的拥有者的kobject的地址。当这个函数应用到字符设备的映射域时，就返回设备驱动程序描述符cdev中所嵌入的kobject的地址。

分配设备号

为了记录目前已经分配了哪些字符设备号，内核使用散列表chrdevs，表的大小不超过设备号范围。两个不同的设备号范围可能共享同一个主设备号，但是范围不能重叠，因此它们的次设备号应该完全不同。chrdevs包含255个表项，由于散列函数屏蔽了主设备号的高四位——因此，主设备号的个数少于255个，它们被散列到不同的表项中。每个表项指向冲突链表的第一个元素，而该链表是按主、次设备号的递增顺序进行排序的。

冲突链表中的每个元素是一个char_device_struct结构，其各字段如表13-10所示。

表13-10：char_device_struct描述符中的字段

类型	字段	说明
unsigned char_device_struct *	next	指向散列冲突链表中下一个元素的指针
unsigned int	major	设备号范围内的主设备号
unsigned int	baseminor	设备号范围内的初始次设备号

表 13-10：char_device_struct 描述符中的字段（续）

类型	字段	说明
int	minorct	设备号范围的大小
const char *	name	处理设备号范围内的设备驱动程序的名称
struct file_operations *	fops	没有使用
struct cdev *	cdev	指向字符设备驱动程序描述符的指针

本质上可以采用两种方法为字符设备驱动程序分配一个范围内的设备号。所有新的设备驱动程序使用第一种方法，该方法使用 register_chrdev_region()函数和 alloc_chrdev_region()函数为驱动程序分配任意范围内的设备号。例如，为了获得从 dev（类型为 dev_t）开始的大小为 size 的一个设备号范围：

```
register_chrdev_region(dev, size, "foo");
```

上述函数并不执行 cdev_add()，因此设备驱动程序在所要求的设备号范围被成功分配时必须执行 cdev_add()函数。

第二种方法使用 register_chrdev()函数，它分配一个固定的设备号范围，该范围包含唯一一个主设备号以及 0~255 的次设备号。在这种情形下，设备驱动程序不必调用 cdev_add()函数。

register_chrdev_region()函数和 alloc_chrdev_region()函数

register_chrdev_region()函数接收三个参数：初始的设备号（主设备号和次设备号）、请求的设备号范围大小（与次设备号的大小一样）以及这个范围内的设备号对应的设备驱动程序的名称。该函数检查请求的设备号范围是否跨越一些次设备号，如果是，则确定其主设备号以及覆盖整个区间的相应设备号范围；然后，在每个相应设备号范围上调用 __register_chrdev_region()函数（参见下文）。

alloc_chrdev_region()函数与 register_chrdev_region()相似，但它可以动态地分配一个主设备号；因此，该函数接收的参数为设备号范围内的初始次设备号、范围的大小以及设备驱动程序的名称。结束时它也调用__ __register_chrdev_region()函数。

__ __register_chrdev_region()函数执行以下步骤：

1. 分配一个新的 char_device_struct 结构，并用 0 填充。
2. 如果设备号范围内的主设备号为0，那么设备驱动程序请求动态分配一个主设备号。

函数从散列表的末尾表项开始继续向后寻找一个与尚未使用的主设备号对应的空冲突链表（NULL指针）。若没有找到空表项，则返回一个错误码（注6）。

3. 初始化char_device_struct结构中的初始设备号、范围大小以及设备驱动程序名称。
4. 执行散列函数计算与主设备号对应的散列表索引。
5. 遍历冲突链表，为新的char_device_struct结构寻找正确的位置。同时，如果找到与请求的设备号范围重叠的一个范围，则返回一个错误码。
6. 将新的char_device_struct描述符插入冲突链表中。
7. 返回新的char_device_struct描述符的地址。

register_chrdev()函数

驱动程序使用register_chrdev()函数时需要一个老式的设备号范围：一个单独的主设备号和0～255的次设备号范围。该函数接收的参数为：请求的主设备号major（如果是0则动态分配）、设备驱动程序的名称name和一个指针fops（它指向设备号范围内的特定字符设备文件的文件操作表）。该函数执行下列操作：

1. 调用__register_chrdev_region()函数分配请求的设备号范围。如果返回一个错误码（不能分配该范围），函数将终止运行。
2. 为设备驱动程序分配一个新的cdev结构。
3. 初始化cdev结构：
   a. 将内嵌的kobject类型设置为ktype_cdev_dynamic类型的描述符（参见前面的“kobject”一节）。
   b. 将owner字段设置为fops->owner的内容。
   c. 将ops字段设置为文件操作表的地址fops。
   d. 将设备驱动程序的名称拷贝到内嵌的kobject结构里的name字段中。
4. 调用cdev_add()函数（在前面解释过）。
5. 将__register_chrdev_region()函数在第1步中返回的char_device_struct描述符的cdev字段设置为设备驱动程序的cdev描述符的地址。
6. 返回分配的设备号范围的主设备号。

注6：注意，内核可以动态分配一个小于255的主设备号，而且在某些情况下，即使存在一个小于255的未被使用的主设备号，分配也可能失败。我们可以期待将来会改变这种限制。

访问字符设备驱动程序

我们在“设备文件的 VFS 处理”一节中曾提到，由 open() 系统调用服务例程触发的 dentry_open() 函数定制字符设备文件的文件对象的 f_op 字段，以使它指向 def_chr_fops 表。这个表几乎为空；它仅仅定义了 chrdev_open() 函数作为设备文件的打开方法。这个方法由 dentry_open() 直接调用。

chrdev_open() 函数接收的参数为索引节点的地址 inode、指向所打开文件对象的指针 filp。本质上它执行以下操作：

1. 检查指向设备驱动程序的 cdev 描述符的指针 inode->i_cdev。如果该字段不为空，则 inode 结构已经被访问：增加 cdev 描述符的引用计数器值并跳转到第 6 步。
2. 调用 kobj_lookup() 函数搜索包括该设备号在内的范围。如果该范围不存在，则返回一个错误码；否则，函数计算与该范围相对应的 cdev 描述符的地址。
3. 将 inode 对象的 inode->i_cdev 字段设置为 cdev 描述符的地址。
4. 将 inode->i_cindex 字段设置为设备驱动程序的设备号范围内的设备号的相关索引（设备号范围内的第一个次设备号的索引值为 0，第二个为 1，依此类推）。
5. 将 inode 对象加入到由 cdev 描述符的 list 字段所指向的链表中。
6. 将 filp->f_ops 文件操作指针初始化为 cdev 描述符的 ops 字段的值。
7. 如果定义了 filp->f_ops->open 方法，chrdev_open() 函数就会执行该方法。若设备驱动程序处理一个以上的设备号，则 chrdev_open() 一般会再次设置 file 对象的文件操作，这样可以为所访问的设备文件安装合适的文件操作。
8. 成功时返回 0 结束。

字符设备的缓冲策略

传统的类 Unix 操作系统把硬件设备划分为块设备和字符设备。但是，这种分类并不能说明整个事实。某些设备在一次单独的 I/O 操作中能够传送大量的数据，而有些设备则只能传送几个字符。

例如，PS/2 鼠标驱动程序在每次读操作中获得几个字节 —— 它们对应鼠标按钮的状态和屏幕上鼠标的指针。这种设备是最容易处理的。首先从设备的输入寄存器中一次读一个字符的输入数据，并存放在合适的内核数据结构中；然后，在空闲时把这个数据拷贝到进程的地址空间。同理，把输出数据首先从进程的地址空间拷贝到合适的内核数据结构中，然后，再一次一个字符地写到 I/O 设备的输出寄存器。显然，这种设备的 I/O 驱

动程序没有使用DMA，因为CPU建立DMA I/O操作所花费的时间跟把数据移到I/O端口所花费的时间差不多。

另一方面，内核也必须准备处理在每次I/O操作中产生大量字节的设备，这些设备或者是诸如声卡或网卡的顺序设备，或者是诸如各类磁盘（软盘、光盘、SCSI磁盘等）的随机访问设备。

例如，假定你已经为自己的计算机配置了声卡，以便能够录下来自麦克风的声音。声卡以固定的频率（比如说44.14 kHz）对来自麦克风的电信号进行采样，并产生一个16位数的输入数据块的流。声卡驱动程序必须能处理所有可能情况下这种蜂拥而至的数据，即使当CPU暂时忙于运行某个其他进程也不例外。

这可以结合两种不同的技术做到：

- 使用DMA方式传送数据块。
- 运用两个或多个元素的循环缓冲区，每个元素具有一个数据块的大小。当一个中断（发送一个信号表明新的数据块已被读入）发生时，中断处理程序把指针移到循环缓冲区的下一个元素，以便将来的数据会存放在一个空元素中。相反，只要驱动程序把数据成功地拷贝到用户地址空间，就释放循环缓冲区中的元素，以便用它来保存从硬件设备传送来的新数据。

循环缓冲区的作用是消除CPU负载的峰值：即使接收数据的用户态应用程序因为其他高优先级任务而慢下来，DMA也要能够继续填充循环缓冲区中的元素，因为中断处理程序代表当前运行的进程执行。

当接收来自网卡的数据包时有类似的情况发生，只是在这种情况下，进入的数据流都是异步的。数据包被互相独立地接收，且两个连续的数据包之间到达的时间间隔是不可预测的。

总而言之，顺序设备的缓冲区是容易处理的，因为同一缓冲区从不会被重用：音频应用程序不能要求麦克风重新传送同一数据块。

我们将在第十五章中看到对随机访问设备（各种各样的磁盘）进行缓冲是相当复杂的。



第十四章

块设备驱动程序

本章主要讨论块设备（例如各类磁盘）的I/O驱动程序。块设备的主要特点是，CPU和总线读写数据所花时间与磁盘硬件的速度不匹配。块设备的平均访问时间很高。每个操作都需要几个毫秒才能完成，主要是因为磁盘控制器必须在磁盘表面将磁头移动到记录数据的确切位置。但是，当磁头到达正确位置时，数据传送就可以稳定在每秒几十MB的速率。

Linux块设备处理程序的组织是相当复杂的。我们不可能对内核的块设备I/O子系统中包含的所有函数都进行详细讨论；但是，我们会提纲挈领地介绍一般软件体系结构。与前一章一样，我们的目标是描述Linux如何支持各种块设备驱动程序的实现，而不只说明如何实现一个具体的驱动程序。

我们将在“块设备的处理”一节首先说明Linux块设备I/O子系统的一般体系结构。然后将在“通用块层”、“I/O调度程序”和“块设备驱动程序”这几节中描述块设备I/O子系统的主要组件。在最后的“打开块设备文件”一节中，我们简要地介绍一下打开一个块设备文件时内核所执行的步骤。

块设备的处理

块设备驱动程序上的每个操作都涉及很多内核组件；其中最重要的一些如图14-1所示。

例如，我们假设一个进程在某个磁盘文件上发出一个`read()`系统调用 —— 我们将会看到处理write请求本质上采用同样的方式。下面是内核对进程请求给予回应的一般步骤：

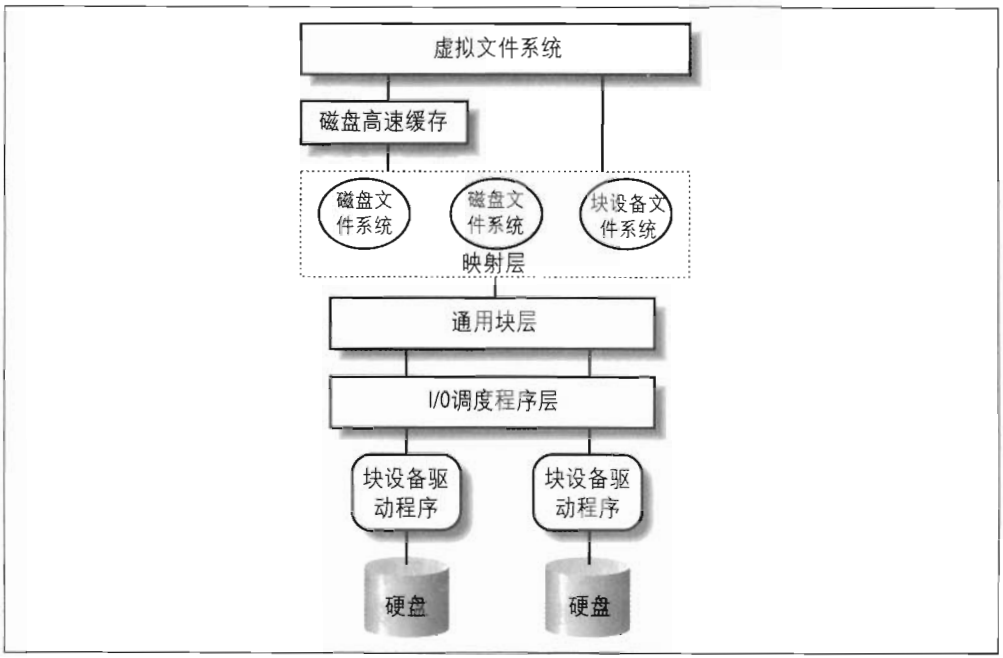


图 14-1：一个块设备操作所涉及的内核组件

1. read()系统调用的服务例程调用一个适当的VFS函数，将文件描述符和文件内的偏移量传递给它。虚拟文件系统位于块设备处理体系结构的上层，它提供一个通用的文件模型，Linux支持的所有文件系统均采用该模型。我们在第十二章已经详细介绍了VFS层。

2. VFS函数确定所请求的数据是否已经存在，如果有必要的话，它决定如何执行read操作。有时候没有必要访问磁盘上的数据，因为内核将大多数最近从块设备读出或写入其中的数据保存在RAM中。第十五章介绍了磁盘高速缓存机制，而第十六章详细说明了VFS如何处理磁盘操作以及如何与磁盘高速缓存和文件系统交互。

3. 我们假设内核从块设备读数据，那么它就必须确定数据的物理位置。为了做到这点，内核依赖映射层（*mapping layer*），主要执行下面两步：

   a. 内核确定该文件所在文件系统的块大小，并根据文件块的大小计算所请求数据的长度。本质上，文件被看作拆分成许多块，因此内核确定请求数据所在的块号（文件开始位置的相对索引）。

   b. 接下来，映射层调用一个具体文件系统的函数，它访问文件的磁盘节点，然后根据逻辑块号确定所请求数据在磁盘上的位置。事实上，磁盘也被看作拆分成许多块，因此内核必须确定存放所请求数据的块对应的号（磁盘或分区开始位

置的相对索引)。由于一个文件可能存储在磁盘上的不连续块中,因此存放在磁盘索引节点中的数据结构将每个文件块号映射为一个逻辑块号(注1)。

我们将在第十六章中说明映射层的功能,在第十八章中将介绍一些典型的磁盘文件系统。

4. 现在内核可以对块设备发出读请求。内核利用通用块层(*generic block lnyer*)启动I/O操作来传送所请求的数据。一般而言,每个I/O操作只针对磁盘上一组连续的块。由于请求的数据不必位于相邻的块中,所以通用块层可能启动几次I/O操作。每次I/O操作是由一个"块I/O"(简称"bio")结构描述,它收集底层组件需要的所有信息以满足所发出的请求。

   通用块层为所有的块设备提供了一个抽象视图,因而隐藏了硬件块设备间的差异性。几乎所有的块设备都是磁盘,所以通用块层也提供了一些通用数据结构来描述"磁盘"或"磁盘分区"。我们将在本章的"通用块层"一节中讨论通用块层和bio数据结构。

5. 通用块层下面的"I/O调度程序"根据预先定义的内核策略将待处理的I/O数据传送请求进行归类。调度程序的作用是把物理介质上相邻的数据请求聚集在一起。我们将在本章后面的"I/O调度程序"一节中介绍调度程序。

6. 最后,块设备驱动程序向磁盘控制器的硬件接口发送适当的命令,从而进行实际的数据传送。我们将在后面的"块设备驱动程序"一节介绍通用块设备驱动程序的总体组织结构。

如你所见,块设备中的数据存储涉及了许多内核组件;每个组件采用不同长度的块来管理磁盘数据:

- 硬件块设备控制器采用称为"扇区"的固定长度的块来传送数据。因此,I/O调度程序和块设备驱动程序必须管理数据扇区。
- 虚拟文件系统、映射层和文件系统将磁盘数据存放在称为"块"的逻辑单元中。一个块对应文件系统中一个最小的磁盘存储单元。
- 我们很快会看到,块设备驱动程序应该能够处理数据的"段":一个段就是一个内存页或内存页的一部分,它们包含磁盘上物理相邻的数据块。
- 磁盘高速缓存作用于磁盘数据的"页"上,每页正好装在一个页框中。
- 通用块层将所有的上层和下层的组件组合在一起,因此它了解数据的扇区、块、段以及页。

注1: 但是,如果是从原始块设备文件进行读访问,映射层就不调用具体文件系统的方法,而是把块设备文件中的偏移量转换成在磁盘或在对应该设备文件的磁盘分区中的位置。

即使有许多不同的数据块，它们通常也是共享相同的物理RAM单元。例如，图14-2显示了一个具有4096字节的页的构造。上层内核组件将页看成是由4个1024字节组成的块缓冲区。块设备驱动程序正在传送页中的后3个块，因此这3块被插入到涵盖了后3072字节的段中。硬盘控制器将该段看成是由6个512字节的扇区组成。

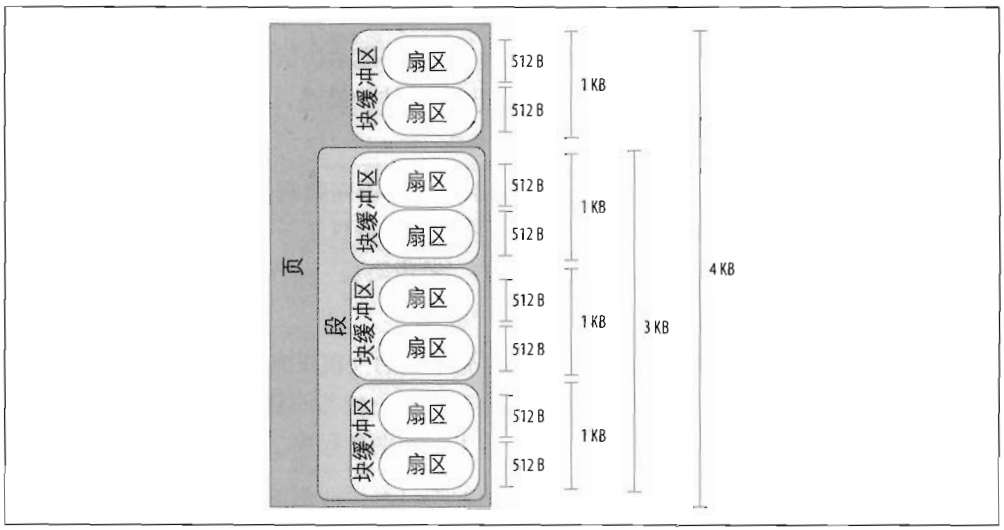


图14-2：包含磁盘数据的页的典型构造

本章我们介绍处理块设备的下层内核组件：通用块层、I/O调度程序以及块设备驱动程序，因此我们将注意力集中在扇区、块和段上。

扇区

为了达到可接受的性能，硬盘和类似的设备快速传送几个相邻字节的数据。块设备的每次数据传送操作都作用于一组称为扇区的相邻字节。在下面的讨论中，我们假定字节按相邻的方式记录在磁盘表面，这样一次搜索操作就可以访问到它们。尽管磁盘的物理构造很复杂，但是硬盘控制器接收到的命令将磁盘看成一大组扇区。

在大部分磁盘设备中，扇区的大小是512字节，但是一些设备使用更大的扇区（1024和2048字节）。注意，应该把扇区作为数据传送的基本单元；不允许传送少于一个扇区的数据，尽管大部分磁盘设备都可以同时传送几个相邻的扇区。

在Linux中，扇区大小按惯例都设为512字节；如果一个块设备使用更大的扇区，那么相应的底层块设备驱动程序将做些必要的变换。因此，对存放在块设备中的一组数据是通过它们在磁盘上的位置来标识，即其首个512字节扇区的下标以及扇区的数目。扇区的下标存放在类型为sector_t的32位或64位的变量中。

块

扇区是硬件设备传送数据的基本单位，而块是VFS和文件系统传送数据的基本单位。例如，内核访问一个文件的内容时，它必须首先从磁盘上读文件的磁盘索引节点所在的块（参见第十二章的“索引节点对象”一节）。该块对应磁盘上一个或多个相邻的扇区，而VFS将其看成是一个单一的数据单元。

在Linux中，块大小必须是2的幂，而且不能超过一个页框。此外，它必须是扇区大小的整数倍，因为每个块必须包含整数个扇区。因此，在80 × 86体系结构中，允许块的大小为512、1024、2048和4096字节。

块设备的块大小不是唯一的。创建一个磁盘文件系统时，管理员可以选择合适的块大小。因此，同一个磁盘上的几个分区可能使用不同的块大小。此外，对块设备文件的每次读或写操作是一种“原始”访问，因为它绕过了磁盘文件系统；内核通过使用最大的块（4096字节）执行该操作。

每个块都需要自己的块缓冲区，它是内核用来存放块内容的RAM内存区。当内核从磁盘读出一个块时，就用从硬件设备中所获得的值来填充相应的块缓冲区；同样，当内核向磁盘中写入一个块时，就用相关块缓冲区的实际值来更新硬件设备上相应的一组相邻字节。块缓冲区的大小通常要与相应块的大小相匹配。

缓冲区首部是一个与每个缓冲区相关的buffer_head类型的描述符。它包含内核处理缓冲区需要了解的所有信息；因此，在对每个缓冲区进行操作之前，内核都要首先检查其缓冲区首部。我们将在第十五章中详细介绍缓冲区首部中的所有字段值；但是在本章中我们仅仅介绍其中的一些字段：b_page、b_data、b_blocknr和b_bdev。

b_page字段存放的是块缓冲区所在页框的页描述符地址。如果页框位于高端内存中，那么b_data字段存放页中块缓冲区的偏移量；否则，b_data存放块缓冲区本身的起始线性地址。b_blocknr字段存放的是逻辑块号（例如磁盘分区中的块索引）。最后，b_bdev字段标识使用缓冲区首部的块设备（参见本章后面的“块设备”一节）。

段

我们知道对磁盘的每个I/O操作就是在磁盘与一些RAM单元之间相互传送一些相邻扇区的内容。大多数情况下，磁盘控制器直接采用DMA方式进行数据传送[参见第十三章中的“直接内存访问（DMA）”一节]。块设备驱动程序只要向磁盘控制器发送一些适当的命令就可以触发一次数据传送，一旦完成数据的传送，控制器就会发出一个中断通知块设备驱动程序。

DMA 方式传送的是磁盘上相邻扇区的数据。这是一个物理约束：磁盘控制器允许DMA传送不相邻的扇区数据，但是这种方式的传送速率很低，因为在磁盘表面上移动读/写磁头是相当慢的。

老式的磁盘控制器仅仅支持“简单的”DMA传送方式：在这种传送方式中，磁盘必须与RAM中的连续内存单元相互传送数据。但是，新的磁盘控制器也支持所谓的分散-聚集（scatter-gather）DMA传送方式：此种方式中，磁盘可以与一些非连续的内存区相互传送数据。

启动一次分散-聚集DMA传送，块设备驱动程序需要向磁盘控制器发送：

- 要传送的起始磁盘扇区号和总的扇区数
- 内存区的描述符链表，其中链表的每项包含一个地址和一个长度

磁盘控制器负责整个数据传送；例如，在读操作中控制器从相邻磁盘扇区中获得数据，然后将它们存放到不同的内存区中。

为了使用分散-聚集DMA传送方式，块设备驱动程序必须能够处理称为段的数据存储单元。一个段就是一个内存页或内存页中的一部分，它们包含一些相邻磁盘扇区中的数据。因此，一次分散-聚集DMA操作可能同时传送几个段。

注意，块设备驱动程序不需要知道块、块大小以及块缓冲区。因此，即使高层将段看成是由几个块缓冲区组成的页，块设备驱动程序也不用对此给予关注。

正如我们所见，如果不同的段在RAM中相应的页框正好是连续的并且在磁盘上相应的数据块也是相邻的，那么通用块层可以合并它们。通过这种合并方式产生的更大的内存区就称为物理段。

然而，在多种体系结构上还允许使用另一个合并方式：通过使用一个专门的总线电路[如IO-MMU；参见第十三章中的“直接内存访问（DMA）”一节]来处理总线地址与物理地址间的映射。通过这种合并方式产生的内存区称为硬件段。由于我们将注意力集中在80 × 86体系结构上，它在总线地址和物理地址之间不存在动态的映射，因此在本章剩余部分我们假定硬件段总是对应物理段。

通用块层

通用块层是一个内核组件，它处理来自系统中的所有块设备发出的请求。由于该层所提供的函数，内核可以容易地做到：

- 将数据缓冲区放在高端内存 —— 仅当CPU访问其数据时，才将页框映射为内核中的线性地址空间，并在数据访问完后取消映射。
- 通过一些附加的手段，实现一个所谓的“零－复制”模式，将磁盘数据直接存放在用户态地址空间而不是首先复制到内核内存区；事实上，内核为I/O数据传送使用的缓冲区所在的页框就映射在进程的用户态线性地址空间中。
- 管理逻辑卷，例如由LVM（逻辑卷管理器）和RAID（廉价磁盘冗余阵列）使用的逻辑卷：几个磁盘分区，即使位于不同的块设备中，也可以被看作是一个单一的分区。
- 发挥大部分新磁盘控制器的高级特性，例如大主板磁盘高速缓存、增强的DMA性能、I/O传送请求的相关调度等等。

bio结构

通用块层的核心数据结构是一个称为*bio*的描述符，它描述了块设备的I/O操作。每个bio结构都包含一个磁盘存储区标识符（存储区中的起始扇区号和扇区数目）和一个或多个描述与I/O操作相关的内存区的段。bio由bio数据结构描述，其各字段如表14-1所示。

表14-1：bio结构中的字段

类型	字段	说明
sector_t	bi_sector	块I/O操作的第一个磁盘扇区
struct bio *	bi_next	链接到请求队列中的下一个bio
struct block_device *	bi_bdev	指向块设备描述符的指针
unsigned long	bi_flags	bio的状态标志
unsigned long	bi_rw	I/O操作标志
unsigned short	bi_vcnt	bio的bio_vec数组中段的数目
unsigned short	bi_idx	bio的bio_vec数组中段的当前索引值
unsigned short	bi_phys_segments	合并之后bio中物理段的数目
unsigned short	bi_hw_segments	合并之后硬件段的数目
unsigned int	bi_size	需要传送的字节数
unsigned int	bi_hw_front_size	硬件段合并算法使用
unsigned int	bi_hw_back_size	硬件段合并算法使用
unsigned int	bi_max_vecs	bio的bio_vec数组中允许的最大段数
struct bio_vec *	bi_io_vec	指向bio的bio_vec数组中的段的指针

表 14-1：bio 结构中的字段（续）

类型	字段	说明
bio_end_io_t *	bi_end_io	bio 的 I/O 操作结束时调用的方法
atomic_t	bi_cnt	bio 的引用计数器
void *	bi_private	通用块层和块设备驱动程序的I/O完成方法使用的指针
bio_destructor_t *	bi_destructor	释放 bio 时调用的析构方法（通常是 bio_destructor()方法）

bio 中的每个段是由一个 bio_vec 数据结构描述的，其中各字段如表 14-2 所示。bio 中的 bi_io_vec 字段指向 bio_vec 数据结构的第一个元素，bi_vcnt 字段则存放了 bio_vec 数组中当前的元素个数。

表 14-2：bio_vec 结构中的字段

类型	字段	说明
struct page *	bv_page	指向段的页框中页描述符的指针
unsigned int	bv_len	段的字节长度
unsigned int	bv_offset	页框中段数据的偏移量

在块 I/O 操作期间 bio 描述符的内容一直保持更新。例如，如果块设备驱动程序在一次分散 - 聚集 DMA 操作中不能完成全部的数据传送，那么 bio 中的 bi_idx 字段会不断更新来指向待传送的第一个段。为了从索引 bi_idx 指向的当前段开始不断重复 bio 中的段，设备驱动程序可以执行宏 bio_for_each_segment。

当通用块层启动一次新的 I/O 操作时，调用 bio_alloc()函数分配一个新的 bio 结构。通常，bio 结构是由 slab 分配器分配的，但是，当内存不足时，内核也会使用一个备用的 bio 小内存池（参见第八章的“内存池”一节）。内核也为 bio_vec 结构分配内存池——毕竟，分配一个bio结构而不能分配其中的段描述符也是没有什么意义的。相应地，bio_put()函数减少bio中引用计数器(bi_cnt)的值，如果该值等于0，则释放bio结构以及相关的 bio_vec 结构。

磁盘和磁盘分区的表示

磁盘是一个由通用块层处理的逻辑块设备。通常一个磁盘对应一个硬件块设备，例如硬盘、软盘或光盘。但是，磁盘也可以是一个虚拟设备，它建立在几个物理磁盘分区之上或一些 RAM 专用页中的内存区上。在任何情形中，借助通用块层提供的服务，上层内核组件可以以同样的方式工作在所有的磁盘上。

磁盘是由 gendisk 对象描述的，其中各字段如表 14-3 所示。

表 14-3：gendisk 对象中的字段

类型	字段	说明
int	major	Major 磁盘主设备号
int	first_minor	与磁盘关联的第一个次设备号
int	minors	与磁盘关联的次设备号范围
char [32]	disk_name	磁盘的标准名称(通常是相应设备文件的规范名称)
struct hd_struct **	part	磁盘的分区描述符数组
struct block_device_operations *	fops	指向块设备操作表的指针
struct request_queue *	queue	指向磁盘请求队列的指针(参见本章后面的“请求队列描述符”一节)
void *	private_data	块设备驱动程序的私有数据
sector_t	capacity	磁盘内存区的大小（扇区数目）
int	flags	描述磁盘类型的标志（见下文）
char [64]	devfs_name	*devfs* 特殊文件系统（现在已不赞成使用）中的设备文件名称
int	number	不再使用
struct device *	driverfs_dev	指向磁盘的硬件设备的 device 对象的指针（参见第十三章的“设备驱动程序模型的组件”一节）
struct kobject	kobj	内嵌的 kobject 结构（参见第十三章的“kobject”一节）
struct timer_rand_state *	random	该指针指向的这个数据结构记录磁盘中断的定时，由内核内置的随机数发生器使用
int	policy	如果磁盘是只读的，则置为 1（写操作禁止），否则为 0
atomic_t	sync_io	写入磁盘的扇区数计数器，仅为 RAID 使用
unsigned long	stamp	统计磁盘队列使用情况的时间戳
unsigned long	stamp_idl	同上
int	in_flight	正在进行的 I/O 操作数
struct disk_stats *	dkstats	统计每个 CPU 使用磁盘的情况

flags字段存放了关于磁盘的信息。其中最重要的标志是GENHD_FL_UP：如果设置它，那么磁盘将被初始化并可以使用。另一个相关的标志是GENHD_FL_REMOVABLE，如果是诸如软盘或光盘这样可移动的磁盘，那么就要设置该标志。

gendisk对象的fops字段指向一个表block_device_operations，该表为块设备的主要操作存放了几个定制的方法（如表14-4所示）。

表14-4：块设备的方法

方法	触发事件
open	打开块设备文件
release	关闭对块设备文件的最后一个引用
ioctl	在块设备文件上发出ioctl()系统调用（使用大内核锁）
compat_ioctl	在块设备文件上发出ioctl()系统调用（不使用大内核锁）
media_changed	检查可移动介质是否已经变化（例如软盘）
revalidate_disk	检查块设备是否持有有效数据

通常硬盘被划分成几个逻辑分区。每个块设备文件要么代表整个磁盘，要么代表磁盘中的某一个分区。例如，一个主设备号为3、次设备号为0的设备文件*/dev/hda*代表的可能是一个主EIDE磁盘，该磁盘中的前两个分区分别由设备文件*/dev/hda1*和*/dev/hda2*代表，它们的主设备号都是3，而次设备号分别为1和2。一般而言，磁盘中的分区是由连续的次设备号来区分的。

如果将一个磁盘分成了几个分区，那么其分区表保存在hd_struct结构的数组中，该数组的地址存放在gendisk对象的part字段中。通过磁盘内分区的相对索引对该数组进行索引。hd_struct描述符中的字段如表14-5所示。

表14-5：hd_struct描述符中的字段

类型	字段	说明
sector_t	start_sect	磁盘中分区的起始扇区
sector_t	nr_sects	分区的长度（扇区数）
struct kobject	kobj	内嵌的kobject(参见第十三章的“kobject”一节)
unsigned int	reads	对分区发出的读操作次数
unsigned int	read_sectors	从分区读取的扇区数
unsigned int	writes	对分区发出的写操作次数
unsigned int	write_sectors	写进分区的扇区数

表 14-5：hd_struct 描述符中的字段（续）

类型	字段	说明
int	policy	如果分区是只读的，则置为 1，否则为 0
int	partno	磁盘中分区的相对索引

当内核发现系统中一个新的磁盘时（在启动阶段，或将一个可移动介质插入一个驱动器中时，或在运行期附加一个外置式磁盘时），就调用 alloc_disk() 函数，该函数分配并初始化一个新的 gendisk 对象，如果新磁盘被分成了几个分区，那么 alloc_disk() 还会分配并初始化一个适当的 hd_struct 类型的数组。然后，内核调用 add_disk() 函数将新的 gendisk 对象插入到通用块层的数据结构中（参见本章后面的“注册和初始化设备驱动程序”一节）。

提交请求

我们介绍一下当向通用块层提交一个 I/O 操作请求时，内核所执行的步骤顺序。我们假设被请求的数据块在磁盘上是相邻的，并且内核已经知道了它们的物理位置。

第一步是执行 bio_alloc() 函数分配一个新的 bio 描述符。然后，内核通过设置一些字段值来初始化 bio 描述符：

- 将 bi_sector 设为数据的起始扇区号（如果块设备分成了几个分区，那么扇区号是相对于分区的起始位置的）。
- 将 bi_size 设为涵盖整个数据的扇区数目。
- 将 bi_bdev 设为块设备描述符的地址（参见本章后面的“块设备”一节）。
- 将 bi_io_vec 设为 bio_vec 结构数组的起始地址，数组中的每个元素描述了 I/O 操作中的一个段（内存缓存）；此外，将 bi_vcnt 设为 bio 中总的段数。
- 将 bi_rw 设为被请求操作的标志。其中最重要的标志指明数据传送的方向：READ（0）或 WRITE（1）。
- 将 bi_end_io 设为当 bio 上的 I/O 操作完成时所执行的完成程序的地址。

一旦 bio 描述符被进行了适当的初始化，内核就调用 generic_make_request() 函数，它是通用块层的主要入口点。该函数主要执行下列操作：

1. 检查 bio->bi_sector 没有超过块设备的扇区数。如果超过，则将 bio->bi_flags 设置为 BIO_EOF 标志，然后打印一条内核错误信息，调用 bio_endio() 函数，并终止。bio_endio() 更新 bio 描述符中的 bi_size 和 bi_sector 值，然后调用 bio

的bi_end_io方法。bi_end_io函数的实现本质上依赖于触发I/O数据传送的内核组件；我们将在下面的章节中看到bi_end_io方法的一些例子。

2. 获取与块设备相关的请求队列q（参见本章后面的“请求队列描述符”一节），其地址存放在块设备描述符的bd_disk字段中，其中的每个元素由bio->bi_bdev指向。

3. 调用block_wait_queue_running()函数检查当前正在使用的I/O调度程序是否可以被动态取代；若可以，则让当前进程睡眠直到启动一个新的I/O调度程序（参见下一节“I/O调度程序”）。

4. 调用blk_partition_remap()函数检查块设备是否指的是一个磁盘分区（bio->bi_bdev不等于bio->bi_dev->bd_contains；参见本章后面的“块设备”一节）。如果是，则从bio->bi_bdev获取分区的hd_struct描述符，从而执行下面的子操作：

   a. 根据数据传送的方向，更新hd_struct描述符中的read_sectors和reads值，或write_sectors和writes值。

   b. 调整bio->bi_sector值使得把相对于分区的起始扇区号转变为相对于整个磁盘的扇区号。

   c. 将bio->bi_bdev设置为整个磁盘的块设备描述符（bio->bd_contains）。

   从现在开始，通用块层、I/O调度程序以及设备驱动程序将忘记磁盘分区的存在，直接作用于整个磁盘。

5. 调用q->make_request_fn方法将bio请求插入请求队列q中。

6. 返回。

在本章后面的“向I/O调度程序发出请求”一节中我们将讨论make_request_fn方法典型实现。

I/O调度程序

虽然块设备驱动程序一次可以传送一个单独的扇区，但是块I/O层并不会为磁盘上每个被访问的扇区都单独执行一次I/O操作；这会导致磁盘性能的下降，因为确定磁盘表面上扇区的物理位置是相当费时的。取而代之的是，只要可能，内核就试图把几个扇区合并在一起，并作为一个整体来处理，这样就减少了磁头的平均移动时间。

当内核组件要读或写一些磁盘数据时，实际上创建一个块设备请求。从本质上说，请求描述的是所请求的扇区以及要对它执行的操作类型（读或写）。然而，并不是请求一发出，内核就满足它——I/O操作仅仅被调度，执行会向后推迟。这种人为的延迟是提高

块设备性能的关键机制。当请求传送一个新的数据块时，内核检查能否通过稍微扩展前一个一直处于等待状态的请求而满足新请求（也就是说，能否不用进一步的寻道操作就能满足新请求）。由于磁盘的访问大都是顺序的，因此这种简单机制就非常高效。

延迟请求复杂化了块设备的处理。例如，假设某个进程打开了一个普通文件，然后，文件系统的驱动程序就要从磁盘读取相应的索引节点。块设备驱动程序把这个请求加入一个队列，并把这个进程挂起，直到存放索引节点的块被传送为止。然而，块设备驱动程序本身不会被阻塞，因为试图访问同一磁盘的任何其他进程也可能被阻塞。

为了防止块设备驱动程序被挂起，每个I/O操作都是异步处理的。特别是块设备驱动程序是中断驱动的（参见第十三章的“监控I/O操作”一节）：通用块层调用I/O调度程序产生一个新的块设备请求或扩展一个已有的块设备请求，然后终止。随后激活的块设备驱动程序会调用一个所谓的策略例程（*strategy routine*）选择一个待处理的请求，并向磁盘控制器发出一条适当的命令来满足这个请求。当I/O操作终止时，磁盘控制器就产生一个中断，如果需要，相应的中断处理程序就又调用策略例程去处理队列中的另一个请求。

每个块设备驱动程序都维持着自己的请求队列，它包含设备待处理的请求链表。如果磁盘控制器正在处理几个磁盘，那么通常每个物理块设备都有一个请求队列。在每个请求队列上单独执行I/O调度，这样可以提高磁盘的性能。

请求队列描述符

请求队列是由一个大的数据结构 `request_queue` 表示的，其字段如表14-6所示。

表14-6：请求队列描述符中的字段

类型	字段	说明
`struct list_head`	`queue_head`	待处理请求的链表
`struct request *`	`last_merge`	指向队列中首先可能合并的请求描述符
`elevator_t *`	`elevator`	指向elevator对象的指针（参见后面的“I/O调度算法”一节）
`struct request_list`	`rq`	为分配请求描述符所使用的数据结构
`request_fn_proc *`	`request_fn`	实现驱动程序的策略例程入口点的方法
`merge_request_fn *`	`back_merge_fn`	检查是否可能将bio合并到请求队列的最后一个请求中的方法
`merge_request_fn *`	`front_merge_fn`	检查是否可能将bio合并到队列的第一个请求中的方法

表 14-6：请求队列描述符中的字段（续）

类型	字段	说明
merge_requests_fn *	merge_requests_fn	试图合并请求队列中两个相邻请求的方法
make_request_fn *	make_request_fn	将一个新请求插入请求队列时调用的方法
prep_rq_fn *	prep_rq_fn	该方法把这个处理请求的命令发送给硬件设备
unplug_fn *	unplug_fn	去掉块设备的方法（参见本章后面的“激活块设备驱动程序”一节）
merge_bvec_fn *	merge_bvec_fn	当增加一个新段时，该方法返回可插入到某个已存在的 bio 结构中的字节数（通常未定义）
activity_fn *	activity_fn	将某个请求加入请求队列时调用的方法（通常未定义）
issue_flush_fn *	issue_flush_fn	刷新请求队列时调用的方法（通过连续处理所有的请求清空队列）
struct timer_list	unplug_timer	插入设备时使用的动态定时器（参见后面的“激活块设备驱动程序”一节）
int	unplug_thresh	如果请求队列中待处理请求数大于该值，将立即去掉请求设备（缺省值是 4）
unsigned long	unplug_delay	去掉设备之前的时间延迟（缺省值是 3ms）
struct work_struct	unplug_work	去掉设备时使用的操作队列（参见后面的“激活块设备驱动程序”一节）
struct backing_dev_info	backing_dev_info	参见本表后面的正文
void *	queuedata	指向块设备驱动程序的私有数据的指针
void *	activity_data	activity_fn 方法使用的私有数据
unsigned long	bounce_pfn	在大于该页框号时必须使用缓冲区回弹（参见本章前面的“提交请求”一节）
int	bounce_gfp	回弹缓冲区的内存分配标志
unsigned long	queue_flags	描述请求队列状态的标志
spinlock_t *	ueue_lock	指向请求队列锁的指针
struct kobject	kobj	请求队列的内嵌 kobject 结构
unsigned long	nr_requests	请求队列中允许的最大请求数

表 14-6：请求队列描述符中的字段（续）

类型	字段	说明
unsigned int	nr_congestion_on	如果待处理请求数超出了该阈值，则认为该队列是拥挤的
unsigned int	nr-congestion-off	如果待处理请求数在这个阈值的范围内，则认为该队列是不拥挤的
unsigned int	nr_batching	即使队列已满，仍可以由特殊进程“batcher”提交的待处理请求的最大值（通常为32）
unsigned short	max_sectors	单个请求所能处理的最大扇区数（可调的）
unsigned short	max_hw_sectors	单个请求所能处理的最大扇区数（硬约束）
unsigned short	max_phys_segments	单个请求所能处理的最大物理段数
unsigned short	max_hw_segments	单个请求所能处理的最大硬段数（分散-聚集DMA操作中的最大不同内存区数）
unsigned short	hardsect_size	扇区中以字节为单位的大小
nsigned int	max_segment_size	物理段的最大长度（以字节为单位）
unsigned long	seg_boundary_mask	段合并的内存边界屏蔽字
unsigned int	dma_alignment	DMA缓冲区的起始地址和长度的对齐位图（缺省值是511）
struct	queue_tags	空闲/忙标记的位图（用于带标记的请求）blk_queue_tag *
atomic_t	refcnt	请求队列的引用计数器
unsigned int	in_flight	请求队列中待处理请求数
unsigned int	sg_timeout	用户定义的命令超时（仅由SCSI通用块设备使用）
unsigned int	sg_reserved_size	基本上没有使用
struct list_head	drain_list	临时延时的请求链表的首部，直到I/O调度程序被动态取代

实质上，请求队列是一个双向链表，其元素就是请求描述符（也就是request数据结构；参见下一节）。请求队列描述符中的queue_head字段存放链表的头（第一个伪元素），而请求描述符中queuelist字段的指针把任一请求链接到链表的前一个和后一个元素之间。

队列链表中元素的排序方式对每个块设备驱动程序是特定的，然而，I/O 调度程序提供了几种预先确定好的元素排序方式，这将在后面的“I/O 调度算法”一节中讨论。

backing_dev_info 字段是一个 backing_dev_info 类型的小对象，它存放了关于基本硬件块设备的 I/O 数据流量的信息。例如，它保存了关于预读以及关于请求队列拥塞状态的信息。

请求描述符

每个块设备的待处理请求都是用一个请求描述符来表示的，请求描述符存放在如表 14-7 所示的 request 数据结构中。

表 14-7：请求描述符的字段

类型	字段	说明
struct list_head	queuelist	请求队列链表的指针
unsigned long	flags	请求标志（参见下面）
sector_t	sector	要传送的下一个扇区号
unsigned long	nr_sectors	整个请求中要传送的扇区数
unsigned int	current_nr_sectors	当前 bio 的当前段中要传送的扇区数
sector_t	hard_sector	要传送的下一个扇区号
unsigned long	hard_nr_sectors	整个请求中要传送的扇区数（由通用块层更新）
unsigned int	hard_cur_sectors	当前 bio 的当前段中要传送的扇区数（由通用块层更新）
struct bio *	bio	请求中第一个没有完成传送操作的bio
struct bio *	biotail	请求链表中末尾的 bio
void *	elevator_private	指向 I/O 调度程序私有数据的指针
int	rq_status	请求状态：实际上，或者是 RQ_ACTIVE，或者是 RQ_INACTIVE
struct gendisk *	rq_disk	请求所引用的磁盘描述符
int	errors	用于记录当前传送中发生的 I/O 失败次数的计数器
unsigned long	start_time	请求的起始时间（用 jiffies 表示）
unsigned short	nr_phys_segments	请求的物理段数
unsigned short	nr_hw_segments	请求的硬段数

表 14-7：请求描述符的字段（续）

类型	字段	说明
int	tag	与请求相关的标记（只适合支持多次数据传送的硬件设备）
char *	buffer	指向当前数据传送的内存缓冲区的指针（如果缓冲区是高端内存区，则为 NULL）
int	ref_count	请求的引用计数器
request_queue_t *	q	指向包含请求的请求队列描述符的指针
struct request_list *	rl	指向 request_list 结构的指针
struct completion *	waiting	等待数据传送终止的Completion结构（参见第五章的“补充原语”一节）
void *	special	对硬件设备发出“特殊”命令的请求所使用的数据的指针
unsigned int	cmd_len	cmd 字段中命令的长度
unsigned char []	cmd	由请求队列的prep_rq_fn方法准备好的预先内置命令所在的缓冲区
unsigned int	data_len	通常，由 data 字段指向的缓冲区中数据的长度
void *	data	设备驱动程序为了跟踪所传送的数据而使用的指针
unsigned int	sense_len	由 sense 字段指向的缓冲区的长度（如果 sense 是 NULL，则为 0）
void *	sense	指向输出sense命令的缓冲区的指针
unsigned int	timeout	请求的超时
struct request_pm_state *	pm	指向电源管理命令所使用的数据结构

每个请求包含一个或多个 bio 结构。最初，通用块层创建一个仅包含一个 bio 结构的请求。然后，I/O 调度程序要么向初始的 bio 中增加一个新段，要么将另一个 bio 结构链接到请求中，从而“扩展”该请求。可能存在新数据与请求中已存在的数据物理相邻的情况。请求描述符的 bio 字段指向请求中的第一个 bio 结构，而 biotail 字段则指向最后一个 bio 结构。rq_for_each_bio 宏执行一个循环，从而遍历请求中的所有 bio 结构。

请求描述符中的几个字段值可能是动态变化的。例如，一旦 bio 中引用的数据块全部传

送完毕，bio字段立即更新从而指向请求链表中的下一个bio。在此期间，新的bio可能被加入到请求链表的尾部，所以biotail的值也可能改变。

当磁盘数据块正在传送时，请求描述符的其它几个字段的值由I/O调度程序或设备驱动程序修改。例如，nr_sectors存放整个请求还需传送的扇区数，current_nr_sectors存放当前bio结构中还需传送的扇区数。

flags中存放了很多标志，如表14-8中所示。到目前为止，最重要的一个标志是REQ_RW，它确定数据传送的方向。

表14-8：请求描述符的标志

标志	说明
REQ_RW	数据传送的方向：READ (0)或WRITE (1)
REQ_FAILFAST	万一出错请求申明不再重试I/O操作
REQ_SOFTBARRIER	请求相当于I/O调度程序的屏障
REQ_HARDBARRIER	请求相当于I/O调度程序和设备驱动程序的屏障——应当在旧请求与新请求之间处理该请求
REQ_CMD	包含一个标准的读或写I/O数据传送的请求
REQ_NOMERGE	不允许扩展或与其它请求合并的请求
REQ_STARTED	正处理的请求
REQ_DONTPREP	不调用请求队列中的prep_rq_fn方法预先准备把命令发送给硬件设备
REQ_QUEUED	请求被标记——也就是说，与该请求相关的硬件设备可以同时管理很多未完成数据的传送
REQ_PC	请求包含发送给硬件设备的直接命令
REQ_BLOCK_PC	与前一个标志功能相同，但发送的命令包含在bio结构中
REQ_SENSE	请求包含一个“sense”请求命令（SCSI和ATAPI设备使用）
REQ_FAILED	当请求中的sense或direct命令的操作与预期的不一致时设置该标志
REQ_QUIET	万一I/O操作出错请求申明不产生内核消息
REQ_SPECIAL	请求包含对硬件设备的特殊命令（例如，重设驱动器）
REQ_DRIVE_CMD	请求包含对IDE磁盘的特殊命令
REQ_DRIVE_TASK	请求包含对IDE磁盘的特殊命令
REQ_DRIVE_TASKFILE	请求包含对IDE磁盘的特殊命令
REQ_PREEMPT	请求取代位于请求队列前面的请求（仅对IDE磁盘而言）

表 14-8：请求描述符的标志（续）

标志	说明
REQ_PM_SUSPEND	请求包含一个挂起硬件设备的电源管理命令
REQ_PM_RESUME	请求包含一个唤醒硬件设备的电源管理命令
REQ_PM_SHUTDOWN	请求包含一个切断硬件设备的电源管理命令
REQ_BAR_PREFLUSH	请求包含一个要发送给磁盘控制器的“刷新队列”命令
REQ_BAR_POSTFLUSH	请求包含一个已发送给磁盘控制器的“刷新队列”命令

对请求描述符的分配进行管理

在重负载和磁盘操作频繁的情况下，固定数目的动态空闲内存将成为进程想要把新请求加入请求队列 q 的瓶颈。为了解决这种问题，每个 request_queue 描述符包含一个 request_list 数据结构，其中包括：

- 一个指针，指向请求描述符的内存池（参见第八章的“内存池”一节）。
- 两个计数器，分别用于记录分配给 READ 和 WRITE 请求的请求描述符数。
- 两个标志，分别用于标记为读或写请求的分配是否失败。
- 两个等待队列，分别存放了为获得空闲的读和写请求描述符而睡眠的进程。
- 一个等待队列，存放等待一个请求队列被刷新（清空）的进程。

blk_get_request()函数试图从一个特定请求队列的内存池中获得一个空闲的请求描述符；如果内存区不足并且内存池已经用完，则要么挂起当前进程，要么返回 NULL（如果不能阻塞内核控制路径）。如果分配成功，则将请求队列的request_list数据结构的地址存放在请求描述符的rl字段中。blk_put_request()函数则释放一个请求描述符；如果该描述符的引用计数器的值为0，则将描述符归还回它原来所在的内存池。

避免请求队列拥塞

每个请求队列都有一个允许处理的最大请求数。请求队列描述符的nr_requests字段存放了每个数据传送方向所允许处理的最大请求数。缺省情况下，一个队列至多有128个待处理读请求和128个待处理写请求。如果待处理的读（写）请求数超过了nr_requests值，那么通过设置请求队列描述符的 queue_flags 字段的 QUEUE_FLAG_READFULL（QUEUE_FLAG_WRITEFULL）标志将该队列标记为已满，试图把请求加入到某个传送方向的可阻塞进程被放置到 request_list 结构所对应的等待队列中睡眠。

一个填满的请求队列对系统性能有负面影响，因为它会强制许多进程去睡眠以等待I/O数据传送的完成。因此，如果给定传送方向上的待处理请求数超过了存放在请求描述符的nr_congestion_on字段中的值（缺省值为113），那么内核认为该队列是拥塞的，并试图降低新请求的创建速率。当待处理请求数小于nr_congestion_off的值（缺省值为111）时，拥塞的请求队列才变为不拥塞。blk_congestion_wait()函数挂起当前进程，直到所有请求队列都变为不拥塞或超时已到。

激活块设备驱动程序

正如我们在前面已经看到的一样，延迟激活块设备驱动程序有利于把相邻块的请求进行集中。这种延迟是通过所谓的设备插入和设备拔出技术（注2）来实现的。在块设备驱动程序被插入时，该驱动程序并不被激活，即使在驱动程序队列中有待处理的请求。

blk_plug_device()函数的功能是插入一个块设备——更准确地说，插入到某个块设备驱动程序处理的请求队列中。本质上，该函数接收一个请求队列描述符的地址q作为其参数。它设置q->queue_flags字段中的QUEUE_FLAG_PLUGGED位；然后，重新启动q->unplug_timer字段中的内嵌动态定时器。

blk_remove_plug()则拔去一个请求队列q：清除QUEUE_FLAG_PLUGGED标志并取消q->unplug_timer动态定时器的执行。当“视线中”所有可合并的请求都被加入请求队列时，内核就会显式地调用该函数。此外，如果请求队列中待处理的请求数超过了请求队列描述符的unplug_thresh字段中存放的值（缺省值为4），那么I/O调度程序也会去掉该请求队列。

如果一个设备保持插入的时间间隔为q->unplug_delay（通常为3ms），那么说明由blk_plug_device()函数激活的动态定时器的时间已用完，因此就会执行blk_unplug_timeout()函数。因而，唤醒内核线程*kblockd*所操作的工作队列kblockd_workqueue（参见第四章的“工作队列”一节）。*kblockd*执行blk-unplug_work()函数，其地址存放在q->unplug_work结构中。接着，该函数会调用请求队列中的q->unplug_fn方法，通常该方法是由generic_unplug_device()函数实现的。generic_unplug_device()函数的功能是拔出块设备：首先，检查请求队列是否仍然活跃；然后，调用blk_remove_plug()函数；最后，执行策略例程request_fn方法来开始处理请求队列中的下一个请求（参见本章后面的“注册和初始化设备驱动程序”一节）。

注2：如果“插入”和“拔出”这两个术语使你费解，你可以把它们分别理解为“激活”和“撤消”。

I/O 调度算法

当向请求队列增加一条新的请求时，通用块层会调用I/O调度程序来确定请求队列中新请求的确切位置。I/O调度程序试图通过扇区将请求队列排序。如果顺序地从链表中提取要处理的请求，那么就会明显减少磁头寻道的次数，因为磁头是按照直线的方式从内磁道移向外磁道(反之亦然)，而不是随意地从一个磁道跳跃到另一个磁道。这可以从电梯算法中得到启发，回想一下，电梯算法处理来自不同层的上下请求。电梯是往一个方向移动的；当朝一个方向上的最后一个预定层到达时，电梯就会改变方向而开始向相反的方向移动。因此，I/O调度程序也被称为电梯算法（*elevator*）。

在重负载情况下，严格遵循扇区号顺序的I/O调度算法运行的并不是很好。在这种情形下，数据传送的完成时间主要取决于磁盘上数据的物理位置。因此，如果设备驱动程序处理的请求位于队列的首部（小扇区号），并且拥有小扇区号的新请求不断被加入队列中，那么队列末尾的请求就很容易会饿死。因而I/O调度算法会非常复杂。

当前，Linux 2.6中提供了四种不同类型的I/O调度程序或电梯算法，分别为“预期(Anticipatory)”算法、“最后期限（Deadline)”算法、“CFQ (Complete Fairness Queueing，完全公平队列)”算法以及“Noop (No Operation)”算法。对大多数块设备而言，内核使用的缺省电梯算法可在引导时通过内核参数elevator=<name>进行再设置，其中<*name*>值可取下列任何一个：as、deadline、cfq和noop。如果没有给定引导参数，那么内核缺省使用“预期” I/O调度程序。总之，设备驱动程序可以用任何一个调度程序取代缺省的电梯算法；设备驱动程序也可以自己定制I/O调度算法，但是这种情况很少见。

此外，系统管理员可以在运行时为一个特定的块设备改变I/O调度程序。例如，为了改变第一个IDE通道的主磁盘所使用的I/O调度程序，管理员可把一个电梯算法的名称写入*sysfs*特殊文件系统的*/sys/block/hda/queue/scheduler*文件中（参见第十三章中的“sysfs文件系统”一节)。

请求队列中使用的I/O调度算法是由一个elevator_t类型的*elevator*对象表示的；该对象的地址存放在请求队列描述符的elevator字段中。elevator对象包含了几个方法，它们覆盖了elevator所有可能的操作：链接和断开elevator，增加和合并队列中的请求，从队列中删除请求，获得队列中下一个待处理的请求等等。elevator对象也存放了一个表的地址，表中包含了处理请求队列所需的所有信息。而且，每个请求描述符包含一个elevator_private字段，该字段指向一个由I/O调度程序用来处理请求的附加数据结构。

现在我们从易到难简要地介绍一下四种I/O调度算法。注意，设计一个I/O调度程序与设计一个CPU调度程序很相似（参见第七章)：启发算法和采用的常量值是测试和基准外延量的结果。

一般而言，所有的算法都使用一个调度队列（*dispatch queue*），队列中包含的所有请求按照设备驱动程序应当处理的顺序进行排序——也即设备驱动程序要处理的下一个请求通常是调度队列中的第一个元素。调度队列实际上是由请求队列描述符的queue_head字段所确定的请求队列。几乎所有的算法都使用另外的队列对请求进行分类和排序。它们允许设备驱动程序将bio结构增加到已存在请求中，如果需要，还要合并两个“相邻的”请求。

“Noop”算法

这是最简单的I/O调度算法。它没有排序的队列：新的请求通常被插在调度队列的开头或末尾，下一个要处理的请求总是队列中的第一个请求。

“CFQ”算法

“CFQ（完全公平队列）”算法的主要目标是在触发I/O请求的所有进程中确保磁盘I/O带宽的公平分配。为了达到这个目标，算法使用许多个排序队列——缺省为64——它们存放了不同进程发出的请求。当算法处理一个请求时，内核调用一个散列函数将当前进程的线程组标识符（通常，它对应其PID，参见第三章“标识一个进程”一节）转换为队列的索引值；然后，算法将一个新的请求插入该队列的末尾。因此，同一个进程发出的请求通常被插入相同的队列中。

为了再填充调度队列，算法本质上采用轮询方式扫描I/O输入队列，选择第一个非空队列，然后将该队列中的一组请求移动到调度队列的末尾。

“最后期限”算法

除了调度队列外，“最后期限”算法还使用了四个队列。其中的两个排序队列分别包含读请求和写请求，其中的请求是根据起始扇区数排序的。另外两个最后期限队列包含相同的读和写请求，但这是根据它们的“最后期限”排序的。引入这些队列是为了避免请求饿死，由于电梯策略优先处理与上一个所处理的请求最近的请求，因而就会对某个请求忽略很长一段时间，这时就会发生这种情况。请求的最后期限本质上就是一个超时定时器，当请求被传给电梯算法时开始计时。缺省情况下，读请求的超时时间是500ms，写请求的超时时间是5s——读请求优先于写请求，因为读请求通常阻塞发出请求的进程。最后期限保证了调度程序照顾等待很长一段时间的那个请求，即使它位于排序队列的末尾。

当算法要补充调度队列时，首先确定下一个请求的数据方向。如果同时要调度读和写两个请求，算法会选择“读”方向，除非该“写”方向已经被放弃很多次了（为了避免写请求饿死）。

接下来，算法检查与被选择方向相关的最后期限队列：如果队列中的第一个请求的最后期限已用完，那么算法将该请求移到调度队列的末尾；也可以从超时的那个请求开始移动来自排序队列的一组请求。如果将要移动的请求在磁盘上物理相邻，那么组的长度会变长，否则就变短。

最后，如果没有请求超时，算法对来自于排序队列的最后一个请求之后的一组请求进行调度。当指针到达排序队列的末尾时，搜索又从头开始（“单方向算法”）。

“预期”算法

“预期”算法是Linux提供的最复杂的一种I/O调度算法。基本上，它是“最后期限”算法的一个演变，借用了“最后期限”算法的基本机制：两个最后期限队列和两个排序队列；I/O调度程序在读和写请之间交互扫描排序队列，不过更倾向于读请求。扫描基本上是连续的，除非有某个请求超时。读请求的缺省超时时间是125ms，写请求的缺省超时时间是250ms。但是，该算法还遵循一些附加的启发式准则：

- 有些情况下，算法可能在排序队列当前位置之后选择一个请求，从而强制磁头从后搜索。这种情况通常发生在这个请求之后的搜索距离小于在排序队列当前位置之后对该请求搜索距离的一半时。
- 算法统计系统中每个进程触发的I/O操作的种类。当刚刚调度了由某个进程p发出的一个读请求之后，算法马上检查排序队列中的下一个请求是否来自同一个进程p。如果是，立即调度下一个请求。否则，查看关于该进程p的统计信息：如果确定进程p可能很快发出另一个读请求，那么就延迟一小段时间（缺省大约为7ms）。因此，算法预测进程p发出的读请求与刚被调度的请求在磁盘上可能是“近邻”。

向I/O调度程序发出请求

正如我们在本章前面的“提交请求”一节中所看到的，generic_make_request()函数调用请求队列描述符的make_request_fn方法向I/O调度程序发送一个请求。通常该方法是由__make_request()函数实现的；该函数接收一个request_queue类型的描述符q和一个bio结构的描述符bio作为其参数，然后执行如下操作：

1. 如果需要，调用blk_queue_bounce()函数建立一个回弹缓冲区（参见后面）。如果回弹缓冲区被建立，__make_request()函数将对该缓冲区而不是原先的bio结构进行操作。
2. 调用I/O调度程序的elv_queue_empty()函数检查请求队列中是否存在待处理请求——注意，调度队列可能是空的，但是I/O调度程序的其他队列可能包含待处

理请求。如果没有待处理请求，那么调用blk_plug_device()函数插入请求队列（参见本章前面的“激活块设备驱动程序”一节），然后跳转到第5步。

3. 插入的请求队列包含待处理请求。调用I/O调度程序的elv_merge()函数检查新的bio结构是否可以并入已存在的请求中。该函数将返回三个可能值：

   - ELEVATOR_NO_MERGE：已经存在的请求中不能包含bio结构；这种情况下，跳转到第5步。
   - ELEVATOR_BACK_MERGE：bio结构可作为末尾的bio而插入到某个请求req中；这种情形下，函数调用q->back_merge_fn方法检查是否可以扩展该请求。如果不行，则跳转到第5步。否则，将bio描述符插入req链表的末尾并更新req的相应字段值。然后，函数试图将该请求与其后面的请求合并（新的bio可能填充在两个请求之间）。
   - ELEVATOR_FRONT_MERGE：bio结构可作为某个请求req的第一个bio被插入；这种情形下，函数调用q->front_merge_fn方法检查是否可以扩展该请求。如果不行，则跳转到第5步。否则，将bio描述符插入req链表的首部并更新req的相应字段值。然后，试图将该请求与其前面的请求合并。

4. bio已经被并入存在的请求中，跳转到第7步终止函数。

5. bio必须被插入到一个新的请求中。分配一个新的请求描述符。如果没有空闲的内存，那么挂起当前进程，直到设置了bio->bi_rw中的BIO_RW_AHEAD标志，该标志表明这个I/O操作是一次预读（参见第十六章）；在这种情形下，函数调用bio_endio()并终止：此时将不会执行数据传送。对bio_endio()函数的描述参见generic_make_request()函数执行的第1步操作（参见前面的“提交请求”一节）。

6. 初始化请求描述符中的字段。主要有：

   a. 根据bio描述符的内容初始化各个字段，包括扇区数、当前bio以及当前段。
   b. 设置flags字段中的REQ_CMD标志（一个标准的读或写操作）。
   c. 如果第一个bio段的页框存放在低端内存，则将buffer字段设置为缓冲区的线性地址。
   d. 将rq_disk字段设置为bio->bi_bdev->bd_disk的地址。
   e. 将bio插入请求链表。
   f. 将start_time字段设置为jiffies的值。

7. 所有操作全部完成。但是，在终止之前，检查是否设置了bio->bi_rw中的

BIO_RW_SYNC标志。如果是，则对“请求队列”调用generic_unplug_device()函数以卸载设备驱动程序（参见本章前面的“激活块设备驱动程序”一节）。

8. 函数终止。

如果在调用__make_request()函数之前请求队列不是空的，那么说明该请求队列要么已经被拔掉过，要么很快将被拔掉——因为每个拥有待处理请求的插入请求队列q都有一个正在运行的动态定时器q->unplug_timer。另一方面，如果请求队列是空的，则__make_request()函数插入请求队列。或迟（最坏的情况是当拔出的定时器到期了）或早（从__make_request()中退出时，如果设置了bio的BIO_RW_SYNC标志），该请求队列都会被拔掉。任何情形下，块设备驱动程序的策略例程最后都将处理调度队列中的请求（参见本章后面的“注册和初始化设备驱动程序”一节）。

blk_queue_bounce()函数

blk_queue_bounce()函数的功能是查看q->bounce_gfp中的标志以及q->bounce_pfn中的阈值，从而确定回弹缓冲区（*buffer bouncing*）是否是必需的。通常当请求中的一些缓冲区位于高端内存而硬件设备不能访问它们时发生这种情况。

ISA总线使用的老式DMA方式只能处理24位的物理地址。因此，回弹缓冲区的上限设为16 MB，也就是说，页框号为4096。然而，当处理老式设备时，块设备驱动程序通常不依赖回弹缓冲区；相反，它们更倾向于直接在ZONE_DMA内存区中分配DMA缓冲区。

如果硬件设备不能处理高端内存中的缓冲区，则blk_queue_bounce()函数检查bio中的一些缓冲区是否真的必须是回弹的。如果是，则将bio描述符复制一份，接着创建一个回弹bio；然后，当段中的页框号等于或大于q->bounce_pfn的值时，执行下列操作：

1. 根据分配的标志，在ZONE_NORMAL或ZONE_DMA内存区中分配一个页框。
2. 更新回弹bio中段的bv_page字段的值，使其指向新页框的描述符。
3. 如果bio->bio_rw代表一个写操作，则调用kmap()临时将高端内存页映射到内核地址空间中，然后将高端内存页复制到低端内存页上，最后调用kunmap()释放该映射。

然后blk_queue_bounce()函数设置回弹bio中的BIO_BOUNCED标志，为其初始化一个特定的bi_end_io方法，最后将它存放在回弹bio的bi_private字段中，该字段是指向初始bio的指针。当在回弹bio上的I/O数据传送终止时，函数执行bi_end_io方法将数据复制到高端内存区中（仅适合读操作），并释放该回弹bio结构。

块设备驱动程序

块设备驱动程序是Linux块子系统中的最底层组件。它们从I/O调度程序中获得请求，然后按要求处理这些请求。

当然，块设备驱动程序是设备驱动程序模型的组成部分（参见第十三章中的“设备驱动程序模型”一节）。因此，每个块设备驱动程序对应一个device_driver类型的描述符；此外，设备驱动程序处理的每个磁盘都与一个device描述符相关联。但是，这些描述符没有什么特别的：块I/O子系统必须为系统中的每个块设备存放附加信息。

块设备

一个块设备驱动程序可能处理几个块设备。例如，IDE设备驱动程序可以处理几个IDE磁盘，其中的每个都是一个单独的块设备。而且，每个磁盘通常是被分区的，每个分区又可以被看作是一个逻辑块设备。很明显，块设备驱动程序必须处理在块设备对应的块设备文件上发出的所有VFS系统调用。

每个块设备都是由一个block_device结构的描述符来表示的，其字段如表14-9所示。

表14-9：块设备描述符中的字段

类型	字段	说明
dev_t	bd_dev	块设备的主设备号和次设备号
struct inode *	bd-inode	指向*bdev*文件系统中块设备对应的文件索引节点的指针
int	bd_openers	计数器，统计块设备已经被打开了多少次
struct semaphore	bd_sem	保护块设备的打开和关闭的信号量
struct semaphore	bd_mount_sem	禁止在块设备上进行新安装的信号量
struct list_head	bd_inodes	已打开的块设备文件的索引节点链表的首部
void *	bd_holder	块设备描述符的当前所有者
int	bd_holders	计数器，统计对bd_holder字段多次设置的次数
struct block_device *	bd_contains	如果块设备是一个分区，则指向整个磁盘的块设备描述符；否则，指向该块设备描述符
unsigned	bd_block_size	块大小
struct hd_struct *	bd_part	指向分区描述符的指针（如果该块设备不是一个分区，则为NULL）

表 14-9：块设备描述符中的字段（续）

类型	字段	说明
unsigned	bd_part_count	计数器，统计包含在块设备中的分区已经被打开了多少次
int	bd_invalidated	当需要读块设备的分区表时设置的标志
struct gendisk *	bd_disk	指向块设备中基本磁盘的gendisk结构的指针
struct list_head *	bd_list	用于块设备描述符链表的指针
struct backing_dev_info *	bd_inode_backing_dev_info	指向块设备的专门描述符backing_dev_info的指针（通常为NULL）
unsigned long	bd_private	指向块设备持有者的私有数据的指针

所有的块设备描述符被插入一个全局链表中，链表首部是由变量all_bdevs表示的，链表链接所用的指针位于块设备描述符的bd_list字段中。

如果块设备描述符对应一个磁盘分区，那么bd_contains字段指向与整个磁盘相关的块设备描述符，而bd_part字段指向hd_struct分区描述符（参见本章前面的“磁盘和磁盘分区的表示”一节）。否则，若块设备描述符对应整个磁盘，那么bd_contains字段指向块设备描述符本身，bd_part_count字段用于记录磁盘上的分区已经被打开了多少次。

bd_holder字段存放代表块设备持有者的线性地址。持有者并不是进行I/O数据传送的块设备驱动程序；准确地说，它是一个内核组件，使用设备并拥有独一无二的特权（例如，它可以自由使用块设备描述符的bd_private字段）。典型地，块设备的持有者是安装在该设备上的文件系统。当块设备文件被打开进行互斥访问时，另一个普遍的问题出现了：持有者就是对应的文件对象。

bd_claim()函数将bd_holder字段设置为一个特定的地址；相反，bd_release()函数将该字段重新设置为NULL。然而，值得注意的是，同一个内核组件可以多次调用bd_claim()函数，每调用一次都增加bd_holders的值。为了释放块设备，内核组件必须调用bd_release()函数bd_holders次。

图14-3对应的是一个整盘，它说明了块设备描述符是如何被链接到块I/O子系统的其他重要数据结构上的。

访问块设备

当内核接收一个打开块设备文件的请求时，必须首先确定该设备文件是否已经是打开的。事实上，如果文件已经是打开的，内核就没有必要创建并初始化一个新的块设备描述符；

相反，内核应该更新这个已经存在的块设备描述符。然而，真正的复杂性在于具有相同主设备号和次设备号但有不同路径名的块设备文件被VFS看作不同的文件，但是它们实际上指向同一个块设备。因此，内核无法通过简单地在一个对象的索引节点高速缓存中检查块设备文件的存在就确定相应的块设备已经在使用。

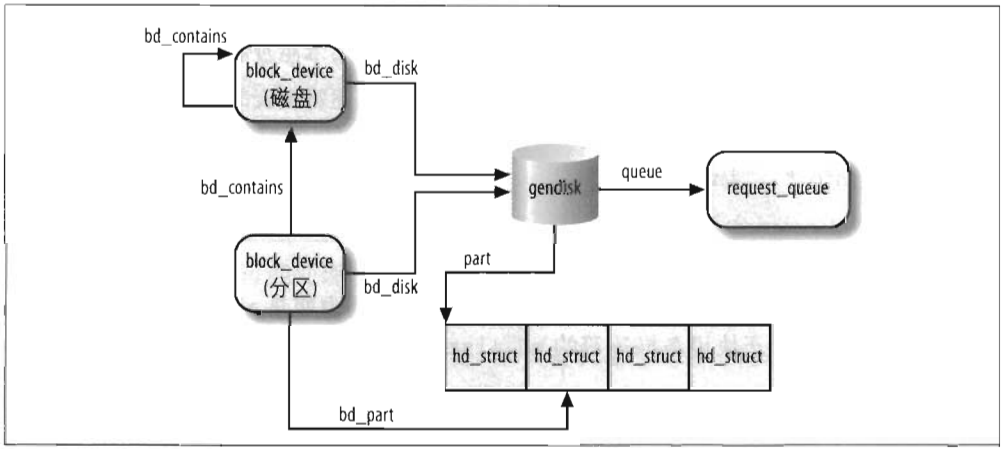


图 14-3：块设备描述符与块子系统其他结构的链接

主、次设备号和相应的块设备描述符之间的关系是通过*bdev*特殊文件系统（参见第十二章的“特殊文件系统”一节）来维护的。每个块设备描述符都对应一个*bdev*特殊文件：块设备描述符的bd_inode字段指向相应的*bdev*索引节点；而该索引节点则将块设备的主、次设备号和相应描述符的地址进行编码。

bdget()接收块设备的主设备号和次设备号作为其参数：在*bdev*文件系统中查寻相关的索引节点；如果不存在这样的节点，那么就分配一个新索引节点和新块设备描述符。在任何情形下，函数都返回一个与给定主、次设备号对应的块设备描述符的地址。

一旦找到了块设备的描述符，那么内核通过检查bd_openers字段的值来确定块设备当前是否在使用：如果值是正的，说明块设备已经在使用（可能通过不同的设备文件）。同时内核也维护一个与已打开的块设备文件对应的索引节点对象的链表。该链表存放在块设备描述符的bd_inodes字段中；索引节点对象的i_devices字段存放用于链接链表中的前后元素的指针。

注册和初始化设备驱动程序

现在我们来说明一下为一个块设备设计一个新的驱动程序所涉及的基本步骤。显然，其描述是比较简单的，但是理解何时并怎样初始化块I/O子系统使用的主要数据结构是很有用的。

我们省略了所有块设备驱动程序需要的但在第十三章中已经讲过的步骤。例如，我们跳过了注册一个驱动程序本身的所有步骤（参见第十三章中的“设备驱动程序模型“一节）。通常，块设备属于一个诸如PCI或SCSI这样的标准总线体系结构，内核提供了相应的辅助函数，作为一个辅助作用，就是在驱动程序模型中注册驱动程序。

自定义驱动程序描述符

首先，设备驱动程序需要一个*foo*_dev_t类型的自定义描述符*foo*，它拥有驱动硬件设备所需的数据。该描述符存放每个设备的相关信息，例如操作设备时使用的I/O端口、设备发出中断的IRQ线、设备的内部状态等等。同时它也包含块I/O子系统所需的一些字段：

```
struct foo_dev_t {
    [...]
    spinlock_t lock;
    struct gendisk *gd;
    [...]
} foo;
```

lock字段是用来保护*foo*描述符中字段值的自旋锁；通常将其地址传给内核辅助函数，从而保护对驱动程序而言特定的块I/O子系统的数据结构。gd字段是指向gendisk描述符的指针，该描述符描述由这个驱动程序处理的整个块设备（磁盘）。

预订主设备号

设备驱动程序必须自己预订一个主设备号。传统上，该操作通过调用register_blkdev()函数完成：

```
err = register_blkdev(FOO_MAJOR, "foo");
if (err) goto error_major_is_busy;
```

该函数类似于第十三章的“分配设备号”一节中出现的register_chrdev()函数：预订主设备号*FOO*_MAJOR并将设备名称*foo*赋给它。注意，这不能分配次设备号范围，因为没有类似的register_chrdev_region()函数；此外，预订的主设备号和驱动程序的数据结构之间也没有建立链接。register_blkdev()函数产生的唯一可见的效果是包含一个新条目，该条目位于*/proc/devices*特殊文件的已注册主设备号列表中。

初始化自定义描述符

在使用驱动程序之前必须适当地初始化*foo*描述符中的所有字段。为了初始化与块I/O子系统相关的字段，设备驱动程序主要执行如下操作：

```
spin_lock_init(&foo.lock);
```