

```

        "value %lx from %p\n", timeout,
        __builtin_return_address(0));
    current->state = TASK_RUNNING;
    goto out;
}

}

expire = timeout + jiffies;

init_timer(&timer);
timer.expires = expire;
timer.data = (unsigned long) current;
timer.function = process_timeout;

add_timer(&timer);
schedule();
del_timer_sync(&timer);

timeout = expire - jiffies;

out:
    return timeout < 0 ? 0 : timeout;
}

```

该函数用原始的名字 `timer` 创建了一个定时器 `timer`；然后设置它的超时时间 `timeout`；设置超时执行函数 `process_timeout()`；接着激活定时器而且调用 `schedule()`。因为任务被标识为 `TASK_INTERRUPTIBLE` 或 `TASK_UNINTERRUPTIBLE`，所以调度程序不会再选择该任务投入运行，而会选择其他新任务运行。

当定时器超时时，`process_timeout()` 函数会被调用：

```

void process_timeout(unsigned long data)
{
    wake_up_process((task_t *)data);
}

```

该函数将任务设置为 `TASK_RUNNING` 状态，然后将其放入运行队列。

当任务重新被调度时，将返回代码进入睡眠前的位置继续执行（正好在调用 `schedule()` 后）。如果任务提前被唤醒（比如收到信号），那么定时器被撤销，`process_timeout()` 函数返回剩余的时间。

在 `switch()` 括号中的代码是为处理特殊情况而写的，正常情况不会用到它们。`MAX_SCHEDULE_TIMEOUT` 是用来检查任务是否无限期地睡眠，如果那样的话，函数不会为它设置定时器（因为睡眠时间没有期限），而这时调度程序会立刻被调用。如果你需要无限期地让任务睡眠，最好使用其他方法唤醒任务。

2. 设置超时时间，在等待队列上睡眠

第 4 章我们已经看到进程上下文中的代码为了等待特定事件发生，可以将自己放入等待队

列，然后调用调度程序去执行新任务。一旦事件发生后，内核调用 `wake_up()` 函数唤醒在睡眠队列上的任务，使其重新投入运行。

有时，等待队列上的某个任务可能既在等待一个特定事件到来，又在等待一个特定时间到期——就看谁来得更快。这种情况下，代码可以简单地使用 `schedule_timeout()` 函数代替 `schedule()` 函数，这样一来，当希望的指定时间到期，任务都会被唤醒。当然，代码需要检查被唤醒的原因（有可能是被事件唤醒，也有可能是因为延迟的时间到期，还可能是因为接收到了信号），然后执行相应的操作。

11.9 小结

在本章中，我们考察了时间的概念，并知道了墙上时钟与计算机的正常运行时间如何管理。我们对比了相对时间和绝对时间以及绝对事件与周期事件。我们还涵盖了诸如时钟中断、时钟节拍、HZ 以及 jiffies 等概念。

我们考察了定时器的实现，了解了如何把这些用到自己的内核代码中。本章最后，我们浏览了开发者用于延迟的其他方法。

你写的大多数内核代码都需要对时间及其走过的时间有一些理解。而最大的可能是，只要你编写驱动程序，就需要处理内核定时器。与其让时间悄悄溜走，还不如阅读本章。



第 12 章

内存管理

在内核里分配内存可不像在其他地方分配内存那么容易。造成这种局面的因素很多。从根本上讲，是因为内核本身不能像用户空间那样奢侈地使用内存。内核与用户空间不同，它不具备这种能力，它不支持简单便捷的内存分配方式。比如，内核一般不能睡眠。此外，处理内存分配错误对内核来说也绝非易事。正是由于这些限制，再加上内存分配机制不能太复杂，所以在内核中获取内存要比在用户空间复杂得多。不过，从程序开发者角度来看，也不是说内核的内存分配就困难得不得了，只是和用户空间中的内存分配不太一样而已。

本章讨论的是在内核之中获取内存的方法。在深入研究实际的分配接口之前，我们需要理解内核是如何管理内存的。

12.1 页

内核把物理页作为内存管理的基本单位。尽管处理器的最小可寻址单位通常为字（甚至字节），但是，内存管理单元（MMU，管理内存并把虚拟地址转换为物理地址的硬件）通常以页为单位进行处理。正因为如此，MMU 以页（page）大小为单位来管理系统中的页表（这也是页表名的来由）。从虚拟内存的角度来看，页就是最小单位。

在第 19 章中我们将会看到，体系结构不同，支持的页大小也不尽相同，还有些体系结构甚至支持几种不同的页大小。大多数 32 位体系结构支持 4KB 的页，而 64 位体系结构一般会支持 8KB 的页。这就意味着，在支持 4KB 页大小并有 1GB 物理内存的机器上，物理内存会被划分为 262144 个页。

内核用 `struct page` 结构表示系统中的每个物理页，该结构位于 `<linux/mm_types.h` 中——我简化了定义，去除了两个容易混淆我们讨论主题的联合结构体：

```
struct page {
    unsigned long    flags;
    atomic_t         _count;
    atomic_t         _mapcount;
    unsigned long    private;
    struct address_space *mapping;
    pgoff_t          index;
    struct list_head lru;
    void             *virtual;
};
```

让我们看一下其中比较重要的域。`flag` 域用来存放页的状态。这些状态包括页是不是脏的，

是不是被锁定在内存中等。flag 的每一位单独表示一种状态，所以它至少可以同时表示出 32 种不同的状态。这些标志定义在 `<linux/page-flags.h>` 中。

`_count` 域存放页的引用计数——也就是这一页被引用了多少次。当计数值变为 -1 时，就说明当前内核并没有引用这一页，于是，在新的分配中就可以使用它。内核代码不应当直接检查该域，而是调用 `page_count()` 函数进行检查，该函数唯一的参数就是 `page` 结构。当页空闲时，尽管该结构内部的 `_count` 值是负的，但是对 `page_count()` 函数而言，返回 0 表示页空闲，返回一个正整数表示页在使用。一个页可以由页缓存使用（这时，`mapping` 域指向和这个页关联的 `addresss_space` 对象），或者作为私有数据（由 `private` 指向），或者作为进程页表中的映射。

`virtual` 域是页的虚拟地址。通常情况下，它就是页在虚拟内存中的地址。有些内存（即所谓的高端内存）并不永久地映射到内核地址空间上。在这种情况下，这个域的值 `NULL`，需要的时候，必须动态地映射这些页。稍后我们将讨论高端内存。

必须要理解的一点是 `page` 结构与物理页相关，而并非与虚拟页相关。因此，该结构对页的描述只是短暂的。即使页中所包含的数据继续存在，由于交换等原因，它们也可能不再和同一个 `page` 结构相关联。内核仅仅用这个数据结构来描述当前时刻在相关的物理页中存放的东西。这种数据结构的目的在于描述物理内存本身，而不是描述包含在其中的数据。

内核用这一结构来管理系统中所有的页，因为内核需要知道一个页是否空闲（也就是页有没有被分配）。如果页已经被分配，内核还需要知道谁拥有这个页。拥有者可能是用户空间进程、动态分配的内核数据、静态内核代码或页高速缓存等。

系统中的每个物理页都要分配一个这样的结构体，开发者常常对此感到惊讶。他们会想“这得浪费多少内存呀”！让我们来算算对所有这些页都这么做，到底要消耗掉多少内存。就算 `struct page` 占 40 字节的内存吧，假定系统的物理页为 8KB 大小，系统有 4GB 物理内存。那么，系统中共有页面 524 288 个，而描述这么多页面的 `page` 结构体消耗的内存只不过是 20MB；也许绝对值不小，但是相对系统 4GB 内存而言，仅是很小的一部分罢了。因此，要管理系统中这么多物理页面，这个代价并不算太高。

12.2 区

由于硬件的限制，内核并不能对所有的页一视同仁。有些页位于内存中特定的物理地址上，所以不能将其用于一些特定的任务。由于存在这种限制，所以内核把页划分为不同的区（`zone`）。内核使用区对具有相似特性的页进行分组。Linux 必须处理如下两种由于硬件存在缺陷而引起的内存寻址问题：

- 一些硬件只能用某些特定的内存地址来执行 DMA（直接内存访问）。
- 一些体系结构的内存的物理寻址范围比虚拟寻址范围大得多。这样，就有一些内存不能永久地映射到内核空间上。

因为存在这些制约条件，Linux 主要使用了四种区：

- `ZONE_DMA`——这个区包含的页能用来执行 DMA 操作。
- `ZONE_DMA32`——和 `ZONE_DMA` 类似，该区包含的页面可用来执行 DMA 操作；而和 `ZONE_DMA` 不同之处在于，这些页面只能被 32 位设备访问。在某些体系结构中，该区将

比 ZONE_DMA 更大。

- ZONE_NORMAL——这个区包含的都是能正常映射的页。
- ZONE_HIGHMEM——这个区包含“高端内存”，其中的页并不能永久地映射到内核地址空间。这些区（还有两种不大重要的）在 <linux/mmzone.h> 中定义。

区的实际使用和分布是与体系结构相关的。例如，某些体系结构在内存的任何地址上执行 DMA 都没有问题。在这些体系结构中，ZONE_DMA 为空，ZONE_NORMAL 就可以直接用于分配。与此相反，在 x86 体系结构上，ISA 设备就不能在整个 32 位^①的地址空间中执行 DMA，因为 ISA 设备只能访问物理内存的前 16MB。因此，ZONE_DMA 在 x86 上包含的页都在 0-16MB 的内存范围里。

ZONE_HIGHMEM 的工作方式也差不多。能否直接映射取决于体系结构。在 32 位 x86 系统上，ZONE_HIGHMEM 为高于 896MB 的所有物理内存。在其他体系结构上，由于所有内存都被直接映射，所以 ZONE_HIGHMEM 为空。ZONE_HIGHMEM 所在的内存就是所谓的高端内存^②（high memory）。系统的其余内存就是所谓的低端内存（low memory）。

前两个区各取所需之后，剩余的就由 ZONE_NORMAL 区独享了。在 x86 上，ZONE_NORMAL 是从 16MB 到 896MB 的所有物理内存。在其他（更幸运）的体系结构上，ZONE_NORMAL 是所有的可用物理内存。表 12-1 是每个区及其在 x86-32 上所占页的列表。

表 12-1 x86-32 上的区

区	描 述	物 理 内 存
ZONE_DMA	DMA 使用的页	<16MB
ZONE_NORMAL	正常可寻址的页	16 ~ 896MB
ZONE_HIGHMEM	动态映射的页	>896MB

Linux 把系统的页划分为区，形成不同的内存池，这样就可以根据用途进行分配了。例如，ZONE_DMA 内存池让内核有能力为 DMA 分配所需的内存。如果需要这样的内存，那么，内核就可以从 ZONE_DMA 中按照请求的数目取出页。注意，区的划分没有任何物理意义，这只不过是内核为了管理页而采取的一种逻辑上的分组。

某些分配可能需要从特定的区中获取页，而另外一些分配则可以从多个区中获取页。比如，尽管用于 DMA 的内存必须从 ZONE_DMA 中进行分配，但是一般用途的内存却既能从 ZONE_DMA 分配，也能从 ZONE_NORMAL 分配，不过不可能同时从两个区分配，因为分配是不能跨区界限的。当然，内核更希望一般用途的内存从常规区分配，这样能节省 ZONE_DMA 中的页，保证满足 DMA 的使用需求。但是，如果可供分配的资源不够用了（如果内存已经变得很少了），那么，内核就会去占用其他可用区的内存。

不是所有的体系结构都定义了全部区，有些 64 位的体系结构，如 Intel 的 x86-64 体系结构

① 有些糟糕的 PCI 设备只能在 24 位地址空间内执行 DMA 操作。

② Linux 的高端内存和 DOS 的高端内存没有关系，DOS 的高端内存是围绕 DOS 和 x86 的“实模式”的空间范围限制而言的。

可以映射和处理 64 位的内存空间，所以 x86-64 没有 ZONE_HIGHMEM 区，所有的物理内存都处于 ZONE_DMA 和 ZONE_NORMAL 区。

每个区都用 struct zone 表示，在 <linux/mmzone.h> 中定义：

```
struct zone {
    unsigned long          watermark[NR_WMARK];
    unsigned long          lowmem_reserve[MAX_NR_ZONES];
    struct per_cpu_pageset pageset[NR_CPUS];
    spinlock_t             lock;
    struct free_area        free_area[MAX_ORDER]
    spinlock_t             lru_lock;
    struct zone_lru {
        struct list_head list;
        unsigned long      nr_saved_scan;
    } lru[NR_LRU_LISTS];
    struct zone_reclaim_stat reclaim_stat;
    unsigned long          pages_scanned;
    unsigned long          flags;
    atomic_long_t          vm_stat[NR_VM_ZONE_STAT_ITEMS];
    int                    prev_priority;
    unsigned               int inactive_ratio;
    wait_queue_head_t      *wait_table;
    unsigned long          wait_table_hash_nr_entries;
    unsigned long          wait_table_bits;
    struct pglist_data      *zone_pgdat;
    unsigned long          zone_start_pfn;
    unsigned long          spanned_pages;
    unsigned long          present_pages;
    const char             *name;
};
```

这个结构体很大，但是，系统中只有三个区，因此，也只有三个这样的结构。让我们看一下其中一些重要的域。

lock 域是一个自旋锁，它防止该结构被并发访问。注意，这个域只保护结构，而不保护驻留在这个区中的所有页。没有特定的锁来保护单个页，但是，部分内核可以锁住在页中驻留的数据。

watermark 数组持有该区的最小值、最低和最高水位值。内核使用水位为每个内存区设置合适的内存消耗基准。该水位随空闲内存的多少而变化。

name 域是一个以 NULL 结束的字符串表示这个区的名字。内核启动期间初始化这个值，其代码位于 mm/page_alloc.c 中。三个区的名字分别为“DMA”、“Normal”和“HighMem”。

12.3 获得页

我们已经对内核如何管理内存（页、区等）有所了解了，现在让我们看一下内核实现的接口，我们正是通过这些接口在内核内分配和释放内存的。

内核提供了一种请求内存的底层机制，并提供了对它进行访问的几个接口。所有这些接口都

以页为单位分配内存，定义于 <linux/gfp.h> 中。最核心的函数是：

```
struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
```

该函数分配 2^{order} ($1 \leq \text{order}$) 个连续的物理页，并返回一个指针，该指针指向第一个页的 page 结构体；如果出错，就返回 NULL。在 12.4 节我们再研究 gfp_t 类型和 gfp_mask 参数。你可以用下面这个函数把给定的页转换成它的逻辑地址：

```
void * page_address(struct page *page)
```

该函数返回一个指针，指向给定物理页当前所在的逻辑地址。如果你无须用到 struct page，你可以调用：

```
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
```

这个函数与 alloc_pages() 作用相同，不过它直接返回所请求的第一个页的逻辑地址。因为页是连续的，所以其他页也会紧随其后。

如果你只需一页，就可以用下面两个封装好的函数，它能让你少敲几下键盘：

```
struct page * alloc_page(gfp_t gfp_mask)
unsigned long __get_free_page(gfp_t gfp_mask)
```

这两个函数与其兄弟函数工作方式相同，只不过传递给 order 的值为 0 ($2^0 = 1$ 页)。

12.3.1 获得填充为 0 的页

如果你需要让返回的页的内容全为 0，请用下面这个函数：

```
unsigned long get_zeroed_page(unsigned int gfp_mask)
```

这个函数与 __get_free_pages() 工作方式相同，只不过把分配好的页都填充成了 0——字节中的每一位都要取消设置。如果分配的页是给用户空间的，这个函数就非常有用了。虽说分配好的页中应该包含的都是随机产生的垃圾信息，但其实这些信息可能并不是完全随机的——它很可能“随机地”包含某些敏感数据。用户空间的页在返回之前，所有数据必须填充为 0，或做其他清理工作，在保障系统安全这一点上，我们决不妥协。表 12-2 是所有底层的页分配方法的列表。

表 12-2 低级页分配方法

标 志	描 述
alloc_page(gfp_mask)	只分配一页，返回指向页结构的指针
alloc_pages(gfp_mask, order)	分配 2^{order} 个页，返回指向第一页页结构的指针
__get_free_page(gfp_mask)	只分配一页，返回指向其逻辑地址的指针
__get_free_pages(gfp_mask, order)	分配 2^{order} 页，返回指向第一页逻辑地址的指针
get_zeroed_page(gfp_mask)	只分配一页，让其内容填充 0，返回指向其逻辑地址的指针

12.3.2 释放页

当你不再需要页时可以用下面的函数释放它们：

```
void __free_pages(struct page *page, unsigned int order)
void free_pages(unsigned long addr, unsigned int order)
void free_page(unsigned long addr)
```

释放页时要谨慎，只能释放属于你的页。传递了错误的 `struct page` 或地址，用了错误的 `order` 值，这些都可能系统崩溃。请记住，内核是完全信赖自己的。这点与用户空间不同，如果你有非法操作，内核会开开心心地把自己挂起来，停止运行。

让我们看一个例子。其中，我们想得到 8 个页：

```
unsigned long page;

page = __get_free_pages(GFP_KERNEL, 3);
if (!page){
    /* 没有足够的内存：你必须处理这种错误！ */
    return -ENOMEM;
}
/* “page” 现在指向 8 个连续页中第 1 个页的地址 ... */
```

在此，我们使用完这 8 个页之后释放它们：

```
free_pages(page, 3);

/*
 * 页现在已经被释放了，我们不应该再访问
 * 存放在 “page” 中的地址了
 */
```

`GFP_KERNEL` 参数是 `gfp_mask` 标志的一个例子。前面我们已经简要讨论过。

调用 `_get_free_pages()` 之后要注意进行错误检查。内核分配可能失败，因此你的代码必须进行检查并做相应的处理。这意味在此之前，你所做的所有工作可能前功尽弃，甚至还需要回归到原来的状态。正因为如此，在程序开始时就先进行内存分配是很有意义的，这能让错误处理得容易一点。如果你不这么做，那么在你想要分配内存的时候如果失败了，局面可能就难以控制了。

当你需要以页为单位的一族连续物理页时，尤其是在你只需要一两页时，这些低级页函数很有用。对于常用的以字节为单位的分配来说，内核提供的函数是 `kmalloc()`。

12.4 kmalloc()

`kmalloc()` 函数与用户空间的 `malloc()` 一族函数非常类似，只不过它多了一个 `flags` 参数。`kmalloc()` 函数是一个简单的接口，用它可以获得以字节为单位的一块内核内存。如果你需要整个页，那么，前面讨论的页分配接口可能是更好的选择。但是，对于大多数内核分配来说，`kmalloc()` 接口用得更多。

`kmalloc()` 在 `<linux/slab.h>` 中声明：

```
void * kmalloc(size_t size, gfp_t flags)
```

这个函数返回一个指向内存块的指针，其内存块至少要有 size 大小。所分配的内存区在物理上是连续的。在出错时，它返回 NULL。除非没有足够的内存可用，否则内核总能分配成功。在对 kmalloc() 调用之后，你必须检查返回的是不是 NULL，如果是，要适当地处理错误。

让我们看一个例子。我们随便假定存在一个 dog 结构体，现在需要为它动态地分配足够的空间：

```
struct dog *p;

p = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!p)
    /* 处理错误 ... */
```

如果 kmalloc() 调用成功，那么，ptr 现在指向一个内存块，内存块的大小至少为所请求的大小。GFP_KERNEL 标志表示在试图获取内存并返回给 kmalloc() 的调用者的过程中，内存分配器将要采取的行为。

12.4.1 gfp_mask 标志

我们已经看过了几个例子，发现不管是在低级页分配函数中，还是在 kmalloc() 中，都用到了分配器标志。现在，我们就深入讨论一下这些标志。

这些标志可分为三类：行为修饰符、区修饰符及类型。行为修饰符表示内核应当如何分配所需的内存。在某些特定情况下，只能使用某些特定的方法分配内存。例如，中断处理程序就要求内核在分配内存的过程中不能睡眠（因为中断处理程序不能被重新调度）。区修饰符表示从哪儿分配内存。前面我们已经看到，内核把物理内存分为多个区，每个区用于不同的目的。区修饰符指明到底从这些区中的哪一区中进行分配。类型标志组合了行为修饰符和区修饰符，将各种可能用到的组合归纳为不同类型，简化了修饰符的使用；这样，你只需指定一个类型标志就可以了。GFP_KERNEL 就是一种类型标志，内核中进程上下文相关的代码可以使用它。我们来看一下这些标志。

1. 行为修饰符

所有这些标志，包括行为描述符都是在 <linux/gfp.h> 中声明的。不过，在 <linux/slab.h> 中包含有这个头文件，因此，你一般不必直接包含引用它。实际上，一般只使用类型修饰符就够了，我们随后会看到这点。因此，最好对每个标志都有所了解。表 12-3 是行为修饰符的列表。

表 12-3 行为修饰符

标 志	描 述
__GFP_WAIT	分配器可以睡眠
__GFP_HIGH	分配器可以访问紧急事件缓冲池
__GFP_IO	分配器可以启动磁盘 I/O
__GFP_FS	分配器可以启动文件系统 I/O
__GFP_COLD	分配器应该使用高速缓存中快要淘汰出去的页
__GFP_NOWARN	分配器将不打印失败警告
__GFP_REPEAT	分配器在分配失败时重复进行分配，但是这次分配还存在失败的可能
__GFP_NOFAIL	分配器将无限地重复进行分配。分配不能失败

(续)

标 志	描 述
<code>__GFP_NORETRY</code>	分配器在分配失败时绝不会重新分配
<code>__GFP_NO_GROW</code>	由 slab 层内部使用
<code>__GFP_COMP</code>	添加混合页元数据，在 hugetlb 的代码内部使用

可以同时指定这些分配标志。例如：

```
ptr = kmalloc(size, __GFP_WAIT | __GFP_IO | __GFP_FS);
```

说明页分配器（最终调用 `alloc_pages()`）在分配时可以阻塞、执行 I/O，在必要时还可以执行文件系统操作。这就让内核有很大的自由度，以便它尽可能找到空闲的内存来满足分配请求。

大多数分配都会指定这些修饰符，但一般不是这样直接指定，而是采用我们随后讨论的类型标志。别担心，你不会在分配内存时为怎样使用这些标志而犯愁的！

2. 区修饰符

区修饰符表示内存区应当从何处分配。通常，分配可以从任何区开始。不过，内核优先从 `ZONE_NORMAL` 开始，这样可以确保其他区在需要时有足够的空闲页可供使用。

实际上只有两个区修饰符，因为除了 `ZONE_NORMAL` 之外只有两个区（默认都是从 `ZONE_NORMAL` 区进行分配）。表 12-4 是区修饰符的列表。

表 12-4 区修饰符

标 志	描 述
<code>__GFP_DMA</code>	从 <code>ZONE_DMA</code> 分配
<code>__GFP_DMA32</code>	只在 <code>ZONE_DMA32</code> 分配
<code>__GFP_HIGHMEM</code>	从 <code>ZONE_HIGHMEM</code> 或 <code>ZONE_NORMAL</code> 分配

指定以上标志中的一个就可以改变内核试图进行分配的区。`__GFP_DMA` 标志强制内核从 `ZONE_DMA` 分配。这个标志在说，有了这种奇怪的标识，我绝对可以拥有进行 DMA 的内存。相反，如果指定 `__GFP_HIGHMEM` 标志，则从 `ZONE_HIGHMEM`（优先）或 `ZONE_NORMAL` 分配。这个标志在说，我可以使用高端内存，因此，我可以是一个玩偶，给你退还一些内存，但是，常规内存还照常工作。如果没有指定任何标志，则内核从 `ZONE_DMA` 或 `ZONE_NORMAL` 进行分配，当然优先从 `ZONE_NORMAL` 进行分配。不管区标志说什么了，只要它行为正常，我就不关心了。

不能给 `_get_free_pages()` 或 `kalloc()` 指定 `ZONE_HIGHMEM`，因为这两个函数返回的都是逻辑地址，而不是 page 结构，这两个函数分配的内存当前有可能还没有映射到内核的虚拟地址空间，因此，也可能根本就没有逻辑地址。只有 `alloc_pages()` 才能分配高端内存。实际上，你的分配在大多数情况下都不必指定修饰符，`ZONE_NORMAL` 就足矣。

3. 类型标志

类型标志指定所需的行为和区描述符以完成特殊类型的处理。正因为这一点，内核代码趋向

于使用正确的类型标志，而不是一味地指定它可能需要用到的多个描述符。这么做既简单又不容易出错误。表 12-5 是类型标志的列表，而表 12-6 显示了每个类型标志与哪些修饰符相关联。

表 12-5 类型标志

标 志	描 述
GFP_ATOMIC	这个标志用在中断处理程序、下半部、持有自旋锁以及其他不能睡眠的地方
GFP_NOWAIT	与 GFP_ATOMIC 类似，不同之处在于，调用不会退给紧急内存池。这就增加了内存分配失败的可能性。
GFP_NOIO	这种分配可以阻塞，但不会启动磁盘 I/O。这个标志在不能引发更多磁盘 I/O 时能阻塞 I/O 代码，这可能导致令人不愉快的递归
GFP_NOFS	这种分配在必要时可能阻塞，也可能启动磁盘 I/O，但是不会启动文件系统操作。这个标志在你不能再启动另一个文件系统的操作时，用在文件系统部分的代码中
GFP_KERNEL	这是一种常规分配方式，可能会阻塞。这个标志在睡眠安全时用在进程上下文代码中。为了获得调用者所需的内存，内核会尽力而为。这个标志应当是首选标志
GFP_USER	这是一种常规分配方式，可能会阻塞。这个标志用于为用户空间进程分配内存时
GFP_HIGHUSER	这是从 ZONE_HIGHMEM 进行分配，可能会阻塞。这个标志用于为用户空间进程分配内存
GFP_DMA	这是从 ZONE_DMA 进行分配。需要获取能供 DMA 使用的内存的设备驱动程序使用这个标志，通常与以上的某个标志组合在一起使用

表 12-6 在每种类型标志后隐含的修饰符列表

标 志	修饰符标志
GFP_ATOMIC	__GFP_HIGH
GFP_NOWAIT	0
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	__GFP_DMA

让我们看一下最常用的标志以及你什么时候、为什么需要使用它们。内核中最常用的标志是 GFP_KERNEL。这种分配可能会引起睡眠，它使用的是普通优先级。因为调用可能阻塞，因此这个标志只用在可以重新安全调度的进程上下文中（也就是没有锁被持有等情况）。因为这个标志对内核如何获取请求的内存没有任何约束，所以内存分配成功的可能性很高。

另一个截然相反的标志是 GFP_ATOMIC。因为这个标志表示不能睡眠的内存分配，因此想要满足调用者获取内存的请求将会受到很严格的限制。即使没有足够的连续内存块可供使用，内核也很可能无法释放出可用内存来，因为内核不能让调用者睡眠。相反，GFP_KERNEL 分配可以让调用者睡眠、交换、刷新一些页到硬盘等。因为 GFP_ATOMIC 不能执行以上任何操作，因此与 GFP_KERNEL 相比较，它分配成功的机会较小（尤其在内存短缺时）。即便如此，在当前代码（例如中断处理程序、软中断和 tasklet）不能睡眠时，也只能选择 GFP_ATOMIC。

在以上两种标志中间的是 GFP_NOIO 和 GFP_NOFS。以这两个标志进行的分配可能会引起阻塞，但它们会避免执行某些其他操作。GFP_NOIO 分配绝不会启动任何磁盘 I/O 来帮助满足

请求。而 GFP_NOFS 可能会启动磁盘 I/O，但是它不会启动文件系统 I/O。你为什么需要这些标志？它们分别用在某些低级块 I/O 或文件系统的代码中。设想，如果文件系统代码中需要分配内存，但没有使用 GFP_NOFS。这种分配可能会引起更多的文件系统操作，而这些操作又会导致另外的分配，从而再引起更多的文件系统操作！这会一直持续下去。这样的代码在调用分配器的时候，必须确保分配器不会再执行到代码本身，否则，分配就可能产生死锁。也别紧张，内核使用这两个标志的地方是极少的。

GFP_DMA 标志表示分配器必须满足从 ZONE_DMA 进行分配的请求。这个标志用在需要 DMA 的内存的设备驱动程序中。一般你会把这个标志与 GFP_ATOMIC 和 GFP_KERNEL 结合起来使用。

在你编写的绝大多数代码中，用到的要么是 GFP_KERNEL，要么是 GFP_ATOMIC。表 12-7 是通常情形和所用标志的列表。不管使用哪种分配类型，你都必须进行检查，并对错误进行处理。

表 12-7 什么时候用哪种标志

情 形	相应标志
进程上下文，可以睡眠	使用 GFP_KERNEL
进程上下文，不可以睡眠	使用 GFP_ATOMIC，在你睡眠之前或之后以 GFP_KERNEL 执行内存分配
中断处理程序	使用 GFP_ATOMIC
软中断	使用 GFP_ATOMIC
tasklet	使用 GFP_ATOMIC
需要用于 DMA 的内存，可以睡眠	使用 (GFP_DMA GFP_KERNEL)
需要用于 DMA 的内存，不可以睡眠	使用 (GFP_DMA GFP_ATOMIC)，或在你睡眠之前执行内存分配

12.4.2 kfree()

kmalloc() 的另一端就是 kfree()，kfree() 声明于 <linux/slab.h> 中：

```
void kfree(const void *ptr)
```

kfree() 函数释放由 kmalloc() 分配出来的内存块。如果想要释放的内存不是由 kmalloc() 分配的，或者想要释放的内存早就被释放了，比如说释放属于内核其他部分的内存，调用这个函数就会导致严重的后果。与用户空间类似，分配和回收要注意配对使用，以避免内存泄漏和其他 bug。注意，调用 kfree(NULL) 是安全的。

让我们看一个在中断处理程序中分配内存的例子。在这个例子中，中断处理程序想分配一个缓冲区来保存输入数据。BUF_SIZE 预定义为以字节为单位的缓冲区长度，它应该是大于两个字节的。

```
char *buf;

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

之后，当我们不再需要这个内存时，别忘了释放它：

```
kfree(buf);
```

12.5 vmalloc()

`vmalloc()` 函数的工作方式类似于 `kmalloc()`，只不过前者分配的内存虚拟地址是连续的，而物理地址则无须连续。这也是用户空间分配函数的工作方式：由 `malloc()` 返回的页在进程的虚拟地址空间内是连续的，但是，这并不保证它们在物理 RAM 中也是连续的。`kmalloc()` 函数确保页在物理地址上是连续的（虚拟地址自然也是连续的）。`vmalloc()` 函数只确保页在虚拟地址空间内是连续的。它通过分配非连续的物理内存块，再“修正”页表，把内存映射到逻辑地址空间的连续区域中，就能做到这点。

大多数情况下，只有硬件设备需要得到物理地址连续的内存。在很多体系结构上，硬件设备存在于内存管理单元以外，它根本不理解什么是虚拟地址。因此，硬件设备用到的任何内存区都必须是物理上连续的块，而不仅仅是虚拟地址连续上的块。而仅供软件使用的内存块（例如与进程相关的缓冲区）就可以使用只有虚拟地址连续的内存块。但在你的编程中，根本察觉不到这种差异。对内核而言，所有内存看起来都是逻辑上连续的。

尽管在某些情况下才需要物理上连续的内存块，但是，很多内核代码都用 `kmalloc()` 来获得内存，而不是 `vmalloc()`。这主要是出于性能的考虑。`vmalloc()` 函数为了把物理上不连续的页转换为虚拟地址空间上连续的页，必须专门建立页表项。糟糕的是，通过 `vmalloc()` 获得的页必须一个一个地进行映射（因为它们物理上是不连续的），这就会导致比直接内存映射大得多的 TLB[⊖] 抖动。因为这些原因，`vmalloc()` 仅在不得已时才会使用——典型的就是为了获得大块内存。时，例如，当模块被动态插入到内核中时，就把模块装载到由 `vmalloc()` 分配的内存上。

`vmalloc()` 函数声明在 `<linux/vmalloc.h>` 中，定义在 `<mm/vmalloc.c>` 中。用法与用户空间的 `malloc()` 相同：

```
void * vmalloc(unsigned long size)
```

该函数返回一个指针，指向逻辑上连续的一块内存区，其大小至少为 `size`。在发生错误时，函数返回 `NULL`。函数可能睡眠，因此，不能从中断上下文中进行调用，也不能从其他不允许阻塞的情况下进行调用。

要释放通过 `vmalloc()` 所获得的内存，使用下面的函数：

```
void vfree(const void *addr)
```

这个函数会释放从 `addr` 开始的内存块，其中 `addr` 是以前由 `vmalloc()` 分配的内存块的地址。这个函数也可以睡眠，因此，不能从中断上下文中调用。它没有返回值。

这个函数用起来比较简单：

⊖ TLB (translation lookaside buffer) 是一种硬缓冲区，很多体系结构用它来缓存虚拟地址到物理地址的映射关系。它极大地提高了系统的性能，因为大多数内存都要进行虚拟寻址。

```
char *buf;

buf = vmalloc(16 * PAGE_SIZE); /* get 16 pages */
if (!buf)
    /* 错误！不能分配内存 */
/*
 * buf 现在指向虚拟地址连续的一块内存区，其大小至少为 16*PAGE_SIZE
 */
```

在分配内存之后，一定要释放它：

```
vfree(buf);
```

12.6 slab 层

分配和释放数据结构是所有内核中最普遍的操作之一。为了便于数据的频繁分配和回收，编程人员常常会用到空闲链表。空闲链表包含可供使用的、已经分配好的数据结构块。当代码需要一个新的数据结构实例时，就可以从空闲链表中抓取一个，而不需要分配内存，再把数据放进去。以后，当不再需要这个数据结构的实例时，就把它放回空闲链表，而不是释放它。从这个意义上说，空闲链表相当于对象高速缓存——快速存储频繁使用的对象类型。

在内核中，空闲链表面临的主要问题之一是不能全局控制。当可用内存变得紧缺时，内核无法通知每个空闲链表，让其收缩缓存的大小以便释放出一些内存来。实际上，内核根本就不知道存在任何空闲链表。为了弥补这一缺陷，也为了使代码更加稳固，Linux 内核提供了 slab 层（也就是所谓的 slab 分配器）。slab 分配器扮演了通用数据结构缓存层的角色。

slab 分配器的概念首先在 Sun 公司的 SunOS 5.4 操作系统中得以实现^①。Linux 数据结构缓存层具有同样的名字和基本设计思想。

slab 分配器试图在几个基本原则之间寻求一种平衡：

- 频繁使用的数据结构也会频繁分配和释放，因此应当缓存它们。
- 频繁分配和回收必然会导致内存碎片（难以找到大块连续的可用内存）。为了避免这种现象，空闲链表的缓存会连续地存放。因为已释放的数据结构又会放回空闲链表，因此不会导致碎片。
- 回收的对象可以立即投入下一次分配，因此，对于频繁的分配和释放，空闲链表能够提高其性能。
- 如果分配器知道对象大小、页大小和总的高速缓存的大小这样的概念，它会做出更明智的决策。
- 如果让部分缓存专属于单个处理器（对系统上的每个处理器独立而唯一），那么，分配和释放就可以在不加 SMP 锁的情况下进行。
- 如果分配器是与 NUMA 相关的，它就可以从相同的内存节点为请求者进行分配。
- 对存放的对象进行着色（color），以防止多个对象映射到相同的高速缓存行（cache line）。

Linux 的 slab 层在设计和实现时充分考虑了上述原则。

^① 参看 J. Bonwick 所著的《The Slab Allocator : An Object-Caching Kernel Memory Allocator》，USENIX，1994。

12.6.1 slab 层的设计

slab 层把不同的对象划分为所谓高速缓存组，其中每个高速缓存组都存放不同类型的对象。每种对象类型对应一个高速缓存。例如，一个高速缓存用于存放进程描述符（task_struct 结构的一个空闲链表），而另一个高速缓存存放索引节点对象（struct inode）。有趣的是，kmalloc() 接口建立在 slab 层之上，使用了一组通用高速缓存。

然后，这些高速缓存又被划分为 slab（这也是这个子系统名字的来由）。slab 由一个或多个物理上连续的页组成。一般情况下，slab 也就仅仅由一页组成。每个高速缓存可以由多个 slab 组成。

每个 slab 都包含一些对象成员，这里的对象指的是被缓存的数据结构。每个 slab 处于三种状态之一：满、部分满或空。一个满的 slab 没有空闲的对象（slab 中的所有对象都已被分配）。一个空的 slab 没有分配出任何对象（slab 中的所有对象都是空闲的）。一个部分满的 slab 有一些对象已分配出去，有些对象还空闲着。当内核的某一部分需要一个新的对象时，先从部分满的 slab 中进行分配。如果没有部分满的 slab，就从空的 slab 中进行分配。如果没有空的 slab，就要创建一个 slab 了。显然，满的 slab 无法满足请求，因为它根本就没有空闲的对象。这种策略能减少碎片。

作为一个例子，让我们考察一下 inode 结构，该结构是磁盘索引节点在内存中的体现（参见第 13 章）。这些数据结构会频繁地创建和释放，因此，用 slab 分配器来管理它们就很有必要。因而 struct inode 就由 inode_cachep 高速缓存（这是一种标准的命名规范）进行分配。这种高速缓存由一个或多个 slab 组成——由多个 slab 组成的可能性大一些，因为这样的对象数量很大。每个 slab 包含尽可能多的 struct inode 对象。当内核请求分配一个新的 inode 结构时，内核就从部分满的 slab 或空的 slab（如果没有部分满的 slab）返回一个指向已分配但未使用的结构的指针。当内核用完 inode 对象后，slab 分配器就把该对象标记为空闲。图 12-1 显示了高速缓存、slab 及对象之间的关系。

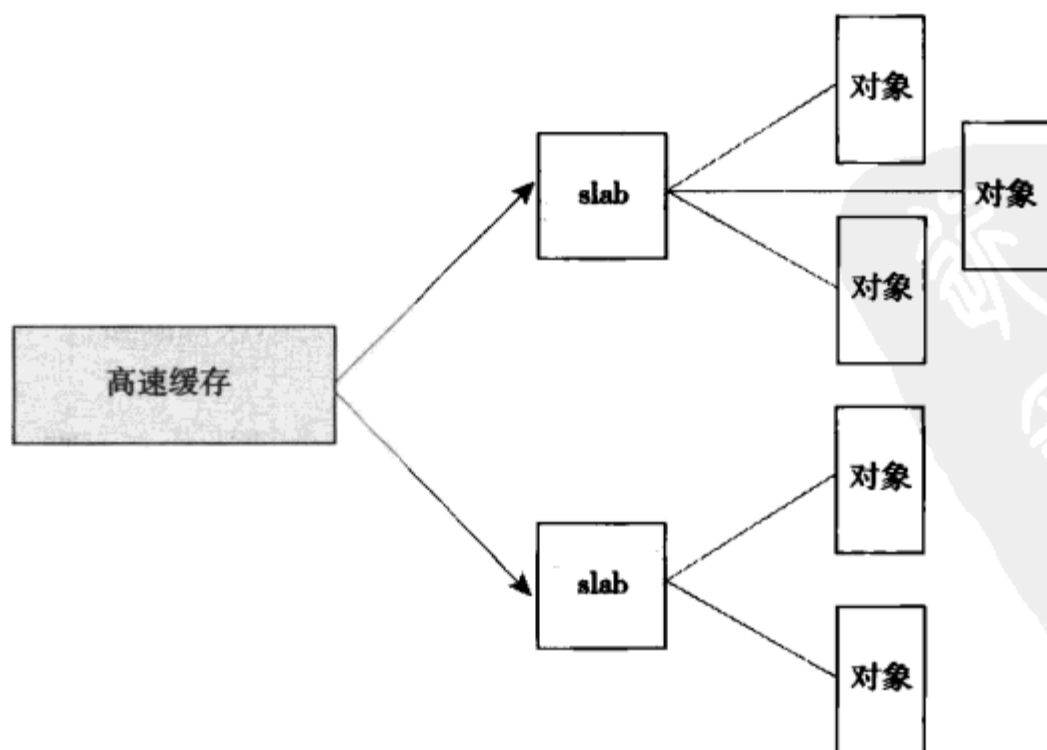


图 12-1 高速缓存、slab 及对象之间的关系

每个高速缓存都使用 `kmem_cache` 结构来表示。这个结构包含三个链表：`slabs_full`、`slabs_partial` 和 `slabs_empty`，均存放在 `kmem_list3` 结构内，该结构在 `mm/slab.c` 中定义。这些链表包含高速缓存中的所有 slab。slab 描述符 `struct slab` 用来描述每个 slab：

```
struct slab {
    struct list_head list;           /* 满、部分满或空链表 */
    unsigned long colouroff;        /* slab 着色的偏移量 */
    void *s_mem;                    /* 在 slab 中的第一个对象 */
    unsigned int inuse;              /* slab 中已分配的对象数 */
    kmem_bufctl_t free;             /* 第一个空闲对象（如果有的话） */
};
```

slab 描述符要么在 slab 之外另行分配，要么就放在 slab 自身开始的地方。如果 slab 很小，或者 slab 内部有足够的空间容纳 slab 描述符，那么描述符就存放在 slab 里面。

slab 分配器可以创建新的 slab，这是通过 `__get_free_pages()` 低级内核页分配器进行的：

```
static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags, int nodeid)
{
    struct page *page;
    void *addr;
    int i;

    flags |= cachep->gfpflags;
    if (likely(nodeid == -1)) {
        addr = (void *)__get_free_pages(flags, cachep->gfporder);
        if (!addr)
            return NULL;
        page = virt_to_page(addr);
    } else {
        page = alloc_pages_node(nodeid, flags, cachep->gfporder);
        if (!page)
            return NULL;
        addr = page_address(page);
    }

    i = (1 << cachep->gfporder);
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        atomic_add(i, &slab_reclaim_pages);
    add_page_state(nr_slab, i);
    while (i--) {
        SetPageSlab(page);
        page++;
    }
    return addr;
}
```

该函数使用 `__get_free_pages()` 来为高速缓存分配足够多的内存。该函数的第一个参数就指向需要很多页的特定高速缓存。第二个参数是要传给 `__get_free_pages()` 的标志，注意这个标志是如何与另一个值进行二进制“或”运算的，这相当于把高速缓存需要的缺省标志加到 `flags` 参数上。

分配的页大小为2的幂次方，存放在 `cachep->gfporder` 中。由于与分配器 NUMA 相关的代码的关系前面这个函数比想象的要复杂一些。当 `nodeid` 是一个非负数时，分配器就试图对从相同的内存节点给发出的请求进行分配。这在 NUMA 系统上提供了较好的性能，但是访问节点之外的内存会导致性能的损失。

为了便于理解，我们可以忽略与 NUMA 相关的代码，写一个简单的 `kmem_getpages()` 函数：

```
static inline void * kmem_getpages(struct kmem_cache *cachep, gfp_t flags)
{
    void *addr;

    flags |= cachep->gfpflags;
    addr = (void*) __get_free_pages(flags, cachep->gfporder);

    return addr;
}
```

接着，调用 `kmem_freepages()` 释放内存，而对给定的高速缓存页，`kmem_freepages()` 最终调用的是 `free_pages()`。当然，slab 层的关键就是避免频繁分配和释放页。由此可知，slab 层只有当给定的高速缓存部分中既没有满也没有空的 slab 时才会调用页分配函数。而只有在下列情况下才会调用释放函数：当可用内存变得紧缺时，系统试图释放出更多内存以供使用；或者当高速缓存显式地被撤销时。

slab 层的管理是在每个高速缓存的基础上，通过提供给整个内核一个简单的接口来完成的。通过接口就可以创建和撤销新的高速缓存，并在高速缓存内分配和释放对象。高速缓存及其内 slab 的复杂管理完全通过 slab 层的内部机制来处理。当你创建了一个高速缓存后，slab 层所起的作用就像一个专用的分配器，可以为具体的对象类型进行分配。

12.6.2 slab 分配器的接口

一个新的高速缓存通过以下函数创建：

```
struct kmem_cache * kmem_cache_create(const char *name,
                                     size_t size,
                                     size_t align,
                                     unsigned long flags,
                                     void (*ctor)(void *));
```

第一个参数是一个字符串，存放着高速缓存的名字；第二个参数是高速缓存中每个元素的大小；第三个参数是 slab 内第一个对象的偏移，它用来确保在页内进行特定的对齐。通常情况下，0 就可以满足要求，也就是标准对齐。flags 参数是可选的设置项，用来控制高速缓存的行为。它可以为 0，表示没有特殊的行为，或者与以下标志中的一个或多个进行“或”运算：

- **SLAB_HWCACHE_ALIGN**——这个标志命令 slab 层把一个 slab 内的所有对象按高速缓存行对齐。这就防止了“错误的共享”（两个或多个对象尽管位于不同的内存地址，但映射到相同的高速缓存行）。这可以提高性能，但以增加内存开销为代价，因为对齐越严格，

浪费的内存就越多。到底会耗费掉多少内存，取决于对象的大小以及对象相对于系统高速缓存行对齐的方式。对于会频繁使用的高速缓存，而且代码本身对性能要求又很严格的情况，设置该选项是理想的选择；否则，请三思而后行。

- **SLAB_POISON**——这个标志使 slab 层用已知的值 (a5a5a5a5) 填充 slab。这就是所谓的“中毒”，有利于对未初始化内存的访问。
- **SLAB_RED_ZONE**——这个标志导致 slab 层在已分配的内存周围插入“红色警戒区”以探测缓冲越界。
- **SLAB_PANIC**——这个标志当分配失败时提醒 slab 层。这在要求分配只能成功的时候非常有用。比如，在系统初启时分配一个 VMA 结构的高速缓存（参见第 15 章）。
- **SLAB_CACHE_DMA**——这个标志命令 slab 层使用可以执行 DMA 的内存给每个 slab 分配空间。只有在分配的对象用于 DMA，而且必须驻留在 ZONE_DMA 区时才需要这个标志。否则，你既不需要也不应该设置这个标志。

最后一个参数 `ctor` 是高速缓存的构造函数。只有在新的页追加到高速缓存时，构造函数才被调用。实际上，Linux 内核的高速缓存不使用构造函数。事实上这里曾经还有一个析构函数参数，但是由于内核代码并不需要它，因此已经被抛弃了。你可以将 `ctor` 参数赋值为 `NULL`。

`kmem_cache_create()` 在成功时会返回一个指向所创建高速缓存的指针；否则，返回 `NULL`。这个函数不能在中断上下文中调用，因为它可能会睡眠。

要撤销一个高速缓存，则调用：

```
int kmem_cache_destroy(struct kmem_cache *cachep)
```

顾名思义，这样就可以撤销给定的高速缓存。这个函数通常在模块的注销代码中被调用，当然，这里指创建了自己的高速缓存的模块。同样，也不能从中断上下文中调用这个函数，因为它也可能睡眠。调用该函数之前必须确保存在以下两个条件：

- 高速缓存中的所有 slab 都必须为空。其实，不管哪个 slab 中，只要还有一个对象被分配出去并正在使用的话，那怎么可能撤销这个高速缓存呢？
- 在调用 `kmem_cache_destroy()` 过程中（更不用说在调用之后了）不再访问这个高速缓存。调用者必须确保这种同步。

该函数在成功时返回 0，否则返回非 0 值。

1. 从缓存中分配

创建高速缓存之后，就可以通过下列函数获取对象：

```
void * kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
```

该函数从给定的高速缓存 `cachep` 中返回一个指向对象的指针。如果高速缓存的所有 slab 中都没有空闲的对象，那么 slab 层必须通过 `kmem_getpages()` 获取新的页，`flags` 的值传递给 `_get_free_pages()`。这与我们前面看到的标志相同，你用到的应该是 `GFP_KERNEL` 或 `GFP_ATOMIC`。

最后释放一个对象，并把它返回给原先的 slab，可以使用下面这个函数：

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp)
```

这样就能把 cachep 中的对象 objp 标记为空闲。

2. slab 分配器的使用实例

让我们考察一个鲜活的实例，这个例子用的是 task_struct 结构（进程描述符）。代码稍微有点复杂，取自 kernel/fork.c。

首先，内核用一个全局变量存放指向 task_struct 高速缓存的指针：

```
struct kmem_cache *task_struct_cachep;
```

在内核初始化期间，在定义于 kernel/fork.c 的 fork_init() 中会创建高速缓存：

```
task_struct_cachep = kmem_cache_create("task_struct",
                                       sizeof(struct task_struct),
                                       ARCH_MIN_TASKALIGN,
                                       SLAB_PANIC | SLAB_NOTRACK,
                                       NULL);
```

这样就创建了一个名为 task_struct 的高速缓存，其中存放的就是类型为 struct task_struct 的对象。该对象被创建后存放在 slab 中偏移量为 ARCH_MIN_TASKALIGN 个字节的地方，ARCH_MIN_TASKALIGN 预定义值与体系结构相关。通常将它定义为 L1_CACHE_BYTES——L1 高速缓存的字节大小。没有构造函数或析构函数。注意不用检查返回值是否为失败标记 NULL，因为 SLAB_PANIC 标志已经被设置了。如果分配失败，slab 分配器就调用 panic() 函数。如果没有提供 SLAB_PANIC 标志，就必须自己检查返回值。SLAB_PANIC 标志用在这儿是因为这是系统操作必不可少的高速缓存（没有进程描述符，机器自然不能正常运行）。

每当进程调用 fork() 时，一定会创建一个新的进程描述符（回忆一下第 3 章）。这是在 dup_task_struct() 中完成的，而该函数会被 do_fork() 调用：

```
struct task_struct *tsk;

tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
if (!tsk)
    return NULL;
```

进程执行完后，如果没有子进程在等待的话，它的进程描述符就会被释放，并返回给 task_struct_cachep slab 高速缓存。这是在 free_task_struct() 中执行的（这里，tsk 是现有的进程）：

```
kmem_cache_free(task_struct_cachep, tsk);
```

由于进程描述符是内核的核心组成部分，时刻都要用到，因此 task_struct_cachep 高速缓存绝不会被撤销掉。即使真能撤销，我们也要通过下列函数阻止其被撤销：

```
int err;

err = kmem_cache_destroy(task_struct_cachep);
if (err)
```

```
/* 出错，撤销高速缓存 */
```

很容易吧？slab 层负责内存紧缺情况下所有底层的对齐、着色、分配、释放和回收等。如果你要频繁创建很多相同类型的对象，那么，就应该考虑使用 slab 高速缓存。也就是说，不要自己去实现空闲链表！

12.7 在栈上的静态分配

在用户空间，我们以前所讨论到的那些分配的例子，有不少都可以在栈上发生。因为我们毕竟可以事先知道所分配空间的大小。用户空间能够奢侈地负担起非常大的栈，而且栈空间还可以动态增长，相反，内核却不能这么奢侈——内核栈小而且固定。当给每个进程分配一个固定大小的小栈后，不但可以减少内存的消耗，而且内核也无须负担太重的栈管理任务。

每个进程的内核栈大小既依赖体系结构，也与编译时的选项有关。历史上，每个进程都有两页的内核栈。因为 32 位和 64 位体系结构的页面大小分别是 4KB 和 8KB，所以通常它们的内核栈的大小分别是 8KB 和 16KB。

12.7.1 单页内核栈

但是，在 2.6 系列内核的早期，引入了一个选项设置单页内核栈。当激活这个选项时，每个进程的内核栈只有一页那么大，根据体系结构的不同，或为 4KB，或为 8KB。这么做出于两个原因：首先，可以让每个进程减少内存消耗。其次，也是最重要的，随着机器运行时间的增加，寻找两个未分配的、连续的页变得越来越困难。物理内存渐渐变为碎片，因此，给一个新进程分配虚拟内存（VM）的压力也在增大。

还有一个更复杂的原因。继续跟随我：我们几乎掌握了关于内核栈的全部知识。现在，每个进程的整个调用链必须放在自己的内核栈中。不过，中断处理程序也曾经使用它们所中断的进程的内核栈，这样，中断处理程序也要放在内核栈中。这当然有效而简单，但是，这同时会把更严格的约束条件加在这可怜的内核栈上。当我们转而使用只有一个页面的内核栈时，中断处理程序就不放在栈中了。

为了矫正这个问题，内核开发者们实现了一个新功能：中断栈。中断栈为每个进程提供一个用于中断处理程序的栈。有了这个选项，中断处理程序不用再和被中断进程共享一个内核栈，它们可以使用自己的栈了。对每个进程来说仅仅耗费了一页而已。

总的来说，内核栈可以是 1 页，也可以是 2 页，这取决于编译时配置选项。栈大小因此在 4 ~ 16KB 的范围内。历史上，中断处理程序和被中断进程共享一个栈。当 1 页栈的选项激活时，中断处理程序获得了自己的栈。在任何情况下，无限制的递归和 `alloca()` 显然是不被允许的。

好，就讲到这里。大家明白了吗？

12.7.2 在栈上光明正大地工作

在任意一个函数中，你都必须尽量节省栈资源。这并不难，也没有什么窍门，只需要在具体的函数中让所有局部变量（即所谓的自动变量）所占空间之和不要超过几百字节。在栈上进行大

量的静态分配（比如分配大型数组或大型结构体）是很危险的。要不然，在内核中和在用户空间中进行的栈分配就没有什么差别了。栈溢出时悄无声息，但势必会引起严重的问题。因为内核没有管理内核栈上做足工作，因此，当栈溢出时，多出的数据就会直接溢出来，覆盖掉紧邻堆栈末端的東西。首先面临考验的就是 `thread_info` 结构（回想一下第 3 章，这个结构就贴着每个进程内核堆栈的末端）。在堆栈之外，任何内核数据都可能存在潜在的危险。当栈溢出时，最好的情况是机器宕机，最坏的情况是悄无声息地破坏数据。

因此，进行动态分配是一种明智的选择，本章前面有关大块内存的分配就是采用这种方式。

12.8 高端内存的映射

根据定义，在高端内存中的页不能永久地映射到内核地址空间上。因此，通过 `alloc_pages()` 函数以 `__GFP_HIGHMEM` 标志获得的页不可能有逻辑地址。

在 x86 体系结构上，高于 896MB 的所有物理内存的范围大都是高端内存，它并不会永久地或自动地映射到内核地址空间，尽管 x86 处理器能够寻址物理 RAM 的范围达到 4GB（启用 PAE[⊖]可以寻址到 64GB）。一旦这些页被分配，就必须映射到内核的逻辑地址空间上。在 x86 上，高端内存中的页被映射到 3GB ~ 4GB。

12.8.1 永久映射

要映射一个给定的 `page` 结构到内核地址空间，可以使用定义在文件 `<linux/highmem.h>` 中的这个函数：

```
void *kmap(struct page *page)
```

这个函数在高端内存或低端内存上都能用。如果 `page` 结构对应的是低端内存中的一页，函数只会单纯地返回该页的虚拟地址。如果页位于高端内存，则会建立一个永久映射，再返回地址。这个函数可以睡眠，因此 `kmap()` 只能用在进程上下文中。

因为允许永久映射的数量是有限的（如果没有这个限制，我们就不必搞得这么复杂，把所有内存通通映射为永久内存就行了），当不再需要高端内存时，应该解除映射，这可以通过下列函数完成：

```
void kunmap(struct page *page)
```

12.8.2 临时映射

当必须创建一个映射而当前的上下文又不能睡眠时，内核提供了临时映射（也就是所谓的原子映射）。有一组保留的映射，它们可以存放新创建的临时映射。内核可以原子地把高端内存中的一个页映射到某个保留的映射中。因此，临时映射可以用在不能睡眠的地方，比如中断处理程序中，因为获取映射时绝不会阻塞。

⊖ PAE 是 Physical Address Extension 的缩写，这是 x86 处理器的特点，这种特点使得 x86 处理器尽管只有 32 位的虚拟地址空间，但从物理上能寻址到 36 位（64GB）的内存空间。

通过下列函数建立一个临时映射：

```
void *kmap_atomic(struct page *page, enum km_type type)
```

参数 `type` 是下列枚举类型之一，这些枚举类型描述了临时映射的目的。它们定义于 `<asm/kmap_types.h>` 中：

```
enum km_type {
    KM_BOUNCE_READ,
    KM_SKB_SUNRPC_DATA,
    KM_SKB_DATA_SOFTIRQ,
    KM_USER0,
    KM_USER1,
    KM_BIO_SRC_IRQ,
    KM_BIO_DST_IRQ,
    KM_PTE0,
    KM_PTE1,
    KM_PTE2,
    KM_IRQ0,
    KM_IRQ1,
    KM_SOFTIRQ0,
    KM_SOFTIRQ1,
    KM_SYNC_ICACHE,
    KM_SYNC_DCACHE,
    KM_UML_USERCOPY,
    KM_IRQ_PTE,
    KM_NMI,
    KM_NMI_PTE,
    KM_TYPE_NR
};
```

这个函数不会阻塞，因此可以用在中断上下文和其他不能重新调度的地方。它也禁止内核抢占，这是有必要的，因为映射对每个处理器都是唯一的（调度可能对哪个处理器执行哪个进程做变动）。

通过下列函数取消映射：

```
void kunmap_atomic(void *kvaddr, enum km_type type)
```

这个函数也不会阻塞。在很多体系结构中，除非激活了内核抢占，否则 `kmap_atomic()` 根本就无事可做，因为只有在下一个临时映射到来前上一个临时映射才有效。因此，内核完全可以“忘掉”`kmap_atomic()` 映射，`kunmap_atomic()` 也无须做什么实际的事情。下一个原子映射将自动覆盖前一个映射。

12.9 每个 CPU 的分配

支持 SMP 的现代操作系统使用每个 CPU 上的数据，对于给定的处理器其数据是唯一的。一般来说，每个 CPU 的数据存放在一个数组中。数组中的每一项对应着系统上一个存在的处理器。按当前处理器号确定这个数组的当前元素，这就是 2.4 内核处理每个 CPU 数据的方式。这种方

式还不错，因此，2.6 内核的很多代码依然用它。可以声明数据如下：

```
unsigned long my_percpu[NR_CPUS];
```

然后，按如下方式访问它：

```
int cpu;
```

```
cpu = get_cpu();           /* 获得当前处理器，并禁止内核抢占 */
my_percpu[cpu]++;          /* ... 或者无论什么 */
printk("my_percpu on cpu=%d is %lu\n", cpu, my_percpu[cpu]);
put_cpu();                 /* 激活内核抢占 */
```

注意，上面的代码中并没有出现锁，这是因为所操作的数据对当前处理器来说是唯一的。除了当前处理器之外，没有其他处理器可接触到这个数据，不存在并发访问问题，所以当前处理器可以在不用锁的情况下安全访问它。

现在，内核抢占成为了唯一需要关注的问题了，内核抢占会引起下面提到的两个问题：

- 如果你的代码被其他处理器抢占并重新调度，那么这时 CPU 变量就会无效，因为它指向的是错误的处理器（通常，代码获得当前处理器后是不可以睡眠的）。
- 如果另一个任务抢占了你的代码，那么有可能在同一个处理器上发生并发访问 `my_percpu` 的情况，显然这属于一个竞争条件。

虽然如此，但是你大可不必惊慌，因为在获取当前处理器号，即调用 `get_cpu()` 时，就已经禁止了内核抢占。相应的在调用 `put_cpu()` 时又会重新激活当前处理器号。注意，只要你总使用上述方法来保护数据安全，那么，内核抢占就不需要你自己去禁止。

12.10 新的每个 CPU 接口

2.6 内核为了方便创建和操作每个 CPU 数据，而引进了新的操作接口，称作 `percpu`。该接口归纳了前面所述的操作行为，简化了创建和操作每个 CPU 的数据。

但前面我们讨论的创建和访问每个 CPU 的方法依然有效，不过大型对称多处理器计算机要求对每个 CPU 数据操作更简单，功能更强大，正是在这种背景下，新接口应运而生。

头文件 `<linux/percpu.h>` 声明了所有的接口操作例程，你可以在文件 `mm/slab.c` 和 `<asm/percpu.h>` 中找到它们的定义。

12.10.1 编译时的每个 CPU 数据

在编译时定义每个 CPU 变量易如反掌：

```
DEFINE_PER_CPU(type, name);
```

这个语句为系统中的每一个处理器都创建了一个类型为 `type`，名字为 `name` 的变量实例，如果你需要在别处声明变量，以防范编译时警告，那么下面的宏将是你的好帮手：

```
DECLARE_PER_CPU(type, name);
```

你可以利用 `get_cpu_var()` 和 `put_cpu_var()` 例程操作变量。调用 `get_cpu_var()` 返回当前处理

器上的指定变量，同时它将禁止抢占；另一方面 `put_cpu_var()` 将相应的重新激活抢占。

```
get_cpu_var(name)++;      /* 增加该处理器上的 name 变量的值 */
put_cpu_var(name);        /* 完成；重新激活内核抢占 */
```

你也可以获得别的处理器上的每个 CPU 数据：

```
per_cpu(name, cpu)++;     /* 增加指定处理器上的 name 变量的值 */
```

使用此方法你需要格外小心，因为 `per_cpu()` 函数既不会禁止内核抢占，也不会提供任何形式的锁保护。如果一些处理器可以接触到其他处理器的数据，那么你就必须要给数据上锁。注意，第 9 章和第 10 章详细讨论了数据上锁问题。

另外还有一个需要提醒的问题：这些编译时每个 CPU 数据的例子并不能在模块内使用，因为连接程序实际上将它们创建在一个唯一的可执行段中（`.data.percpu`）。如果你需要从模块中访问每个 CPU 数据，或者如果你需要动态创建这些数据，那还是有希望的。

12.10.2 运行时的每个 CPU 数据

内核实现每个 CPU 数据的动态分配方法类似于 `kmalloc()`。该例程为系统上的每个处理器创建所需内存的实例，其原型在文件 `<linux/percpu.h>` 中：

```
void *alloc_percpu(type); /* 一个宏 */
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(const void *);
```

宏 `alloc_percpu()` 给系统中的每个处理器分配一个指定类型对象的实例。它其实是宏 `__alloc_percpu()` 的一个封装，这个原始宏接收的参数有两个：一个是要分配的实际字节数，一个是分配时要按多少字节对齐。而封装后的 `alloc_percpu()` 按照单字节对齐——按照给定类型的自然边界对齐。这种对齐方式最为常用。比如：

```
struct rabid_cheetah = alloc_percpu(struct rabid_cheetah);
```

它等价于

```
struct rabid_cheetah = __alloc_percpu(sizeof (struct rabid_cheetah),
__alignof__ (struct rabid_cheetah));
```

`__alignof__` 是 gcc 的一个功能，它会返回指定类型或 lvalue 所需的（或建议的，要知道有些古怪的体系结构并没有字节对齐的要求）对齐字节数。它的语义和 `sizeof` 一样，比如，下列程序在 x86 体系中将返回 4：

```
__alignof__ (unsigned long)
```

如果指定一个 lvalue，那么将返回 lvalue 的最大对齐字节数。比如一个结构中的 lvalue 相比结构外的 lvalue 可能有更大的对齐字节需求，这是结构本身的对齐要求的缘故。有关对齐的进一步讨论我们放在第 19 章中介绍。

相应的调用 `free_percpu()` 将释放所有处理器上指定的每个 CPU 数据。

无论是 `alloc_percpu()` 或是 `__alloc_percpu()` 都会返回一个指针，它用来间接引用动态创建的每个 CPU 数据，内核提供了两个宏来利用指针获取每个 CPU 数据：

```
get_cpu_var(ptr);          /* 返回一个 void 类型指针，该指针指向处理器的 ptr 的拷贝 */
put_cpu_var(ptr);          /* 完成：重新激活内核抢占 */
```

`get_cpu_var()` 宏返回了一个指向当前处理器数据的特殊实例，它同时会禁止内核抢占；而在 `et_cpu_var()` 宏中会重新激活内核抢占。

我们来看一个使用这些函数的完整例子。当然这个例子有点无聊，因为通常你会一次分配够内存（比如，在某些初始化函数中），就可以在各种地方使用它，或再一次释放（比如，在一些清理函数中）。不过，这个例子可清楚地说明如何使用这些函数。

```
void *percpu_ptr;
unsigned long *foo;

percpu_ptr = alloc_percpu(unsigned long);
if (!ptr)
    /* 内存分配错误 ... */

foo = get_cpu_var(percpu_ptr);
/* 操作 foo ... */
put_cpu_var(percpu_ptr);
```

12.11 使用每个 CPU 数据的原因

使用每个 CPU 数据具有不少好处。首先是减少了数据锁定。因为按照每个处理器访问每个 CPU 数据的逻辑，你可以不再需要任何锁。记住“只有这个处理器能访问这个数据”的规则纯粹是一个编程约定。你需要确保本地处理器只会访问它自己的唯一数据。系统本身并不存在任何措施禁止你从事欺骗活动。

第二个好处是使用每个 CPU 数据可以大大减少缓存失效。失效发生在处理器试图使它们的缓存保持同步时。如果一个处理器操作某个数据，而该数据又存放在其他处理器缓存中，那么存放该数据的那个处理器必须清理或刷新自己的缓存。持续不断的缓存失效称为缓存抖动，这样对系统性能影响颇大。使用每个 CPU 数据将使得缓存影响降至最低，因为理想情况下只会访问自己的数据。`percpu` 接口缓存一对齐（`cache-align`）所有数据，以便确保在访问一个处理器的数据时，不会将另一个处理器的数据带入同一个缓存线上。

综上所述，使用每个 CPU 数据会省去许多（或最小化）数据上锁，它唯一的安全要求就是要禁止内核抢占。而这点代价相比上锁要小得多，而且接口会自动帮你完成这个步骤。每个 CPU 数据在中断上下文或进程上下文中使用都很安全。但要注意，不能在访问每个 CPU 数据过程中睡眠——否则，你就可能醒来后已经到了其他处理器上了。

目前并不要求必须使用每个 CPU 的新接口。只要你禁止了内核抢占，用手动方法（利用我们原来讨论的数组）就很好，但是新接口在将来更容易使用，而且功能也会得到长足的优化。如果确实决定在你的内核中使用每个 CPU 数据，请考虑使用新接口。但我要提醒的是——新接口并不向后兼容之前的内核。

12.12 分配函数的选择

在这么多分配函数和方法中，有时并不能搞清楚到底该选择那种方式分配——但这确实很重要。如果你需要连续的物理页，就可以使用某个低级页分配器或 `kmalloc()`。这是内核中内存分配的常用方式，也是大多数情况下你自己应该使用的内存分配方式。回忆一下，传递给这些函数的两个最常用的标志是 `GFP_ATOMIC` 和 `GFP_KERNEL`。`GFP_ATOMIC` 表示进行不睡眠的高优先级分配，这是中断处理程序和其他不能睡眠的代码段的需要。对于可以睡眠的代码，（比如没有持自旋锁的进程上下文代码）则应该使用 `GFP_KERNEL` 获取所需的内存。这个标志表示如果有必要，分配时可以睡眠。

如果你想从高端内存进行分配，就使用 `alloc_pages()`。`alloc_pages()` 函数返回一个指向 `struct page` 结构的指针，而不是一个指向某个逻辑地址的指针。因为高端内存很可能并没有被映射，因此，访问它的唯一方式就是通过相应的 `struct page` 结构。为了获得真正的指针，应该调用 `kmap()`，把高端内存映射到内核的逻辑地址空间。

如果你不需要物理上连续的页，而仅仅需要虚拟地址上连续的页，那么就使用 `vmalloc()`（不过要记住 `vmalloc()` 相对 `kmalloc()` 来说，有一定的性能损失）。`vmalloc()` 函数分配的内存虚地址是连续的，但它本身并不保证物理上的连续。这与用户空间的分配非常类似，它也是把物理内存块映射到连续的逻辑地址空间上。

如果你要创建和撤销很多大的数据结构，那么考虑建立 slab 高速缓存。slab 层会给每个处理器维持一个对象高速缓存（空闲链表），这种高速缓存会极大地提高对象分配和回收的性能。slab 层不是频繁地分配和释放内存，而是为你把事先分配好的对象存放到高速缓存中。当你需要一块新的内存来存放数据结构时，slab 层一般无须另外去分配内存，而只需要从高速缓存中得到一个对象就可以了。

12.13 小结

本章中，我们学习了 Linux 内核如何管理内存。我们首先看到了内存空间的各种不同的描述单位，包括字节、页面和区（在第 15 章的进程地址空间中可看到 4 种不同层次的内存单位）。我们接着讨论了各种内存分配机制，其中包括页分配器和 slab 分配器。在内核中分配内存并非总是轻而易举，因为你必须小心地确保分配过程遵从内核特定的状态约束。比如分配过程中不得堵塞，或者访问文件系统等约束。为此我们讨论了 gfp 标识以及使用每个标识的针对场景。分配内存相对复杂是内核开发和用户程序开发的最大区别之一，本章使用大量篇幅描述内存分配的各种不同接口——通过这些不同调用接口，你应该能感觉到内核中分配内存为什么更复杂的原因。在本章基础上，在第 13 章我们讨论虚拟文件系统（VFS）——负责管理文件系统且为用户空间程序提供一致性接口的内核子系统。我们继续深入！

第 13 章

虚拟文件系统

虚拟文件系统（有时也称作虚拟文件交换，更常见的是简称 VFS）作为内核子系统，为用户空间程序提供了文件和文件系统相关的接口。系统中所有文件系统不但依赖 VFS 共存，而且也依靠 VFS 系统协同工作。通过虚拟文件系统，程序可以利用标准的 Unix 系统调用对不同的文件系统，甚至不同介质上的文件系统进行读写操作，如图 13-1 所示。

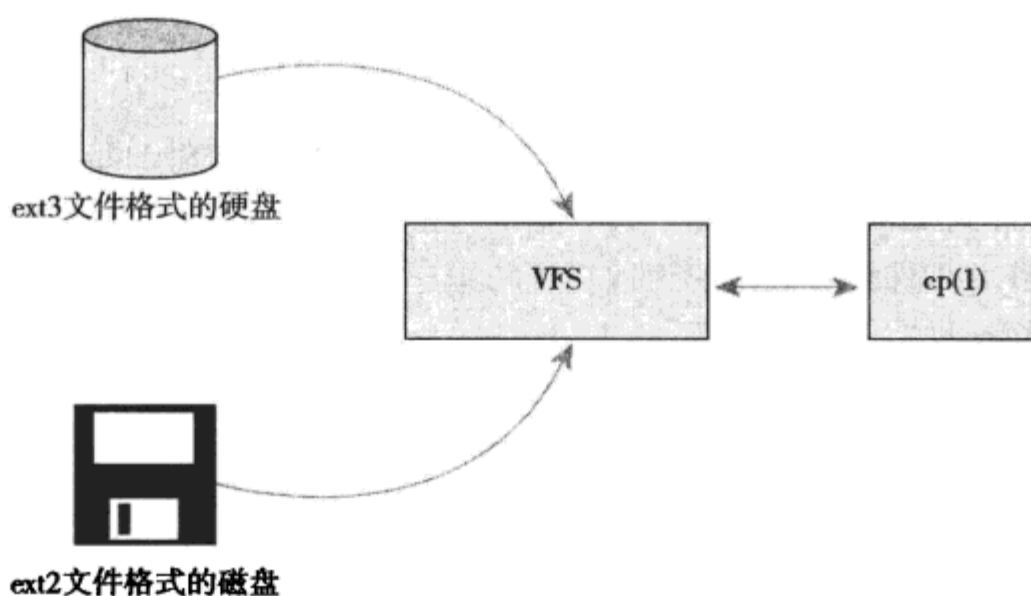


图 13-1 VFS 执行的动作：使用 `cp(1)` 命令从 ext3 文件系统格式的硬盘拷贝数据到 ext2 文件系统格式的可移动磁盘上。两种不同的文件系统，两种不同的介质，连接到同一个 VFS 上

13.1 通用文件系统接口

VFS 使得用户可以直接使用 `open()`、`read()` 和 `write()` 这样的系统调用而无须考虑具体文件系统和实际物理介质。现在听起来这也没什么新奇的（我们早就认为这是理所当然的），但是，使得这些通用的系统调用可以跨越各种文件系统和不同介质执行，绝非是微不足道的成绩。更了不起的是，系统调用可以在这些不同的文件系统和介质之间执行——我们可以使用标准的系统调用从一个文件系统拷贝或移动数据到另一个文件系统。老式的操作系统（比如 DOS）是无力完成上述工作的，任何对非本地文件系统的访问都必须依靠特殊工具才能完成。正是由于现代操作系统引入抽象层，比如 Linux，通过虚拟接口访问文件系统，才使得这种协作性和泛型存取成为可能。

新的文件系统和新类型的存储介质都能找到进入 Linux 之路，程序无需重写，甚至无须重新编译。在本章中，我们将讨论 VFS，它把各种不同的文件系统抽象后采用统一的方式进行操作。在第 14 章中，我们将讨论块 I/O 层，它支持各种各样的存储设备——从 CD 到蓝光光盘，从硬件设备再到压缩闪存。VFS 与块 I/O 相结合，提供抽象、接口以及交融，使得用户空间的程序调

用统一的系统调用访问各种文件，不管文件系统是什么，也不管文件系统位于何种介质，采用的命名策略是统一的。

13.2 文件系统抽象层

之所以可以使用这种通用接口对所有类型的文件系统进行操作，是因为内核在它的底层文件系统接口上建立了一个抽象层。该抽象层使 Linux 能够支持各种文件系统，即便是它们在功能和行为上存在很大差别。为了支持多文件系统，VFS 提供了一个通用文件系统模型，该模型囊括了任何文件系统的常用功能集和行为。当然，该模型偏重于 Unix 风格的文件系统（我们将在后面的小节看到 Unix 风格的文件系统的构成）。但即使这样，Linux 仍然可以支持很多种差异很大的文件系统，从 DOS 系统的 FAT 到 Windows 系统的 NTFS，再到各种 Unix 风格文件系统和 Linux 特有的文件系统。

VFS 抽象层之所以能衔接各种各样的文件系统，是因为它定义了所有文件系统都支持的、基本的、概念上的接口和数据结构。同时实际文件系统也将自身的诸如“如何打开文件”，“目录是什么”等概念在形式上与 VFS 的定义保持一致。因为实际文件系统的代码在统一的接口和数据结构下隐藏了具体的实现细节，所以在 VFS 层和内核的其他部分看来，所有文件系统都是相同的，它们都支持像文件和目录这样的概念，同时也支持像创建文件和删除文件这样的操作。

内核通过抽象层能够方便、简单地支持各种类型的文件系统。实际文件系统通过编程提供 VFS 所期望的抽象接口和数据结构，这样，内核就可以毫不费力地和任何文件系统协同工作，并且这样提供给用户空间的接口，也可以和任何文件系统无缝地连接在一起，完成实际工作。

其实在内核中，除了文件系统本身外，其他部分并不需要了解文件系统的内部细节。比如一个简单的用户空间程序执行如下的操作：

```
ret = write(fd, buf, len);
```

该系统调用将 buf 指针指向的长度为 len 字节的数据写入文件描述符 fd 对应的文件的当前位置。这个系统调用首先被一个通用系统调用 sys_write() 处理，sys_write() 函数要找到 fd 所在的文件系统实际给出的是哪个写操作，然后再执行该操作。实际文件系统的写方法是文件系统实现的一部分，数据最终通过该操作写入介质（或执行这个文件系统想要完成的写动作）。图 13-2 描述了从用户空间的 write() 调用到数据被写入磁盘介质的整个流程。一方面，系统调用是通用 VFS 接口，提供给用户空间的前端；另一方面，系统调用是具体文件系统的后端，处理实现细节。接下来的小节中我们会具体看到 VFS 抽象模型以及它提供的接口。

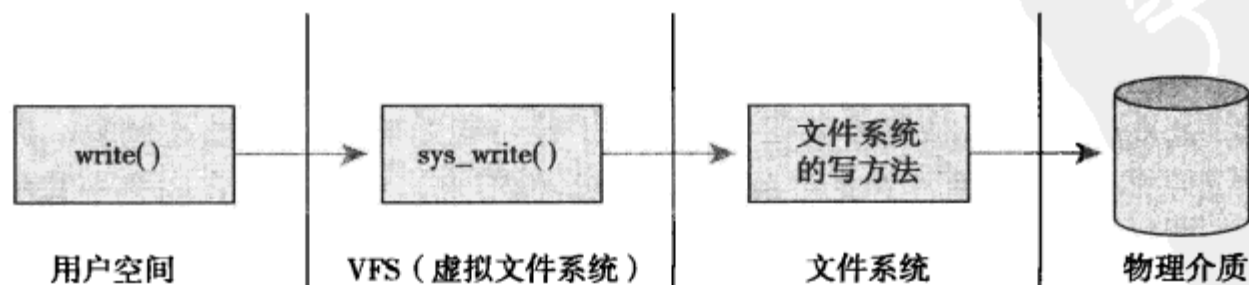


图 13-2 write() 调用将来自用户空间的数据流，首先通过 VFS 的通用系统调用，其次通过文件系统的特殊写法，最后写入物理介质中

13.3 Unix 文件系统

Unix 使用了四种和文件系统相关的传统抽象概念：文件、目录项、索引节点和安装点 (mount point)。

从本质上讲文件系统是特殊的数据分层存储结构，它包含文件、目录和相关的控制信息。文件系统的通用操作包含创建、删除和安装等。在 Unix 中，文件系统被安装在一个特定的安装点上，该安装点在全局层次结构^①中被称作命名空间，所有的已安装文件系统都作为根文件系统树的枝叶出现在系统中。与这种单一、统一的树形成鲜明对照的就是 DOS 和 Windows 的表现，它们将文件的命名空间分类为驱动字母，例如 C:。这种将命名空间划分为设备和分区的做法，相当于把硬件细节“泄露”给文件系统抽象层。对用户而言，如此的描述有点随意，甚至产生混淆，这是 Linux 统一命名空间所不屑一顾的。

文件其实可以做一个有序字节串，字节串中第一个字节是文件的头，最后一个字节是文件的尾。每一个文件为了便于系统和用户识别，都被分配了一个便于理解的名字。典型的文件操作有读、写、创建和删除等。Unix 文件的概念与面向记录的文件系统（如 OpenVMS 的 File-11）形成鲜明的对照。面向记录的文件系统提供更丰富、更结构化的表示，而简单的面向字节流抽象的 Unix 文件则以简单性和相当的灵活性为代价。

文件通过目录组织起来。文件目录好比一个文件夹，用来容纳相关文件。因为目录也可以包含其他目录，即子目录，所以目录可以层层嵌套，形成文件路径。路径中的每一部分都被称作目录条目。“/home/wolfman/butter”是文件路径的一个例子——根目录 /，目录 home，wolfman 和文件 butter 都是目录条目，它们统称为目录项。在 Unix 中，目录属于普通文件，它列出包含在其中的所有文件。由于 VFS 把目录当作文件对待，所以可以对目录执行和文件相同的操作。

Unix 系统将文件的相关信息和文件本身这两个概念加以区分，例如访问控制权限、大小、拥有者、创建时间等信息。文件相关信息，有时被称作文件的元数据（也就是说，文件的相关数据），被存储在一个单独的数据结构中，该结构被称为索引节点 (inode)，它其实是 index node 的缩写，不过近来术语“inode”使用得更为普遍一些。

所有这些信息都和文件系统的控制信息密切相关，文件系统的控制信息存储在超级块中，超级块是一种包含文件系统信息的数据结构。有时，把这些收集起来的信息称为文件系统数据元，它集单独文件信息和文件系统的信息于一身。

一直以来，Unix 文件系统在它们物理磁盘布局中也是按照上述概念实现的。比如说在磁盘上，文件（目录也属于文件）信息按照索引节点形式存储在单独的块中；控制信息被集中存储在磁盘的超级块中，等等。Unix 中文件的概念从物理上被映射到存储介质。Linux 的 VFS 的设计目标就是要保证能与支持和实现了这些概念的文件系统协同工作。像如 FAT 或 NTFS 这样的非 Unix 风格的文件系统，虽然也可以在 Linux 上工作，但是它们必须经过封装，提供一个符合这些概念的界面。比如，即使一个文件系统不支持索引节点，它也必须在内存中装配索引节点结构体，就像它本身包含索引节点一样。再比如，如果一个文件系统将目录看做一种特殊对象，那

^① 近来，Linux 已经将这种层次化概念引入了单个进程中，每个进程都指定一个唯一的命名空间。因为每个进程都会继承父进程的命名空间（除非是特别声明的情况），所以所有进程往往都只有一个全局命名空间。

么要想使用 VFS，就必须将目录重新表示为文件形式。通常，这种转换需要在使用现场（on the fly）引入一些特殊处理，使得非 Unix 文件系统能够兼容 Unix 文件系统的使用规则并满足 VFS 的需求。这种文件系统当然仍能工作，但是其带来的开销则不可思议（开销太大了）。

13.4 VFS 对象及其数据结构

VFS 其实采用的是面向对象^①的设计思路，使用一组数据结构来代表通用文件对象。这些数据结构类似于对象。因为内核纯粹使用 C 代码实现，没有直接利用面向对象的语言，所以内核中的数据结构都使用 C 语言的结构体实现，而这些结构体包含数据的同时也包含操作这些数据的函数指针，其中的操作函数由具体文件系统实现。

VFS 中有四个主要的对象类型，它们分别是：

- 超级块对象，它代表一个具体的已安装文件系统。
- 索引节点对象，它代表一个具体文件。
- 目录项对象，它代表一个目录项，是路径的一个组成部分。
- 文件对象，它代表由进程打开的文件。

注意，因为 VFS 将目录作为一个文件来处理，所以不存在目录对象。回忆本章前面所提到的目录项代表的是路径中的一个组成部分，它可能包括一个普通文件。换句话说，目录项不同于目录，但目录却是另一种形式的文件，明白了吗？

每个主要对象中都包含一个操作对象，这些操作对象描述了内核针对主要对象可以使用的方法：

- `super_operations` 对象，其中包括内核针对特定文件系统所能调用的方法，比如 `write_inode()` 和 `sync_fs()` 等方法。
- `inode_operations` 对象，其中包括内核针对特定文件所能调用的方法，比如 `create()` 和 `link()` 等方法。
- `dentry_operations` 对象，其中包括内核针对特定目录所能调用的方法，比如 `d_compare()` 和 `d_delete()` 等方法。
- `file_operations` 对象，其中包括进程针对已打开文件所能调用的方法，比如 `read()` 和 `write()` 等方法。

操作对象作为一个结构体指针来实现，此结构体中包含指向操作其父对象的函数指针。对于其中许多方法来说，可以继承使用 VFS 提供的通用函数，如果通用函数提供的基本功能无法满足需要，那么就必须使用实际文件系统的独有方法填充这些函数指针，使其指向文件系统实例。

再次提醒，我们这里所说的对象就是指结构体，而不是像 C++ 或 Java 那样的真正的对象数据类型。但是这些结构体的确代表的是一个对象，它含有相关的数据和对这些数据的操作，所以可以说它们就是对象。

^① 人们时常忽略，甚至会否认，但是在内核中确实存在很多利用面向对象思想编程的例子。虽然内核开发者可能有意避免 C++ 和其他面向对象语言，但是面向对象的思想仍然经常被借鉴——虽然 C 语言缺乏面向对象的机制。VFS 就是一个利用 C 代码来有效和简洁地实现 OOP 的例子。

VFS 使用了大量结构体对象，它所包括的对象远远多于上面提到的这几种主要对象。比如每个注册的文件系统都由 `file_system_type` 结构体来表示，它描述了文件系统及其性能；另外，每一个安装点也都用 `vfsmount` 结构体表示，它包含的是安装点的相关信息，如位置和安装标志等。

在本章的最后还要介绍两个与进程相关的结构体，它们描述了文件系统以及和进程相关的文件，分别是 `fs_struct` 结构体和 `file` 结构体。

13.5 节将讨论这些对象以及它们在 VFS 层的实现中扮演的角色。

13.5 超级块对象

各种文件系统都必须实现超级块对象，该对象用于存储特定文件系统的信息，通常对应于存放在磁盘特定扇区中的文件系统超级块或文件系统控制块（所以称为超级块对象）。对于并非基于磁盘的文件系统（如基于内存的文件系统，比如 `sysfs`），它们会在使用现场创建超级块并将其保存到内存中。

超级块对象由 `super_block` 结构体表示，定义在文件 `<linux/fs.h>` 中，下面给出它的结构和各个域的描述：

```
struct super_block {
    struct list_head    s_list;           /* 指向所有超级块的链表 */
    dev_t               s_dev;           /* 设备标识符 */
    unsigned long       s_blocksize;     /* 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;      /* 文件大小上限 */
    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;      /* 磁盘限额方法 */
    struct quotactl_ops  *s_qcop;        /* 限额控制方法 */
    struct export_operations *s_export_op; /* 导出方法 */
    unsigned long       s_flags;         /* 挂载标志 */
    unsigned long       s_magic;         /* 文件系统的幻数 */
    struct dentry        *s_root;        /* 目录挂载点 */
    struct rw_semaphore  s_umount;       /* 卸载信号量 */
    struct semaphore     s_lock;         /* 超级块信号量 */
    int                  s_count;        /* 超级块引用计数 */
    int                  s_need_sync;    /* 尚未同步标志 */
    atomic_t             s_active;       /* 活动引用计数 */
    void                 *s_security;    /* 安全模块 */
    struct xattr_handler **s_xattr;      /* 扩展的属性操作 */
    struct list_head     s_inodes;       /* inodes 链表 */
    struct list_head     s_dirty;        /* 脏数据链表 */
    struct list_head     s_io;           /* 回写链表 */
    struct list_head     s_more_io;      /* 更多回写的链表 */
    struct hlist_head     s_anon;        /* 匿名目录项 */
    struct list_head     s_files;        /* 被分配文件链表 */
    struct list_head     s_dentry_lru;   /* 未被使用目录项链表 */
    int                  s_nr_dentry_unused; /* 链表中目录项的数目 */
    struct block_device  *s_bdev;        /* 相关的块设备 */
}
```

```

struct mtd_info          *s_mtd;          /* 存储磁盘信息 */
struct list_head         s_instances;     /* 该类型文件系统 */
struct quota_info        s_dquot;        /* 限额相关选项 */
int                      s_frozen;        /* frozen 标志位 */
wait_queue_head_t        s_wait_unfrozen; /* 冻结的等待队列 */
char                     s_id[32];        /* 文本名字 */
void                     *s_fs_info;       /* 文件系统特殊信息 */
fmode_t                  s_mode;          /* 安装权限 */
struct semaphore         s_vfs_rename_sem; /* 重命名信号量 */
u32                      s_time_gran;     /* 时间戳粒度 */
char                     *s_subtype;      /* 子类型名称 */
char                     *s_options;      /* 已存安装选项 */
};

```

创建、管理和撤销超级块对象的代码位于文件 `fs/super.c` 中。超级块对象通过 `alloc_super()` 函数创建并初始化。在文件系统安装时，文件系统会调用该函数以便从磁盘读取文件系统超级块，并且将其信息填充到内存中的超级块对象中。

13.6 超级块操作

超级块对象中最重要的一个域是 `s_op`，它指向超级块的操作函数表。超级块操作函数表由 `super_operations` 结构体表示，定义在文件 `<linux/fs.h>` 中，其形式如下：

```

struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*dirty_inode) (struct inode *);
    int (*write_inode) (struct inode *, int);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*freeze_fs) (struct super_block *);
    int (*unfreeze_fs) (struct super_block *);
    int (*statfs) (struct dentry *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
    int (*show_options)(struct seq_file *, struct vfsmount *);
    int (*show_stats)(struct seq_file *, struct vfsmount *);
    ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
    int (*bdev_try_to_free_page)(struct super_block *, struct page *, gfp_t);
};

```

该结构体中的每一项都是一个指向超级块操作函数的指针，超级块操作函数执行文件系统和索引节点的低层操作。

当文件系统需要对其超级块执行操作时，首先要在超级块对象中寻找需要的操作方法。比

如，如果一个文件系统要写自己的超级块，需要调用：

```
sb->s_op->write_super(sb);
```

在这个调用中，sb 是指向文件系统超级块的指针，沿着该指针进入超级块操作函数表 s_op，并从表中取得希望得到的 write_super() 函数，该函数执行写入超级块的实际操作。注意，尽管 write_super() 方法来自超级块，但是在调用时，还是要把超级块作为参数传递给它，这是因为 C 语言中缺少对面向对象的支持，而在 C++ 中，使用如下的调用就足够了：

```
sb.write_super();
```

由于在 C 语言中无法直接得到操作函数的父对象，所以必须将父对象以参数形式传给操作函数。下面给出 super_operation 中，超级块操作函数的用法。

- struct inode *alloc_inode(struct super_block *sb)

在给定的超级块下创建和初始化一个新的索引节点对象。

- void destroy_inode(struct inode *inode)

用于释放给定的索引节点。

- void dirty_inode(struct inode *inode)

VFS 在索引节点脏（被修改）时会调用此函数。日志文件系统（如 ext3 和 ext4）执行该函数进行日志更新。

- void write_inode(struct inode *inode, int wait)

用于将给定的索引节点写入磁盘。wait 参数指明写操作是否需要同步。

- void drop_inode(struct inode *inode)

在最后一个指向索引节点的引用被释放后，VFS 会调用该函数。VFS 只需要简单地删除这个索引节点后，普通 Unix 文件系统就不会定义这个函数了。

- void delete_inode(struct inode *inode)

用于从磁盘上删除给定的索引节点。

- void put_super(struct super_block *sb)

在卸载文件系统时由 VFS 调用，用来释放超级块。调用者必须一直持有 s_lock 锁。

- void write_super(struct super_block *sb)

用给定的超级块更新磁盘上的超级块。VFS 通过该函数对内存中的超级块和磁盘中的超级块进行同步。调用者必须一直持有 s_lock 锁。

- int sync_fs(struct super_block *sb, int wait)

使文件系统的数据元与磁盘上的文件系统同步。wait 参数指定操作是否同步。

- void write_super_lockfs(struct super_block *sb)

首先禁止对文件系统做改变，再使用给定的超级块更新磁盘上的超级块。目前 LVM(逻辑卷标管理)会调用该函数。

- void unlockfs(struct super_block *sb)

对文件系统解除锁定，它是 write_super_lockfs() 的逆操作。

- int statfs(struct super_block *sb, struct statfs *statfs)

VFS 通过调用该函数获取文件系统状态。指定文件系统相关的统计信息将放置在 statfs 中。

- int remount_fs(struct super_block *sb, int *flags, char *data)

当指定新的安装选项重新安装文件系统时，VFS 会调用该函数。调用者必须一直持有 s_lock 锁。

- void clear_inode(struct inode *inode)

VFS 调用该函数释放索引节点，并清空包含相关数据的所有页面。

- void umount_begin(struct super_block *sb)

VFS 调用该函数中断安装操作。该函数被网络文件系统使用，如 NFS。

所有以上函数都是由 VFS 在进程上下文中调用。除了 dirty_inode(), 其他函数在必要时都可以阻塞。

这其中的一些函数是可选的。在超级块操作表中，文件系统可以将不需要的函数指针设置成 NULL。如果 VFS 发现操作函数指针是 NULL，那它要么就会调用通用函数执行相应操作，要么什么也不做，如何选择取决于具体操作。

13.7 索引节点对象

索引节点对象包含了内核在操作文件或目录时需要的全部信息。对于 Unix 风格的文件系统来说，这些信息可以从磁盘索引节点直接读入。如果一个文件系统没有索引节点，那么，不管这些相关信息在磁盘上是怎么存放的，文件系统都必须从中提取这些信息。没有索引节点的文件系统通常将文件的描述信息作为文件的一部分来存放。这些文件系统与 Unix 风格的文件系统不同，没有将数据与控制信息分开存放。有些现代文件系统使用数据库来存储文件的数据。不管哪种情况、采用哪种方式，索引节点对象必须在内存中创建，以便于文件系统使用。

索引节点对象由 inode 结构体表示，它定义在文件 <linux/fs.h> 中，下面给出它的结构体和各项的描述。

```
struct inode {
    struct hlist_node    i_hash;           /* 散列表 */
    struct list_head     i_list;          /* 索引节点链表 */
    struct list_head     i_sb_list;       /* 超级块链表 */
    struct list_head     i_dentry;        /* 目录项链表 */
    unsigned long         i_ino;           /* 节点号 */
}
```

```

atomic_t          i_count;          /* 引用计数 */
unsigned int       i_nlink;          /* 硬链接数 */
uid_t             i_uid;             /* 使用者的 id */
gid_t             i_gid;             /* 使用组的 id */
kdev_t            i_rdev;            /* 实际设备标识符 */
u64               i_version;         /* 版本号 */
loff_t            i_size;            /* 以字节为单位的文件大小 */
seqcount_t        i_size_seqcount;  /* 对 i_size 进行串行计数 */
struct timespec   i_atime;           /* 最后访问时间 */
struct timespec   i_mtime;           /* 最后修改时间 */
struct timespec   i_ctime;           /* 最后改变时间 */
unsigned int       i_blkbits;        /* 以位为单位的块大小 */
blkcnt_t          i_blocks;          /* 文件的块数 */
unsigned short     i_bytes;          /* 使用的字节数 */
umode_t           i_mode;            /* 访问权限 */
spinlock_t        i_lock;            /* 自旋锁 */
struct rw_semaphore i_alloc_sem;     /* 嵌入 i_sem 内部 */
struct semaphore   i_sem;            /* 索引节点信号量 */
struct inode_operations *i_op;       /* 索引节点操作表 */
struct file_operations *i_fop;       /* 缺省的索引节点操作 */
struct super_block *i_sb;            /* 相关的超级块 */
struct file_lock   *i_flock;         /* 文件锁链表 */
struct address_space *i_mapping;     /* 相关的地址映射 */
struct address_space i_data;         /* 设备地址映射 */
struct dquot       *i_dquot[MAXQUOTAS]; /* 索引节点的磁盘限额 */
struct list_head    i_devices;       /* 块设备链表 */
union {
    struct pipe_inode_info *i_pipe;   /* 管道信息 */
    struct block_device    *i_bdev;   /* 块设备驱动 */
    struct cdev             *i_cdev;   /* 字符设备驱动 */
};
unsigned long         i_dnotify_mask; /* 目录通知掩码 */
struct dnotify_struct *i_dnotify;     /* 目录通知 */
struct list_head      inotify_watches; /* 索引节点通知监测链表 */
struct mutex          inotify_mutex;   /* 保护 inotify_watches */
unsigned long         i_state;         /* 状态标志 */
unsigned long         dirtied_when;    /* 第一次弄脏数据的时间 */
unsigned int          i_flags;         /* 文件系统标志 */
atomic_t             i_writecount;     /* 写者计数 */
void                 *i_security;      /* 安全模块 */
void                 *i_private;      /* fs 私有指针 */
};

```

一个索引节点代表文件系统中（但是索引节点仅当文件被访问时，才在内存中创建）的一个文件，它也可以是设备或管道这样的特殊文件。因此索引节点结构体中有一些和特殊文件相关的项，比如 `i_pipe` 项就指向一个代表有名管道的数据结构，`i_bdev` 指向块设备结构体，`i_cdev` 指向字符设备结构体。这三个指针被存放在一个公用体中，因为一个给定的索引节点每次只能表示三者之一（或三者均不）。

有时，某些文件系统可能并不能完整地包含索引节点结构体要求的所有信息。举个例子，有的文件系统可能并不记录文件的访问时间，这时，该文件系统就可以在实现中选择任意合适的办法来解决这个问题。它可以在 `i_atime` 中存储 0，或者让 `i_atime` 等于 `i_mtime`，或者只在内存中更新 `i_atime` 而不将其写回磁盘，或者由文件系统的实现者来决定。

13.8 索引节点操作

和超级块操作一样，索引节点对象中的 `inode_operations` 项也非常重要，因为它描述了 VFS 用以操作索引节点对象的所有方法，这些方法由文件系统实现。与超级块类似，对索引节点的操作调用方式如下：

```
i->i_op->truncate(i)
```

`i` 指向给定的索引节点，`truncate()` 函数是由索引节点 `i` 所在的文件系统定义的。`inode_operations` 结构体定义在文件 `<linux/fs.h>` 中：

```
struct inode_operations {
    int (*create) (struct inode *, struct dentry *, int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *, struct dentry *, struct nameidata *);
    int (*link) (struct dentry *, struct inode *, struct dentry *);
    int (*unlink) (struct inode *, struct dentry *);
    int (*symlink) (struct inode *, struct dentry *, const char *);
    int (*mkdir) (struct inode *, struct dentry *, int);
    int (*rmdir) (struct inode *, struct dentry *);
    int (*mknod) (struct inode *, struct dentry *, int, dev_t);
    int (*rename) (struct inode *, struct dentry *,
                  struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char __user *, int);
    void * (*follow_link) (struct dentry *, struct nameidata *);
    void (*put_link) (struct dentry *, struct nameidata *, void *);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
    int (*setxattr) (struct dentry *, const char *, const void *, size_t, int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*removexattr) (struct dentry *, const char *);
    void (*truncate_range) (struct inode *, loff_t, loff_t);
    long (*fallocate) (struct inode *inode, int mode, loff_t offset,
                      loff_t len);
    int (*fiemap) (struct inode *, struct fiemap_extent_info *, u64 start,
                  u64 len);
};
```

下面这些接口由各种函数组成，在给定的节点上，可能由 VFS 执行这些函数，也可能由具体的文件系统执行：

- `int create(struct inode *dir, struct dentry *dentry, int mode)`

VFS 通过系统调用 `create()` 和 `open()` 来调用该函数，从而为 `dentry` 对象创建一个新的索引节点。在创建时使用 `mode` 指定的初始模式。

- `struct dentry * lookup(struct inode *dir, struct dentry *dentry)`

该函数在特定目录中寻找索引节点，该索引节点要对应于 dentry 中给出的文件名。

- `int link(struct dentry *old_dentry,
 struct inode *dir,
 struct dentry *dentry)`

该函数被系统调用 `link()` 调用，用来创建硬连接。硬连接名称由 `dentry` 参数指定，连接对象是 `dir` 目录中 `old_dentry` 目录项所代表的文件。

- `int unlink(struct inode *dir, struct dentry *dentry)`

该函数被系统调用 `unlink()` 调用，从目录 `dir` 中删除由目录项 `dentry` 指定的索引节点对象。

- `int symlink(struct inode *dir,
 struct dentry *dentry,
 const char *symname)`

该函数被系统调用 `symlink()` 调用，创建符号连接。该符号连接名称由 `symname` 指定，连接对象是 `dir` 目录中的 `dentry` 目录项。

- `int mkdir(struct inode *dir,
 struct dentry *dentry,
 int mode)`

该函数被系统调用 `mkdir()` 调用，创建一个新目录。创建时使用 `mode` 指定的初始模式。

- `int rmdir(struct inode *dir,
 struct dentry *dentry)`

该函数被系统调用 `rmdir()` 调用，删除 `dir` 目录中的 `dentry` 目录项代表的文件。

- `int mknod(struct inode *dir,
 struct dentry *dentry,
 int mode, dev_t rdev)`

该函数被系统调用 `mknod()` 调用，创建特殊文件（设备文件、命名管道或套接字）。要创建的文件放在 `dir` 目录中，其目录项为 `dentry`，关联的设备为 `rdev`，初始权限由 `mode` 指定。

- `int rename(struct inode *old_dir,
 struct dentry *old_dentry,
 struct inode *new_dir,
 struct dentry *new_dentry)`

VFS 调用该函数来移动文件。文件源路径在 `old_dir` 目录中，源文件由 `old_dentry` 目录项指定，目标路径在 `new_dir` 目录中，目标文件由 `new_dentry` 指定。

- `int readlink(struct dentry *dentry,
 char *buffer, int buflen)`

该函数被系统调用 `readlink()` 调用，拷贝数据到特定的缓冲 `buffer` 中。拷贝的数据来自 `dentry` 指定的符号连接，拷贝大小最大可达 `buflen` 字节。

- `int follow_link(struct dentry *dentry,
 struct nameidata *nd)`

该函数由 VFS 调用，从一个符号连接查找它指向的索引节点。由 dentry 指向的连接被解析，其结果存放在由 nd 指向的 nameidata 结构体中。

- int put_link(struct dentry *dentry,
 struct nameidata *nd)

在 follow_link() 调用之后，该函数由 VFS 调用进行清除工作。

- void truncate(struct inode *inode)

该函数由 VFS 调用，修改文件的大小。在调用前，索引节点的 i_size 项必须设置为预期的大小。

- int permission(struct inode *inode, int mask)

该函数用来检查给定的 inode 所代表的文件是否允许特定的访问模式。如果允许特定的访问模式，返回零，否则返回负值的错误码。多数文件系统都将此区域设置为 NULL，使用 VFS 提供的通用方法进行检查。这种检查操作仅仅比较索引节点对象中的访问模式位是否和给定的 mask 一致。比较复杂的系统（比如支持访问控制链（ACLS）的文件系统），需要使用特殊的 permission() 方法。

- int setattr(struct dentry *dentry,
 struct iattr *attr)

该函数被 notify_change() 调用，在修改索引节点后，通知发生了“改变事件”。

- int getattr(struct vfsmount *mnt,
 struct dentry *dentry,
 struct kstat *stat)

在通知索引节点需要从磁盘中更新时，VFS 会调用该函数。

扩展属性允许 key/value 这样的一对值与文件相关联。

- int setxattr(struct dentry *dentry,
 const char *name,
 const void *value,
 size_t size, int flags)

该函数由 VFS 调用，给 dentry 指定的文件设置扩展属性。属性名为 name，值为 value。

- ssize_t getxattr(struct dentry *dentry,
 const char *name,
 void *value, size_t size)

该函数由 VFS 调用，向 value 中拷贝给定文件的扩展属性 name 对应的数值。

- ssize_t listxattr(struct dentry *dentry,
 char *list, size_t size)

该函数将特定文件的所有属性列表拷贝到一个缓冲列表中。

- int removexattr(struct dentry *dentry,
 const char *name)

该函数从给定文件中删除指定的属性。

13.9 目录项对象

VFS 把目录当作文件对待，所以在路径 `/bin/vi` 中，`bin` 和 `vi` 都属于文件——`bin` 是特殊的目录文件而 `vi` 是一个普通文件，路径中的每个组成部分都由一个索引节点对象表示。虽然它们可以统一由索引节点表示，但是 VFS 经常需要执行目录相关的操作，比如路径名查找等。路径名查找需要解析路径中的每一个组成部分，不但要确保它有效，而且还需要再进一步寻找路径中的下一个部分。

为了方便查找操作，VFS 引入了目录项的概念。每个 `dentry` 代表路径中的一个特定部分。对前一个例子来说，`/`、`bin` 和 `vi` 都属于目录项对象。前两个是目录，最后一个是普通文件。必须明确一点：在路径中（包括普通文件在内），每一个部分都是目录项对象。解析一个路径并遍历其分量绝非简单的演练，它是耗时的、常规的字符串比较过程，执行耗时、代码繁琐。目录项对象的引入使得这个过程更加简单。

目录项也可包括安装点。在路径 `/mnt/cdrom/foo` 中，构成元素 `/`、`mnt`、`cdrom` 和 `foo` 都属于目录项对象。VFS 在执行目录操作时（如果需要的话）会现场创建目录项对象。

目录项对象由 `dentry` 结构体表示，定义在文件 `<linux/dcache.h>` 中。下面给出该结构体和其中各项的描述：

```
struct dentry {
    atomic_t                d_count;                /* 使用记数 */
    unsigned int            d_flags;                /* 目录项标识 */
    spinlock_t              d_lock;                /* 单目录项锁 */
    int                     d_mounted;             /* 是登录点的目录项吗？ */
    struct inode             *d_inode;              /* 相关联的索引节点 */
    struct hlist_node       d_hash;                /* 散列表 */
    struct dentry            *d_parent;             /* 父目录的目录项对象 */
    struct qstr             d_name;                /* 目录项名称 */
    struct list_head        d_lru;                 /* 未使用的链表 */
    union {
        struct list_head    d_child;              /* 目录项内部形成的链表 */
        struct rcu_head     d_rcu;                /* RCU 加锁 */
    } d_u;
    struct list_head        d_subdirs;             /* 子目录链表 */
    struct list_head        d_alias;               /* 索引节点别名链表 */
    unsigned long           d_time;                /* 重置时间 */
    struct dentry_operations *d_op;                /* 目录项操作指针 */
    struct super_block       *d_sb;                /* 文件的超级块 */
    void                    *d_fsdata;             /* 文件系统特有数据 */
    unsigned char           d_iname[DNAME_INLINE_LEN_MIN]; /* 短文件名 */
};
```

与前面的两个对象不同，目录项对象没有对应的磁盘数据结构，VFS 根据字符串形式的路径名现场创建它。而且由于目录项对象并非真正保存在磁盘上，所以目录项结构体没有是否被修改的标志（也就是是否为脏、是否需要写回磁盘的标志）。

13.9.1 目录项状态

目录项对象有三种有效状态：被使用、未被使用和负状态。

一个被使用的目录项对应一个有效的索引节点（即 `d_inode` 指向相应的索引节点）并且表明该对象存在一个或多个使用者（即 `d_count` 为正值）。一个目录项处于被使用状态，意味着它正被 VFS 使用并且指向有效的数据，因此不能被丢弃。

一个未被使用的目录项对应一个有效的索引节点（`d_inode` 指向一个索引节点），但是应指明 VFS 当前并未使用它（`d_count` 为 0）。该目录项对象仍然指向一个有效对象，而且被保留在缓存中以便需要时再使用它。由于该目录项不会过早地被撤销，所以以后再需要它时，不必重新创建，与未缓存的目录项相比，这样使路径查找更迅速。但如果要回收内存的话，可以撤销未使用的目录项。

一个负状态的目录项^①没有对应的有效索引节点（`d_inode` 为 NULL），因为索引节点已被删除了，或路径不再正确了，但是目录项仍然保留，以便快速解析以后的路径查询。比如，一个守护进程不断地去试图打开并读取一个不存在的配置文件。`open()` 系统调用不断地返回 `ENOENT`，直到内核构建了这个路径、遍历磁盘上的目录结构体并检查这个文件的确不存在为止。即便这个失败的查找很浪费资源，但是将负状态缓存起来还是非常值得的。虽然负状态的目录项有些用处，但是如果有需要，可以撤销它，因为毕竟实际上很少用到它。

目录项对象释放后也可以保存到 slab 对象缓存中去，这点在第 12 章讨论过。此时，任何 VFS 或文件系统代码都没有指向该目录项对象的有效引用。

13.9.2 目录项缓存

如果 VFS 层遍历路径名中所有的元素并将它们逐个地解析成目录项对象，还要到达最深层目录，将是一件非常费力的工作，会浪费大量的时间。所以内核将目录项对象缓存在目录项缓存（简称 `dcache`）中。

目录项缓存包括三个主要部分：

- “被使用的”目录项链表。该链表通过索引节点对象中的 `i_dentry` 项连接相关的索引节点，因为一个给定的索引节点可能有多个链接，所以就可能有多个目录项对象，因此用一个链表来连接它们。
- “最近被使用的”双向链表。该链表含有未被使用的和负状态的目录项对象。由于该链总是在头部插入目录项，所以链头节点的数据总比链尾的数据要新。当内核必须通过删除节点项回收内存时，会从链尾删除节点项，因为尾部的节点最旧，所以它们在近期内再次被使用的可能性最小。
- 散列表和相应的散列函数用来快速地将给定路径解析为相关目录项对象。

散列表由数组 `dentry_hashtable` 表示，其中每一个元素都是一个指向具有相同键值的目录项对象链表的指针。数组的大小取决于系统中物理内存的大小。

实际的散列值由 `d_hash()` 函数计算，它是内核提供给文件系统的唯一的一个散列函数。

查找散列表要通过 `d_lookup()` 函数，如果该函数在 `dcache` 中发现了与其相匹配的目录项对象，则匹配的对象被返回；否则，返回 NULL 指针。

^① 这个名字容易产生误导，其实它和任何负数或负状态并没有联系。更准确的名称应该是无效目录项。

举例说明, 假设你需要在自己目录中编译一个源文件, `/home/dracula/src/the_sun_sucks.c`, 每一次对文件进行访问 (比如说, 首先要打开它, 然后要存储它, 还要进行编译等), VFS 都必须沿着嵌套的目录依次解析全部路径: `/`、`home`、`dracula`、`src` 和最终的 `the_sun_sucks.c`。为了避免每次访问该路径名都进行这种耗时的操作, VFS 会先在目录项缓存中搜索路径名, 如果找到了, 就无须花费那么大的力气了。相反, 如果该目录项在目录项缓存中并不存在, VFS 就必须自己通过遍历文件系统为每个路径分量解析路径, 解析完毕后, 再将目录项对象加入 `dcache` 中, 以便以后可以快速查找到它。

而 `dcache` 在一定意义上也提供对索引节点的缓存, 也就是 `icache`。和目录项对象相关的索引节点对象不会被释放, 因为目录项会让相关索引节点的使用计数为正, 这样就可以确保索引节点留在内存中。只要目录项被缓存, 其相应的索引节点也就被缓存了。所以像前面的例子, 只要路径名在缓存中找到了, 那么相应的索引节点肯定也在内存中缓存着。

因为文件访问呈现空间和时间的局部性, 所以对目录项和索引节点进行缓存非常有益。文件访问有时间上的局部性, 是因为程序可能会一次又一次地访问相同的文件。因此, 当一个文件被访问时, 所缓存的相关目录项和索引节点不久被命中的概率较高。文件访问具有空间的局部性是因为程序可能在同一个目录下访问多个文件, 因此一个文件对应的目录项缓存后极有可能被命中, 因为相关的文件可能在下次又被使用。

13.10 目录项操作

`dentry_operation` 结构体指明了 VFS 操作目录项的所有方法。

该结构定义在文件 `<linux/dcache.h>` 中。

```
struct dentry_operations {
    int (*d_revalidate) (struct dentry *, struct nameidata *);
    int (*d_hash) (struct dentry *, struct qstr *);
    int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
    int (*d_delete) (struct dentry *);
    void (*d_release) (struct dentry *);
    void (*d_iput) (struct dentry *, struct inode *);
    char *(*d_dname) (struct dentry *, char *, int);
};
```

下面给出函数的具体用法:

- `int d_revalidate(struct dentry *dentry, struct nameidata*);`

该函数判断目录对象是否有效。VFS 准备从 `dcache` 中使用一个目录项时, 会调用该函数。大部分文件系统将该方法置 `NULL`, 因为它们认为 `dcache` 中的目录项对象总是有效的。

- `int d_hash(struct dentry *dentry, struct qstr *name)`

该函数为目录项生成散列值, 当目录项需要加入到散列表中时, VFS 调用该函数。

- `int d_compare(struct dentry *dentry,`

```
struct qstr *name1,
struct qstr *name2)
```

VFS 调用该函数来比较 `name1` 和 `name2` 这两个文件名。多数文件系统使用 VFS 默认的操作，仅仅作字符串比较。对有些文件系统，比如 FAT，简单的字符串比较不能满足其需要。因为 FAT 文件系统不区分大小写，所以需要实现一种不区分大小写的字符串比较函数。注意使用该函数时需要加 `dcache_lock` 锁。

- `int d_delete(struct dentry *dentry)`

当目录项对象的 `d_count` 计数值等于 0 时，VFS 调用该函数。注意使用该函数需要加 `dcache_lock` 锁和目录项的 `d_lock`。

- `void d_release(struct dentry *dentry)`

当目录项对象将要被释放时，VFS 调用该函数，默认情况下，它什么也不做。

- `void d_iput(struct dentry *dentry,
struct inode *inode)`

当一个目录项对象丢失了其相关的索引节点时（也就是说磁盘索引节点被删除了），VFS 调用该函数。默认情况下 VFS 会调用 `iput()` 函数释放索引节点。如果文件系统重载了该函数，那么除了执行此文件系统特殊的工作外，还必须调用 `iput()` 函数。

13.11 文件对象

VFS 的最后一个主要对象是文件对象。文件对象表示进程已打开的文件。如果我们站在用户角度来看待 VFS，文件对象会首先进入我们的视野。进程直接处理的是文件，而不是超级块、索引节点或目录项。所以不必奇怪：文件对象包含我们非常熟悉的信息（如访问模式，当前偏移等），同样道理，文件操作和我们非常熟悉的系统调用 `read()` 和 `write()` 等也很类似。

文件对象是已打开的文件在内存中的表示。该对象（不是物理文件）由相应的 `open()` 系统调用创建，由 `close()` 系统调用撤销，所有这些文件相关的调用实际上都是文件操作表中定义的方法。因为多个进程可以同时打开和操作同一个文件，所以同一个文件也可能存在多个对应的文件对象。文件对象仅仅在进程观点上代表已打开文件，它反过来指向目录项对象（反过来指向索引节点），其实只有目录项对象才表示已打开的实际文件。虽然一个文件对应的文件对象不是唯一的，但对应的索引节点和目录项对象无疑是唯一的。

文件对象由 `file` 结构体表示，定义在文件 `<linux/fs.h>` 中，下面给出该结构体和各项的描述。

```
struct file {
    union {
        struct list_head    fu_list;        /* 文件对象链表 */
        struct rcu_head      fu_rcuhead;    /* 释放之后的 RCU 链表 */
    } f_u;
    struct path              f_path;        /* 包含目录项 */
    struct file_operations   *f_op;        /* 文件操作表 */
    spinlock_t               f_lock;        /* 单个文件结构锁 */
    atomic_t                 f_count;       /* 文件对象的使用计数 */
}
```



```

                                unsigned long,
                                unsigned long,
                                unsigned long);

int (*check_flags) (int);
int (*flock) (struct file *, int, struct file_lock *);
ssize_t (*splice_write) (struct pipe_inode_info *,
                        struct file *,
                        loff_t *,
                        size_t,
                        unsigned int);
ssize_t (*splice_read) (struct file *,
                        loff_t *,
                        struct pipe_inode_info *,
                        size_t,
                        unsigned int);
int (*setlease) (struct file *, long, struct file_lock **);
};

```

具体的文件系统可以为每一种操作做专门的实现，或者如果存在通用操作，也可以使用通用操作。一般在基于 Unix 的文件系统上，这些通用操作效果都不错。并不要求实际文件系统实现文件操作函数表中的所有方法——虽然不实现最基础的那些操作显然是很不明智的，对不感兴趣的操作完全可以简单地将该函数指针置为 NULL。

下面给出操作的用法说明：

- `loff_t lseek(struct file *file, loff_t offset, int origin)`

该函数用于更新偏移量指针，由系统调用 `lseek()` 调用它。

- `ssize_t read(struct file *file, char *buf, size_t count, loff_t *offset)`

该函数从给定文件的 `offset` 偏移处读取 `count` 字节的数据到 `buf` 中，同时更新文件指针。由系统调用 `read()` 调用它。

- `ssize_t aio_read(struct kiocb *iocb, char *buf, size_t count, loff_t offset)`

该函数从 `iocb` 描述的文件里，以同步方式读取 `count` 字节的数据到 `buf` 中。由系统调用 `aio_read()` 调用它。

- `ssize_t write(struct file *file, const char *buf, size_t count, loff_t *offset)`

该函数从给定的 `buf` 中取出 `count` 字节的数据，写入给定文件的 `offset` 偏移处，同时更新文件指针。由系统调用 `write()` 调用它。

- `ssize_t aio_write(struct kiocb *iocb,
 const char *buf,
 size_t count, loff_t offset)`

该函数以同步方式从给定的 buf 中取出 count 字节的数据，写入由 iocb 描述的文件中。由系统调用 `aio_write()` 调用它。

- `int readdir(struct file *file, void *dirent, filldir_t filldir)`

该函数返回目录列表中的下一个目录。由系统调用 `readdir()` 调用它。

- `unsigned int poll(struct file *file,
 struct poll_table_struct *poll_table)`

该函数睡眠等待给定文件活动。由系统调用 `poll()` 调用它。

- `int ioctl(struct inode *inode,
 struct file *file,
 unsigned int cmd,
 unsigned long arg)`

该函数用来给设备发送命令参数对。当文件是一个被打开的设备节点时，可以通过它进行设置操作。由系统调用 `ioctl()` 调用它。调用者必须持有 BKL。

- `int unlocked_ioctl(struct file *file,
 unsigned int cmd,
 unsigned long arg)`

其实现与 `ioctl()` 有类似的功能，只不过不需要调用者持有 BKL。如果用户空间调用 `ioctl()` 系统调用，VFS 便可以调用 `unlocked_ioctl()`（凡是 `ioctl()` 出现的场所）。因此文件系统只需要实现其中的一个，一般优先实现 `unlocked_ioctl()`。

- `int compat_ioctl(struct file *file,
 unsigned int cmd,
 unsigned long arg)`

该函数是 `ioctl()` 函数的可移植变种，被 32 位应用程序用在 64 位系统上。这个函数被设计成即使在 64 位的体系结构上对 32 位也是安全的，它可以进行必要的字大小转换。新的驱动程序应该设计自己的 `ioctl` 命令以便所有的驱动程序都是可移植的，从而使得 `compat_ioctl()` 和 `unlocked_ioctl()` 指向同一个函数。像 `compat_ioctl()` 和 `unlocked_ioctl()` 一样都不必持有 BKL。

- `int mmap(struct file *file, struct vm_area_struct *vma)`

该函数将给定的文件映射到指定的地址空间上。由系统调用 `mmap()` 调用它。

- `int open(struct inode *inode,
 struct file *file)`

该函数创建一个新的文件对象，并将它和相应的索引节点对象关联起来。由系统调用 `open()` 调用它。

- `int flush(struct file *file)`

当已打开文件的引用计数减少时，该函数被 VFS 调用。它的作用根据具体文件系统而定。

- `int release(struct inode *inode,
 struct file *file)`

当文件的最后一个引用被注销时（比如，当最后一个共享文件描述符的进程调用了 `close()` 或退出时），该函数会被 VFS 调用。它的作用根据具体文件系统而定。

- `int fsync(struct file *file,
 struct dentry *dentry,
 int datasync)`

将给定文件的所有被缓存数据写回磁盘。由系统调用 `fsync()` 调用它。

- `int aio_fsync(struct kiocb *iocb,
 int datasync)`

将 `iocb` 描述的文件的所有被缓存数据写回到磁盘。由系统调用 `aio_fsync()` 调用它。

- `int fasync(int fd, struct file *file, int on)`

该函数用于打开或关闭异步 I/O 的通告信号。

- `int lock(struct file *file, int cmd, struct file_lock *lock)`

该函数用于给指定文件上锁。

- `ssize_t readv(struct file *file,
 const struct iovec *vector,
 unsigned long count,
 loff_t *offset)`

该函数从给定文件中读取数据，并将其写入由 `vector` 描述的 `count` 个缓冲中去，同时增加文件的偏移量。由系统调用 `readv()` 调用它。

- `ssize_t writev(struct file *file,
 const struct iovec *vector,
 unsigned long count,
 loff_t *offset)`

该函数将由 `vector` 描述的 `count` 个缓冲中的数据写入到由 `file` 指定的文件中去，同时减小文件的偏移量。由系统调用 `writev()` 调用它。

- `ssize_t sendfile(struct file *file,
 loff_t *offset,
 size_t size,
 read_actor_t actor,
 void *target)`

该函数用于从一个文件拷贝数据到另一个文件中，它执行的拷贝操作完全在内核中完成，避免了向用户空间进行不必要的拷贝。由系统调用 `sendfile()` 调用它。

- `ssize_t sendpage(struct file *file,
 struct page *page,
 int offset, size_t size,
 loff_t *pos, int more)`

该函数用来从一个文件向另一个文件发送数据。

- unsigned long get_unmapped_area(struct file *file,
 unsigned long addr,
 unsigned long len,
 unsigned long offset,
 unsigned long flags)

该函数用于获取未使用的地址空间来映射给定的文件。

- int check_flags(int flags)

当给出 SETFL 命令时，这个函数用来检查传递给 fcntl() 系统调用的 flags 的有效性。与大多数 VFS 操作一样，文件系统不必实现 check_flags()——目前，只有在 NFS 文件系统中实现了。这个函数能使文件系统限制无效的 SETFL 标志，不进行限制的话，普通的 fcntl() 函数能使标志生效。在 NFS 文件系统中，不允许把 O_APPEND 和 O_DIRECT 相结合。

- int flock(struct file *filp,
 int cmd,
 struct file_lock *fl)

这个函数用来实现 flock() 系统调用，该调用提供忠告锁。

如此之多的 ioctl

不久之前，只有一个单独的 ioctl 方法。如今，有三个相关的方法。unlocked_ioctl() 和 ioctl 相同，不过前者在无大内核锁（BKL）情况下被调用。因此函数的作者必须确保适当的同步。因为大内核锁是粗粒度、低效的锁，驱动程序应当实现 unlocked_ioctl() 而不是 ioctl()。

compat_ioctl() 也在无大内核锁的情况下被调用，但是它的目的是为 64 位的系统提供 32 位 ioctl 的兼容方法。至于你如何实现它取决于现有的 ioctl 命令。早期的驱动程序隐含有确定大小的类型（如 long），应该实现适用于 32 位应用的 compat_ioctl() 方法。这通常意味着把 32 位值转换为 64 位内核中合适的类型。新驱动程序重新设计 ioctl 命令，应该确保所有的参数和数据都有明确大小的数据类型，在 32 位系统上运行 32 位应用是安全的，在 64 位系统上运行 32 位应用也是安全的，在 64 位系统上运行 64 位应用更是安全的。这些驱动程序可以让 compat_ioctl() 函数指针和 unlocked_ioctl() 函数指针指向同一函数。

13.13 和文件系统相关的数据结构

除了以上几种 VFS 基础对象外，内核还使用了另外一些标准数据结构来管理文件系统的其他相关数据。第一个对象是 file_system_type，用来描述各种特定文件系统类型，比如 ext3、ext4 或 UDF。第二个结构体是 vfsmount，用来描述一个安装文件系统的实例。

因为 Linux 支持众多不同的文件系统，所以内核必须由一个特殊的结构来描述每种文件系统的功能和行为。file_system_type 结构体被定义在 <linux/fs.h> 中，具体实现如下：

```
struct file_system_type {
    const char      *name;          /* 文件系统的名字 */
    int             fs_flags;       /* 文件系统类型标志 */
};
```

```

/* 下面的函数用来从磁盘中读取超级块 */
struct super_block      *(*get_sb)      (struct file_system_type *, int,
                                          char *, void *);

/* 下面的函数用来终止访问超级块 */
void                    (*kill_sb)      (struct super_block *);

struct module           *owner;         /* 文件系统模块 */
struct file_system_type *next;         /* 链表中下一个文件系统类型 */
struct list_head        fs_supers;     /* 超级块对象链表 */

/* 剩下的几个字段运行时使锁生效 */
struct lock_class_key   s_lock_key;
struct lock_class_key   s_umount_key;
struct lock_class_key   i_lock_key;
struct lock_class_key   i_mutex_key;
struct lock_class_key   i_mutex_dir_key;
struct lock_class_key   i_alloc_sem_key;
};

```

get_sb() 函数从磁盘上读取超级块，并且在文件系统被安装时，在内存中组装超级块对象。剩余的函数描述文件系统的属性。

每种文件系统，不管有多少个实例安装到系统中，还是根本就没有安装到系统中，都只有一个 file_system_type 结构。

更有趣的事情是，当文件系统被实际安装时，将有一个 vfsmount 结构体在安装点被创建。该结构体用来代表文件系统的实例——换句话说，代表一个安装点。

vfsmount 结构被定义在 <linux/mount.h> 中，下面是具体结构：

```

struct vfsmount {
    struct list_head    mnt_hash;           /* 散列表 */
    struct vfsmount     *mnt_parent;        /* 父文件系统 */
    struct dentry        *mnt_mountpoint;   /* 安装点的目录项 */
    struct dentry        *mnt_root;         /* 该文件系统的根目录项 */
    struct super_block   *mnt_sb;           /* 该文件系统的超级块 */
    struct list_head     mnt_mounts;        /* 子文件系统链表 */
    struct list_head     mnt_child;         /* 子文件系统链表 */
    int                  mnt_flags;         /* 安装标志 */
    char                 *mnt_devname;      /* 设备文件名 */
    struct list_head     mnt_list;          /* 描述符链表 */
    struct list_head     mnt_expire;        /* 在到期链表中的入口 */
    struct list_head     mnt_share;         /* 在共享安装链表中的入口 */
    struct list_head     mnt_slave_list;    /* 从安装链表 */
    struct list_head     mnt_slave;        /* 从安装链表中的入口 */
    struct vfsmount     *mnt_master;       /* 从安装链表的主人 */
    struct mnt_namespace *mnt_namespace;   /* 相关的命名空间 */
    int                  mnt_id;            /* 安装标识符 */
    int                  mnt_group_id;     /* 组标识符 */
    atomic_t             mnt_count;        /* 使用计数 */
    int                  mnt_expiry_mark;  /* 如果标记为到期，则值为真 */
    int                  mnt_pinned;       /* “钉住”进程计数 */
    int                  mnt_ghosts;       /* “镜像”引用计数 */
    atomic_t             __mnt_writers;    /* 写者引用计数 */
};

```

理清文件系统和所有其他安装点间的关系，是维护所有安装点链表中最复杂的工作。所以

vfsmount 结构体中维护的各种链表就是为了能够跟踪这些关联信息。

vfsmount 结构还保存了在安装时指定的标志信息，该信息存储在 mnt_flags 域中。表 13-1 列出了标准的安装标志。

表 13-1 标准安装标志列表

标 志	描 述
MNT_NOSUID	禁止该文件系统的可执行文件设置 setuid 和 setgid 标志
MNT_MODEV	禁止访问该文件系统上的设备文件
MNT_NOEXEC	禁止执行该文件系统上的可执行文件

安装那些管理员不充分信任的移动设备时，这些标志很有用处。它们和其他一些很少用的标志一起定义在 <linux/mount.h> 中。

13.14 和进程相关的数据结构

系统中的每一个进程都有自己的一组打开的文件，像根文件系统、当前工作目录、安装点等。有三个数据结构将 VFS 层和系统的进程紧密联系在一起，它们分别是：file_struct、fs_struct 和 namespace 结构体。

file_struct 结构体定义在文件 <linux/fdtable.h> 中。该结构体由进程描述符中的 files 目录项指向。所有与单个进程（per-process）相关的信息（如打开的文件及文件描述符）都包含在其中，其结构和描述如下：

```
struct files_struct {
    atomic_t          count;                /* 结构的使用计数 */
    struct fdtable     *fdt;                /* 指向其他 fd 表的指针 */
    struct fdtable     fdtab;               /* 基 fd 表 */
    spinlock_t        file_lock;           /* 单个文件的锁 */
    int                next_fd;             /* 缓存下一个可用的 fd */
    struct embedded_fd_set close_on_exec_init; /* exec() 时关闭的文件描述符链表 */
    struct embedded_fd_set open_fds_init;    /* 打开的文件描述符链表 */
    struct file        *fd_array[NR_OPEN_DEFAULT]; /* 缺省的文件对象数组 */
};
```

fd_array 数组指针指向已打开的文件对象。因为 NR_OPEN_DEFAULT 等于 BITS_PER_LONG，在 64 位机器体系结构中这个宏的值为 64，所以该数组可以容纳 64 个文件对象。如果一个进程所打开的文件对象超过 64 个，内核将分配一个新数组，并且将 fdt 指针指向它。所以对适当数量的文件对象的访问会执行得很快，因为它是对静态数组进行的操作；如果一个进程打开的文件数量过多，那么内核就需要建立新数组。所以如果系统中有大量的进程都要打开超过 64 个文件，为了优化性能，管理员可以适当增大 NR_OPEN_DEFAULT 的预定义值。

和进程相关的第二个结构体是 fs_struct。该结构由进程描述符的 fs 域指向。它包含文件系统和进程相关的信息，定义在文件 <linux/fs_struct.h> 中，下面是它的具体结构体和各项描述：

```
struct fs_struct {
    int                users;               /* 用户数目 */
    rwlock_t          lock;                /* 保护该结构体的锁 */
};
```

```

int          umask;          /* 掩码 */
int          in_exec;        /* 当前正在执行的文件 */
struct path root;           /* 根目录路径 */
struct path pwd;            /* 当前工作目录的路径 */
};

```

该结构包含了当前进程的当前工作目录 (pwd) 和根目录。

第三个也是最后一个相关结构体是 namespace 结构体。它定义在文件 <linux/mmt_namespace.h> 中, 由进程描述符中的 mmt_namespace 域指向。2.4 版内核以后, 单进程命名空间被加入到内核中, 它使得每一个进程在系统中都看到唯一的安装文件系统——不仅是唯一的根目录, 而且是唯一的文件系统层次结构。下面是其具体结构和描述:

```

struct mmt_namespace {
    atomic_t          count; /* 结构的使用计数 */
    struct vfsmount    *root; /* 根目录的安装点对象 */
    struct list_head    list; /* 安装点链表 */
    wait_queue_head_t poll; /* 轮询的等待队列 */
    int                event; /* 事件计数 */
};

```

list 域是连接已安装文件系统的双向链表, 它包含的元素组成了全体命名空间。

上述这些数据结构都是通过进程描述符连接起来的。对多数进程来说, 它们的描述符都指向唯一的 files_struct 和 fs_struct 结构体。但是, 对于那些使用克隆标志 CLONE_FILES 或 CLONE_FS 创建的进程, 会共享^①这两个结构体。所以多个进程描述符可能指向同一个 files_struct 或 fs_struct 结构体。每个结构体都维护一个 count 域作为引用计数, 它防止在进程正使用该结构时, 该结构被撤销。

namespace 结构体的使用方法却和前两种结构体完全不同, 默认情况下, 所有的进程共享同样的命名空间 (也就是, 它们都从相同的挂载表中看到同一个文件系统层次结构)。只有在进行 clone() 操作时使用 CLONE_NEWNS 标志, 才会给进程一个唯一的命名空间结构体的拷贝。因为大多数进程不提供这个标志, 所有进程都继承其父进程的命名空间。因此, 在大多数系统上只有一个命名空间, 不过, CLONE_NEWNS 标志可以使这一功能失效。

13.15 小结

Linux 支持了相当多种类的文件系统。从本地文件系统 (如 ext3 和 ext4) 到网络文件系统 (如 NFS 和 Coda), Linux 在标准内核中已支持的文件系统超过 60 种。VFS 层提供给这些不同文件系统一个统一的实现框架, 而且也提供了能和标准系统调用交互工作的统一接口。由于 VFS 层的存在, 使得在 Linux 上实现新文件系统的工作变得简单起来, 它可以轻松地使这些文件系统通过标准 Unix 系统调用而协同工作。

本章描述了 VFS 的目的, 讨论了各种数据结构, 包括最重要的索引节点、目录项以及超级块对象。第 14 章将讨论数据如何从物理上存放在文件系统中。

① 线程通常在创建时使用 CLONE_FILES 和 CLONE_FS 标志, 所以多个线程共享一个 file_struct 结构体和 fs_struct 结构体。但另一方面, 普通进程没有指定这些标志, 所以它们有自己的文件系统信息和打开文件表。

第 14 章

块 I/O 层

系统中能够随机（不需要按顺序）访问固定大小数据片（chunks）的硬件设备称作块设备，这些固定大小的数据片就称作块。最常见的块设备是硬盘，除此以外，还有软盘驱动器、蓝光光驱和闪存等许多其他块设备。注意，它们都是以安装文件系统的方式使用的——这也是块设备一般的访问方式。

另一种基本的设备类型是字符设备。字符设备按照字符流的方式被有序访问，像串口和键盘就属于字符设备。如果一个硬件设备是以字符流的方式被访问的话，那就应该将它归于字符设备；反过来，如果一个设备是随机（无序的）访问的，那么它就属于块设备。

对于这两种类型的设备，它们的区别在于是否可以随机访问数据——换句话说，就是能否在访问设备时随意地从一个位置跳转到另一个位置。举个例子，键盘这种设备提供的就是一个数据流，当你输入“wolf”这个字符串时，键盘驱动程序会按照和输入完全相同的顺序返回这个由四个字符组成的数据流。如果让键盘驱动程序打乱顺序来读字符串，或读取其他字符，都是没有意义的。所以键盘就是一种典型的字符设备，它提供的就是用户从键盘输入的字符流。对键盘进行读操作会得到一个字符流，首先是“w”，然后是“o”，再是“l”，最后是“x”。当没人敲键盘时，字符流就是空的。硬盘设备的情况就不大一样了。硬盘设备的驱动可能要求读取磁盘上任意块的内容，然后又转去读取别的块的内容，而被读取的块在磁盘上位置不一定要连续。所以说硬盘的数据可以被随机访问，而不是以流的方式被访问，因此它是一个块设备。

内核管理块设备要比管理字符设备细致得多，需要考虑的问题和完成的工作相对于字符设备来说要复杂许多。这是因为字符设备仅仅需要控制一个位置——当前位置，而块设备访问的位置必须能够在介质的不同区间前后移动。所以事实上内核不必提供一个专门的子系统来管理字符设备，但是对块设备的管理却必须要有一个专门的提供服务的子系统。不仅仅是因为块设备的复杂性远远高于字符设备，更重要的原因是块设备对执行性能的要求很高；对硬盘每多一份利用都会对整个系统的性能带来提升，其效果要远远比键盘吞吐速度成倍的提高大得多。另外，我们将会看到，块设备的复杂性会为这种优化留下很大的施展空间。这一章的主题就是讨论内核如何对块设备和块设备的请求进行管理。该部分在内核中称作块 I/O 层。有趣的是，改写块 I/O 层正是 2.5 开发版内核的主要目标。本章涵盖了 2.6 版内核中所有新的块 I/O 层。

14.1 剖析一个块设备

块设备中最小的可寻址单元是扇区。扇区大小一般是 2 的整数倍，而最常见的是 512 字节。扇区的大小是设备的物理属性，扇区是所有块设备的基本单元——块设备无法对比它还小的单元进行寻址和操作，尽管许多块设备能够一次对多个扇区进行操作。虽然大多数块设备的扇区大小

都是 512 字节，不过其他大小的扇区也很常见。比如，很多 CD-ROM 盘的扇区都是 2KB 大小。

因为各种软件的用途不同，所以它们都会用到自己的最小逻辑可寻址单元——块。块是文件系统的一种抽象——只能基于块来访问文件系统。虽然物理磁盘寻址是按照扇区级进行的，但是内核执行的所有磁盘操作都是按照块进行的。由于扇区是设备的最小可寻址单元，所以块不能比扇区还小，只能数倍于扇区大小。另外，内核（对有扇区的硬件设备）还要求块大小是 2 的整数倍，而且不能超过一个页的长度（请看第 12 章和第 19 章）^①。所以，对块大小的最终要求是，必须是扇区大小的 2 的整数倍，并且要小于页面大小。所以通常块大小是 512 字节、1KB 或 4KB。

扇区和块还有一些不同的叫法，为了不引起混淆，我们在这里简要介绍一下它们的其他名称。扇区——设备的最小寻址单元，有时会称作“硬扇区”或“设备块”；同样的，块——文件系统的最小寻址单元，有时会称作“文件块”或“I/O 块”。在这一章里，会一直使用“扇区”和“块”这两个术语，但你还是应该记住它们的这些别名。图 14-1 是扇区和缓冲区之间的关系图。

至少相对于硬盘而言，另外一些术语更通用——如簇、柱面以及磁头。这些表示是针对某些特定的块设备的，大多数情况下，对用户空间的软件是不可见的。扇区这一术语之所以对内核重要，是因为所有设备的 I/O 必须以扇区为单位进行操作。以此类推，内核所使用的“块”这一高级概念就是建立在扇区之上的。

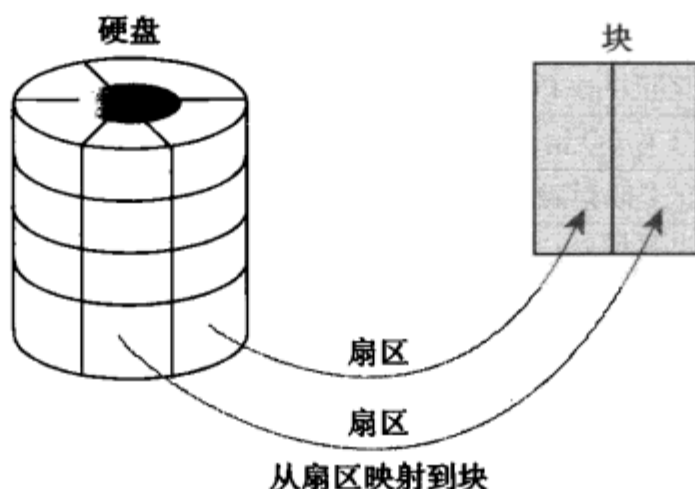


图 14-1 扇区和缓冲区之间的关系

14.2 缓冲区和缓冲区头

当一个块被调入内存时（也就是说，在读入后或等待写出时），它要存储在一个缓冲区中。每个缓冲区与一个块对应，它相当于是磁盘块在内存中的表示。前面提到过，块包含一个或多个扇区，但大小不能超过一个页面，所以一个页可以容纳一个或多个内存中的块。由于内核在处理数据时需要一些相关的控制信息（比如块属于哪一个块设备，块对应于哪个缓冲区等），所以每一个缓冲区都有一个对应的描述符。该描述符用 `buffer_head` 结构体表示，称作缓冲区头，在文件 `<linux/buffer_head.h>` 中定义，它包含了内核操作缓冲区所需要的全部信息。

^① 这个认为的限制可能会遗留到以后，但是强制块的大小等于或小于页大小无疑简化了内核。

下面给出缓冲区头结构体和其中各个域的说明：

```
struct buffer_head {
    unsigned long b_state;           /* 缓冲区状态标志 */
    struct buffer_head *b_this_page; /* 页面中的缓冲区 */
    struct page *b_page;             /* 存储缓冲区的页面 */
    sector_t b_blocknr;             /* 起始块号 */
    size_t b_size;                  /* 映像的大小 */
    char *b_data;                   /* 页面内的数据指针 */
    struct block_device *b_bdev;     /* 相关联的块设备 */
    bh_end_io_t *b_end_io;          /* I/O 完成方法 */
    void *b_private;                /* io 完成方法 */
    struct list_head b_assoc_buffers; /* 相关的映射链表 */
    struct address_space *b_assoc_map; /* 相关的地址空间 */
    atomic_t b_count;               /* 缓冲区使用计数 */
};
```

b_state 域表示缓冲区的状态，可以是表 14-1 中一种标志或多种标志的组合。合法的标志存放在 bh_state_bits 枚举中，该枚举在 <linux/buffer_head.h> 中定义。

表 14-1 bh_state 标志

状态标志	意 义
BH_Uptodate	该缓冲区包含可用数据
BH_Dirty	该缓冲区是脏的（缓存中的内容比磁盘中的块内容新，所以缓冲区内容必须被写回磁盘）
BH_Lock	该缓冲区正在被 I/O 操作使用，被锁定以防被并发访问
BH_Req	该缓冲区有 I/O 请求操作
BH_Mapped	该缓冲区是映射磁盘块的可用缓冲区
BH_New	缓冲区是通过 get_block() 刚刚映射的，尚且不能访问
BH_Async_Read	该缓冲区正通过 end_buffer_async_read() 被异步 I/O 读操作使用
BH_Async_write	该缓冲区正通过 end_buffer_async_write() 被异步 I/O 写操作使用
BH_Delay	该缓冲区尚未和磁盘块关联
BH_Boundary	该缓冲区处于连续块区的边界——下一个块不再连续
BH_Write_EIO	该缓冲区在写的时候遇到 I/O 错误
BH_Ordered	顺序写
BH_Eopnotsupp	该缓冲区发生“不被支持”错误
BH_Unwritten	该缓冲区在硬盘上的空间已被申请但是没有实际的数据写出
BH_Quiet	此缓冲区禁止错误

bh_state_bits 列表还包含了一个特殊标志——BH_PrivateStart，该标志不是可用状态标志，使用它是为了指明可被其他代码使用的起始位。块 I/O 层不会使用 BH_PrivateStart 或更高的位。那么某个驱动程序希望通过 b_state 域存储信息时就可以安全地使用这些位。驱动程序可以在这些位中定义自己的状态标志，只要保证自定义的状态标志不与块 I/O 层的专用位发生冲突就可以了。

b_count 域表示缓冲区的使用记数，可通过两个定义在文件 <linux/buffer_head.h> 中的内联函数对此域进行增减。

```
static inline void get_bh(struct buffer_head *bh)
{
    atomic_inc(&bh->b_count);
}

static inline void put_bh(struct buffer_head *bh)
{
    atomic_dec(&bh->b_count);
}
```

在操作缓冲区头之前，应该先使用 `get_bh()` 函数增加缓冲区头的引用计数，确保该缓冲区头不会再被分配出去；当完成对缓冲区头的操作之后，还必须使用 `put_bh()` 函数减少引用计数。

与缓冲区对应的磁盘物理块由 `b_blocknr-th` 域索引，该值是 `b_bdev` 域指明的块设备中的逻辑块号。

与缓冲区对应的内存物理页由 `b_page` 域表示，另外，`b_data` 域直接指向相应的块（它位于 `b_page` 域所指明的页面中的某个位置上），块的大小由 `b_size` 域表示，所以块在内存中的起始位置在 `b_data` 处，结束位置在 `(b_data + b_size)` 处。

缓冲区头的目的在于描述磁盘块和物理内存缓冲区（在特定页面上的字节序列）之间的映射关系。这个结构体在内核中只扮演一个描述符的角色，说明从缓冲区到块的映射关系。

在 2.6 内核以前，缓冲区头的作用比现在还要重要。因为缓冲区头作为内核中的 I/O 操作单元，不仅仅描述了从磁盘块到物理内存的映射，而且还是所有块 I/O 操作的容器。可是，将缓冲区头作为 I/O 操作单元带来了两个弊端。首先，缓冲区头是一个很大且不易控制的数据结构体（现在是缩减过的了），而且缓冲区头对数据的操作既不方便也不清晰。对内核来说，它更倾向于操作页面结构，因为页面操作起来更为简便，同时效率也高。使用一个巨大的缓冲区头表示每一个独立的缓冲区（可能比页面小）效率低下，所以在 2.6 版本中，许多 I/O 操作都是通过内核直接对页面或地址空间进行操作来完成，不再使用缓冲区头了。这其中所做的一些工作会在第 16 章中进行讨论，具体情况请参考 `address_space` 结构和 `pdflush` 等守护进程（daemon）部分。

缓冲区头带来的第二个弊端是：它仅能描述单个缓冲区，当作为所有 I/O 的容器使用时，缓冲区头会促使内核把对大块数据的 I/O 操作（比如写操作）分解为对多个 `buffer_head` 结构体进行操作。这样做必然会造成不必要的负担和空间浪费。所以 2.5 开发版内核的主要目标就是为块 I/O 操作引入一种新型、灵活并且轻量级的容器，也就是 14.3 节要介绍的 `bio` 结构体。

14.3 bio 结构体

目前内核中块 I/O 操作的基本容器由 `bio` 结构体表示，它定义在文件 `<linux/bio.h>` 中。该结构体代表了正在现场的（活动的）以片断（segment）链表形式组织的块 I/O 操作。一个片段是一小块连续的内存缓冲区。这样的话，就不需要保证单个缓冲区一定要连续。所以通过用片段来描述缓冲区，即使一个缓冲区分散在内存的多个位置上，`bio` 结构体也能对内核保证 I/O 操作的执行。像这样的向量 I/O 就是所谓的聚散 I/O。

`bio` 结构体定义于 `<linux/bio.h>` 中，下面给出 `bio` 结构体和各个域的描述。

```

struct bio {
    sector_t                bi_sector;           /* 磁盘上相关的扇区 */
    struct bio              *bi_next;           /* 请求链表 */
    struct block_device      *bi_bdev;          /* 相关的块设备 */
    unsigned long            bi_flags;          /* 状态和命令标志 */
    unsigned long            bi_rw;             /* 读还是写 */
    unsigned short           bi_vcnt;           /* bio_vecs 偏移的个数 */
    unsigned short           bi_idx;            /* bio_io_vect 的当前索引 */
    unsigned short           bi_phys_segments;  /* 结合后的片断数目 */
    unsigned int             bi_size;           /* I/O 计数 */
    unsigned int             bi_seg_front_size; /* 第一个可合并的段大小 */
    unsigned int             bi_seg_back_size;  /* 最后一个可合并的段大小 */
    unsigned int             bi_max_vecs;       /* bio_vecs 数目上限 */
    unsigned int             bi_comp_cpu;       /* 结束 CPU */
    atomic_t                 bi_cnt;            /* 使用计数 */
    struct bio_vec            *bi_io_vec;       /* bio_vecs 链表 */
    bio_end_io_t             *bi_end_io;        /* I/O 完成方法 */
    void                     *bi_private;       /* 拥有者的私有方法 */
    bio_destructor_t         *bi_destructor;    /* 撤销方法 */
    struct bio_vec            bi_inline_vecs[0]; /* 内嵌 bio 向量 */
};

```

使用 bio 结构体的目的主要是代表正在现场执行的 I/O 操作，所以该结构体中的主要域都是用来管理相关信息的，其中最重要的几个域是 bi_io_vecs、bi_vcnt 和 bi_idx。图 14-2 显示了 bio 结构体及其他结构体之间的关系。

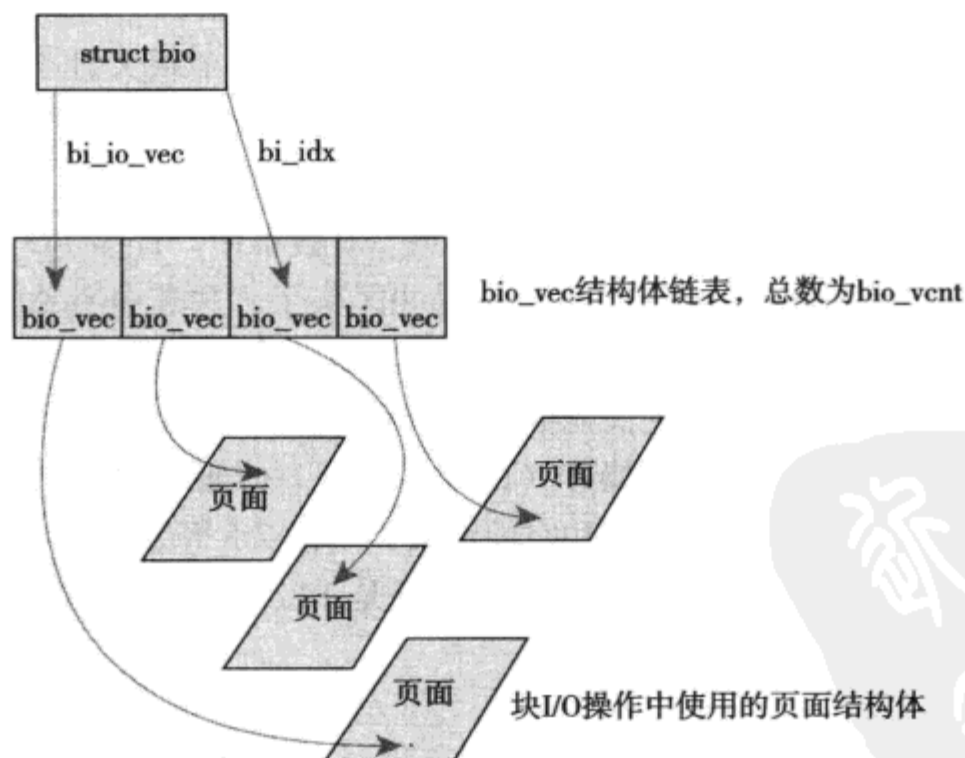


图 14-2 bio 结构体、bio_vec 结构体和 page 结构体之间的关系

14.3.1 I/O 向量

bi_io_vec 域指向一个 bio_vec 结构体数组，该结构体链表包含了一个特定 I/O 操作所需要使用的片断。每个 bio_vec 结构都是一个形式为 <page, offset, len> 的向量，它描述的是一个特定的片断：片断所在的物理页、块在物理页中的偏移位置、从给定偏移量开始的块长度。整个

bio_io_vec 结构体数组表示了一个完整的缓冲区。bio_vec 结构定义在 <linux/bio.h> 文件中：

```
struct bio_vec {
    /* 指向这个缓冲区所驻留的物理页 */
    struct page    *bv_page;

    /* 这个缓冲区以字节为单位的大小 */
    unsigned int    bv_len;

    /* 缓冲区所驻留的页中以字节为单位的偏移量 */
    unsigned int    bv_offset;
};
```

在每个给定的块 I/O 操作中，bi_vcnt 域用来描述 bi_io_vec 所指向的 bio_vec 数组中的向量数目。当块 I/O 操作执行完毕后，bi_idx 域指向数组的当前索引。

总而言之，每一个块 I/O 请求都通过一个 bio 结构体表示。每个请求包含一个或多个块，这些块存储在 bio_vec 结构体数组中。这些结构体描述了每个片段在物理页中的实际位置，并且像向量一样被组织在一起。I/O 操作的第一个片段由 b_io_vec 结构体所指向，其他的片段在其后依次放置，共有 bi_vcnt 个片段。当块 I/O 层开始执行请求、需要使用各个片段时，bi_idx 域会不断更新，从而总指向当前片段。

bi_idx 域指向数组中的当前 bio_vec 片段，块 I/O 层通过它可以跟踪块 I/O 操作的完成进度。但该域更重要的作用在于分割 bio 结构体。像冗余廉价磁盘阵列 (RAID，出于提高性能和可靠性的目的，将单个磁盘的卷扩展到多个磁盘上) 这样的驱动器可以把单独的 bio 结构体 (原本是为单个设备使用准备的)，分割到 RAID 阵列中的各个硬盘上去。RAID 设备驱动只需要拷贝这个 bio 结构体，再把 bi_idx 域设置为每个独立硬盘操作时需要的位置就可以了。

bi_cnt 域记录 bio 结构体的使用计数，如果该域值减为 0，就应该撤销该 bio 结构体，并释放它占用的内存。通过下面两个函数管理使用计数。

```
void bio_get(struct bio *bio)
void bio_put(struct bio *bio)
```

前者增加使用计数，后者减少使用计数 (如果计数减到 0，则撤销 bio 结构体)。在操作正在活动的 bio 结构体时，一定要首先增加它的使用计数，以免在操作过程中该 bio 结构体被释放；相反，在操作完毕后，要减少使用计数。

最后要说明的是 bi_private 域，这是一个属于拥有者 (也就是创建者) 的私有域，只有创建了 bio 结构的拥有者可以读写该域。

14.3.2 新老方法对比

缓冲区头和新的 bio 结构体之间存在显著差别。bio 结构体代表的是 I/O 操作，它可以包括内存中的一个或多个页；而另一方面，buffer_head 结构体代表的是一个缓冲区，它描述的仅仅是磁盘中的一个块。因为缓冲区头关联的是单独页中的单独磁盘块，所以它可能会引起不必要的分割，将请求按块为单位划分，只能靠以后才能再重新组合。由于 bio 结构体是轻量级的，它描述的块可以不需要连续存储区，并且不需要分割 I/O 操作。

利用 bio 结构体代替 buffer_head 结构体还有以下好处：

- bio 结构体很容易处理高端内存，因为它处理的是物理页而不是直接指针。
- bio 结构体既可以代表普通页 I/O，同时也可以代表直接 I/O（指那些不通过页高速缓存的 I/O 操作——请参考第 16 章中对页高速缓存的讨论）。
- bio 结构体便于执行分散-集中（矢量化的）块 I/O 操作，操作中的数据可取自多个物理页面。
- bio 结构体相比缓冲区头属于轻量级的结构体。因为它只需要包含块 I/O 操作所需的信息就行了，不用包含与缓冲区本身相关的不必要信息。

但是还是需要缓冲区头这个概念，毕竟它还负责描述磁盘块到页面的映射。bio 结构体不包含任何和缓冲区相关的状态信息——它仅仅是一个矢量数组，描述一个或多个单独块 I/O 操作的数据片段和相关信息。在当前设置中，当 bio 结构体描述当前正在使用的 I/O 操作时，buffer_head 结构体仍然需要包含缓冲区信息。内核通过这两种结构分别保存各自的信息，可以保证每种结构所含的信息量尽可能地少。

14.4 请求队列

块设备将它们挂起的块 I/O 请求保存在请求队列中，该队列由 request_queue 结构体表示，定义在文件 <linux/blkdev.h> 中，包含一个双向请求链表以及相关控制信息。通过内核中像文件系统这样高层的代码将请求加入到队列中。请求队列只要不为空，队列对应的块设备驱动程序就会从队列头获取请求，然后将其送入对应的块设备上去。请求队列表中的每一项都是一个单独的请求，由 request 结构体表示。

队列中的请求由结构体 request 表示，它定义在文件 <linux/blkdev.h> 中。因为一个请求可能要操作多个连续的磁盘块，所以每个请求可以由多个 bio 结构体组成。注意，虽然磁盘上的块必须连续，但是在内存中这些块并不一定要连续——每个 bio 结构体都可以描述多个片段（回忆一下，片段是内存中连续的小区域），而每个请求也可以包含多个 bio 结构体。

14.5 I/O 调度程序

如果简单地以内核产生请求的次序直接将请求发向块设备的话，性能肯定让人难以接受。磁盘寻址是整个计算机中最慢的操作之一，每一次寻址（定位硬盘磁头到特定块上的某个位置）需要花费不少时间。所以尽量缩短寻址时间无疑是提高系统性能的关键。

为了优化寻址操作，内核既不会简单地按请求接收次序，也不会立即将其提交给磁盘。相反，它会在提交前，先执行名为合并与排序的预操作，这种预操作可以极大地提高系统的整体性能^①。在内核中负责提交 I/O 请求的子系统称为 I/O 调度程序。

I/O 调度程序将磁盘 I/O 资源分配给系统中所有挂起的块 I/O 请求。具体地说，这种资源分配是通过将请求队列中挂起的请求合并和排序来完成的。注意不要将 I/O 调度程序和进程调度程序（请看第 4 章）混淆。进程调度程序的作用是将处理器资源分配给系统中的运行进程。这两种

① 这一点需要强调。如果一个系统没有这些功能，或者这些功能实现得很差，那么即使是数量不大的块 I/O 操作，执行性能也会很糟糕。

子系统看起来非常相似，但并不相同。进程调度程序和 I/O 调度程序都是将一个资源虚拟给多个对象，对进程调度程序来说，处理器被虚拟并被系统中的运行进程共享。这种虚拟提供给用户的就是多任务和分时操作系统，像 Unix 系统。相反，I/O 调度程序虚拟块设备给多个磁盘请求，以便降低磁盘寻址时间，确保磁盘性能的最优化。

14.5.1 I/O 调度程序的工作

I/O 调度程序的工作是管理块设备的请求队列。它决定队列中的请求排列顺序以及在什么时候派发请求到块设备。这样做有利于减少磁盘寻址时间，从而提高全局吞吐量。注意，全局这个定语很重要，坦率地讲，一个 I/O 调度器可能为了提高系统整体性能，而对某些请求不公。

I/O 调度程序通过两种方法减少磁盘寻址时间：合并与排序。合并指将两个或多个请求结合成一个新请求。考虑一下这种情况，文件系统提交请求到请求队列——从文件中读取一个数据区（当然，最终所有的操作都是针对扇区和块进行的，而不是文件，还假定请求的块都是来自文件块），如果这时队列中已经存在一个请求，它访问的磁盘扇区和当前请求访问的磁盘扇区相邻（比如，同一个文件中早些时候被读取的数据区），那么这两个请求就可以合并为一个对单个和多个相邻磁盘扇区操作的新请求。通过合并请求，I/O 调度程序将多次请求的开销压缩成一次请求的开销。更重要的是，请求合并后只需要传递给磁盘一条寻址命令，就可以访问到请求合并前必须多次寻址才能访问完的磁盘区域了，因此合并请求显然能减少系统开销和磁盘寻址次数。

现在，假设在读请求被提交给请求队列的时候，队列中并不需要操作相邻扇区的其他请求，此时就无法将当前请求与其他请求合并，当然，可以将其插入请求队列的尾部。但是如果有其他请求需要操作磁盘上类似的位置呢？如果存在一个请求，它要操作的磁盘扇区位置与当前请求比较接近，那么是不是该让这两个请求在请求队列上也相邻呢？事实上，I/O 调度程序的确是这样处理上述情况的，整个请求队列将按扇区增长方向有序排列。使所有请求按硬盘上扇区的排列顺序有序排列（尽可能的）的目的不仅是为了缩短单独一次请求的寻址时间，更重要的优化在于，通过保持磁盘头以直线方向移动，缩短了所有请求的磁盘寻址时间。该排序算法类似于电梯调度——电梯不能随意地从一层跳到另一层，它应该向一个方向移动，当抵达了同一方向上的最后一层后，再掉头向另一个方向移动。出于这种相似性，所以 I/O 调度程序（或这种排序算法）称作电梯调度。

14.5.2 Linux 电梯

下面看看 Linux 中实际使用的 I/O 调度程序。我们看到的第一个 I/O 调度程序称为 Linux 电梯（没错，Linux 确实是用他的名字命名了这个电梯）。在 2.4 版内核中，Linux 电梯是默认的 I/O 调度程序。虽然后来在 2.6 版内核中它被另外两种调度程序取代了，但是由于这个电梯比后来的调度程序简单，而且它们执行的许多功能都相似，所以它可以作为一个优秀的入门介绍程序。

Linux 电梯能执行合并与排序预处理。当有新的请求加入队列时，它首先会检查其他每一个挂起的请求是否可以和新请求合并。Linux 电梯 I/O 调度程序可以执行向前和向后合并，合并类型描述的是请求向前面还是向后面，这一点和已有请求相连。如果新请求正好连在一个现存的请求前，就是向前合并；相反如果新请求直接连在一个现存的请求后，就是向后合并。鉴于文件的分布（通常以扇区号的增长表现）特点和 I/O 操作执行方式具有典型性（一般都是从头读向尾，

很少从反方向读), 所以向前合并相比向后合并要少得多, 但是 Linux 电梯还是会对两种合并类型都进行检查。

如果合并尝试失败, 那么就需要寻找可能的插入点 (新请求在队列中的位置必须符合请求以扇区方向有序排序的原则)。如果找到, 新请求将被插入到该点; 如果没有合适的位置, 那么新请求就被加入到队列尾部。另外, 如果发现队列中有驻留时间过长的请求, 那么新请求也将被加入到队列尾部, 即使插入后还要排序。这样做是为了避免由于访问相近磁盘位置的请求太多, 而造成访问磁盘其他位置的请求难以得到执行机会这一问题。不幸的是, 这种“年龄”检测方法并不很有效, 因为它并非是给等待了一段时间的请求提供实质性服务, 它仅仅是在经过了一段时间后停止插入—排序请求, 这改善了等待时间但最终还是会导致请求饥饿现象的发生, 所以这是一个 2.4 内核 I/O 调度程序中必须要修改的缺陷。

总而言之, 当一个请求加入到队列中时, 有可能发生四种操作, 它们依次是:

- 1) 如果队列中已存在一个对相邻磁盘扇区操作的请求, 那么新请求将和这个已经存在的请求合并成一个请求。
- 2) 如果队列中存在一个驻留时间过长的请求, 那么新请求将被插入到队列尾部, 以防止其他旧的请求饥饿发生。
- 3) 如果队列中以扇区方向为序存在合适的插入位置, 那么新的请求将被插入到该位置, 保证队列中的请求是以被访问磁盘物理位置为序进行排列的。
- 4) 如果队列中不存在合适的请求插入位置, 请求将被插入到队列尾部。

14.5.3 最终期限 I/O 调度程序

最终期限 (deadline) I/O 调度程序是为了解决 Linux 电梯所带来的饥饿问题而提出的。出于减少磁盘寻址时间的考虑, 对某个磁盘区域上的繁重操作, 无疑会使得磁盘其他位置上的操作请求得不到运行机会。实际上, 一个对磁盘同一位置操作的请求流可以造成较远位置的其他请求永远得不到运行机会, 这是一种很不公平的饥饿现象。

更糟糕的是, 普通的请求饥饿还会带来名为写—饥饿—读 (writes-starving-reads) 这种特殊问题。写操作通常是在内核有空时才将请求提交给磁盘的, 写操作完全和提交它的应用程序异步执行; 读操作则恰恰相反, 通常当应用程序提交一个读请求时, 应用程序会发生堵塞直到读请求被满足, 也就是说, 读操作是和提交它的应用程序同步执行的。所以虽然写反应时间 (提交写请求花费的时间) 不会给系统响应速度造成很大影响, 但是读响应时间 (提交读请求花费的时间) 对系统响应时间来说却非同小可。虽然写请求时间对应用程序性能^①带来的影响不大, 但是应用程序却必须等待读请求完成后才能运行其他程序, 所以读操作响应时间对系统的性能非常重要。

问题还可能更严重, 这是因为读请求往往会相互依靠。比如, 要读大量的文件, 每次都是针对一块很小的缓冲区数据区进行读操作, 而应用程序只有将上一个数据区从磁盘中读取并返回之后, 才能继续读取下一个数据区 (或下一个文件)。糟糕的是, 不管是读还是写, 二者都需要读

① 不过, 我们还是打算把写请求无限期地延迟下去, 因为内核想确保数据最终能写到磁盘, 以避免在内存缓冲区中的数据变得越来越多或者太陈旧。

取像索引节点这样的元数据。从磁盘进一步读取这些块会使 I/O 操作串行化。所以如果每一次请求都发生饥饿现象，那么对读取文件的应用程序来说，全部延迟加起来会造成过长的等待时间，让用户无法忍受。综上所述，读操作具有同步性，并且彼此之间往往相互依靠，所以读请求响应时间直接影响系统性能，因此 2.6 版本内核新引入了最后期限 I/O 调度程序来减少请求饥饿现象，特别是读请求饥饿现象。

注意，减少请求饥饿必须以降低全局吞吐量为代价。Linux 电梯调度程序虽然也做了这样的折中，但显然不够——Linux 电梯可以提供更好的系统吞吐量（通过最小化寻址），可是它总按照扇区顺序将请求插入到队列，从不检查驻留时间过长的请求，更不会将请求插入到列队尾部，所以它虽然能让寻址时间最短，但是却会带来同样不可取的请求饥饿问题。为了避免饥饿同时提供良好的全局吞吐量，最后期限 I/O 调度程序做了更多的努力。既要尽量提高全局吞吐量，又要使请求得到公平处理，这是很困难的。

在最后期限 I/O 调度程序中，每个请求都有一个超时时间。默认情况下，读请求的超时时间是 500ms，写请求的超时时间是 5s。最后期限 I/O 调度请求类似于 Linux 电梯，也以磁盘物理位置为次序维护请求队列，这个队列称为排序队列。当一个新请求递交给排序队列时，最后期限 I/O 调度程序在执行合并和插入请求时类似于 Linux 电梯^①，但是最后期限 I/O 调度程序同时也会以请求类型为依据将它们插入到额外队列中。读请求按次序被插入到特定的读 FIFO 队列中，写请求被插入到特定的写 FIFO 队列中。虽然普通队列以磁盘扇区为序进行排列，但是这些队列是以 FIFO（很有效，以时间为基准排序）形式组织的，结果新队列总是被加入到队列尾部。对于普通操作来说，最后期限 I/O 调度程序将请求从排序队列的头部取下，再推入到派发队列中，派发队列然后将请求提交给磁盘驱动，从而保证了最小化的请求寻址。

如果在写 FIFO 队列头，或是在读 FIFO 队列头的请求超时（也就是，当前时间超过了请求指定的超时时间），那么最后期限 I/O 调度程序便从 FIFO 队列中提取请求进行服务。依靠这种方法，最后期限 I/O 调度程序试图保证不会发生有请求在明显超期的情况下仍不能得到服务的现象，参见图 14-3。

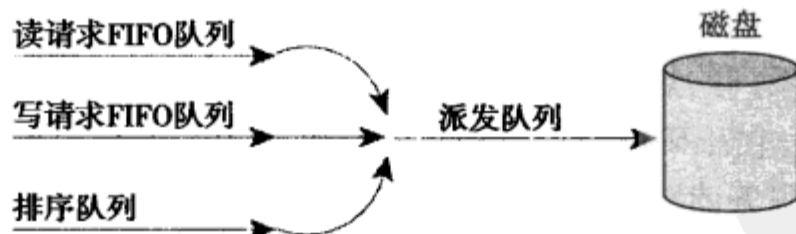


图 14-3 最后期限 I/O 调度程序的三个队列

注意，最后期限 I/O 调度算法并不能严格保证请求的响应时间，但是通常情况下，可以在请求超时或超时前提提交和执行，以防止请求饥饿现象的发生。由于读请求给定的超时时间要比写请求短许多，所以最后期限 I/O 调度器也确保了写请求不会因为堵塞读请求而使读请求发生饥饿。这种对读操作的照顾确保了读响应时间尽可能短。

^① 最后期限 I/O 排序执行向前合并是一个可选项。因为读操作请求通常很少需要向前合并，所以向前合并通常不必考虑。

最后期限 I/O 调度程序的实现在文件 `block/deadline-iosched.c` 中。

14.5.4 预测 I/O 调度程序

虽然最后期限 I/O 调度程序为降低读操作响应时间做了许多工作，但是它同时也降低了系统吞吐量。假设一个系统处于很繁重的写操作期间，每次提交读请求，I/O 调度程序都会迅速处理读请求，所以磁盘首先为读操作进行寻址，执行读操作，然后返回再寻址进行写操作，并且对每个读请求都重复这个过程。这种做法对读请求来说是件好事，但是两次寻址操作（一次对读操作定位，一次返回来进行写操作定位）却损害了系统全局吞吐量。预测（Anticipatory）I/O 调度程序的目标就是在保持良好的读响应的同时也能提供良好的全局吞吐量。

预测 I/O 调度的基础仍然是最后期限 I/O 调度程序，所以它们有很多相同之处。预测 I/O 调度程序也实现了三个队列（加上一个派发队列），并为每个请求设置了超时时间，这点与最后期限 I/O 调度程序一样。预测 I/O 调度程序最主要的改进是它增加了预测启发（anticipation-heuristic）能力。

预测 I/O 调度试图减少在进行 I/O 操作期间，处理新到的读请求所带来的寻址数量。和最后期限 I/O 调度程序一样，读请求通常会在超时前得到处理，但是预测 I/O 调度程序的不同之处在于，请求提交后并不直接返回处理其他请求，而是会有意空闲片刻（实际空闲时间可以设置，默认为 6ms）。这几 ms，对应用程序来说是个提交其他读请求的好机会——任何对相邻磁盘位置操作的请求都会立刻得到处理。在等待时间结束后，预测 I/O 调度程序重新返回原来的位置，继续执行以前剩下的请求。

要注意，如果等待可以减少读请求所带来的向后再向前（back-and-forth）寻址操作，那么完全值得花一些时间来等待更多的请求；如果一个相邻的 I/O 请求在等待期到来，那么 I/O 调度程序可以节省两次寻址操作。如果存在愈来愈多的访问同样区域的读请求到来，那么片刻等待无疑会避免大量的寻址操作。

当然，如果没有 I/O 请求在等待期到来，那么预测 I/O 调度程序会给系统性能带来轻微的损失，浪费掉几 ms。预测 I/O 调度程序所能带来的优势取决于能否正确预测应用程序和文件系统的行为。这种预测依靠一系列的启发和统计工作。预测 I/O 调度程序跟踪并且统计每个应用程序块 I/O 操作的习惯行为，以便正确预测应用程序的未来行为。如果预测准确率足够高，那么预测调度程序便可以大大减少服务读请求所需的寻址开销，而且同时仍能满足请求所需要的系统响应时间要求。这样的话，预测 I/O 调度程序既减少了读响应时间，又能减少寻址次数和时间，所以说它既缩短了系统响应时间，又提高了系统吞吐量。

预测 I/O 调度程序的实现在文件内核源代码树的 `block/as-iosched.c` 中，它是 Linux 内核中缺省的 I/O 调度程序，对大多数工作负荷来说都执行良好，对服务器也是理想的。不过，在某些非常见而又有严格工作负荷的服务器（包括数据库挖掘服务器）上，这个调度程序执行的效果不好。

14.5.5 完全公正的排队 I/O 调度程序

完全公正的排队 I/O 调度程序（Complete Fair Queuing，CFQ）是为专有工作负荷设计的，不过，在实际中，也为多种工作负荷提供了良好的性能。但是，它与前面介绍的 I/O 调度程序有

根本的不同。

CFQ I/O 调度程序把进入的 I/O 请求放入特定的队列中，这种队列是根据引起 I/O 请求的进程组织的。例如，来自 foo 进程的 I/O 请求进入 foo 队列，而来自 bar 进程的 I/O 请求进入 bar 队列。在每个队列中，刚进入的请求与相邻请求合并在一起，并进行插入分类。队列由此按扇区方式分类，这与其他 I/O 调度程序队列类似。CFQ I/O 调度程序的差异在于每一个提交 I/O 的进程都有自己的队列。

CFQ I/O 调度程序以时间片轮转调度队列，从每个队列中选取请求数（默认值为 4，可以进行配置），然后进行下一轮调度。这就在进程级提供了公平，确保每个进程接收公平的磁盘带宽片断。预定的工作负荷是多媒体，在这种媒体中，这种公平的算法可以得到保证，比如，音频播放器总能够及时从磁盘再填满它的音频缓冲区。不过，实际上，CFQ I/O 调度程序在很多场合都能很好地执行。

完全公正的排队 I/O 调度程序位于 block/cfq-iosched.c。尽管这主要推荐给桌面工作负荷使用，但是，如果没有其他异常情况，它在几乎所有的工作负荷中都能很好地执行。

14.5.6 空操作的 I/O 调度程序

第四种也是最后一种 I/O 调度程序是空操作（Noop）I/O 调度程序，之所以这样命名是因为它基本上是一个空操作，不做多少事情。空操作 I/O 调度程序不进行排序，或者也不进行什么其他形式的预寻址操作。依此类推，它也没必要实现那些老套的算法，也就是在以前的 I/O 调度程序中看到的为了最小化请求周期而采用的算法。

不过，空操作 I/O 调度程序忘不了执行合并，这就像它的家务事。当一个新的请求提交到队列时，就把它与任一相邻的请求合并。除了这一操作，空操作 I/O 调度程序的确再不做什么，只是维护请求队列以近乎 FIFO 的顺序排列，块设备驱动程序便可以从这种队列中摘取请求。

空操作 I/O 调度程序不勤奋工作是有道理的。因为它打算用在块设备，那是真正的随机访问设备，比如闪存卡。如果块设备只有一点或者没有“寻道”的负担，那么，就没有必要对进入的请求进行插入排序，因此，空操作 I/O 调度程序是理想的候选者。

空操作 I/O 调度程序位于 block/noop-iosched.c，它是专为随机访问设备而设计的。

14.5.7 I/O 调度程序的选择

你已经看到 2.6 内核中四种不同的 I/O 调度程序。其中的每一种 I/O 调度程序都可以被启用，并内置在内核中。作为缺省，块设备使用完全公平的 I/O 调度程序。在启动时，可以通过命令行选项 `elevator=foo` 来覆盖缺省，这里 foo 是一个有效而激活的 I/O 调度程序，参看表 14-2。

表 14-2 给定 elevator 选项的参数

参 数	I/O 调度程序
as	预测
cfq	完全公正的排队
deadline	最终期限
noop	空操作

例如，内核命令行选项 `elevator=as` 会启用预测 I/O 调度程序给所有的块设备，从而覆盖默认的完全公正调度程序。

14.6 小结

在本章中，我们讨论了块设备的基本知识，并考察了块 I/O 层所用的数据结构：`bio`，表示活动的 I/O 操作；`buffer_head`，表示块到页的映射；还有请求结构，表示具体的 I/O 请求。我们追寻了 I/O 请求简单但重要的生命历程，其生命的重要点就是 I/O 调度程序。我们讨论了 I/O 调度程序所涉及的困惑问题，同时仔细推敲了当前内核的 4 种 I/O 调度程序，以及 2.4 版本中原有的 `linux` 电梯调度。

我们将在第 15 章讨论进程地址空间。



第 15 章

进程地址空间

第 12 章介绍了内核如何管理物理内存。其实内核除了管理本身的内存外，还必须管理用户空间中进程的内存。我们称这个内存为进程地址空间，也就是系统中每个用户空间进程所看到的内存。Linux 操作系统采用虚拟内存技术，因此，系统中的所有进程之间以虚拟方式共享内存。对一个进程而言，它好像都可以访问整个系统的所有物理内存。更重要的是，即使单独一个进程，它拥有的地址空间也可以远远大于系统物理内存。本章将集中讨论内核如何管理进程地址空间。

15.1 地址空间

进程地址空间由进程可寻址的虚拟内存组成，而且更为重要的特点是内核允许进程使用这种虚拟内存中的地址。每个进程都有一个 32 位或 64 位的平坦（flat）地址空间，空间的具体大小取决于体系结构。术语“平坦”指的是地址空间范围是一个独立的连续区间（比如，地址从 0 扩展到 4294967295 的 32 位地址空间）。一些操作系统提供了段地址空间，这种地址空间并非是一个独立的线性区域，而是被分段的，但现代采用虚拟内存的操作系统通常都使用平坦地址空间而不是分段式的内存模式。通常情况下，每个进程都有唯一的这种平坦地址空间。一个进程的地址空间与另一个进程的地址空间即使有相同的内存地址，实际上也彼此互不相干。我们称这样的进程为线程。

内存地址是一个给定的值，它要在地址空间范围之内，比如 4021f000。这个值表示的是进程 32 位地址空间中的一个特定的字节。尽管一个进程可以寻址 4GB 的虚拟内存（在 32 位的地址空间中），但这并不代表它就有权访问所有的虚拟地址。在地址空间中，我们更为关心的是一些虚拟内存的地址区间，比如 08048000-0804c000，它们可被进程访问。这些可被访问的合法地址空间称为内存区域（memory areas）。通过内核，进程可以给自己的地址空间动态地添加或减少内存区域。

进程只能访问有效内存区域内的内存地址。每个内存区域也具有相关权限如对相关进程有可读、可写、可执行属性。如果一个进程访问了不在有效范围中的内存区域，或以不正确的方式访问了有效地址，那么内核就会终止该进程，并返回“段错误”信息。

内存区域可以包含各种内存对象，比如：

- 可执行文件代码的内存映射，称为代码段（text section）。
- 可执行文件的已初始化全局变量的内存映射，称为数据段（data section）。

- 包含未初始化全局变量，也就是 bss 段[⊖]的零页（页面中的信息全部为 0 值，所以可用于映射 bss 段等目的）的内存映射。
- 用于进程用户空间栈（不要和进程内核栈混淆，进程的内核栈独立存在并由内核维护）的零页的内存映射。
- 每一个诸如 C 库或动态连接程序等共享库的代码段、数据段和 bss 也会被载入进程的地址空间。
- 任何内存映射文件。
- 任何共享内存段。
- 任何匿名的内存映射，比如由 malloc()[⊖]分配的内存。

进程地址空间中的任何有效地址都只能位于唯一的区域，这些内存区域不能相互覆盖。可以看到，在执行的进程中，每个不同的内存片段都对应一个独立的内存区域：栈、对象代码、全局变量、被映射的文件等。

15.2 内存描述符

内核使用内存描述符结构体表示进程的地址空间，该结构包含了和进程地址空间有关的全部信息。内存描述符由 mm_struct 结构体表示，定义在文件 <linux/sched.h> 中。下面给出内存描述符的结构和各个域的描述：

```
struct mm_struct{
    struct vm_area_struct    *mmap;                /* 内存区域链表 */
    struct rb_root            mm_rb;                /* VMA 形成的红黑树 */
    struct vm_area_struct    *mmap_cache;           /* 最近使用的内存区域 */
    unsigned long             free_area_cache;       /* 地址空间第一个空洞 */
    pgd_t                    *pgd;                 /* 页全局目录 */
    atomic_t                  mm_users;              /* 使用地址空间的用户数 */
    atomic_t                  mm_count;              /* 主使用计数器 */
    int                       map_count;             /* 内存区域的个数 */
    struct rw_semaphore       mmap_sem;             /* 内存区域的信号量 */
    spinlock_t                page_table_lock;       /* 页表锁 */
    struct list_head          mmlist;               /* 所有 mm_struct 形成的链表 */
    unsigned long             start_code;            /* 代码段的开始地址 */
    unsigned long             end_code;              /* 代码段的结束地址 */
    unsigned long             start_data;            /* 数据的首地址 */
    unsigned long             end_data;              /* 数据的尾地址 */
    unsigned long             start_brk;            /* 堆的首地址 */
    unsigned long             brk;                  /* 堆的尾地址 */
    unsigned long             start_stack;           /* 进程栈的首地址 */
    unsigned long             arg_start;             /* 命令行参数的首地址 */
    unsigned long             arg_end;               /* 命令行参数的尾地址 */
    unsigned long             env_start;            /* 环境变量的首地址 */
    unsigned long             env_end;              /* 环境变量的尾地址 */
}
```

⊖ 术语“BSS”已经有些年头了，它是 block started by symbol 的缩写。因为未初始化的变量没有对应的值，所以并不需要存放在可执行对象中。但是因为 C 标准强制规定未初始化的全局变量要被赋予特殊的默认值（基本上是 0 值），所以内核要将变量（未赋值的）从可执行代码载入到内存中，然后将零页映射到该片内存上，于是这些未初始化变量就被赋予了 0 值。这样做避免了在目标文件中显式地进行初始化，减少了空间浪费。

⊖ 在最新版本的 glibc 中，通过 mmap() 和 brk() 来实现 malloc() 函数。

```

unsigned long      rss;                /* 所分配的物理页 */
unsigned long      total_vm;           /* 全部页面数目 */
unsigned long      locked_vm;         /* 上锁的页面数目 */
unsigned long      saved_auxv[AT_VECTOR_SIZE]; /* 保存的 auxv */
cpumask_t          cpu_vm_mask;       /* 懒惰 (lazy) TLB 交换掩码 */
mm_context_t       context;           /* 体系结构特殊数据 */
unsigned long      flags;             /* 状态标志 */
int               core_waiters;       /* 内核转储等待线程 */
struct core_state  *core_state;       /* 核心转储的支持 */
spinlock_t         ioctx_lock;        /* AIO I/O 链表锁 */
struct hlist_head  ioctx_list;        /* AIO I/O 链表 */
};

```

mm_users 域记录正在使用该地址的进程数目。比如，如果两个线程共享该地址空间，那么 mm_users 的值便等于 2；mm_count 域是 mm_struct 结构体的主引用计数。所有的 mm_users 都等于 mm_count 的增加量。这样，在前面的例子中，mm_count 就仅仅为 1。如果有 9 个线程共享某个地址空间，那么 mm_users 将会是 9，而 mm_count 的值将再次为 1。当 mm_users 的值减为 0（即所有正使用该地址空间的线程都退出）时，mm_count 域的值才变为 0。当 mm_count 的值等于 0，说明已经没有任何指向该 mm_struct 结构体的引用了，这时该结构体会被撤销。当内核在一个地址空间上操作，并需要使用与该地址相关联的引用计数时，内核便增加 mm_count。内核同时使用这两个计数器是为了区别主使用计数（mm_count）器和使用该地址空间的进程的数目（mm_users）。

mmap 和 mm_rb 这两个不同数据结构体描述的对象是相同的：该地址空间中的全部内存区域。但是前者以链表形式存放而后者以红—黑树的形式存放。红—黑树是一种二叉树，与其他二叉树一样，搜索它的时间复杂度为 $O(\log n)$ 。在本章后续部分“内存区域的树型结构和内存区域的链表结构”一节中，我们将进一步讨论红—黑树。

内核通常会避免使用两种数据结构组织同一种数据，但此处内核这样的冗余确实派得上用场。mmap 结构体作为链表，利于简单、高效地遍历所有元素；而 mm_rb 结构体作为红—黑树，更适合搜索指定元素。对内存区域的具体操作将在本章的后续部分详细介绍。内核并没有复制 mm_struct 结构体，而仅仅被包含其中。覆盖树上的链表并用这两个结构体同时访问相同的数据集，有时候我们将此操作称作线索树。

所有的 mm_struct 结构体都通过自身的 mmlist 域连接在一个双向链表中，该链表的首元素是 init_mm 内存描述符，它代表 init 进程的地址空间。另外要注意，操作该链表的时候需要使用 mmlist_lock 锁来防止并发访问，该锁定义在文件 kernel/fork.c 中。

15.2.1 分配内存描述符

在进程的进程描述符（在 <linux/sched.h> 中定义的 task_struct 结构体就表示进程描述符）中，mm 域存放着该进程使用的内存描述符，所以 current->mm 便指向当前进程的内存描述符。fork() 函数利用 copy_mm() 函数复制父进程的内存描述符，也就是 current->mm 域给其子进程，而子进程中的 mm_struct 结构体实际是通过文件 kernel/fork.c 中的 allocate_mm() 宏从 mm_cachep slab 缓存中分配得到的。通常，每个进程都有唯一的 mm_struct 结构体，即唯一的

进程地址空间。

如果父进程希望和其子进程共享地址空间，可以在调用 `clone()` 时，设置 `CLONE_VM` 标志。我们把这样的进程称作线程。回忆第 3 章，是否共享地址空间几乎是进程和 Linux 中所谓的线程间本质上的唯一区别。除此以外，Linux 内核并不区别对待它们，线程对内核来说仅仅是一个共享特定资源的进程而已。

当 `CLONE_VM` 被指定后，内核就不再需要调用 `allocate_mm()` 函数了，而仅仅需要在调用 `copy_mm()` 函数中将 `mm` 域指向其父进程的内存描述符就可以了：

```
if(clone_flags & CLONE_VM){
    /*
     * current 是父进程而 tsk 在 fork() 执行期间是子进程
     */
    atomic_inc(&current->mm->mm_users);
    tsk->mm= current ->mm;
}
```

15.2.2 撤销内存描述符

当进程退出时，内核会调用定义在 `kernel/exit.c` 中的 `exit_mm()` 函数，该函数执行一些常规的撤销工作，同时更新一些统计量。其中，该函数会调用 `mmapput()` 函数减少内存描述符中的 `mm_users` 用户计数，如果用户计数降到零，将调用 `mmdrop()` 函数，减少 `mm_count` 使用计数。如果使用计数也等于零了，说明该内存描述符不再有任何使用者了，那么调用 `free_mm()` 宏通过 `kmem_cache_free()` 函数将 `mm_struct` 结构体归还到 `mm_cache` slab 缓存中。

15.2.3 mm_struct 与内核线程

内核线程没有进程地址空间，也没有相关的内存描述符。所以内核线程对应的进程描述符中 `mm` 域为空。事实上，这也正是内核线程的真实含义——它们没有用户上下文。

省了进程地址空间再好不过了，因为内核线程并不需要访问任何用户空间的内存（那它们访问谁的呢？）而且因为内核线程在用户空间中没有任何页，所以实际上它们并不需要有自己的内存描述符和页表（后面的内容将讲述页表）。尽管如此，即使访问内核内存，内核线程也还是需要使用一些数据的，比如页表。为了避免内核线程为内存描述符和页表浪费内存，也为了当新内核线程运行时，避免浪费处理器周期向新地址空间进行切换，内核线程将直接使用前一个进程的内存描述符。

当一个进程被调度时，该进程的 `mm` 域指向的地址空间被装载到内存，进程描述符中的 `active_mm` 域会被更新，指向新的地址空间。内核线程没有地址空间，所以 `mm` 域为 `NULL`。于是，当一个内核线程被调度时，内核发现它的 `mm` 域为 `NULL`，就会保留前一个进程的地址空间，随后内核更新内核线程对应的进程描述符中的 `active_mm` 域，使其指向前一个进程的内存描述符。所以在需要时，内核线程便可以使用前一个进程的页表。因为内核线程不访问用户空间的内存，所以它们仅仅使用地址空间中和内核内存相关的信息，这些信息的含义和普通进程完全相同。

15.3 虚拟内存区域

内存区域由 `vm_area_struct` 结构体描述，定义在文件 `<linux/mm_types.h>` 中。内存区域在 Linux 内核中也经常称作虚拟内存区域（virtual memory Areas, VMAs）。

`vm_area_struct` 结构体描述了指定地址空间内连续区间上的一个独立内存范围。内核将每个内存区域作为一个单独的内存对象管理，每个内存区域都拥有一致的属性，比如访问权限等，另外，相应的操作也都一致。按照这样的方式，每一个 VMA 就可以代表不同类型的内存区域（比如内存映射文件或者进程用户空间栈），这种管理方式类似于使用 VFS 层的面向对象方法（请看第 13 章），下面给出该结构定义和各个域的描述：

```
struct vm_area_struct {
    struct mm_struct      *vm_mm;           /* 相关的 mm_struct 结构体 */
    unsigned long         vm_start;         /* 区间的首地址 */
    unsigned long         vm_end;          /* 区间的尾地址 */
    struct vm_area_struct *vm_next;         /* VMA 链表 */
    pgprot_t              vm_page_prot;     /* 访问控制权限 */
    unsigned long         vm_flags;        /* 标志 */
    struct rb_node         vm_rb;          /* 树上该 VMA 的节点 */
    union {                          /* 或者是关联于 address_space->i_mmap 字段，或者是关联于
        address_space->i_mmap_nonlinear 字段 */
        struct {
            struct list_head list;
            void *parent;
            struct vm_area_struct *head;
        } vm_set;
        struct prio_tree_node prio_tree_node;
    } shared;
    struct list_head      anon_vma_node;    /* anon_vma 项 */
    struct anon_vma       *anon_vma;       /* 匿名 VMA 对象 */
    struct vm_operations_struct *vm_ops;    /* 相关的操作表 */
    unsigned long         vm_pgoff;         /* 文件中的偏移量 */
    struct file           *vm_file;         /* 被映射的文件（如果存在） */
    void                  *vm_private_data; /* 私有数据 */
};
```

每个内存描述符都对应于进程地址空间中的唯一区间。`vm_start` 域指向区间的首地址（最低地址），`vm_end` 域指向区间的尾地址（最高地址）之后的第一个字节，也就是说，`vm_start` 是内存区间的开始地址（它本身在区间内），而 `vm_end` 是内存区间的结束地址（它本身在区间外），因此，`vm_end - vm_start` 的大小便是内存区间的长度，内存区域的位置就在 `[vm_start, vm_end]` 之中。注意，在同一个地址空间内的不同内存区间不能重叠。

`vm_mm` 域指向和 VMA 相关的 `mm_struct` 结构体，注意，每个 VMA 对其相关的 `mm_struct` 结构体来说都是唯一的，所以即使两个独立的进程将同一个文件映射到各自的地址空间，它们分别都会有一个 `vm_area_struct` 结构体来标志自己的内存区域；反过来，如果两个线程共享一个地址空间，那么它们也同时共享其中的所有 `vm_area_struct` 结构体。

15.3.1 VMA 标志

VMA 标志是一种位标志，其定义见 `<linux/mm.h>`。它包含在 `vm_flags` 域内，标志了内存区

域所包含的页面的行为和信息。和物理页的访问权限不同，VMA 标志反映了内核处理页面所需要遵守的行为准则，而不是硬件要求。而且，`vm_flags` 同时也包含了内存区域中每个页面的信息，或内存区域的整体信息，而不是具体的独立页面。表 15-1 列出了所有 VMA 标志的可能取值。

表 15-1 VMA 标志

标 志	对 VMA 及其页面的影响
VM_READ	页面可读取
VM_WRITE	页面可写
VM_EXEC	页面可执行
VM_SHARED	页面可共享
VM_MAYREAD	VM_READ 标志可被设置
VM_MAYWRITE	VM_WRITE 标志可被设置
VM_MAYEXEC	VM_EXEC 标志可被设置
VM_MAYSHARE	VM_SHARE 标志可被设置
VM_GROWSDOWN	区域可向下增长
VM_GROWSUP	区域可向上增长
VM_SHM	区域可用作共享内存
VM_DENYWRITE	区域映射一个不可写文件
VM_EXECUTABLE	区域映射一个可执行文件
VM_LOCKED	区域中的页面被锁定
VM_IO	区域映射设备 I/O 空间
VM_SEQ_READ	页面可能会被连续访问
VM_RAND_READ	页面可能会被随机访问
VM_DONTCOPY	区域不能在 <code>fork()</code> 时被拷贝
VM_DONTEXPAND	区域不能通过 <code>mremap()</code> 增加
VM_RESERVED	区域不能被换出
VM_ACCOUNT	该区域是一个记账 VM 对象
VM_HUGETLB	区域使用了 <code>hugetlb</code> 页面
VM_NONLINEAR	该区域是非线性映射的

让我们进一步看看其中有趣和重要的几种标志，VM_READ、VM_WRITE 和 VM_EXEC 标志了内存区域中页面的读、写和执行权限。这些标志根据要求组合构成 VMA 的访问控制权限，当访问 VMA 时，需要查看其访问权限。比如进程的对象代码映射区域可能会标志为 VM_READ 和 VM_EXEC，而没有标志为 VM_WRITE；另一方面，可执行对象数据段的映射区域标志为 VM_READ 和 VM_WRITE，而 VM_EXEC 标志对它就毫无意义。也就是说，只读文件数据段的映射区域仅可被标志为 VM_READ。

VM_SHARED 指明了内存区域包含的映射是否可以在多进程间共享，如果该标志被设置，则我们称其为共享映射；如果未被设置，而仅仅只有一个进程可以使用该映射的内容，我们称它为私有映射。

VM_IO 标志内存区域中包含对设备 I/O 空间的映射。该标志通常在设备驱动程序执行

mmap() 函数进行 I/O 空间映射时才被设置，同时该标志也表示该内存区域不能被包含在任何进程的存放转存 (core dump) 中。VM_RESERVED 标志规定了内存区域不能被换出，它也是在设备驱动程序进行映射时被设置。

VM_SEQ_READ 标志暗示内核应用程序对映射内容执行有序的（线性和连续的）读操作；这样，内核可以有选择地执行预读文件。VM_RANDOM_READ 标志的意义正好相反，暗示应用程序对映射内容执行随机的（非有序的）读操作。因此内核可以有选择地减少或彻底取消文件预读，所以这两个标志可以通过系统调用 madvise() 设置，设置参数分别是 MADV_SEQUENTIAL 和 MADV_RANDOM。文件预读是指在读数据时有意地按顺序多读取一些本次请求以外的数据——希望多读的数据能够很快就被用到。这种预读行为对那些顺序读取数据的应用程序有很大的好处，但是如果数据的访问是随机的，那么预读显然就多余了。

15.3.2 VMA 操作

vm_area_struct 结构体中的 vm_ops 域指向与指定内存区域相关的操作函数表，内核使用表中的方法操作 VMA。vm_area_struct 作为通用对象代表了任何类型的内存区域，而操作表描述针对特定的对象实例的特定方法。

操作函数表由 vm_operations_struct 结构体表示，定义在文件 <linux/mm.h> 中：

```
struct vm_operations_struct {
    void (*open)(struct vm_area_struct *);
    void (*close)(struct vm_area_struct *);
    int (*fault)(struct vm_area_struct *, struct vm_fault *);
    int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
    int (*access)(struct vm_area_struct *, unsigned long,
                  void *, int, int);
};
```

下面介绍具体方法：

`*void open(struct vm_area_struct *area)`

当指定的内存区域被加入到一个地址空间时，该函数被调用。

`*void close(struct vm_area_struct *area)`

当指定的内存区域从地址空间删除时，该函数被调用。

`*int fault(struct vm_area_struct *area, struct vm_fault *vmf)`

当没有出现在物理内存中的页面被访问时，该函数被页面故障处理调用。

`*int page_mkwrite(struct vm_area_struct *area, struct vm_fault *vmf)`

当某个页面为只读页面时，该函数被页面故障处理调用。

`*int access(struct vm_area_struct *vma, unsigned long address, void *buf, int len, int write)`

当 get_user_pages() 函数调用失败时，该函数被 access_process_vm() 函数调用。

15.3.3 内存区域的树型结构和内存区域的链表结构

上文讨论过，可以通过内存描述符中的 `mmap` 和 `mm_rb` 域之一访问内存区域。这两个域各自独立地指向与内存描述符相关的全体内存区域对象。其实，它们包含完全相同的 `vm_area_struct` 结构体的指针，仅仅组织方法不同。

`mmap` 域使用单独链表连接所有的内存区域对象。每一个 `vm_area_struct` 结构体通过自身的 `vm_next` 域被连入链表，所有的区域按地址增长的方向排序，`mmap` 域指向链表中第一个内存区域，链中最后一个结构体指针指向空。

`mm_rb` 域使用红-黑树连接所有的内存区域对象。`mm_rb` 域指向红-黑树的根节点，地址空间中每一个 `vm_area_struct` 结构体通过自身的 `vm_rb` 域连接到树中。

红-黑树是一种二叉树，树中的每一个元素称为一个节点，最初的节点称为树根。红-黑树的多数节点都由两个子节点：一个左子节点和一个右子节点，不过也有节点只有一个子节点的情况。树末端的节点称为叶子节点，它们没有子节点。红-黑树中的所有节点都遵从：左边节点值小于右边节点值；另外每个节点都被配以红色或黑色（要么红要么黑，所以叫做红-黑树）。分配的规则为：红节点的子节点为黑色，并且树中的任何一条从节点到叶子的路径必须包含同样数目的黑色节点。记住根节点总为红色。红-黑树的搜索、插入、删除等操作的复杂度都为 $O(\log(n))$ 。

链表用于需要遍历全部节点的时候，而红-黑树适用于在地址空间中定位特定内存区域的时候。内核为了内存区域上的各种不同操作都能获得高性能，所以同时使用了这两种数据结构。

15.3.4 实际使用中的内存区域

可以使用 `/proc` 文件系统和 `pmap (1)` 工具查看给定进程的内存空间和其中所含的内存区域。我们来看一个非常简单的用户空间程序的例子，它其实什么也不做，仅仅是为了做说明：

```
int main(int argc, char *argv[])
{
    return 0;
}
```

下面列出该进程地址空间中包含的内存区域。其中有代码段、数据段和 `bss` 段等。假设该进程与 C 库动态连接，那么地址空间中还将分别包含 `libc.so` 和 `ld.so` 对应的上述三种内存区域。此外，地址空间中还要包含进程栈对应的内存区域。

`/proc/<pid>/maps` 的输出显示了该进程地址空间中的全部内存区域：

```
rlove@wolf:~$ cat /proc/1426/maps
00e80000-00faf000 r-xp 00000000 03:01 208530 /lib/tls/libc-2.5.1.so
00faf000-00fb2000 rw-p 0012f000 03:01 208530 /lib/tls/libc-2.5.1.so
00fb2000-00fb4000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 03:03 439029 /home/rlove/src/example
08049000-0804a000 rw-p 00000000 03:03 439029 /home/rlove/src/example
40000000-40015000 r-xp 00000000 03:01 80276 /lib/ld-2.5.1.so
40015000-40016000 rw-p 00015000 03:01 80276 /lib/ld-2.5.1.so
4001e000-4001f000 rw-p 00000000 00:00 0
bffffe000-c0000000 rwxp fffff000 00:00 0
```

每行数据格式如下：

开始—结束 访问权限 偏移 主设备号：次设备号 i 节点 文件

pmap(1) 工具^①将上述信息以更方便阅读的形式输出：

```
rlove@wolf:~$ pmap 1426
example [1426]
00e80000(1212KB)  r-xp    (03:01 208530)      /lib/tls/libc-2.5.1.so
00faf000(12KB)   rw-p    (03:01 208530)      /lib/tls/libc-2.5.1.so
00fbz000(8KB)    rw-p    (00:000)
08048000(4KB)    r-xp    (03:03 439029)      /home/rlove/src/example
08049000(4KB)    rw-p    (03:03 439029)      /home/rlove/src/example
40000000(84KB)   r-xp    (03:01 80276)       /lib/ld-2.5.1.so
40015000(4KB)    rw-p    (03:01 80276)       /lib/ld-2.5.1.so
4001e000(4KB)    rw-p    (00:00 0)
bffffe000(8KB)   rwxp    (00:00 0)             [stack]
mapped :1340KB   writable/private : 40KB      shared :0KB
```

前三行分别对应 C 库中 libc.so 的代码段、数据段和 bss 段，接下来的两个行为可执行对象的代码段和数据段，再下来三个行为动态连接程序 ld.so 的代码段、数据段和 bss 段，最后一行是进程的栈。

注意，代码段具有我们所要求的可读且可执行权限；另一方面，数据段和 bss（它们都包含全局数据变量）具有可读、可写但不可执行权限。而堆栈则可读、可写，甚至还可执行——虽然这点并不常用到。

该进程的全部地址空间大约为 1340KB，但是只有大约 40KB 的内存区域是可写和私有的。如果一片内存范围是共享的或不可写的，那么内核只需要在内存中为文件（backing file）保留一份映射。对于共享映射来说，这样做没什么特别的，但是对于不可写内存区域也这样做，就有些让人奇怪了。如果考虑到映射区域不可写意味着该区域不可被改变（映射只用来读），就应该清楚只把该映像读入一次是很安全的。所以 C 库在物理内存中仅仅需要占用 1212KB 空间，而不需要为每个使用 C 库的进程在内存中都保存一个 1212KB 的空间。进程访问了 1340KB 的数据和代码空间，然而仅仅消耗了 40KB 的物理内存，可以看出利用这种共享不可写内存的方法节约了大量的内存空间。

注意没有映射文件的内存区域的设备标志为 00:00，索引接点标志也为 0，这个区域就是零页——零页映射的内容全为零。如果将零页映射到可写的内存区域，那么该区域将全被初始化为 0。这是零页的一个重要用处，而 bss 段需要的就是全 0 的内存区域。由于内存未被共享，所以只要一有进程写该处数据，那么该处数据就将被拷贝出来（就是我们所说的写时拷贝），然后才被更新。

每个和进程相关的内存区域都对应于一个 vm_area_struct 结构体。另外进程不同于线程，进程结构体 task_struct 包含唯一的 mm_struct 结构体引用。

15.4 操作内存区域

内核时常需要在某个内存区域上执行一些操作，比如某个指定地址是否包含在某个内存区域

^① pmap(1) 工具将进程的内存区域格式化后显示，虽然结果中的信息和 /proc 中的是一样的信息，但它的输出形式比 /proc 的输出形式更具可读性。在新版本的 procps 包中可以找到这个工具。

中。这类操作非常频繁，另外它们也是 `mmap()` 例程的基础——我们在 15.5 节会讨论 `mmap()` 操作。为了方便执行这类对内存区域的操作，内核定义了许多的辅助函数。

它们都声明在文件 `<linux/mm.h>` 中。

15.4.1 `find_vma()`

为了找到一个给定的内存地址属于哪一个内存区域，内核提供了 `find_vma()` 函数。该函数定义在文件 `<mm/mmap.c>` 中：

```
struct vm_area_struct * find_vma(struct mm_struct *mm, unsigned long addr);
```

该函数在指定的地址空间中搜索第一个 `vm_end` 大于 `addr` 的内存区域。换句话说，该函数寻找第一个包含 `addr` 或首地址大于 `addr` 的内存区域，如果没有发现这样的区域，该函数返回 `NULL`；否则返回指向匹配的内存区域的 `vm_area_struct` 结构体指针。注意，由于返回的 VMA 首地址可能大于 `addr`，所以指定的地址并不一定就包含在返回的 VMA 中。因为很有可能在对某个 VMA 执行操作后，还有其他更多的操作会对该 VMA 接着进行操作，所以 `find_vma()` 函数返回的结果被缓存在内存描述符的 `mmap_cache` 域中。实践证明，被缓存的 VMA 会有相当好的命中率（实践中大约 30% ~ 40%），而且检查被缓存的 VMA 速度会很快，如果指定的地址不在缓存中，那么必须搜索和内存描述符相关的所有内存区域。这种搜索通过红-黑树进行：

```
struct vm_area_struct * find_vma(struct mm_struct *mm, unsigned long addr)
{
    struct vm_area_struct *vma = NULL;

    if (mm) {
        vma = mm->mmap_cache;
        if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
            struct rb_node *rb_node;

            rb_node = mm->mm_rb.rb_node;
            vma = NULL;
            while (rb_node) {
                struct vm_area_struct * vma_tmp;

                vma_tmp = rb_entry(rb_node,
                                   struct vm_area_struct, vm_rb);
                if (vma_tmp->vm_end > addr) {
                    vma = vma_tmp;
                    if (vma_tmp->vm_start <= addr)
                        break;
                    rb_node = rb_node->rb_left;
                } else
                    rb_node = rb_node->rb_right;
            }
            if (vma)
                mm->mmap_cache = vma;
        }
    }
}
```

```

    }

    return vma;
}

```

首先，该函数检查 `mmap_cache`，看看缓存的 VMA 是否包含了所需地址。注意简单地检查 VMA 的 `vm_end` 是否大于 `addr`，并不能保证该 VMA 是第一个大于 `addr` 的内存区域，所以缓存要想发挥作用，就要求指定的地址必须包含在被缓存的 VMA 中——幸好，这也正是连续操作同一 VMA 必然发生的情况。

如果缓存中并未包含希望的 VMA，那么该函数必须搜索红-黑树。如果当前 VMA 的 `vm_end` 大于 `addr`，进入左子节点继续搜索；否则，沿右边子节点搜索，直到找到包含 `addr` 的 VMA 为止。如果没有包含 `addr` 的 VMA 被找到，那么该函数继续搜索树，并且返回大于 `addr` 的第一个 VMA。如果也不存在满足要求的 VMA，那该函数返回 `NULL`。

15.4.2 find_vma_prev()

`find_vma_prev()` 函数和 `find_vma()` 工作方式相同，但是它返回第一个小于 `addr` 的 VMA。该函数定义和声明分别在文件 `mm/mmap.c` 中和文件 `<linux/mm.h>` 中：

```

struct vm_area_struct * find_vma_prev(struct mm_struct *mm, unsigned long addr,
                                     struct vm_area_struct **pprev)

```

`pprev` 参数存放指向先于 `addr` 的 VMA 指针。

15.4.3 find_vma_intersection()

`find_vma_intersection()` 函数返回第一个和指定地址区间相交的 VMA。因为该函数是内联函数，所以定义在文件 `<linux/mm.h>` 中：

```

static inline struct vm_area_struct *
find_vma_intersection(struct mm_struct *mm,
                     unsigned long start_addr,
                     unsigned long end_addr)
{
    struct vm_area_struct *vma;

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

```

第一个参数 `mm` 是要搜索的地址空间，`start_addr` 是区间的开始首位置，`end_addr` 是区间的尾位置。

显然，如果 `find_vma()` 返回 `NULL`，那么 `find_vma_intersection()` 也会返回 `NULL`。但是如果 `find_vma()` 返回有效的 VMA，`find_vma_intersection()` 只有在该 VMA 的起始位置于给定的地

址区间结束位置之前，才将其返回。如果 VMA 的起始位置大于指定地址范围的结束位置，则该函数返回 NULL。

15.5 mmap() 和 do_mmap()：创建地址区间

内核使用 do_mmap() 函数创建一个新的线性地址区间。但是说该函数创建了一个新 VMA 并不非常准确，因为如果创建的地址区间和一个已经存在的地址区间相邻，并且它们具有相同的访问权限的话，两个区间将合并为一个。如果不能合并，就确实需要创建一个新的 VMA 了。但无论哪种情况，do_mmap() 函数都会将一个地址区间加入到进程的地址空间中——无论是扩展已存在的内存区域还是创建一个新的区域。

do_mmap() 函数定义在文件 <linux/mm.h> 中。

```
unsigned long do_mmap(struct file *file, unsigned long addr,
                     unsigned long len, unsigned long prot,
                     unsigned long flag, unsigned long offset)
```

该函数映射由 file 指定的文件，具体映射的是文件中从偏移 offset 处开始，长度为 len 字节的范围内的数据。如果 file 参数是 NULL 并且 offset 参数也是 0，那么就代表这次映射没有和文件相关，该情况称作匿名映射（anonymous mapping）。如果指定了文件名和偏移量，那么该映射称为文件映射（file-backed mapping）。

addr 是可选参数，它指定搜索空闲区域的起始位置。

prot 参数指定内存区域中页面的访问权限。访问权限标志定义在文件 <asm/mman.h> 中，不同体系结构标志的定义有所不同，但是对所有体系结构而言，都会包含表 15-2 中所列举的标志。

表 15-2 页保护标志

标 志	对新建区间中页的要求
PROT_READ	对应于 VM_READ
PROT_WRITE	对应于 VM_WRITE
PROT_EXEC	对应于 VM_EXEC
PROT_NONE	页不可被访问

flag 参数指定了 VMA 标志，这些标志指定类型并改变映射的行为。它们也在文件 <asm/mman.h> 中定义，请参看表 15-3。

表 15-3 页保护标志

标 志	对新区间的要求
MAP_SHARED	映射可以被共享
MAP_PRIVATE	映射不能被共享
MAP_FIXED	新区间必须开始于指定的地址 addr
MAP_ANONYMOUS	映射不是 file-backed，而是匿名的
MAP_GROWSDOWN	对应于 VM_GROWSDOWN
MAP_DENYWRITE	对应于 VM_DENYWRITE

(续)

标 志	对新区间的要求
MAP_EXECUTABLE	对应于 VM_EXECUTABLE
MAP_LOCKED	对应于 VM_LOCKED
MAP_NORESERVE	不需要为映射保留空间
MAP_POPULATE	填充页表
MAP_NONBLOCK	在 I/O 操作上不堵塞

如果系统调用 `do_mmap()` 的参数中有无效参数, 那么它返回一个负值; 否则, 它会在虚拟内存中分配一个合适的新内存区域。如果有可能的话, 将新区域和邻近区域进行合并, 否则内核从 `vm_area_cachep` 长字节 (slab) 缓存中分配一个 `vm_area_struct` 结构体, 并且使用 `vma_link()` 函数将新分配的内存区域添加到地址空间的内存区域链表和红-黑树中, 随后还要更新内存描述符中的 `total_vm` 域, 然后才返回新分配的地址区间的初始地址。

在用户空间可以通过 `mmap()` 系统调用获取内核函数 `do_mmap()` 的功能。`mmap()` 系统调用定义如下:

```
void * mmap2(void *start,
             size_t length,
             int prot,
             int flags,
             int fd,
             off_t pgoff)
```

由于该系统调用是 `mmap()` 调用的第二种变种, 所以起名为 `mmap2()`。最原始的 `mmap()` 调用中最后一个参数是字节偏移量, 而目前这个 `mmap2()` 使用页面偏移作最后一个参数。使用页面偏移量可以映射更大的文件和更大的偏移位置。原始的 `mmap()` 调用由 POSIX 定义, 仍然在 C 库中作为 `mmap()` 方法使用, 但是内核中已经没有对应的实现了, 而实现的是新方法 `mmap2()`。虽然 C 库仍然可以使用原始版本的映射方法, 但是它其实还是基于函数 `mmap2()` 进行的, 因为对原始 `mmap()` 方法的调用是通过将字节偏移转化为页面偏移, 从而转化为对 `mmap2()` 函数的调用来实现的。

15.6 `mummap()` 和 `do_mummap()`: 删除地址区间

`do_mummap()` 函数从特定的进程地址空间中删除指定地址区间, 该函数定义在文件 `<linux/mm.h>` 中:

```
int do_mummap(struct mm_struct *mm, unsigned long start, size_t len)
```

第一个参数指定要删除区域所在的地址空间, 删除从地址 `start` 开始, 长度为 `len` 字节的地址区间。如果成功, 返回零。否则, 返回负的错误码。

系统调用 `mummap()` 给用户空间程序提供了一种从自身地址空间中删除指定地址区间的方法, 它和系统调用 `mmap()` 的作用相反:

```
int munmap(void *start, size_t length)
```

该系统调用定义在文件 `mm/mmap.c` 中，它是对 `do_mummap()` 函数的一个简单的封装：

```
asmlinkage long sys_munmap(unsigned long addr, size_t len)
{
    int ret;
    struct mm_struct *mm;

    mm = current->mm;
    down_write(&mm->mmap_sem);
    ret = do_munmap(mm, addr, len);
    up_write(&mm->mmap_sem);
    return ret;
}
```

15.7 页表

虽然应用程序操作的对象是映射到物理内存之上的虚拟内存，但是处理器直接操作的却是物理内存。所以当用程序访问一个虚拟地址时，首先必须将虚拟地址转化成物理地址，然后处理器才能解析地址访问请求。地址的转换工作需要通过查询页表才能完成，概括地讲，地址转换需要将虚拟地址分段，使每段虚拟地址都作为一个索引指向页表，而页表项则指向下一级别的页表或者指向最终的物理页面。

Linux 中使用三级页表完成地址转换。利用多级页表能够节约地址转换需占用的存放空间。如果利用三级页表转换地址，即使是 64 位机器，占用的空间也很有限。但是如果使用静态数组实现页表，那么即便在 32 位机器上，该数组也将占用巨大的存放空间。Linux 对所有体系结构，包括对那些不支持三级页表的体系结构（比如，有些体系结构只使用两级页表或者使用散列表完成地址转换）都使用三级页表管理，因为使用三级页表结构可以利用“最大公约数”的思想——一种设计简单的体系结构，可以按照需要在编译时简化使用页表的三级结构，比如只使用两级。

顶级页表是页全局目录（PGD），它包含了一个 `pgd_t` 类型数组，多数体系结构中 `pgd_t` 类型等同于无符号长整型类型。PGD 中的表项指向二级页目录中的表项：PMD。

二级页表是中间页目录（PMD），它是个 `pmd_t` 类型数组，其中的表项指向 PTE 中的表项。

最后一级的页表简称页表，其中包含了 `pte_t` 类型的页表项，该页表项指向物理页面。

多数体系结构中，搜索页表的工作是由硬件完成的（至少某种程度上）。虽然通常操作中，很多使用页表的工作都可以由硬件执行，但是只有在内核正确设置页表的前提下，硬件才能方便地操作它们。图 15-1 描述了虚拟地址通过页表找到物理地址的过程。

每个进程都有自己的页表（当然，线程会共享页表）。内存描述符的 `pgd` 域指向的就是进程的页全局目录。注意，操作和检索页表时必须使用 `page_table_lock` 锁，该锁在相应的进程的内存描述符中，以防止竞争条件。

页表对应的结构体依赖于具体的体系结构，所以定义在文件 `<asm/page.h>` 中。

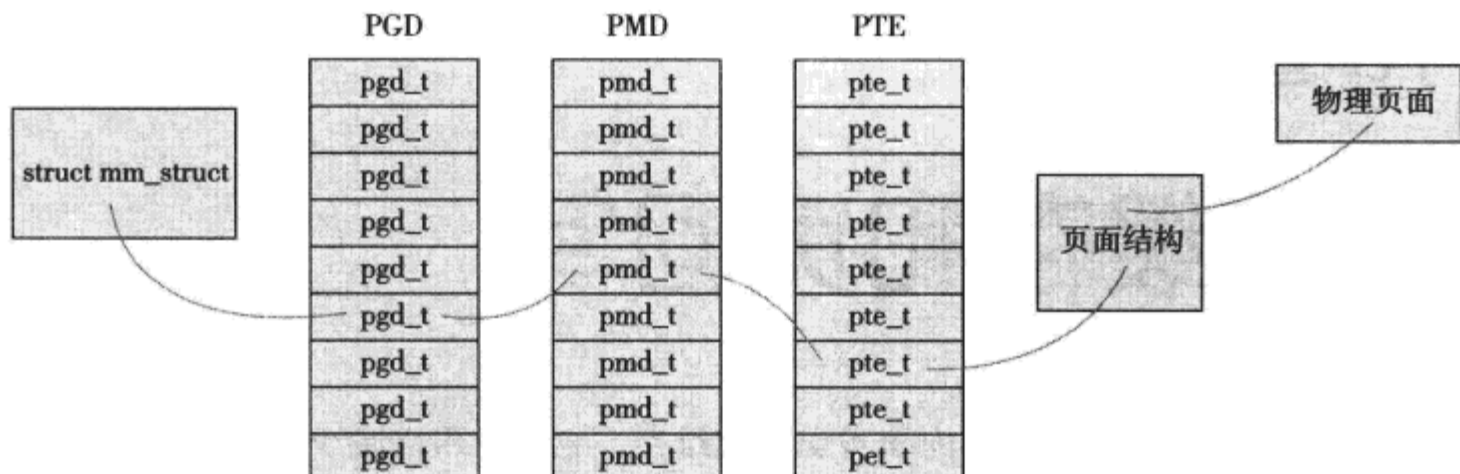


图 15-1 虚拟-物理地址查询

由于几乎每次对虚拟内存中的页面访问都必须先解析它，从而得到物理内存中的对应地址，所以页表操作的性能非常关键。但不幸的是，搜索内存中的物理地址速度很有限，因此为了加快搜索，多数体系结构都实现了一个翻译后缓冲器（translate lookaside buffer, TLB）。TLB 作为一个将虚拟地址映射到物理地址的硬件缓存，当请求访问一个虚拟地址时，处理器将首先检查 TLB 中是否缓存了该虚拟地址到物理地址的映射，如果在缓存中直接命中，物理地址立刻返回；否则，就需要再通过页表搜索需要的物理地址。

虽然硬件完成了有关页表的部分工作，但是页表的管理仍然是内核的关键部分——而且在不断改进。2.6 版内核对页表管理的主要改进是：从高端内存分配部分页表。今后可能的改进包括通过在写时拷贝（copy-on-write）的方式共享页表。这种机制使得在 `fork()` 操作中可由父子进程共享页表。因为只有当子进程或父进程试图修改特定页表项时，内核才去创建该页表项的新拷贝，此后父子进程才不再共享该页表项。可以看到，利用共享页表可以消除 `fork()` 操作中页表拷贝所带来的消耗。

15.8 小结

这章的内容不能不说是很“难缠”啦。其中，我们看到了抽象出来的进程虚拟内存，看到了内核如何表示进程空间（通过 `mm_struct`）以及内核如何表示该空间中的内存区域（通过结构体 `vm_area_struct`）。除此以外，我们还了解了内核如何创建（通过 `mmap()`）和撤销（通过 `munmap()`）这些内存区域，最后还讨论了页表。因为 Linux 是一个基于虚拟内存的操作系统，所以这些概念对于系统运行来说都是非常基础的，一定要仔细领会。

第 16 章，我们要讨论页缓存——一种用于所有页 I/O 操作的内存数据缓存，而且还要涵盖内核执行基于页的数据回写。

第 16 章

页高速缓存和页回写

页高速缓存 (cache) 是 Linux 内核实现磁盘缓存。它主要用来减少对磁盘的 I/O 操作。具体地讲, 是通过把磁盘中的数据缓存到物理内存中, 把对磁盘的访问变为对物理内存的访问。这一章将讨论页高速缓存与页回写 (将页高速缓存中的变更数据刷新回磁盘的操作)。

磁盘高速缓存之所以在任何现代操作系统中尤为重要源自两个因素: 第一, 访问磁盘的速度要远远低于 (差好几个数量级) 访问内存的速度——ms 和 ns 的差距, 因此, 从内存访问数据比从磁盘访问速度更快, 若从处理器的 L1 和 L2 高速缓存访问则更快。第二, 数据一旦被访问, 就很有可能在短期内再次被访问到。这种在短时期内集中访问同一片数据的原理称作临时局部原理 (temporal locality)。临时局部原理能保证: 如果在第一次访问数据时缓存它, 那就极有可能在短期内再次被高速缓存命中 (访问到高速缓存中的数据)。正是由于内存访问要比磁盘访问快得多, 再加上数据一次被访问后更可能再次被访问的特点, 所以磁盘的内存缓存将给系统存储性能带来质的飞跃。

16.1 缓存手段

页高速缓存是由内存中的物理页面组成的, 其内容对应磁盘上的物理块。页高速缓存大小能动态调整——它可以通过占用空闲内存以扩张大小, 也可以自我收缩以缓解内存使用压力。我们称正被缓存的存储设备为后备存储, 因为缓存背后的磁盘无疑才是所有缓存数据的归属。当内核开始一个读操作 (比如, 进程发起一个 `read()` 系统调用), 它首先会检查需要的数据是否在页高速缓存中。如果在, 则放弃访问磁盘, 而直接从内存中读取。这个行为称作缓存命中。如果数据没有在缓存中, 称为缓存未命中, 那么内核必须调度块 I/O 操作从磁盘去读取数据。然后内核将读来的数据放入页缓存中, 于是任何后续相同的数据读取都可命中缓存了。注意, 系统并不一定要将整个文件都缓存。缓存可以持有某个文件的全部内容, 也可以存储另一些文件的一页或者几页。到底该缓存谁取决于谁被访问到。

16.1.1 写缓存

上面解释了在读操作过程中页高速缓存的作用, 那么在进程写磁盘时, 比如执行 `write()` 系统调用, 缓存如何被使用呢? 通常来讲, 缓存一般被实现成下面三种策略之一: 第一种策略称为不缓存 (nowrite), 也就是说高速缓存不去缓存任何写操作。当对一个缓存中的数据片进行写时, 将直接跳过缓存, 写到磁盘, 同时也使缓存中的数据失效。那么如果后续读操作进行时, 需要再重新从磁盘中读取数据。不过这种策略很少使用, 因为该策略不但不去缓存写操作, 而且需要额外费力去使缓存数据失效。

第二种策略，写操作将自动更新内存缓存，同时也更新磁盘文件。这种方式，通常称为写透缓存（write-through cache），因为写操作会立刻穿透缓存到磁盘中。这种策略对保持缓存一致性很有好处——缓存数据时刻和后备存储保持同步，所以不需要让缓存失效，同时它的实现也最简单。

第三种策略，也是 Linux 所采用的，称为“回写”^①。在这种策略下，程序执行写操作直接写到缓存中，后端存储不会立刻直接更新，而是将页高速缓存中被写入的页面标记成“脏”，并且被加入到脏页链表中。然后由一个进程（回写进程）周期行将脏页链表中的页写回到磁盘，从而让磁盘中的数据和内存中最终一致。最后清理“脏”页标识。注意这里“脏页”这个词可能引起混淆，因为实际上脏的并非页高速缓存中的数据（它们是干干净净的），而是磁盘中的数据（它们已过时了）。也许更好的描述应该是“未同步”吧。尽管如此，我们说缓存内容是脏的，而不是说磁盘内容。回写策略通常认为要好于写透策略，因为通过延迟写磁盘，方便在以后的时间内合并更多的数据和再一次刷新。当然，其代价是实现复杂度高了许多。

16.1.2 缓存回收

缓存算法最后涉及的重要内容是缓存中的数据如何清除；或者是为更重要的缓存项腾出位置；或者是收缩缓存大小，腾出内存给其他地方使用。这个工作，也就是决定缓存中什么内容将被清除的策略，称为缓存回收策略。Linux 的缓存回收是通过选择干净页（不脏）进行简单替换。如果缓存中没有足够的干净页面，内核将强制地进行回写操作，以腾出更多的干净可用页。最困难的事情在于决定什么页应该回收。理想的回收策略应该是回收那些以后最不可能使用的页面。当然要知道以后的事情你必须是先知。也正是这个原因，理想的回收策略称为预测算法。但这种策略太理想了，无法真正实现。

1. 最近最少使用

缓存回收策略通过所访问的数据特性，尽量追求预测效率。最成功的算法（特别是对于通用目的的页高速缓存）称作最近最少使用算法，简称 LRU。LRU 回收策略需要跟踪每个页面的访问踪迹（或者至少按照访问时间为序的页链表），以便能回收最老时间戳的页面（或者回收排序链表头所指的页面）。该策略的良好效果源自于缓存的数据越久未被访问，则越不大可能近期再被访问，而最近被访问的最有可能被再次访问。但是，LRU 策略并非是放之四海而皆准的法则，对于许多文件被访问一次，再不被访问的情景，LRU 尤其失败。将这些页面放在 LRU 链的顶端显然不是最优，当然，内核并没办法知道一个文件只会被访问一次，但是它却知道过去访问了多少次。

2. 双链策略

Linux 实现的是一个修改过的 LRU，也称为双链策略。和以前不同，Linux 维护的不再是一个 LUR 链表，而是维护两个链表：活跃链表和非活跃链表。处于活跃链表上的页面被认为是

^① 有些书或者操作系统称这个策略是 copy-write 或者 write-behind 缓存。所有这些命名都是同义词。Linux 和其他 Unix 系统使用 write-back 称谓来描述缓存策略，使用 writeback 称谓描述写缓存数据到后备存储这一动作。本书遵循这种称谓。

“热”的且不会被换出，而在非活跃链表上的页面则是可以被换出的。在活跃链表中的页面必须在其被访问时就处于非活跃链表中。两个链表都被伪 LRU 规则维护：页面从尾部加入，从头部移除，如同队列。两个链表需要维持平衡——如果活跃链表变得过多而超过了非活跃链表，那么活跃链表的头页面将被重新移回到非活跃链表中，以便能再被回收。双链表策略解决了传统 LRU 算法中对仅一次访问的窘境。而且也更加简单的实现了伪 LRU 语义。这种双链表方式也称作 LUR/2。更普遍的是 n 个链表，故称 LRU/ n 。

我们现在知道页缓存如何构建（通过读和写），如何在写时被同步（通过回写）以及旧数据如何被回收来容纳新数据（通过双链表）。现在让我们看看真实世界应用场景中，页高速缓存如何帮助系统。假定你在开发一个很大的软件工程（比如 Linux 内核）那么你将有大量的源文件被打开，只要你打开读取源文件，这些文件就将被存储在页高速缓存中。只要数据被缓存，那么从一个文件跳到另一个文件将瞬间完成。当你编辑文件时，存储文件也会瞬间完成，因为写操作只需要写到内存，而不是磁盘。当你编译项目时，缓存的文件将使得编译过程更少访问磁盘，所以编译速度也就更快了。如果整个源码树太大了，无法一次性放入内存，那么其中一部分必须被回收——由于双链表策略，任何回收的文件都将处于非活跃链表，而且不大可能是你正在编译的文件。幸运的是，在你没在编译的时候，内核会执行页回写，刷新你所修改文件的磁盘副本。由此可见，缓存将极大地提高系统性能。为了看到差别，对比一下缓存冷（cache cold）时（也就是说重启后，编译你的大软件工程的时间）和缓存热（cache warm）时的差别吧。

16.2 Linux 页高速缓存

从名字可以看出，页高速缓存缓存的是内存页面。缓存中的页来自对正规文件、块设备文件和内存映射文件的读写。如此一来，页高速缓存就包含了最近被访问过的文件的数据块。在执行一个 I/O 操作前（比如 `read()` ^①操作），内核会检查数据是否已经在页高速缓存中了，如果所需要的数据确实在高速缓存中，那么内核可以从内存中迅速地返回需要的页，而不再需要从相对较慢的磁盘上读取数据。在接下来的章节里，我们将剖析具体的数据结构以及内核如何使用它们管理缓存。

16.2.1 address_space 对象

在页高速缓存中的页可能包含了多个不连续的物理磁盘块^②。也正是由于页面中映射的磁盘块不一定连续，所以在页高速缓存中检查特定数据是否已经被缓存是件颇为困难的工作。因为不能用设备名称和块号来做页高速缓存中的数据的索引，要不然这将是最简单的定位办法。

另外，Linux 页高速缓存对被缓存的页面范围定义非常宽泛。实际上，在最初 System V

① 如你在第 13 章所见，并非 `read()`、`write()` 系统调用执行实际的页 I/O 操作，而是通过文件系统提供的特定操作 `file->f_op->read()` 和 `file->f_op->write()` 完成。

② 比如，x86 体系结构中一个物理页大小是 4KB。而大多数文件系统的块大小仅仅 512KB。所以八个块才可以填满一个页面。另外因为文件本身可能分布在磁盘上的各个位置，所以页面中映射的块也不需要连续。

Release 4 引入页高速缓存时，仅仅只用作缓存文件系统数据，所以 SVR4 的页高速缓存使用它的等价文件对象（称为 vnode 结构体）管理页高速缓存。Linux 页高速缓存的目标是缓存任何基于页的对象，这包含各种类型的文件和各种类型的内存映射。

虽然 Linux 页高速缓存可以通过扩展 inode 结构体（见第 13 章）支持页 I/O 操作，但这种做法会将页高速缓存局限于文件。为了维持页高速缓存的普遍性（不应该将其绑定到物理文件或者 inode 结构体），Linux 页高速缓存使用了一个新对象管理缓存项和页 I/O 操作。这个对象是 address_space 结构体。该结构体是第 15 章介绍的虚拟地址 vm_area_struct 的物理地址对等体。当一个文件可以被 10 个 vm_area_struct 结构体标识（比如有 5 个进程，每个调用 mmap() 映射它两次），那么这个文件只能有一个 address_space 数据结构——也就是文件可以有多个虚拟地址，但是只能在物理内存有一份。与 Linux 内核中其他结构一样，address_space 也是文不对题，也许更应该叫它 page_cache_entity 或者 physical_pages_of_a_file。

该结构定义在文件 <linux/fs.h> 中，下面给出具体形式：

```
struct address_space {
    struct inode          *host;                /* 拥有节点 */
    struct radix_tree_root page_tree;           /* 包含全部页面的 radix 树 */
    spinlock_t           tree_lock;             /* 保护 page_tree 的自旋锁 */
    unsigned int          i_mmap_writable;       /* VM_SHARED 计数 */
    struct prio_tree_root i_mmap;               /* 私有映射链表 */
    struct list_head      i_mmap_nonlinear;     /* VM_NONLINEAR 链表 */
    spinlock_t           i_mmap_lock;           /* 保护 i_mmap 的自旋锁 */
    atomic_t             truncate_count;        /* 截断计数 */
    unsigned long         nrpages;              /* 页总数 */
    pgoff_t              writeback_index;       /* 回写的起始偏移 */
    struct address_space_operations *a_ops;     /* 操作表 */
    unsigned long         flags;                /* gfp_mask 掩码与错误标识 */
    struct backing_dev_info *backing_dev_info;  /* 预读信息 */
    spinlock_t           private_lock;          /* 私有 address_space 锁 */
    struct list_head      private_list;         /* 私有 address_space 链表 */
    struct address_space  *assoc_mapping;       /* 相关的缓冲 */
};
```

其中 i_mmap 字段是一个优先搜索树，它的搜索范围包含了在 address_space 中所有共享的与私有的映射页面。优先搜索树是一种巧妙地将堆与 radix 树结合^①的快速检索树。回忆早些提到的：一个被缓存的文件只和一个 address_space 结构体相关联，但它可以有多个 vm_area_struct 结构体——物理页到虚拟页是个一对多的映射。i_map 字段可帮助内核高效地找到关联的被缓存文件。

address_space 页总数由 nrpages 字段描述。

address_space 结构往往会和某些内核对象关联。通常情况下，它会与一个索引节点（inode）关联，这时 host 域就会指向该索引节点；如果关联对象不是一个索引节点的话，比如 address_space 和 swapper 关联时，host 域会被置为 NULL。

① 在内核中采用 Radix 优先搜索树是由 Edward M. McCreight 1985 年 5 月于 SIAM 计算机杂志第 14 期，第 2 集，257-276 页中提出的。

16.2.2 address_space 操作

a_ops 域指向地址空间对象中的操作函数表，这与 VFS 对象及其操作表关系类似，操作函数表定义在文件 <linux/fs.h> 中，由 address_space_operations 结构体来表示：

```
struct address_space_operations {
    int (*writepage)(struct page *, struct writeback_control *);
    int (*readpage)(struct file *, struct page *);
    int (*sync_page)(struct page *);
    int (*writepages)(struct address_space *,
                      struct writeback_control *);
    int (*set_page_dirty)(struct page *);
    int (*readpages)(struct file *, struct address_space *,
                     struct list_head *, unsigned);
    int (*write_begin)(struct file *, struct address_space *mapping,
                       loff_t pos, unsigned len, unsigned flags,
                       struct page **pagep, void **fsdata);
    int (*write_end)(struct file *, struct address_space *mapping,
                     loff_t pos, unsigned len, unsigned copied,
                     struct page *page, void *fsdata);
    sector_t (*bmap)(struct address_space *, sector_t);
    int (*invalidatepage)(struct page *, unsigned long);
    int (*releasepage)(struct page *, int);
    int (*direct_IO)(int, struct kiocb *, const struct iovec *,
                     loff_t, unsigned long);
    int (*get_xip_mem)(struct address_space *, pgoff_t, int,
                       void **, unsigned long *);
    int (*migratepage)(struct address_space *,
                       struct page *, struct page *);
    int (*launder_page)(struct page *);
    int (*is_partially_uptodate)(struct page *,
                                read_descriptor_t *,
                                unsigned long);
    int (*error_remove_page)(struct address_space *,
                             struct page *);
};
```

这些方法指针指向那些为指定缓存对象实现的页 I/O 操作。每个后备存储都通过自己的 address_space_operation 描述自己如何与页高速缓存交互。比如 ext3 文件系统在文件 fs/ext3/inode.c 中定义自己的操作表。这些方法提供了管理页高速缓存的各种行为，包括最常用的读页到缓存、更新缓存数据。这里面 readpage() 和 writepage() 两个方法最为重要。我们下面就来看看一个页面的读操作会包含哪些步骤。首先 linux 内核试图在页高速缓存中找到需要的数据；find_get_page() 方法负责完成这个检查动作。一个 address_space 对象和一个偏移量会传给 find_get_page() 方法，用于在页高速缓存中搜索需要的数据：

```
page = find_get_page(mapping, index);
```

这里 mapping 是指定的地址空间，index 是文件中的指定位置，以页面为单位（是的，称 address_space 结构体为 mapping，又是一个容易混淆的命名，虽然我也在重复这种内核命名不一

致的问题，但我还是鄙视它)。如果搜索的页并没在高速缓存中，`find_get_page()` 将会返回一个 `NULL`，并且内核将分配一个新页面，然后将之前搜索的页加入到页高速缓存中。

```
struct page *page;
int error;

/* 分配页...*/
page = page_cache_alloc_cold(mapping);
if(!page)
    /* 内存分配出错 */

/*.....然后将其加入到页面调整缓存 */
error = add_to_page_cache_lru(page, mapping, index, GFP_KERNEL);
if(error)
    /* 页面被加入到页面高速缓存时，出错 */
```

最后，需要的数据从磁盘被读入，再被加入页高速缓存，然后返回给用户：

```
error= mapping->a_ops->readpage(file, page);
```

写操作和读操作有少许不同。对于文件映射来说，当页被修改了，VM 仅仅需要调用：

```
SetPageDirty(page);
```

内核会在晚些时候通过 `writpage()` 方法把页写出。对特定文件的写操作比较复杂，它的代码在文件 `mm/filemap.c` 中，通常写操作路径要包含以下各步：

```
page = __grab_cache_page(mapping, index, &cached_page, &lru_pvec);
status = a_ops->prepare_write(file, page, offset, offset+bytes);
page_fault = filemap_copy_from_user(page, offset, buf, bytes);
status = a_ops->commit_write(file, page, offset, offset+bytes);
```

首先，在页高速缓存中搜索需要的页。如果需要的页不在高速缓存中，那么内核在高速缓存中新分配一空闲项；下一步，内核创建一个写请求；接着数据被从用户空间拷贝到了内核缓冲；最后将数据写入磁盘。

因为所有的页 I/O 操作都要执行上面这些步骤，这就保证了所有的页 I/O 操作必然都是通过页高速缓存进行的。因此，内核也总是试图先通过页高速缓存来满足所有的读请求。如果在页高速缓存中未搜索到需要的页，则内核将从磁盘读入需要的页，然后将该页加入到页高速缓存中；对于写操作，页高速缓存更像是一个存储平台，所有要被写出的页都要加入页高速缓存中。

16.2.3 基树

因为在任何页 I/O 操作前内核都要检查页是否已经在页高速缓存中了，所以这种频繁进行的检查必须迅速、高效，否则搜索和检查页高速缓存的开销可能抵消页高速缓存带来的好处（至少在缓存命中率很低的时候，搜索的开销足以抵消以内存代替磁盘进行检索数据带来的好处）。

正如在 16.2.2 节所看到的，页高速缓存通过两个参数 `address_space` 对象加上一个偏移量进行搜索。每个 `address_space` 对象都有唯一的基树（radix tree），它保存在 `page_tree` 结构体中。基树是一个二叉树，只要指定了文件偏移量，就可以在基树中迅速检索到希望的页。页高速缓存的搜索函数 `find_get_page()` 要调用函数 `radix_tree_lookup()`，该函数会在指定基树中搜索指定页面。

基树核心代码的通用形式可以在文件 `lib/radix-tree.c` 中找到。另外，要想使用基树，需要包含头文件 `<linux/radix_tree.h>`。

16.2.4 以前的页散列表

在 2.6 版本以前，内核页高速缓存不是通过基树检索，而是通过一个维护了系统中所有页的全局散列表进行检索。对于给定的一个键值，该散列表会返回一个双向链表的入口对应于这个所给定的值。如果需要的页贮存在缓存中，那么链表中的一项就会与其对应。否则，页就不在页面高速缓存中，散列函数返回 `NULL`。

全局散列表主要存在四个问题：

- 由于使用单个的全局锁保护散列表，所以即使在中等规模的机器中，锁的争用情况也会相当严重，造成性能受损。
- 由于散列表需要包含所有页高速缓存中的页，可是搜索需要的只是和当前文件相关的那些页，所以散列表包含的页面相比搜索需要的页面要大得多。
- 如果散列搜索失败（也就是给定的页不在页高速缓存中），执行速度比希望的要慢得多，这是因为检索必须遍历指定散列键值对应的整个链表。
- 散列表比其他方法会消耗更多的内存。

2.6 版本内核中引入基于基树的页高速缓存来解决这些问题。

16.3 缓冲区高速缓存

独立的磁盘块通过块 I/O 缓冲也要被存入页高速缓存。回忆一下第 14 章，一个缓冲是一个物理磁盘块在内存里的表示。缓冲的作用就是映射内存中的页面到磁盘块，这样一来页高速缓存在块 I/O 操作时也减少了磁盘访问，因为它缓存磁盘块和减少块 I/O 操作。这个缓存通常称为缓冲区高速缓存，虽然实现上它没有作为独立缓存，而是作为页高速缓存的一部分。

块 I/O 操作一次操作一个单独的磁盘块。普遍的块 I/O 操作是读写 `i` 节点。内核提供了 `bread()` 函数实现从磁盘读一个块的底层操作。通过缓存，磁盘块映射到它们相关的内存页，并缓存到页高速缓存中。

缓冲和页高速缓存并非天生就是统一的，2.4 内核的主要工作之一就是统一它们。在更早的内核中，有两个独立的磁盘缓存：页高速缓存和缓冲区高速缓存。前者缓存页面，后者缓存缓冲区，这两个缓存并没有统一。一个磁盘块可以同时存于两个缓存中，这导致必须同步操作两个缓冲中的数据，而且浪费了内存，去存储重复的缓存项。今天我们只有一个磁盘缓存，即页高速缓存。虽然如此，内核仍然需要在内存中使用缓冲来表示磁盘块，幸好，缓冲是用页映射块的，所以它正好在页高速缓存中。

16.4 flusher 线程

由于页高速缓存的缓存作用，写操作实际上会被延迟。当页高速缓存中的数据比后台存储的数据更新时，该数据就称作脏数据。在内存中累积起来的脏页最终必须被写回磁盘。在以下 3 种情况发生时，脏页被写回磁盘：

- 当空闲内存低于一个特定的阈值时，内核必须将脏页写回磁盘以便释放内存，因为只有干净（不脏的）内存才可以被回收。当内存干净后，内核就可以从缓存清理数据，然后收缩缓存，最终释放出更多的内存。
- 当脏页在内存中驻留时间超过一个特定的阈值时，内核必须将超时的脏页写回磁盘，以确保脏页不会无限期地驻留在内存中。
- 当用户进程调用 `sync()` 和 `fsync()` 系统调用时，内核会按要求执行回写动作。

上面三种工作的目的完全不同。实际上，在旧内核中，这是由两个独立的内核线程（请看后面章节）分别完成的。但是在 2.6 内核中，由一群[⊖]内核线程（flusher 线程）执行这三种工作。

首先，flusher 线程在系统中的空闲内存低于一个特定的阈值时，将脏页刷新写回磁盘。该后台回写例程的目的在于——在可用物理内存过低时，释放脏页以重新获得内存。这个特定的内存阈值可以通过 `dirty_background_ratio` sysctl 系统调用设置。当空闲内存比阈值 `dirty_background_ratio` 还低时，内核便会调用函数 `flusher_threads()`[⊖] 唤醒一个或多个 flusher 线程，随后 flusher 线程进一步调用函数 `bdi_writeback_all()` 开始将脏页写回磁盘。该函数需要一个参数——试图写回的页面数目。函数连续地写出数据，直到满足以下两个条件：

- 已经有指定的最小数目的页被写出到磁盘。
- 空闲内存数已经回升，超过了阈值 `dirty_background_ratio`。

上述条件确保了 flusher 线程操作可以减轻系统中内存不足的压力。回写操作不会在达到这两个条件前停止，除非刷新者线程写回了所有的脏页，没有剩下的脏页可再被写回了。

为了满足第二个目标，flusher 线程后台例程会被周期性唤醒（和空闲内存是否过低无关），将那些在内存中驻留时间过长的脏页写出，确保内存中不会有长期存在的脏页。如果系统发生崩溃，由于内存处于混乱之中，所以那些在内存中还没来得及写回磁盘的脏页就会丢失，所以周期性同步页高速缓存和磁盘非常重要。在系统启动时，内核初始化一个定时器，让它周期地唤醒 flusher 线程，随后使其运行函数 `wb_writeback()`。该函数将把所有驻留时间超过 `dirty_expire_interval` ms 的脏页写回。然后定时器将再次被初始化为 `dirty_expire_centisecs` 秒后唤醒 flusher 线程。总而言之，flusher 线程周期性地被唤醒并且把超过特定期限的脏页写回磁盘。

系统管理员可以在 `/proc/sys/vm` 中设置回写相关的参数，也可以通过 sysctl 系统调用设置它们。表 16-1 列出了与 `pdflush` 相关的所有可设置变量。

表 16-1 页回写设置

变 量	描 述
<code>dirty_background_ratio</code>	占全部内存的百分比。当内存中空闲页达到这个比例时， <code>pdflush</code> 线程开始回写脏页
<code>dirty_expire_interval</code>	该数值以百分之一秒为单位，它描述超时多久的数据将被周期性执行的 <code>pdflush</code> 线程写出
<code>dirty_ratio</code>	占全部内存百分比，当一个进程产生的脏页达到这个比例时，就开始被写出
<code>dirty_writeback_interval</code>	该数值以百分之一秒为单位，它描述 <code>pdflush</code> 线程的运行频率
<code>laptop_mode</code>	一个布尔值，用于控制膝上型计算机模式，具体请见后续内容

⊖ 术语“群”通常在计算机科学中指的是一组可以并行执行的事情。

⊖ 是的，它的确命名错了，它应该称为 `wakeup_bdflush()`。原因请看后面关于这个调用的继承部分。

flusher 线程的实现代码在文件 `mm/page-writeback.c` 和 `mm/backing-dev.c` 中，回写机制的实现代码在文件 `fs/fs-writeback.c` 中。

16.4.1 膝上型计算机模式

膝上型计算机模式是一种特殊的页回写策略，该策略主要意图是将硬盘转动的机械行为最小化，允许硬盘尽可能长时间地停滞，以此延长电池供电时间。该模式可通过 `/proc/sys/vm/laptop_mode` 文件进行配置。通常，上述配置文件内容为 0，也就是说膝上型计算机模式关闭，如果需要启用膝上型计算机模式，则向配置文件中写入 1。

膝上型计算机模式的页回写行为与传统方式相比只有一处变化。除了当缓存中的页面太旧时要执行回写脏页以外，flusher 还会找准磁盘运转的时机，把所有其他的物理磁盘 I/O、刷新脏缓冲等通通写回到磁盘，以便保证不会专门为了写磁盘而去主动激活磁盘运行。

上述回写行为变化要求 `dirty_expire_interval` 和 `dirty_writeback_interval` 两阈值必须设置得更大，比如 10 分钟。因为磁盘运转并不很频繁，所以用这样长的回写延迟就能保证膝上型计算机模式可以等到磁盘运转机会写入数据。因为关闭磁盘驱动器是节电的重要手段，膝上模式可以延长膝上计算机依靠电池的续航能力。其坏处则是系统崩溃或者其他错误会使得数据丢失。

多数 Linux 发布版会在计算机接上电池或拔掉电池时，自动开启或禁止膝上型计算机模式以及其他需要的回写可调节开关。因此机器可在使用电池电源时自动进入膝上型计算机模式，而在插上交流电源时恢复到常规的页回写模式。

16.4.2 历史上的 bdflush、kupdated 和 pdflush

在 2.6 版本前，flusher 线程的工作是分别由 bdflush 和 kupdated 两个线程共同完成。

当可用内存过低时，bdflush 内核线程在后台执行脏页回写操作。类似 flusher，它也有一组阈值参数，当系统中空闲内存消耗到特定阈值以下时，bdflush 线程就被 `wakeup_bdflush()` 函数唤醒。

bdflush 和当前的 flusher 线程之间存在两个主要区别。第一个区别是系统中只有一个 bdflush 后台线程，而 flusher 线程的数目却是根据磁盘数量变化的（这在 16.5 节中会谈到）；第二个区别是 bdflush 线程基于缓冲，它将脏缓冲写回磁盘。相反，flusher 线程基于页面，它将整个脏页写回磁盘。当然，页面可能包含缓冲，但是实际 I/O 操作对象是整页，而不是块。因为页在内存中是更普遍和普通的概念，所以管理页相比管理块要简单。

因为只有在内存过低和缓冲数量过大时，bdflush 例程才刷新缓冲，所以 kupdated 例程被引入，以便周期地写回脏页。它和 pdflush 线程的 `wb_writeback()` 函数提供同样的服务。

在 2.6 内核中，bdflush 和 kupdated 已让路给了 pdflush 线程——page dirty flush（比以前两个更容易令人混淆的名字）的缩写。Pdflush 线程的执行和今天的 flusher 线程类似。其主要区别在于，pdflush 线程数目是动态的，默认是 2 个到 8 个，具体多少取决于系统 I/O 的负载。Pdflush 线程与任何任务都无关，它们是面向系统所有磁盘的全局任务。这样做的好处是实现简单，可带来的问题是，pdflush 线程很容易在拥塞的磁盘上绊住，而现代硬件发生拥塞更是家常便饭。采用每个磁盘一个刷新线程可以使得 I/O 操作同步执行，简化了拥塞逻辑，也提升

了性能。Flusher 线程在 2.6.32 内核系列中取代了 pdflush 线程（针对每个磁盘独立执行回写操作是其和 pdflush 的主要区别）。本节中剩下部分的讨论，仍然适用于 pdflush，而且也适用于所有 2.6 内核系列。

16.4.3 避免拥塞的方法：使用多线程

使用 bdflush 线程最主要的一个缺点就是，bdflush 仅仅包含了一个线程，因此很有可能在页回写任务很重时，造成拥塞。这是因为单一的线程有可能堵塞在某个设备的已拥塞请求队列（正在等待将请求提交给磁盘的 I/O 请求队列）上，而其他设备的请求队列却没法得到处理。如果系统有多个磁盘和较强的处理能力，内核应该能使得每个磁盘都处于忙状态。不幸的是，即使还有许多数据需要回写，单个的 bdflush 线程也可能会堵塞在某个队列的处理上，不能使所有磁盘都处于饱和的工作状态，原因在于磁盘的吞吐量是非常有限的。正是因为磁盘的吞吐量很有限，所以如果只有唯一线程执行页回写操作，那么这个线程很容易苦苦等待对一个磁盘上的操作。为了避免出现这种情况，内核需要多个回写线程并发执行，这样单个设备队列的拥塞就不会成为系统瓶颈了。

2.6 内核通过使用多个 flusher 线程来解决上述问题。每个线程可以相互独立地将脏页刷新回磁盘，而且不同的 flusher 线程处理不同的设备队列。pdflush 线程策略中，线程数是动态变化的。每一个线程试图尽可能忙地从每个超级块的脏页链表中回收数据，并且写回到磁盘。pdflush 方式避免了因为一个忙磁盘，而使得其余磁盘饥饿的状况。通常情况下这样是不错的，但是如果每个 pdflush 线程在同一个拥塞的队列上挂起了又该如何呢？在这种情况下，多个 pdflush 线程可能并不比一个线程更好，就浪费的内存而言就要多许多。为了减轻上述影响，pdflush 线程采用了拥塞回避策略：它们会主动尝试从那些没有拥塞的队列回写页。从而，pdflush 线程将其工作调度开来，防止了仅仅欺负某一个忙碌设备。

这种方式效果确实不错，但是拥塞回避并不完美。在现代操作系统中，因为 I/O 总线技术和计算机其他部分相比发展要缓慢得多，所以拥塞现象时常发生——处理器发展速度遵循摩尔定理，但是硬盘驱动器则仅仅比 20 年前快一点点。要知道，目前除了 pdflush 以外，I/O 系统中还没有其他地方使用这种拥塞回避处理。不过在很多情况下，pdflush 确实可以避免向特定盘回写的时间和期望时间相比太久。当前 flusher 线程模型（自 2.6.32 内核系列以后采用）和具体块设备关联，所以每个给定线程从每个给定设备的脏页链表收集数据，并写回到对应磁盘。回写于是更趋于同步了，而且由于每个磁盘对应一个线程，所以线程也不需要采用复杂的拥塞避免策略，因为一个磁盘就一个线程操作。该方法提高了 I/O 操作的公平性，而且降低了饥饿风险。

因为使用 pdflush 以及后来的 flusher 线程提升了页回写性能。2.6 内核系列相比早期内核可以让磁盘利用更饱和。在系统 I/O 很重的时候，flusher 线程可以在每个磁盘上都维护更高的吞吐量。

16.5 小结

本章中我们看到了 Linux 的页高速缓存和页回写。了解了内核如何通过页缓存执行页 I/O 操作以及这些页高速缓存（通过存储数据在内存中）可以利用减少磁盘 I/O，从而极大地提升系统

的性能。我们讨论了通过称为“回写缓存”的进程维护在缓存中的更新页面——具体做法是标记内存中的页面为脏，然后找时机延迟写到磁盘中。Flusher 内核线程将负责处理这些最终的页回写操作。

通过最近几章的学习，你应该已经对内存与文件系统有了深刻认识，那么接下来我们将进入模块专题，去学习 Linux 的设备驱动以及内核如何被模块化、在运行时插入和删除内核代码的动态机制。



第 17 章

设备与模块

在本章中，关于设备驱动和设备管理，我们讨论四种内核成分。

- 设备类型：在所有 Unix 系统中为了统一普通设备的操作所采用的分类。
- 模块：Linux 内核中用于按需加载和卸载目标码的机制。
- 内核对象：内核数据结构中支持面向对象的简单操作，还支持维护对象之间的父子关系。
- sysfs：表示系统中设备树的一个文件系统。

17.1 设备类型

在 Linux 以及所有 Unix 系统中，设备被分为以下三种类型：

- 块设备
- 字符设备
- 网络设备

块设备通常缩写为 blkdev，它是可寻址的，寻址以块为单位，块大小随设备不同而不同；块设备通常支持重定位（seeking）操作，也就是对数据的随机访问。块设备的例子有硬盘、蓝光光碟，还有如 Flash 这样的存储设备。块设备是通过称为“块设备节点”的特殊文件来访问的，并且通常被挂载为文件系统。我们在第 13 章已经讨论过了文件系统，在第 14 章已经讨论过了块设备。

字符设备通常缩写为 cdev，它是不可寻址的，仅提供数据的流式访问，就是一个个字符，或者一个个字节。字符设备的例子有键盘、鼠标、打印机，还有大部分伪设备。字符设备是通过称为“字符设备节点”的特殊文件来访问的。与块设备不同，应用程序通过直接访问设备节点与字符设备交互。

网络设备最常见的类型有时也以以太网设备 (ethernet devices) 来称呼，它提供了对网络（例如 Internet）的访问，这是通过一个物理适配器（如你的膝上型计算机的 802.11 卡）和一种特定的协议（如 IP 协议）进行的。网络设备打破了 Unix 的“所有东西都是文件”的设计原则，它不是通过设备节点来访问，而是通过套接字 API 这样的特殊接口来访问。

Linux 还提供了不少其他设备类型，但都是针对单个任务，而非通用的。一个特例是“杂项设备” (miscellaneous device)，通常简写为 miscdev，它实际上是个简化的字符设备。杂项设备使驱动程序开发者能够很容易地表示一个简单设备——实际上是对通用基本架构的一种折中。

并不是所有设备驱动都表示物理设备。有些设备驱动是虚拟的，仅提供访问内核功能而已。我们称为“伪设备” (pseudo device)，最常见的如内核随机数发生器（通过 /dev/random 和 /dev/urandom 访问）、空设备（通过 /dev/null 访问）、零设备（通过 /dev/zero 访问）、满设备

(通过 /dev/full 访问), 还有内存设备 (通过 /dev/mem 访问)。然而, 大部分设备驱动是表示物理设备的。

17.2 模块

尽管 Linux 是“单块内核”(monolithic)的操作系统——这是说整个系统内核都运行于一个单独的保护域中, 但是 Linux 内核是模块化组成的, 它允许内核在运行时动态地向其中插入或从中删除代码。这些代码 (包括相关的子例程、数据、函数入口和函数出口) 被一并组合在一个单独的二进制镜像中, 即所谓的可装载内核模块中, 或简称为模块。支持模块的好处是基本内核镜像可以尽可能地小, 因为可选的功能和驱动程序可以利用模块形式再提供。模块允许我们方便地删除和重新载入内核代码, 也方便了调试工作。而且当热插拔新设备时, 可通过命令载入新的驱动程序。

本章我们将探寻内核模块的奥秘, 同时也学习如何编写自己的内核模块。

17.2.1 Hello, World

与开发我们已经讨论过的大多数内核核心子系统不同, 模块开发更接近编写新的应用系统, 因为至少在模块文件中具有入口和出口点。

虽然编写“Hello, World”程序作为实例实属陈词滥调了, 但它的确很让人喜爱。内核模块 Hello, World 出场了:

```
/*
 * hello.c _The Hello, World! 我们的第一个内核模块
 */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
/*
 * hello_init- 初始化函数, 当模块装载时被调用, 如果成功装载返回零, 否
 * 则返回非零值
 */
static int hello_init(void)
{
    printk(KERN_ALERT "I bear a charmed life.\n");
    return 0;
}

/*
 * hello_exit- 退出函数, 当模块卸载时被调用
 */
static void hello_exit(void)
{
    printk(KERN_ALERT "Out, out, brief candle!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Shakespeare");  
MODULE_DESCRIPTION("A Hello, World Module");
```

这大概是我们所能见到的最简单的内核模块了，`hello_init()` 函数是模块的入口点，它通过 `module_init()` 例程注册到系统中，在内核装载时被调用。调用 `module_init()` 实际上不是真正的函数调用，而是一个宏调用，它唯一的参数便是模块的初始化函数。模块的所有初始化函数必须符合下面的形式：

```
int my_init(void);
```

因为 `init` 函数通常不会被外部函数直接调用，所以你不必导出该函数，故它可标记为 `static` 类型。

`init` 函数会返回一个 `int` 型数值，如果初始化（或你的 `init` 函数想做的事情）顺利完成，那么它的返回值为零；否则返回一个非零值。

这个 `init` 函数仅仅打印了一条简单的消息，然后返回零。在实际的模块中，`init` 函数还会注册资源、初始化硬件、分配数据结构等。如果这个文件被静态编译进内核映像中，其 `init` 函数将存放在内核映像中，并在内核启动时运行。

`hello_exit()` 函数是模块的出口函数，它由 `module_exit()` 例程注册到系统。在模块从内存卸载时，内核便会调用 `hello_exit()`。退出函数可能会在返回前负责清理资源，以保证硬件处于一致状态；或者做其他的一些操作。简单说来，`exit` 函数负责对 `init` 函数以及在模块生命周期过程中所做的一切事情进行撤销工作，基本上就是清理工作。在退出函数返回后，模块就被卸载了。

退出函数必须符合以下形式：

```
void my_exit(void);
```

与 `init` 函数一样，你也可以标记其为 `static`。

如果上述文件被静态地编译到内核映像中，那么退出函数将不被包含，而且永远都不会被调用（因为如果不是编译成模块的话，那么代码就不需从内核中卸载）。

`MODULE_LICENSE()` 宏用于指定模块的版权。如果载入非 GPL 模块到系统内存，则会在内核中设置被污染标识——这个标识只起到记录信息的作用。版权许可证具有两大目的。首先，它具有通告的目的。当 `oops` 中设置了被污染的标识时，很多内核开发者对 `bug` 的报告缺乏信任，因为他们认为二进制模块（也就是开发者不能调试它）被装载到了内核。其次，非 GPL 模块不能调用 `GPL_only` 符号，本章后续的“导出符号表”一节将对其加以描述。

最后还要说明，`MODULE_AUTHOR()` 宏和 `MODULE_DESCRIPTION()` 宏指定了代码作者和模块的简要描述，它们完全是用作信息记录目的。

17.2.2 构建模块

在 2.6 内核中，由于采用了新的“`kbuild`”构建系统，现在构建模块相比从前更加容易。构建过程的第一步是决定在哪里管理模块源码。你可以把模块源码加入到内核源代码树中，或者是作为一个补丁或者是最终把你的代码合并到正式的内核代码树中；另一种可行的方式是在内核源

代码树之外维护和构建你的模块源码。

1. 放在内核源代码树中

最理想的情况莫过于你的模块正式成为 Linux 内核的一部分，这样就会被存放入内核源代码树中。把你的模块代码正确地置于内核中，开始的时候难免需要更多的维护，但这样通常是一劳永逸的解决之道。

当你决定了把你的模块放入内核源代码树中，下一步要清楚你的模块应在内核源代码树中处于何处。设备驱动程序存放在内核源码树根目录下 `/drivers` 的子目录下，在其内部，设备驱动文件被进一步按照类别、类型或特殊驱动程序等更有序地组织起来。如字符设备存在于 `drivers/char/` 目录下，而块设备存放在 `drivers/block/` 目录下，USB 设备则存放在 `drivers/usb/` 目录下。文件的具体组织规则并不须绝对墨守成规，不容打破，你可看到许多 USB 设备也属于字符设备。但是不管怎么样，这些组织关系对我们来说相当容易理解，而且很也准确。

假定你有一个字符设备，而且希望将它存放在 `drivers/char/` 目录下，那么要注意，在该目录下同时会存在大量的 C 源代码文件和许多其他目录。所以对于仅仅只有一两个源文件的设备驱动程序，可以直接存放在该目录下；但如果驱动程序包含许多源文件和其他辅助文件，那么可以创建一个新子目录。这期间并没有什么金科玉律。假设想建立自己代码的子目录，你的驱动程序是一个钓鱼竿和计算机的接口，名为 Fish Master XL 3000，那么你需要在 `drivers/char/` 目录下建立一个名为 `fishing` 的子目录。

接下来需要向 `drivers/char/` 下的 Makefile 文件中添加一行。编辑 `drivers/char/Makefile/` 并加入：

```
obj-m += fishing/
```

这行编译指令告诉模块构建系统，在编译模块时需要进入 `fishing/` 子目录中。更可能发生的情况是，你的驱动程序的编译取决于一个特殊配置选项；比如，可能的 `CONFIG_FISHING_POLE`（请看 17.2.6 节，它会告诉你如何加入一个新的编译选项）。如果这样，你需要用下面的指令代替刚才那条指令：

```
obj-$(CONFIG_FISHING_POLE) += fishing/
```

最后，在 `drivers/char/fishing/` 下，需要添加一个新 Makefile 文件，其中需要有下面这行指令：

```
obj-m += fishing.o
```

一切就绪了，此刻构建系统运行将会进入 `fishing/` 目录下，并且将 `fishing.c` 编译为 `fishing.ko` 模块。虽然你写的扩展名是 `.o`，但是模块被编译后的扩展名却是 `.ko`。

再一个可能，要是你的钓鱼竿驱动程序编译时有编译选项，那么你可能需要这么来做：

```
obj-$(CONFIG_FISHING_POLE) += fishing.o
```

以后，假如你的钓鱼竿驱动程序需要更加智能化——它可以自动检测钓鱼线，这可是最新的鱼竿“必备要求”呀。这时驱动程序源文件可能就不再只有一个了。别怕，朋友，你只要把你的 Makefile 做如下修改就可搞定：

```
obj-$(CONFIG_FISHING_POLE) += fishing.o
fishing-objs := fishing-main.o fishing-line.o
```

每当设置了 `CONFIG_FISHING_POLE`, `fishing-main.c` 和 `fishing-line.c` 就一起被编译和连接到 `fishing.ko` 模块内。

最后一个注意事项是, 在构建文件时你可能需要额外的编译标记, 如果这样, 你只需在 `Makefile` 中添加如下指令:

```
EXTRA_CFLAGS += -DTITANIUM_POLE
```

如果喜欢把你的源文件置于 `drivers/char/` 目录下, 并且不建立新目录的话, 那么你要做的便是将前面提到的行 (也就是原来处于 `drivers/char/fishing/` 下你自己的 `Makefile` 中的) 都加入到 `drivers/char/Makefile` 中。

开始编译吧, 运行内核构建过程和原来一样。如果你的模块编译取决于配置选项, 比如有 `CONFIG_FISHING_POLE` 约束, 那么在编译前首先要确保选项被允许。

2. 放在内核代码外

如果你喜欢脱离内核源代码树来维护和构建你的模块, 把自己作为一个圈外人, 那你要做的就是在你自己的源代码树目录中建立一个 `Makefile` 文件, 它只需要一行指令:

```
obj-m := fishing.o
```

这条指令就可把 `fishing.c` 编译成 `fishing.ko`。如果你有多个源文件, 那么用两行就足够:

```
obj-m := fishing.o
fishing-objs := fishing-main.o fishing-line.o
```

这样一来, `fishing-main.c` 和 `fishing-line.c` 就一起被编译和连接到 `fishing.ko` 模块内了。

模块在内核内和在内核外构建的最大区别在于构建过程。当模块在内核源代码树外围时, 你必须告诉 `make` 如何找到内核源代码文件和基础 `Makefile` 文件。不过要完成这个工作同样不难:

```
make -C /kernel/source/location SUBDIRS=$PWD modules
```

在这个例子中, `/kernel/source/location` 是你配置的内核源代码树。回想一下, 不要把要处理的内核源代码树放在 `/usr/src/linux` 下, 而要移到你 `home` 目录下某个方便访问的地方。

17.2.3 安装模块

编译后的模块将被装入到目录 `/lib/modules/version/kernel/` 下, 在 `kernel/` 目录下的每一个目录都对应着内核源码树中的模块位置。如果使用的是 2.6.34 内核, 而且将你的模块源代码直接放在 `drivers/char/` 下, 那么编译后的钓鱼竿驱动程序的存放路径将是: `/lib/modules/2.6.34/kernel/drivers/char/fishing.ko`。

下面的构建命令用来安装编译的模块到合适的目录下:

```
make modules_install
```

通常需要以 `root` 权限运行。

17.2.4 产生模块依赖性

Linux 模块之间存在依赖性, 也就是说钓鱼模块依赖于鱼饵模块, 那么当你载入钓鱼模块

时，鱼饵模块会被自动载入。这里需要的依赖信息必须事先生成。多数 Linux 发布版都能自动产生这些依赖关系信息，而且在每次启动时更新。若想产生内核依赖关系的信息，root 用户可运行命令

```
depmod
```

为了执行更快的更新操作，那么可以只为新模块生成依赖信息，而不是生成所有的依赖关系，这时 root 用户可运行命令

```
depmod -A
```

模块依赖关系信息存放在 `/lib/modules/version/modules.dep` 文件中。

17.2.5 载入模块

载入模块最简单的方法是通过 `insmod` 命令，这是个功能很有限的命令，它能做的就是请求内核载入指定的模块。`insmod` 程序不执行任何依赖性分析或进一步的错误检查。它用法简单，以 root 身份运行命令：

```
insmod module.ko
```

这里，`module.ko` 是要载入的模块名称。比如装载钓鱼竿模块，那你就执行命令：

```
insmod fishing.ko
```

类似的，卸载一个模块，你可使用 `rmmod` 命令，它同样需要以 root 身份运行：

```
rmmod module
```

比如，`rmmod fishing` 命令将卸载钓鱼竿模块。

```
rmmod fishing
```

这两个命令是很简单，但是它们一点也不智能。先进工具 `modprobe` 提供了模块依赖性分析、错误智能检查、错误报告以及许多其他功能和选项。我强烈建议大家用这个命令。

为了在内核 via `modprobe` 中插入模块，需要以 root 身份运行：

```
modprobe module [ module parameters ]
```

其中，参数 `module` 指定了需要载入的模块名称，后面的参数将在模块加载时传入内核。（请看 17.2.7 一节对模块参数的讨论）。

`modprobe` 命令不但会加载指定的模块，而且会自动加载任何它所依赖的有关模块。所以说它是加载模块的最佳机制。

`modprobe` 命令也可用来从内核中卸载模块，当然这也需要以 root 身份运行：

```
modprobe -r modules
```

参数 `modules` 指定一个或多个需要卸载的模块。与 `rmmod` 命令不同，`modprobe` 也会卸载给定模块所依赖的相关模块，但其前提是这些相关模块没有被使用。Linux 用户手册第 8 部分提供

了上述命令的使用参考，里面包括了命令选项和用法。

17.2.6 管理配置选项

在前面的内容中我们看到，只要设置了 CONFIG_FISHING_POLE 配置选项，钓鱼竿模块将被自动编译。虽然配置选项在前面已经讨论过了，但这里我们将继续以钓鱼竿驱动程序为例，再看看一个新的配置选项如何加入。

由于 2.6 内核中新引入了“kbuild”系统，因此，加入一个新配置选项现在可以说是易如反掌。你所需做的全部就是向 kconfig 文件中添加一项，用以对应内核源码树。对驱动程序而言，kconfig 通常和源代码处于同一目录。如果钓鱼竿驱动程序在目录 drivers/char/ 下，那么你便会发现 drivers/char/kconfig 也同时存在。

如果你建立了一个新子目录，而且也希望 kconfig 文件存在于该目录中的话，那么你必须在一个已存在的 kconfig 文件中将它引入。你需要加入下面一行指令：

```
source "drivers/char/fishing/Kconfig"
```

这里所谓存在的 Kconfig 文件可能是 drivers/char/Kconfig。

Kconfig 文件很方便加入一个配置选型，请看钓鱼竿模块的选项，如下所示：

```
config FISHING_POLE
    tristate "Fish Master 3000 support"
    default n
    help
        If you say Y here, support for the Fish Master 3000 with computer
        interface will be compiled into the kernel and accessible via a
        device node. You can also say M here and the driver will be built as a
        module named fishing.ko.

    If unsure, say N.
```

配置选项第一行定义了该选项所代表的配置目标。注意 CONFIG_ 前缀并不需要写上。

第二行声明选项类型为 tristate，也就是说可以编译进内核（Y），也可作为模块编译（M），或者干脆不编译它（N）。如果编译选项代表的是一个系统功能，而不是一个模块，那么编译选项将用 bool 指令代替 tristate，这说明它不允许被编译成模块。处于指令之后的引号内文字为该选项指定了名称。

第三行指定了该选项的默认选择，这里默认操作是不编译它（N）。也可以把默认选择指定为编译进内核（Y），或者编译成一个模块（M）。对驱动程序而言，默认选择通常为不编译进内核（N）。

Help 指令的目的是为该选项提供帮助文档。各种配置工具都可以按要求显示这些帮助。因为这些帮助是面向编译内核的用户和开发者的，所以帮助内容简洁扼要。一般的用户通常不会编译内核，但如果他们想试试，往往也能理解配置帮助的意思。

除了上述选项以外，还存在其他选项。比如 depends 指令规定了在该选项被设置前，首先要设置的选项。假如依赖性不满足，那么该选项就被禁止。比如，如果你加入指令：

```
depends on FISH_TANK
```

到配置选项中,那么就意味着在 CONFIG_FISH_TANK 被选择前,我们的钓鱼竿模块是不能使用的 (Y 或者 M)。Select 指令和 depends 类似,它们只有一点不同之处——只要是 select 指定了谁,它就会强行将被指定的选项打开。所以这个指令可不能向 depends 那样滥用一通,因为它会自动的激活其他配置选项。它的用法和 depends 一样。比如:

```
select BAIT
```

意味着当 CONFIG_FISHING_POLE 被激活时,配置选项 CONFIG_BAIT 必然一起被激活。

如果 select 和 depends 同时指定多个选项,那就需要通过 && 指令来进行多选。使用 depends 时,你还可以利用叹号前缀来指明禁止某个选项。比如:

```
depends on EXAMPLE_DRIVERS && !NO_FISHING_ALLOWED
```

这行指令就指定驱动程序安装要求打开 CONFIG_EXAMPLE_DRIVERS 选项,同时要禁止 CONFIG_NO_FISHING_ALLOWED 选项。

tristate 和 bool 选项往往会结合 if 指令一起使用,这表示某个选项取决于另一个配置选项。如果条件不满足,配置选项不但会被禁止,甚至不会显示在配置工具中,比如,要求配置系统只有在 CONFIG_x86 配置选项设置时才显示某选项。请看下面指令:

```
bool "Deep Sea Mode" if OCEAN
```

If 指令也可与 default 指令结合使用,强制只有在条件满足时 default 选项才有效。

配置系统导出了一些元选项 (meta-option) 以简化生成配置文件。比如选项 CONFIG_EMBEDDED 是用于关闭那些用户想要禁止的关键功能 (比如要在嵌入系统中节省珍贵的内存); 选项 CONFIG_BROKEN_ON_SMP 用来表示驱动程序并非多处理器安全的。通常该项不应设置,标记它的目的是确保用户能知道该驱动程序的弱点。当然,新的驱动程序不应该使用该标志。最后要说明 CONFIG_EXPERIMENTAL 选项,它是一个用于说明某项功能尚在试验或处于 beta 阶段的标志选项。该选项默认情况下关闭,同样,标记它的目的是为了用户在使用驱动程序前明白潜在风险。

17.2.7 模块参数

Linux 提供了这样一个简单框架——它可允许驱动程序声明参数,从而用户可以在系统启动或者模块装载时再指定参数值,这些参数对于驱动程序属于全局变量。值得一提的是模块参数时也将出现在 sysfs 文件系统中 (见本章后面的介绍),这样一来,无论是生成模块参数,还是管理模块参数的方式都变得灵活多样了。

定义一个模块参数可通过宏 module_param() 完成:

```
module_param(name, type, perm);
```

参数 name 既是用户可见的参数名,也是你模块中存放模块参数的变量名。参数 type 则存放了参数的类型,它可以是 byte、short、ushort、int、uint、long、ulong、charp、bool 或 invbool,它们分别代表字节型、短整型、无符号短整形、整型、无符号整型、长整形、无符号长整型、字

符指针、布尔型，以及应用户要求转换得来的布尔型。其中 `byte` 类型存放在 `char` 类型变量中，`boolean` 类型存放在 `int` 变量中，其余的类型都一致对应 C 语言的变量类型。最后一个参数 `perm` 指定了模块在 `sysfs` 文件系统下对应文件的权限，该值可以是八进制的格式，比如 `0644`（所有者可以读写，组内可以读，其他人可以读）；或是 `S_Ifoo` 的定义形式，比如 `S_IRUGO | S_IWUSR`（任何人可读，`user` 可写）；如果该值是零，则表示禁止所有的 `sysfs` 项。

上面的宏其实并没有定义变量，你必须在使用该宏前进行变量定义。通常使用类似下面的语句完成定义：

```
/* 在模块参数控制下，我们允许在钓鱼竿上用活鱼饵 */
static int allow_live_bait = 1;          /* 默认功能允许 */
module_param(allow_live_bait, bool, 0644); /* 一个 Boolean 类型 */
```

这个值处于模块代码文件外部，换句话说，`allow_live_bait` 是个全局变量。

有可能模块的外部参数名称不同于它对应的内部变量名称，这时就该使用宏 `module_param_named()` 定义了：

```
module_param_named(name, variable, type, perm);
```

参数 `name` 是外部可见的参数名称，参数 `variable` 是参数对应的内部全局变量名称。比如：

```
static unsigned int max_test = DEFAULT_MAX_LINE_TEST;
module_param_named(maximum_line_test, max_test, int, 0);
```

通常，需要用 `charp` 类型来定义模块参数（一个字符串），内核将用户提供的这个字符串拷贝到内存，而且将变量指向该字符串。比如：

```
static char *name;
module_param(name, charp, 0);
```

如果需要，也可使内核直接拷贝字符串到指定的字符数组。宏 `module_param_string()` 可完成上述任务：

```
module_param_string(name, string, len, perm);
```

这里参数 `name` 为外部参数名称，参数 `string` 是对应的内部变量名称，参数 `len` 是 `string` 命名缓冲区的长度（或更小的长度，但是没什么太大的意义），参数 `perm` 是 `sysfs` 文件系统的访问权限（如果为零，则表示完全禁止 `sysfs` 项），比如：

```
static char species[BUF_LEN];
module_param_string(specifies, species, BUF_LEN, 0);
```

你可接受逗号分隔的参数序列，这些参数序列可通过宏 `module_param_array()` 存储在 C 数组中：

```
module_param_array(name, type, nump, perm);
```

参数 `name` 仍然是外部参数以及对应内部变量名，参数 `type` 是数据类型，参数 `perm` 是 `sysfs` 文件系统访问权限，这里新参数是 `nump`，它是一个整型指针，该整型存放数组项数。注意由参数 `name` 指定的数组必须是静态分配的，内核需要在编译时确定数组大小，从而保证不会造成溢

出。该函数用法相当简单，比如：

```
static int fish[MAX_FISH];
static int nr_fish;
module_param_array(fish, int, &nr_fish, 0444);
```

你可以将内部参数数组命名区别于外部参数，这时你需使用宏：

```
module_param_array_named(name, array, type, nump, perm);
```

其中参数和其他宏一致。

最后，你可使用 `MODULE_PARM_DESC()` 描述你的参数：

```
static unsigned short size = 1;
module_param(size, ushort, 0644);
MODULE_PARM_DESC(size, "The size in inches of the fishing pole.");
```

上述所有宏需要包含 `<linux/module.h>` 头文件。

17.2.8 导出符号表

模块被载入后，就会被动态地连接到内核。注意，它与用户空间中的动态链接库类似，只有当被显式导出后的外部函数，才可以被动态库调用。在内核中，导出内核函数需要使用特殊的指令：`EXPORT_SYMBOL()` 和 `EXPORT_SYMBOL_GPL()`。

导出的内核函数可以被模块调用，而未导出的函数模块则无法被调用。模块代码的链接和调用规则相比核心内核镜像中的代码而言，要更加严格。核心代码在内核中可以调用任意非静态接口，因为所有的核心源代码文件被链接成了同一个镜像。当然，被导出的符号表所含的函数必然也要是非静态的。

导出的内核符号表被看做导出的内核接口，甚至称为内核 API。导出符号相当简单，在声明函数后，紧跟上 `EXPORT_SYMBOL()` 指令就搞定了，比如：

```
/*
 * get_pirate_beard_color - 返回当前 pirate 胡须的颜色，
 * @pirate 是一个指向 pirate 结构体的指针；颜色定义在文件 <linux/beard_colors.h> 中
 */
int get_pirate_beard_color(struct pirate *p)
{
    return p->beard.color;
}
EXPORT_SYMBOL(get_pirate_beard_color);
```

假定 `get_pirate_beard_color()` 同时也定义在一个可访问的头文件中，那么现在任何模块都可以访问它。有一些开发者希望自己的接口仅仅对 GPL 兼容的模块可见，内核连接器使用 `MODULE_LICENSE()` 宏可满足这个要求。如果你希望先前的函数仅仅对标记为 GPL 协议的模块可见，那么你就需要用：

```
EXPORT_SYMBOL_GPL(get_pirate_beard_color);
```

如果你的代码被配置为模块，那么你就必须确保当它被编译为模块时，它所用的全部接口都

已被导出，否则就会产生连接错误（而且模块不能成功编译）。

17.3 设备模型

2.6 内核增加了一个引人注目的新特性——统一设备模型（device model）。设备模型提供了一个独立的机制专门来表示设备，并描述其在系统中的拓扑结构，从而使得系统具有以下优点：

- 代码重复最小化。
- 提供诸如引用计数这样的统一机制。
- 可以列举系统中所有的设备，观察它们的状态，并且查看它们连接的总线。
- 可以将系统中的全部设备结构以树的形式完整、有效地展现出来——包括所有的总线和内部连接。
- 可以将设备和其对应的驱动联系起来，反之亦然。
- 可以将设备按照类型加以归类，比如分类为输入设备，而无需理解物理设备的拓扑结构。
- 可以沿设备树的叶子向其根的方向依次遍历，以保证能以正确顺序关闭各设备的电源。

最后一点是实现设备模型的最初动机。若想在内核中实现智能的电源管理，就需要建立表示系统中设备拓扑关系的树结构。当在树上端的设备关闭电源时，内核必须首先关闭该设备节点以下的（处于叶子上的）设备电源。比如内核需要先关闭一个 USB 鼠标，然后才可关闭 USB 控制器；同样内核也必须在关闭 PCI 总线前先关闭 USB 控制器。简而言之，若要准确而又高效地完成上述电源管理目标，内核无疑需要一棵设备树。

17.3.1 kobject

设备模型的核心部分就是 kobject（kernel object），它由 struct kobject 结构体表示，定义于头文件 <linux/kobject.h> 中。kobject 类似于 C# 或 Java 这些面向对象语言中的对象（object）类，提供了诸如引用计数、名称和父指针等字段，可以创建对象的层次结构。

看下面的具体结构：

```
struct kobject {
    const char          *name;
    struct list_head    entry;
    struct kobject      *parent;
    struct kset         *kset;
    struct kobj_type    *ktype;
    struct sysfs_dirent *sd;
    struct kref         kref;
    unsigned int        state_initialized:1;
    unsigned int        state_in_sysfs:1;
    unsigned int        state_add_uevent_sent:1;
    unsigned int        state_remove_uevent_sent:1;
    unsigned int        uevent_suppress:1;
};
```

name 指针指向此 kobject 的名称。

parent 指针指向 kobject 的父对象。这样一来，kobject 就会在内核中构造一个对象层次结构，并且可以将多个对象间的关系表现出来。就如你所看到的，这便是 sysfs 的真正面目：一个用户空间的文件系统，用来表示内核中 kobject 对象的层次结构。

sd 指针指向 sysfs_dirent 结构体，该结构体在 sysfs 中表示的就是这个 kobject。从 sysfs 文件系统内部看，这个结构体是表示 kobject 的一个 inode 结构体。

kref 提供引用计数。ktype 和 kset 结构体对 kobject 对象进行描述和分类。在下面的内容中将详细介绍它们。

kobject 通常是嵌入其他结构中的，其单独意义其实并不大。相反，那些更为重要的结构体，比如定义于 <linux/cdev.h> 中的 struct cdev 中才真正需要用到 kobj 结构。

```
/* cdev structure - 该对象代表一个字符设备 */
struct cdev {
    struct kobject          kobj;
    struct module          *owner;
    const struct file_operations *ops;
    struct list_head       list;
    dev_t                  dev;
    unsigned int            count;
};
```

当 kobject 被嵌入到其他结构中时，该结构便拥有了 kobject 提供的标准功能。更重要的一点是，嵌入 kobject 的结构体可以成为对象层次架构中的一部分。比如 cdev 结构体就可通过其父指针 cdev->kobj.parent 和链表 cdev->kobj.entry 插入到对象层次结构中。

17.3.2 ktype

kobject 对象被关联到一种特殊的类型，即 ktype (kernel object type 的缩写)。ktype 由 kobj_type 结构体表示，定义于头文件 <linux/kobject.h> 中：

```
struct kobj_type {
    void (*release)(struct kobject *);
    const struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

ktype 的存在是为了描述一族 kobject 所具有的普遍特性。如此一来，不再需要每个 kobject 都分别定义自己的特性，而是将这些普遍的特性在 ktype 结构中一次定义，然后所有“同类”的 kobject 都能共享一样的特性。

release 指针指向在 kobject 引用计数减至零时要被调用的析构函数。该函数负责释放所有 kobject 使用的内存和其他相关清理工作。

sysfs_ops 变量指向 sysfs_ops 结构体。该结构体描述了 sysfs 文件读写时的特性。有关其细节参见 17.3.9 节。

最后，default_attrs 指向一个 attribute 结构体数组。这些结构体定义了该 kobject 相关的默认属性。属性描述了给定对象的特征，如果该 kobject 导出到 sysfs 中，那么这些属性都将相应地作为文件而导出。数组中的最后一项必须为 NULL。

17.3.3 kset

kset 是 kobject 对象的集合体。把它看成是一个容器，可将所有相关的 kobject 对象，比如“全部的块设备”置于同一位置。听起来 kset 与 ktype 非常类似，好像没有多少实质内容。那么“为什么会需要这两个类似的东西呢？”kset 可把 kobject 集中到一个集合中，而 ktype 描述相关类型 kobject 所共有的特性，它们之间的重要区别在于：具有相同 ktype 的 kobject 可以被分组到不同的 kset。就是说，在 Linux 内核中，只有少数一些的 ktype，却有多多个 kset。

kobject 的 kset 指针指向相应的 kset 集合。kset 集合由 kset 结构体表示，定义于头文件 `<linux/kobject.h>` 中：

```
struct kset {
    struct list_head    list;
    spinlock_t          list_lock;
    struct kobject       kobj;
    struct kset_uevent_ops *uevent_ops;
};
```

在这个结构中，其中 list 连接该集合（kset）中所有的 kobject 对象，list_lock 是保护这个链表中元素的自旋锁（关于自旋锁的讨论，详见第 10 章），kobj 指向的 kobject 对象代表了该集合的基类。uevent_ops 指向一个结构体——用于处理集合中 kobject 对象的热插拔操作。uevent 就是用户事件（user event）的缩写，提供了与用户空间热插拔信息进行通信的机制。

17.3.4 kobject、ktype 和 kset 的相互关系

上文反复讨论的这一组结构体很容易令人混淆，这可不是因为它们数量繁多（其实只有三个），也不是它们太复杂（它们都相当简单），而是由于它们内部相互交织。要了解 kobject，很难只讨论其中一个结构而不涉及其他相关结构。然而在这些结构的相互作用下，会更有助你深刻理解它们之间的关系。

这里最重要的家伙是 kobject，它由 struct kobject 表示。kobject 为我们引入了诸如引用计数（reference counting）、父子关系和对象名称等基本对象道具，并且是以一个统一的方式提供这些功能。不过 kobject 本身意义并不大，通常情况下它需要被嵌入到其他数据结构中，让那些包含它的结构具有了 kobject 的特性。

kobject 与一个特别的 ktype 对象关联，ktype 由 struct kobj_type 结构体表示，在 kobject 中 ktype 字段指向该对象。ktype 定义了一些 kobject 相关的默认特性：析构行为（反构造功能）、sysfs 行为（sysfs 的操作表）以及别的一些默认属性。

kobject 又归入了称作 kset 的集合，kset 集合由 struct kset 结构体表示。kset 提供了两个功能。第一，其中嵌入的 kobject 作为 kobject 组的基类。第二，kset 将相关的 kobject 集合在一起。在 sysfs 中，这些相关的 kobject 将以独立的目录出现在文件系统中。这些相关的目录，也许是给定目录的所有子目录，它们可能处于同一个 kset。

图 17-1 描述了这些数据结构的内在关系。

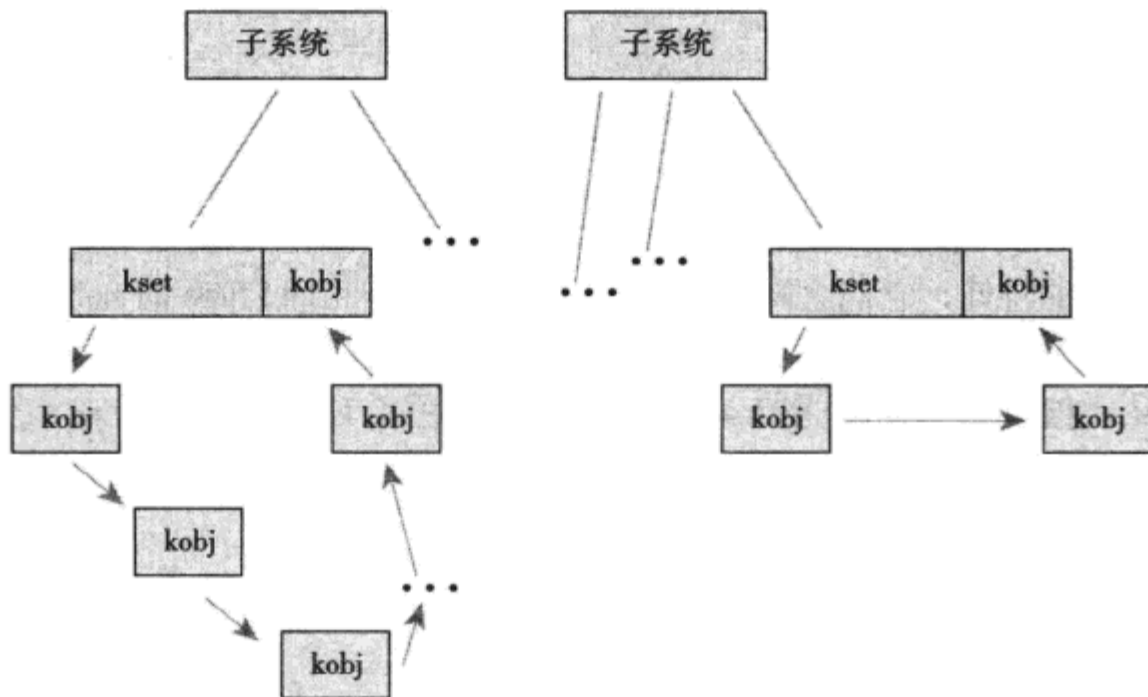


图 17-1 kobject、kset 和子系统的内在关系

17.3.5 管理和操作 kobject

当了解了 kobject 的内部基本细节后，我们来看管理和操作它的外部接口了。多数时候，驱动程序开发者并不必直接处理 kobject，因为 kobject 是被嵌入到一些特殊类型结构体中的（就如在字符设备结构体中看到的情形），而且会由相关的设备驱动程序在“幕后”管理。即便如此，kobject 并不是有意在隐藏自己，它可以出现在设备驱动代码中，或者可以在设备驱动子系统本身中使用它。

使用 kobject 的第一步需要先来声明和初始化。kobject 通过函数 kobject_init 进行初始化，该函数定义在文件 <linux/kobject.h> 中：

```
void kobject_init(struct kobject *kobj, struct kobj_type *ktype);
```

该函数的第一个参数就是需要初始化的 kobject 对象，在调用初始化函数前，kobject 必须清空。这个工作往往会在 kobject 所在的上层结构体初始化时完成。如果 kobject 未被清空，那么只需要调用 memset() 即可：

```
memset(kobj, 0, sizeof (*kobj));
```

在清零后，就可以安全的初始化 parent 和 kset 字段。例如，

```
struct kobject *kobj;

kobj = kmalloc(sizeof (*kobj), GFP_KERNEL);
if (!kobj)
    return -ENOMEM;
memset(kobj, 0, sizeof (*kobj));
kobj->kset = my_kset;
kobject_init(kobj, my_ktype);
```

这多步操作也可以由 kobject_create() 来自动处理，它返回一个新分配的 kobject：

```
struct kobject *kobject_create(void);
```

使用相当简单：

```
struct kobject *kobj;

kobj = kobject_create();
if (!kobj)
    return -ENOMEM;
```

大多数情况下，应该调用 `kobject_create()` 创建 `kobject`，或者是调用相关的辅助函数，而不是直接操作这个结构体。

17.3.6 引用计数

`kobject` 的主要功能之一就是为我们提供了一个统一的引用计数系统。初始化后，`kobject` 的引用计数设置为 1。只要引用计数不为零，那么该对象就会继续保留在内存中，也可以说是被“钉住”了。任何包含对象引用的代码首先要增加该对象的引用计数，当代码结束后则减少它的引用计数。增加引用计数称为获得（getting）对象的引用，减少引用计数称为释放（putting）对象的引用。当引用计数跌到零时，对象便可以撤销，同时相关内存也都被释放。

1. 递增和递减引用计数

增加一个引用计数可通过 `kobject_get()` 函数完成：

```
struct kobject * kobject_get(struct kobject *kobj);
```

该函数正常情况下将返回一个指向 `kobject` 的指针，如果失败则返回 `NULL` 指针；

减少引用计数通过 `kobject_put()` 完成，这个指令也声明在 `<linux/kobject.h>` 中：

```
void kobject_put(struct kobject *kobj);
```

如果对应的 `kobject` 的引用计数减少到零，则与该 `kobject` 关联的 `ktype` 中的析构函数将被调用。

2. kref

我们深入到引用计数系统的内部去看，会发现 `kobject` 的引用计数是通过 `kref` 结构体实现的，该结构体定义在头文件 `<linux/kref.h>` 中：

```
struct kref {
    atomic_t refcount;
};
```

其中唯一的字段是用来存放引用计数的原子变量。那为什么采用结构体？这是为了便于进行类型检测。在使用 `kref` 前，你必须先通过 `kref_init()` 函数来初始化它：

```
void kref_init(struct kref *kref)
{
    atomic_set(&kref->refcount, 1);
}
```

正如你所看到的，这个函数简单地将原子变量置 1，所以 `kref` 一旦被初始化，它表示的引用计数便固定为 1。这点和 `kobject` 中的计数行为一致。

要获得对 kref 的引用，需要调用 kref_get() 函数，这个函数声明在 <linux/kref.h> 中：

```
void kref_get(struct kref *kref)
{
    WARN_ON(!atomic_read(&kref->refcount));
    atomic_inc(&kref->refcount);
}
```

该函数增加引用计数值，它没有返回值。减少对 kref 的引用，调用声明在 <Linux/kref.h> 中的函数 kref_put()：

```
int kref_put(struct kref *kref, void (*release) (struct kref *kref))
{
    WARN_ON(release == NULL);
    WARN_ON(release == (void (*)(struct kref *))kfree);

    if (atomic_dec_and_test(&kref->refcount)) {
        release(kref);
        return 1;
    }
    return 0;
}
```

该函数将使得引用计数减 1，如果计数减少到零，则要调用作为参数提供的 release() 函数。注意 WARN_ON() 声明，提供的 release() 函数不能简单地采用 kfree()，它必须是一个仅接收一个 kref 结构体作为参数的特有函数，而且还没有返回值。kref_put() 函数返回 0，但有一种情况下它返回 1，那就是在对该对象的最后一个引用减 1 时。通常情况下，kref_put() 的调用者不关心这个返回值。

开发者现在不必在内核代码中利用 atomic_t 类型来实现自己的引用计数和简单的“get”、“put”这些封装函数。对开发者而言，在内核代码中最好的方法是利用 kref 类型和它相应的辅助函数，为自己提供一个通用的、正确的引用计数机制。

上述的所有函数定义与声明分别在文件 lib/kref.c 和文件 <linux/kref.h> 中。

17.4 sysfs

sysfs 文件系统是一个处于内存中的虚拟文件系统，它为我们提供了 kobject 对象层次结构的视图。帮助用户能以一个简单文件系统的方式来观察系统中各种设备的拓扑结构。借助属性对象，kobject 可以用导出文件的方式，将内核变量提供给用户读取或写入（可选）。

虽然设备模型的初衷是为了方便电源管理而提出的一种设备拓扑结构，但是 sysfs 是颇为意外的收获。为了方便调试，设备模型的开发者决定将设备结构树导出为一个文件系统。这个举措很快被证明是非常明智的，首先 sysfs 代替了先前处于 /proc 下的设备相关文件；另外它为系统对象提供了一个很有效的视图。实际上，sysfs 起初被称为 driverfs，它早于 kobject 出现。最终 sysfs 使得我们认识到一个全新的对象模型非常有利于系统，于是 kobject 应运而生。今天所有 2.6 内核的系统都拥有 sysfs 文件系统，而且几乎都毫无例外的将其挂载在 sys 目录下。

sysfs 的诀窍是把 kobject 对象与目录项 (directory entries) 紧密联系起来，这点是通过 kobject 对象中的 dentry 字段实现的。回忆第 12 章，dentry 结构体表示目录项，通过连接 kobject 到指定的目录项上，无疑方便地将 kobject 映射到该目录上。从此，把 kobject 导出形成文件系统

就变得如同在内存中构建目录项一样简单。好了，kobject 其实已经形成了一棵树——就是我们心爱的对象模型体系。由于 kobject 被映射到目录项，同时对象层次结构也已经在内存中形成了一棵树，因此 sysfs 的生成便水到渠成般地简单了。

```

-- block
|  -- loop0 -> ../devices/virtual/block/loop0
|  -- md0 -> ../devices/virtual/block/md0
|  -- nbd0 -> ../devices/virtual/block/nbd0
|  -- ram0 -> ../devices/virtual/block/ram0
|  -- xvda -> ../devices/vbd-51712/block/xvda
-- bus
|  -- platform
|  -- serio
-- class
|  -- bdi
|  -- block
|  -- input
|  -- mem
|  -- misc
|  -- net
|  -- ppp
|  -- rtc
|  -- tty
|  -- vc
|  -- vtconsole
-- dev
|  -- block
|  -- char
-- devices
|  -- console-0
|  -- platform
|  -- system
|  -- vbd-51712
|  -- vbd-51728
|  -- vif-0
|  -- virtual
-- firmware
-- fs
|  -- ecryptfs
|  -- ext4
|  -- fuse
|  -- gfs2
-- kernel
|  -- config
|  -- dlm
|  -- mm
|  -- notes
|  -- uevent_helper
|  -- uevent_seqnum
|  -- uids
-- module
|  -- ext4
|  -- i8042
|  -- kernel
|  -- keyboard
|  -- mousedev
|  -- nbd
|  -- printk
|  -- psmouse
|  -- sch_htb
|  -- tcp_cubic
|  -- vt
|  -- xt_recent

```

图 17-2 挂载于 /sys 目录下的 sysfs 文件系统的局部视图

sysfs 的根目录下包含了至少十个目录：block、bus、class、dev、devices、firmware、fs、kernel、module 和 power。block 目录下的每个子目录都对应着系统中的一个已注册的块设备。反过来，每个目录下又都包含了该块设备的所有分区。bus 目录提供了一个系统总线视图。class 目录包含了以高层功能逻辑组织起来的系统设备视图。dev 目录是已注册设备节点的视图。devices 目录是系统中设备拓扑结构视图，它直接映射出了内核中设备结构体的组织层次。firmware 目录包含了一些诸如 ACPI、EDD、EFI 等低层子系统的特殊树。fs 目录是已注册文件系统的视图。kernel 目录包含内核配置项和状态信息，module 目录则包含系统已加载模块的信息。power 目录包含系统范围的电源管理数据。并不是所有的系统都包含所有这些目录，还有些系统含有其他目录，但在这里尚未提到。

其中最重要的目录是 devices，该目录将设备模型导出到用户空间。目录结构就是系统中实际的设备拓扑。其他目录中的很多数据都是将 devices 目录下的数据加以转换加工而得。比如，/sys/class/net/ 目录是以注册网络接口这一高层概念来组织设备关系的，在这个目录中可能会有目录 eth0，它里面包含的 devices 文件其实就是一个指回到 devices 下实际设备目录的符号连接。

随便看看你可访问到的任何 Linux 系统的 sys 目录，这种系统设备视图相当准确和漂亮，而且可以看到 class 中的高层概念与 devices 中的低层物理设备，以及 bus 中的实际驱动程序之间互相联络是非常广泛的。当你认识到这种数据是开放的，换句话说，这是内核中维持系统的很好表示方式^①时，整个经历都弥足珍贵。

17.4.1 sysfs 中添加和删除 kobject

仅仅初始化 kobject 是不能自动将其导出到 sysfs 中的，想要把 kobject 导入 sysfs，你需要用到函数 kobject_add()：

```
int kobject_add(struct kobject *kobj, struct kobject *parent, const char *fmt, ...);
```

kobject 在 sysfs 中的位置取决于 kobject 在对象层次结构中的位置。如果 kobject 的父指针被设置，那么在 sysfs 中 kobject 将被映射为其父目录下的子目录；如果 parent 没有设置，那么 kobject 将被映射为 kset->kobj 中的子目录。如果给定的 kobject 中 parent 或 kset 字段都没有被设置，那么就认为 kobject 没有父对象，所以就会被映射成 sysfs 下的根级目录。这往往不是你所需要的，所以在调用 kobject_add() 前 parent 或 kset 字段应该进行适当的设置。不管怎么样，sysfs 中代表 kobject 的目录名字是由 fmt 指定的，它也接受 printf() 样式的格式化字符串。

辅助函数 kobject_create_and_add() 把 kobject_create() 和 kobject_add() 所做的工作放在一个函数中：

```
struct kobject *kobject_create_and_add(const char *name, struct kobject *parent);
```

注意 kobject_create_and_add() 函数接受直接的指针 name 作为 kobject 所对应的目录名称，而 kobject_add() 使用 printf() 风格的格式化字符串。

① 如果你对 sysfs 感兴趣，你可能也会对 HAL 感兴趣，它是一个硬件抽象层，可以在 <http://hal.freedesktop.org/wiki/software/hal> 找到它。HAL 基于 sysfs 中的数据建立起了一个内存数据库，将 class 概念、设备概念和驱动概念联系在一起。在这些数据之上，HAL 提供了丰富的 API 以使得应用程序更灵活。

从 sysfs 中删除一个 kobject 对应文件目录，需使用函数 `kobject_del()`：

```
void kobject_del(struct kobject *kobj);
```

上述这些函数都定义于文件 `lib/kobject.c` 中，声明于头文件 `<linux/kobject.h>` 中。

17.4.2 向 sysfs 中添加文件

我们已经看到 kobject 被映射为文件目录，而且所有的对象层次结构都优雅地、一个不少地映射成 sys 下的目录结构。但是里面的文件是什么？sysfs 仅仅是一个漂亮的树，但是没有提供实际数据的文件。

1. 默认属性

默认的文件集合是通过 kobject 和 kset 中的 `ktype` 字段提供的。因此所有具有相同类型的 kobject 在它们对应的 sysfs 目录下都拥有相同的默认文件集合。`kobj_type` 字段含有一个字段——`default_attrs`，它是一个 `attribute` 结构体数组。这些属性负责将内核数据映射成 sysfs 中的文件。

`attribute` 结构体定义在文件 `<linux/sysfs.h>` 中：

```
/* attribute 结构体 - 内核数据映射成 sysfs 中的文件 */
struct attribute {
    const char      *name;      /* 属性名称 */
    struct module    *owner;     /* 所属模块，如果存在 */
    mode_t           mode;       /* 权限 */
};
```

其中名称字段提供了该属性的名称，最终出现在 sysfs 中的文件名就是它。`owner` 字段在存在所属模块的情况下指向其所属的 `module` 结构体。如果一个模块没有该属性，那么该字段为 `NULL`。`mode` 字段类型为 `mode_t`，它表示了 sysfs 中该文件的权限。对于只读属性而言，如果是所有人都可读它，那么该字段被设为 `S_IRUGO`；如果只限于所有者可读，则该字段被设置为 `S_IRUSR`。同样对于可写属性，可能会设置该字段为 `S_IRUGO | S_IWUSR`。sysfs 中的所有文件和目录的 `uid` 与 `gid` 标志均为零。

虽然 `default_attrs` 列出了默认的属性，`sysfs_ops` 字段则描述了如何使用它们。`sysfs_ops` 字段指向了一个定义于文件 `<linux/sysfs.h>` 的同名的结构体：

```
struct sysfs_ops {
    /* 在读 sysfs 文件时该方法被调用 */
    ssize_t (*show) (struct kobject *kobj,
                    struct attribute *attr,
                    char *buffer);

    /* 在写 sysfs 文件时该方法被调用 */
    ssize_t (*store) (struct kobject *kobj,
                    struct attribute *attr,
                    const char *buffer,
                    size_t size);
};
```

当从用户空间读取 sysfs 的项时调用 show() 方法。它会拷贝由 attr 提供的属性值到 buffer 指定的缓冲区中，缓冲区大小为 PAGE_SIZE 字节；在 x86 体系中，PAGE_SIZE 为 4096 字节。该函数如果执行成功，则将返回实际写入 buffer 的字节数；如果失败，则返回负的错误码。

store() 方法在写操作时调用，它会从 buffer 中读取 size 大小的字节，并将其存放入 attr 表示的属性结构体变量中。缓冲区的大小总是为 PAGE_SIZE 或更小些。该函数如果执行成功，则将返回实际从 buffer 中读取的字节数；如果失败，则返回负数的错误码。

由于这组函数必须对所有的属性都进行文件 I/O 请求处理，所以它们通常需要维护某些通用映射来调用每个属性所特有的处理函数。

2. 创建新属性

通常来讲，由 kobject 相关 ktype 所提供的默认属性是充足的，事实上，因为所有具有相同 ktype 的 kobject，在本质上区别不大的情况下，都应是相互接近的。也就是说，比如对于所有的分区而言，它们完全可以具有同样的属性集合。这不但可以让事情简单，有助于代码合并，还使类似对象在 sysfs 目录中外观一致。

但是，有时在一些特别情况下会碰到特殊的 kobject 实例。它希望（甚至是必须）有自己的属性——也许是通用属性没包含那些需要的数据或者函数。为此，内核为能在默认集合之上，再添加新属性而提供了 sysfs_create_file() 接口：

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
```

这个接口通过 attr 参数指向相应的 attribute 结构体，而参数 kobj 则指定了属性所在的 kobject 对象。在该函数被调用前，给定的属性将被赋值，如果成功，该函数返回零，否则返回负的错误码。

注意，kobject 中 ktype 所对应的 sysfs_ops 操作将负责处理新属性。现有的 show() 和 store() 方法必须能够处理新属性。

除了添加文件外，还有可能需要创建符号连接。在 sysfs 中创建一个符号连接相当简单：

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char *name);
```

该函数创建的符号连接名由 name 指定，连接则由 kobj 对应的目录映射到 target 指定的目录。如果成功该函数返回零，如果失败返回负的错误码。

3. 删除新属性

删除一个属性需通过函数 sysfs_remove_file() 完成：

```
void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr);
```

一旦调用返回，给定的属性将不再存在于给定的 kobject 目录中。另外由 sysfs_create_link() 创建的符号连接可通过函数 sysfs_remove_link() 删除：

```
void sysfs_remove_link(struct kobject *kobj, char *name);
```

调用一旦返回，在 kobj 对应目录中的名为 name 的符号连接将不复存在。

上述的四个函数在文件 <linux/kobject.h> 中声明；sysfs_create_file() 和 sysfs_remove_file()

函数定义于文件 `fs/sysfs/file.c` 中；`sysfs_create_link()` 和 `sysfs_remove_link()` 函数定义于文件 `fs/sysfs/symlink.c` 中。

4. sysfs 约定

当前 `sysfs` 文件系统代替了以前需要由 `ioctl()`（作用于设备节点）和 `procfs` 文件系统完成的功能。目前，在合适目录下实现 `sysfs` 属性这样的功能的确别具一格。比如利用在设备映射的 `sysfs` 目录中添加一个 `sysfs` 属性，代替在设备节点上实现一新的 `ioctl()`。采用这种方法避免了在调用 `ioctl()` 时使用类型不正确的参数和弄乱 `/proc` 目录结构。

但是为了保持 `sysfs` 干净和直观，开发者必须遵从以下约定。

首先，`sysfs` 属性应该保证每个文件只导出一个值，该值应该是文本形式而且映射为简单 C 类型。其目的是为了避免数据的过度结构化或太凌乱，现在 `/proc` 中就混乱而不具有可读性。每个文件提供一个值，这使得从命令行读写变得简洁，同时也使 C 语言程序轻易地将内核数据从 `sysfs` 导入到自身的变量中去。但有些时候，一值一文件的规则不能很有效地表示数据，那么可以将同一类型的多个值放入一个文件中。不过这时需要合理地表述它们，比如利用一个空格也许就可使其意义清晰明了。总的来讲，应考虑 `sysfs` 属性要映射到独立的内核变量（正如通常所做），而且要记住应保证从用户空间操作简单，尤其是从 `shell` 操作简单。

其次，在 `sysfs` 中要以一个清晰的层次组织数据。父子关系要正确才能将 `kobject` 层次结构直观地映射到 `sysfs` 树中。另外，`kobject` 相关属性同样需要正确，并且要记住 `kobject` 层次结构不仅仅存在于内核，而且也要作为一个树导出到用户空间，所以要保证 `sysfs` 树健全无误。

最后，记住 `sysfs` 提供内核到用户空间的服务，这多少有些用户空间的 ABI（应用程序二进制接口）的作用。用户程序可以检测和获得其存在性、位置、取值以及 `sysfs` 目录和文件的行为。任何情况下都不应改变现有的文件，另外更改给定属性，但保留其名称和位置不变无疑是在自找麻烦。

这些简单的约定保证 `sysfs` 可为用户空间提供丰富和直观的接口。正确使用 `sysfs`，其他应用程序的开发者绝不会对你的代码抱有微辞，相反会赞美它。

17.4.3 内核事件层

内核事件层实现了内核到用户的消息通知系统——就是建立在上文一直讨论的 `kobject` 基础之上。在 2.6.0 版本以后，显而易见，系统确实需要一种机制来帮助将事件传出内核输送到用户空间，特别是对桌面系统而言，因为它需要更完整和异步的系统。为此就要让内核将其事件压到堆栈：硬盘满了！处理器过热了！分区挂载了！

早期的事件层没有采用 `kobject` 和 `sysfs`，它们如过眼烟云，没有存在多久。现在的事件层借助 `kobject` 和 `sysfs` 实现已证明相当理想。内核事件层把事件模拟为信号——从明确的 `kobject` 对象发出，所以每个事件源都是一个 `sysfs` 路径。如果请求的事件与你的第一个硬盘相关，那么 `/sys/block/hda` 便是源树。实质上，在内核中我们认为事件都是从幕后的 `kobject` 对象产生的。

每个事件都被赋予了一个动词或动作字符串表示信号。该字符串会以“被修改过”或“未挂载”等词语来描述事件。

最后，每个事件都有一个可选的负载（payload）。相比传递任意一个表示负载的字符串到用户空间而言，内核事件层使用 sysfs 属性代表负载。

从内部实现来讲，内核事件由内核空间传递到用户空间需要经过 netlink。netlink 是一个用于传送网络信息的多点传送套接字。使用 netlink 意味着从用户空间获取内核事件就如同在套接字上堵塞一样易如反掌。方法就是用户空间实现一个系统后台服务用于监听套接字，处理任何读到的信息，并将事件传送到系统栈里。对于这种用户后台服务来说，一个潜在的目的就是将事件融入 D-BUS 系统^①。D-BUS 系统已经实现了一套系统范围的消息总线，这种总线可帮助内核如同系统中其他组件一样地发出信号。

在内核代码中向用户空间发送信号使用函数 kobject_uevent()：

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action);
```

第一个参数指定发送该信号的 kobject 对象。实际的内核事件将包含该 kobject 映射到 sysfs 的路径。

第二个参数指定了描述该信号的“动作”或“动词”。实际的内核事件将包含一个映射成枚举类型 kobject_action 的字符串。该函数不是直接提供一个字符串，而是利用一个枚举变量来提高可重用性和保证类型安全，而且也消除了打字错误或其他错误。该枚举变量定义于文件 <linux/kobject_uevent.c> 中，其形式为 kOBJ_foo。当前值包含 kOBJ_MOUNT、kOBJ_UNMOUNT、kOBJ_ADD、kOBJ_REMOVE 和 kOBJ_CHANGE 等，这些值分别映射为字符串“mount”、“unmount”、“add”、“remove”和“change”等。当这些现有的值不够用时，允许添加新动作。

使用 kobject 和属性不但有利于很好的实现基于 sysfs 的事件，同时也有利于创建新 kobjects 对象和属性来表示新对象和数据——它们尚未出现在 sysfs 中。

这两个函数分别定义和声明于文件 lib/kobject_uevent.c 与文件 <linux/kobject.h> 中。

17.5 小结

本章中，我们考察的内核功能涉及设备驱动的实现和设备树的管理，包括模块、kobject（以及相关的 kset 和 ktype）和 sysfs。这些功能对于设备驱动程序的开发来说是非常重要的，因为这能够让他们写出更为模块化、更为高级的驱动程序。

这章讨论了内核中我们要学习的最后一个子系统，从下面开始要介绍一些普遍的但却重要的主题，这些主题是任何一个内核开发者都需要了解的，首先要讲的就是调试！

^① 想了解 D-BUS 更多的信息，参见 <http://dbus.freedesktop.org/>。

第 18 章

调 试

调试工作艰难是内核级开发区别于用户级开发的一个显著特点。相比于用户级开发，内核调试的难度确实要艰苦得多。更可怕的是，它带来的风险比用户级别更高，内核的一个错误往往立刻就能让系统崩溃。

驾驭内核调试的能力（当然，最终是为了能够成功地开发内核）很大程度上取决于经验和对整个操作系统的把握。没错，玉树临风可能会对别的事情有帮助，但是调试内核的关键还是在于你对内核的深刻理解。然而我们必须找到可以开始着手的地方，所以，在这一章里我们从调试内核的一种可能步骤开始。

18.1 准备开始

内核调试往往是一个令人挠头不已的漫长过程。不少 bug 已经让整个开发社区几个月都食不甘味了。幸运的是，在这些费劲的问题中也有不少比较简单，而且容易消灭的小 bug。运气好时，你可能面对的是些简单的小 bug。开始做一些调查之前，不会清楚到底面对的是什么。现在，需要的只是：

- 一个 bug。听起来很可笑，但确实需要一个确定的 bug。如果错误总是能够重现的话，那对我们会有很大的帮助（有一部分错误确实如此）。然而不幸的是，大部分 bug 通常都不是行为可靠而且定义明确的。
- 一个藏匿 bug 的内核版本。如果你知道这个 bug 最早出现在哪个内核版本中那就再理想不过了。如果你还不知道的话，别着急，本章会教你一个快速找出这个 bug 首先出现在哪个内核版本中的方法。
- 相关内核代码的知识和运气。调试内核其实是一个棘手的问题。不过对周围的代码理解得越多，调试起来也就越轻松。

本章中的大多数方法都假定能够让 bug 重现。因此，想要成功地进行调试，就取决于是否能让这些错误重现。如果不能，消灭 bug 就只能通过抽象出问题，再从代码中搜索蛛丝马迹来进行。虽然有时也得这么做，但如果你能够让错误重现，成功的机会要大许多。

有一些 bug 存在而且有人没办法让它重现，这听起来可能感觉挺奇怪。在用户级的程序里，bug 常常表现得很直截了当。比如，执行 foo 就会让程序立即产生核心信息转储（dump core）。但是内核中的 bug 表现却不是那么清晰。内核、用户程序和硬件之间的交互常常会很微妙。一个竞争条件可能在几百万次的算法迭代中才露出一次狰狞的面孔。设计不佳的（甚至是包含错误的）代码在某些系统上可能还让人可以忍受，而在其他的一些系统中却表现得相当糟糕。在一些特定的配置、一些特定的机器上，通常都需要付出额外的努力来触发某个 bug，不然的话，根本

看不到它。在跟踪 bug 的时候，掌握的信息越多越好。许多时候，当可以精确地重现一个 bug 的时候，就已经成功了一大半了。

18.2 内核中的 bug

内核中的 bug 多种多样。它们的产生可以有无数的原因，同时它们的表象也变化多端。从明白无误的错误代码（比如，没有把正确的值存放在恰当的位置）到同步时发生的错误（比如，共享变量锁定不当），再到错误地管理硬件（比如，给错误的控制寄存器发送错误的指令）；从降低所有程序的运行性能到毁坏数据再到使得系统处于死锁状态，都可能是 bug 发作时的症状。

从隐藏在源代码中的错误到展现在目击者面前的 bug，往往是经历一系列连锁反应的事件才可能触发的。举个例子，一个被共享的结构体，如果它没有引用计数，那么它就有可能引发竞争条件。因为没有引用计数的话，一个进程可以在另外一个进程仍然需要使用该结构的时候就释放掉它。继而，第二个进程就有可能试图通过无效的指针去使用一个不存在的数据结构。这样做可能导致引用一个空指针，也可能导致读出一些垃圾数据，还可能并不产生什么恶果（如果该数据并没有被其他什么覆盖的话）。引用空指针会导致产生一个 oops，而垃圾数据可能会导致系统崩溃（这种情形比 oops 还坏）。用户报告了 oops 或系统的错误现象之后，开发者回过头来观察错误情形，发现在释放数据之后还会对它进行读写，存在着一个竞争条件，于是就会进行修正，给这个共享的结构加上适当的引用计数。

内核调试听起来很难，但事实上 Linux 内核与其他大型的软件项目也没有什么太大的不同。内核确实有一些独特的问题需要考虑，像定时限制和竞争条件等，它们都是允许多个线程在内核中同时运行产生的结果。

18.3 通过打印来调试

内核提供的打印函数 `printk()` 和 C 库提供的 `printf()` 函数功能几乎相同。实际上，在本书中我们都没有用到这两个函数的不同部分。从它实现的大部分意图来说，这个名字很不错，`printk()` 就是内核的格式化打印函数。但是，`printk()` 确实还有一些自身特殊的功能。

18.3.1 健壮性

健壮性是 `printk()` 函数最容易让人们接受的一个特质。任何时候，任何地方都能调用它，内核中的 `printk()` 比比皆是。可以在中断上下文和进程上下中被调用；可以在任何持有锁时被调用；可以在多处理器上同时被调用，而且调用者连锁都不必使用。

它是一个弹性极佳的函数。这一点相当重要，`printk()` 之所以这么有用，就在于它随时都能被调用。

`printk()` 函数的健壮躯壳下也难免会有漏洞。在系统启动过程中，终端还没有初始化之前，在某些地方不能使用它。不过说实在的，如果终端没有初始化，你又能输出到什么地方去呢？

这一般不是一个什么问题，除非你要调试的是启动过程最开始的那些步骤（比如说在负责执行硬件体系结构相关的初始化动作的 `setup_arch()` 函数中）。着手进行这样的调试挑战性很强——没有任何打印函数能用，确实让问题更加棘手。

不过还是有一些可以指望的（虽然不多）。核心硬件部分的黑客依靠此时能够工作的硬件设备（比如说一个串口）与外界通信。绝大部分人对此都不会感兴趣。解决的办法是提供一个 `printk()` 的变体函数——`early_printk()`，这个函数在启动过程的初期就具有在终端上打印的能力。它的功能与 `printk()` 完全相同，区别仅仅在于名字和能够更早地工作。不过，由于该函数在一些内核支持的硬件体系结构上无法实现，所以这种办法缺少可移植性。但是，如果所使用的硬件体系可以实现这个函数（大多数硬件体系都可以，包括 x86），它就是最好的指望。

除非在启动过程的初期就要在终端上输出，否则可以认为 `printk()` 在什么情况下都能工作。

18.3.2 日志等级

`printk()` 和 `printf()` 在使用上最主要的区别就是前者可以指定一个日志级别。内核根据这个级别来判断是否在终端上打印消息。内核把级别比某个特定值低的所有消息显示在终端上。

可以通过下面这种方式指定一个记录级别：

```
printk(KERN_WARNING "This is a warning!\n");
printk(KERN_DEBUG "This is a debug notice!\n");
printk("I did not specify a loglevel!\n");
```

`KERN_WARNING` 和 `KERN_DEBUG` 都是 `<linux/kernel.h>` 中的简单宏定义。它们扩展开是像 “<4>” 或 “<7>” 这样的字符串，加进 `printk()` 函数要打印的消息的开头。内核用这个指定的记录等级和当前终端的记录等级 `console_loglevel` 来决定是不是向终端上打印。表 18-1 列举了所有可供使用的记录等级。

表 18-1 可供使用的记录等级

记录等级	描 述
<code>KERN_EMERG</code>	一个紧急情况
<code>KERN_ALERT</code>	一个需要立即被注意到的错误
<code>KERN_CRIT</code>	一个临界情况
<code>KERN_ERR</code>	一个错误
<code>KERN_WARNING</code>	一个警告
<code>KERN_NOTICE</code>	一个普通的，不过也有可能需要注意的情况
<code>KERN_INFO</code>	一条非正式的消息
<code>KERN_DEBUG</code>	一条调试信息——一般是冗余信息

如果你没有特别指定一个记录等级，函数会选用默认的 `DEFAULT_MESSAGE_LOGLEVEL`，现在默认等级是 `KERN_WARNING`。由于这个默认值将来存在变化的可能性，所以还是应该给自己的消息指定一个记录等级。

内核将最重要的记录等级 `KERN_EMERG` 定为 “<0>”，将无关紧要的记录等级 “`KERN_DEBUG`” 定为 “<7>”。举例来说，当编译预处理完成之后，前例中的代码实际被编译成如下格式：

```
printk("<4> This is a warning!\n");
printk("<7> This is a debug notice!\n");
printk("<4> did not specify a loglevel!\n");
```

怎样给调用的 `printk()` 赋记录等级完全取决于自己。那些正式的、需要你保持的消息应该有合适的记录等级。但是那些当你试图解决一个问题时加得到处都是的调试信息（必须承认，我们都这么干而且也确实行得通），可以按照你的想法赋予记录等级。一种选择是保持终端的默认记录等级不变，给所有调试信息 `KERN_CRIT` 或更低的等级。相反，也可以给所有调试信息 `KERN_DEBUG` 等级，而调整终端的默认记录等级。两种方法各有利弊，自己拿主意吧。

18.3.3 记录缓冲区

内核消息都被保存在一个 `LOG_BUF_LEN` 大小的环形队列中。该缓冲区大小可以在编译时通过设置 `CONFIG_LOG_BUF_SHIFT` 进行调整。在单处理器的系统上其默认值是 16KB。换句话说，就是内核在同一时间只能保存 16KB 的内核消息。如果消息队列已经达到最大值，那么如果再有 `printk()` 调用时，新消息将覆盖队列中的老消息。这个记录缓冲区之所以称为环形，是因为它的读写都是按照环形队列方式进行操作。

使用环形队列有许多好处。由于同时读写环形缓冲区时，其同步问题很容易解决，所以即使在中断上下文中也可以方便地使用 `printk()`。此外，它使记录维护起来也更容易。如果有大量的消息同时产生，新消息只需覆盖掉旧消息即可。在某个问题引发大量消息的时候，记录只会覆盖掉它本身，而不会因为失控而消耗掉大量内存。而环形缓冲区的唯一缺点——可能会丢失消息，但是与简单性和健壮性的好处相比，这点代价是值得的。

18.3.4 syslogd 和 klogd

在标准的 Linux 系统上，用户空间的守护进程 `klogd` 从记录缓冲区中获取内核消息，再通过 `syslogd` 守护进程将它们保存在系统日志文件中。`klogd` 程序既可以从 `/proc/kmsg` 文件中，也可以通过 `syslog()` 系统调用读取这些消息。默认情况下，它选择读取 `/proc` 方式实现。不管是哪种方法，`klogd` 都会阻塞，直到有新的内核消息可供读出。在被唤醒之后，它会读取出新的内核消息并进行处理。默认情况下，它就是把消息传给 `syslogd` 守护进程。

`syslogd` 守护进程把它接收到的所有消息添加进一个文件中，该文件默认是 `/var/log/messages`。也可以通过 `/etc/syslog.conf` 配置文件重新指定。

在启动 `klogd` 的时候，可以通过指定 `-c` 标志来改变终端的记录等级。

18.3.5 从 printf() 到 printk() 的转换

当刚开始开发内核代码的时候，往往会把 `printk()` 输入成 `printf()`。这很正常，你无法抗拒多年来在用户级程序中使用 `printf()` 的习惯。幸而这种错误不会持续很长时间，反复出现的链接错误很快就会让你在心烦意乱中开始培养新的习惯。

在编写用户级程序的时候，你输入 `printf()` 的时候不小心输入了 `printk()`。恭喜你，成为一个真正的内核黑客的时刻终于到来了。

18.4 oops

`oops` 是内核告知用户有不幸发生的最常用的方式。由于内核是整个系统的管理者，所以它

不能采取像在用户空间出现运行错误时使用的那些简单手段，因为它很难自行修复，它也不能将自己杀死。内核只能发布 oops。这个过程包括向终端上输出错误消息，输出寄存器中保存的信息并输出可供跟踪的回溯线索。内核中出现的故障很难处理，所以内核往往要经历严峻的考验才能发送出 oops 和靠它自己完成的一些清理工作。通常，发送完 oops 之后，内核会处于一种不稳定状态。举例来说，oops 发生的时候内核可能正在处理非常重要的数据。它可能持有一把锁或正在和硬件设备交互。内核必须适当地从当前的上下文环境中退出并尝试恢复对系统的控制。多数时候，这种尝试都会失败。因为如果 oops 在中断上下文时发生，内核根本无法继续，它会陷入混乱。混乱的结果就是系统死机。如果 oops 在 idle 进程（pid 为 0）或 init 进程（pid 为 1）时发生，结果同样是系统陷入混乱，因为内核缺了这两个重要的进程根本就无法工作。不过，要是 oops 在其他进程运行时发生，内核就会杀死该进程并尝试着继续执行。

oops 的产生有很多可能原因，其中包括内存访问越界或者非法的指令等。作为一个内核开发者，你将会经常处理（毫无疑问，也将导致）oops。

紧接着的是一个 oops 的实例，它是在一台 PPC 机器上的 tulip 网卡的定时器处理函数运行时发生的：

```
Oops: Exception in kernel mode, sig: 4
Unable to handle kernel NULL pointer dereference at virtual address 00000001

NIP: C013A7F0 LR: C013A7F0 SP: C0685E00 REGS: c0905d10 TRAP: 0700
Not tainted
MSR: 00089037 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last syscall: 120
GPR00: C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08: 000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00000005 00000000 00001032 C3F7C000 00000032 FFFFFFFF C3F7C1C0
Call trace:
[c013ab30] tulip_timer+0x128/0x1c4
[c0020744] run_timer_softirq+0x10c/0x164
[c001b864] do_softirq+0x88/0x104
[c0007e80] timer_interrupt+0x284/0x298
[c00033c4] ret_from_except+0x0/0x34
[c0007b84] default_idle+0x20/0x60
[c0007bf8] cpu_idle+0x34/0x38
[c0003ae8] rest_init+0x24/0x34
```

使用 PC 的读者可能对这么多的寄存器感到惊奇（居然有 32 个之多）。你可能对 x86-32 系统更熟悉一些，在这种系统上，oops 会简单一点。但是，oops 中包含的重要信息对于所有体系结构都是完全相同的：寄存器上下文和回溯线索。

回溯线索显示了导致错误发生的函数调用链。这样我们就可以观察究竟发生了什么：机器处于空闲状态，正在执行 idle 循环，由 cpu_idle() 循环调用 default_idle()。此时定时器中断产生了，它引起了对定时器的处理。tulip_timer() 这个定时器处理函数被调用，而就是它引用了空指针。甚至可以通过偏移量（像 0x128/0x1c4 这些出现在函数左侧的数字）找出导致问题的语句。

寄存器上下文信息可能同样有用，尽管使用起来不那么方便。如果你有函数的汇编代码，这

些寄存器数据可以帮助你重建引发问题的现场。在寄存器中发现一个本不应该出现的数值可能会在黑暗中给你带来第一丝光明。在上面的例子中，我们可以查看是哪个寄存器包含了 NULL（一个所有位都为零的数值），进而找出是函数的哪个变量的值不正常。一般在这种情况下问题往往是竞争引起的，在本例中，是指定时器和这块网卡驱动的其他部分之间的竞争。调试一个竞争条件往往很有挑战性。

18.4.1 ksymoops

前面列举的 oops 可以说是一个经过解码的 oops，因为内存地址都已经转换成了它们对应的函数。下面是其未解码版本：

```
NIP: C013A7F0 LR: C013A7F0 SP: C0685E00 REGS: c0905d10 TRAP: 0700
Not tainted
MSR: 00089037 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last syscall: 120
GPR00: C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08: 000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00000005 00000000 00001032 C3F7C000 00000032 FFFFFFFF C3F7C1C0
Call trace: [c013ab30] [c0020744] [c001b864] [c0007e80] [c00061c4]
[c0007b84] [c0007bf8] [c0003ae8]
```

回溯线索中的地址需要转化成有意义的符号名称才方便使用。这需要调用 ksymoops 命令，并且还必须提供编译内核时产生的 System.map。如果使用的是模块，还需要一些模块信息。ksymoops 通常会自行解析这些信息，所以一般可以这样调用它：

```
ksymoops saved_oops.txt
```

然后该程序就会吐出解码版的 oops。如果 ksymoops 无法找到默认位置上的信息，或者想提供不同信息，该程序可以接受许多参数。ksymoops 的使用手册上提供了许多说明信息，使用之前最好先行查阅。ksymoops 一般会随 Linux 发行版本提供。

18.4.2 kallsyms

谢天谢地，现在已经无须使用 ksymoops 工具了，这是一个了不起的工作。因为尽管开发者使用它的时候一般很少出现问题，但是最终用户常常会错误地匹配 System.map 文件或错误地对 oops 进行解码。

开发版的 2.5 版内核引入了 kallsyms 特性，它可以通过定义 CONFIG_KALLSYMS 配置选项启用。该选项存放着内核镜像中相应函数地址的符号名称，所以内核可以打印解码好的跟踪线索。相应地，解码 oops 也不再需要 System.map 或者 ksymoops 工具了。但是，这样做会使内核变大一些，因为从函数的地址到符号名称的映射必须永久地驻留在内核所映射的内存地址上。然而，不管是在开发的过程中还是在部署的过程中，占用这些内存都是值得的。配置选项 CONFIG_KALLSYMS_ALL 表示不仅存放函数名称，还存放所有的

符号名称。但一般只有那些特殊的调试器才会有此需要。CONFIG_KALLSYMS_EXTRA_PASS 选项会引起内核构建过程中再次忽略内核的目标代码。这个选项只有在调试 kallsyms 本身时才会有用。

18.5 内核调试配置选项

在编译的时候，为了方便调试和测试内核代码，内核提供了许多配置选项。这些选项都在内核配置编辑器的内核开发（Kernel hacking）菜单项中，它们都依赖于 CONFIG_DEBUG_KERNEL。当开发内核的时候，作为一种练习，不妨打开所有这些选项。

有些选项确实有用，应该启用 slab layer debugging（slab 层调试选项）、high-memory debugging（高端内存调试选项）、I/O mapping debugging（I/O 映射调试选项）、spin-lock debugging（自旋锁调试选项）和 stack-overflow checking（栈溢出检查选项）。其中最有用的一个是 sleep-inside-spinlock checking（自旋锁内睡眠选项），这些选项确实能完成不少调试工作。

从 2.5 版开始，为了检查各类由原子操作引发的问题，内核提供了极佳的工具。回忆一下第 9 章，原子操作指那些能够不分隔执行的东西；在执行时不能中断否则就是完不成的代码。正在使用一个自旋锁或禁止抢占的代码进行的就原子操作。在进行此类操作的时候，代码不能睡眠——使用锁时睡眠是引发死锁的元凶。

托内核抢占的福，内核提供了一个原子操作计数器。它可以被配置成一旦在原子操作过程中进程进入睡眠或者做了一些可能引起睡眠的操作，就打印警告信息并提供追踪线索。所以，包括正使用锁的时候调用 schedule()，正使用锁的时候以阻塞方式请求分配内存和在引用单 CPU 数据时睡眠在内，各种潜在的 bug 都能够被探测到。这种调试方法捕获了大量 bug，它也受到了大家极力推荐使用。

下面这些选项可以最大限度地利用该特性：

```
CONFIG_PREEMPT=y
CONFIG_DEBUG_KERNEL=y
CONFIG_KALLSYMS=y
CONFIG_DEBUG_SPINLOCK_SLEEP=y
```

18.6 引发 bug 并打印信息

一些内核调用可以用来方便标记 bug，提供断言并输出信息。最常用的两个是 BUG() 和 BUG_ON()。当被调用时，它们会引发 oops，导致栈的回溯和错误信息的打印。这些声明会导致 oops 跟硬件的体系结构是相关的。大部分体系结构把 BUG() 和 BUG_ON() 定义成某种非法操作，这样自然会产生需要的 oops。可以把这些调用当做断言使用，想要断言某种情况不该发生：

```
if (bad_thing)
    BUG();
```

或者使用更好的形式：

```
BUG_ON(bad_thing);
```

多数内核开发者相信 `BUG_ON()` 比 `BUG()` 更清晰、更可读，而且 `BUG_ON()` 会将其声明作为一个语句放入 `unlikely()` 中。请注意，有些开发者在讨论是否能用一个编译选项将 `BUG_ON()` 声明在编译时剔除，以便能在嵌入内核中节约空间。这就意味着你可以放心地使用 `BUG_ON()`，而不用担心 `BUG_ON()` 内的声明可能带来的任何“不良反应”。`BUILD_BUG_ON()` 与 `BUG_ON()` 作用相同，仅在编译时调用。如果在编译阶段已提供的声明为真，那么编译将会因为一个错误而中止。

可以用 `panic()` 引发更严重的错误。调用 `panic()` 不但会打印错误消息，而且还会挂起整个系统。显然，只应该在最糟糕的情况下使用它：

```
if (terrible_thing)
    panic("terrible thing is %ld\n", terrible_thing);
```

有些时候，只是需要在终端上打印一下栈的回溯信息来帮助调试。这个时候，`dump_stack()` 就很有用了。它只在终端上打印寄存器上下文和函数的跟踪线索：

```
if (!debug_check) {
    printk(KERN_DEBUG "provide some information...\n");
    dump_stack();
}
```

18.7 神奇的系统请求键

神奇的系统请求键 (Magic SysRq key) 是另外一根救命稻草，该功能可以通过定义 `CONFIG_MAGIC_SYSRQ` 配置选项来启用。SysRq (系统请求) 键在大多数键盘上都是标准键。在 i386 和 PPC 上，它可以通过 ALT-PrintScreen 访问。当该功能被启用的时候，无论内核处于什么状态，都可以通过特殊的组合键跟内核进行通信。这种功能可以让你在面对一台奄奄一息的系统时能完成一些有用的工作。

除了配置选项以外，还要通过一个 `sysctl` 用来标记该特性的开或关。需要启用它时使用如下命令：

```
echo 1 > /proc/sys/kernel/sysrq
```

从终端上，你可以输入 `Sysrq-h` 获取一份可用的选项列表。SysRq-s 将“脏”缓冲区跟硬盘交换分区同步，SysRq-u 卸载所有的文件系统，SysRq-b 重启设备。在一行内发送这三个键的组合可以重新启动濒临死亡的系统，这比直接按下机器的 Reset 键要安全一些。

如果机器已经完全锁死了，它也可能不会再响应神奇系统请求键，或者无法完成给定的命令。不过如果运气稍好的话，这些选项或许可以保存数据或者进行调试。表 18-2 列举了所有支持的系统请求命令。

内核代码中的 `Documentation/sysrq.txt` 对此有更详细的说明。实际的实现在 `drivers/char/sysrq.c` 中。神奇系统请求键是调试和挽救垂危系统所必需的一种工具。由于该功能对终端上的任何用户都提供服务，所以在重要的机器上启用它需要三思而行。可是对于自己用于开发的机器，启用它确实帮助很大。

表 18-2 支持 SysRq 的命令

主 要 命 令	描 述
SysRq-b	重新启动机器
SysRq-e	向 init 以外的所有进程发送 SIGTERM 信号
SysRq-h	在控制台显示 SysRq
SysRq-i	向 init 以外的所有进程发送 SIGKILL 信号
SysRq-k	安全访问键：杀死这个控制台上的所有程序
SysRq-l	向包括 init 的所有进程发送 SIGKILL 信号
SysRq-m	把内存信息输出到控制台
SysRq-o	关闭机器
SysRq-p	把寄存器的信息输出到控制台
SysRq-r	关闭键盘原始模式
SysRq-s	把所有已安装文件系统都刷新到磁盘
SysRq-t	把任务信息输出到控制台
SysRq-u	卸载所有已加载文件系统

18.8 内核调试器的传奇

很多内核开发者一直以来都希望能拥有一个用于内核的调试器。不幸的是，Linus 不愿意在它的内核源代码树中加入一个调试器。他认为调试器会误导开发者，从而导致引入不良的修正。没有人能对他的逻辑提出异议——从真正理解代码出发，确实更能保证修正的正确性。然而，许多内核开发者们还是希望有一个官方发布的、用于内核的调试器。因为这个要求看起来不会马上被满足，所以许多补丁应运而生了，它们为标准内核附加上了内核调试的支持。虽然这都是一些不被官方认可的附加补丁，但它们确实功能完善，十分强大。在我们深入这些解决方案之前，先看看标准的 Linux 调试器 gdb 能够给我们一些什么帮助是一个不错的选择。

18.8.1 gdb

可以使用标准的 GNU 调试器对正在运行的内核进行查看。针对内核启动调试器的方法与针对进程的方法大致相同：

```
gdb vmlinux /proc/kcore
```

其中 vmlinux 文件是未经压缩的内核映像，不是压缩过的 zImage 或 bzImage，它存放在源代码树的根目录上。

/proc/kcore 作为一个参数选项，是作为 core 文件来用的，通过它能够访问到内核驻留的高端内存。只有超级用户才能读取此文件的数据。

可以使用 gdb 的所有命令来获取信息。举个例子，为了打印一个变量的值，你可以用下面的命令：

```
p global_variable
```

反汇编一个函数：

```
disassemble function
```

如果编译内核的时候使用了 `-g` 参数（在内核的 Makefile 文件的 CFLAGS 变量中加入 `-g`），gdb 还可以提供更多的信息。比如，你可以打印出结构体中存放的信息或是跟踪指针。当然，编译出的内核会大很多，所以不要把编译带调试信息的内核当做一种习惯。

接下来，就要说不幸的那一面了，gdb 还是有很多局限性的。它没有任何办法修改内核数据。它也不能单步执行内核代码，不能加断点。不能修改内核数据是个非常大的缺陷。尽管在必要时反汇编函数无疑是个非常有用的功能，但是能够修改数据的却更为有用。

18.8.2 kgdb

kgdb 是一个补丁，它可以让我们在远端主机上通过串口利用 gdb 的所有功能对内核进行调试。这需要两台计算机：第一台运行带有 kgdb 补丁的内核，第二台通过串行线（不通过 modem，直接连接两台机器的电缆）使用 gdb 对第一台进行调试。通过 kgdb、gdb 的所有功能都能使用：读取或修改变量值，设置断点，设置关注变量，单步执行等。某些版本的 gdb 甚至允许执行函数。

设置 kgdb 和连接串行线比较麻烦，但是一旦做完了，调试就变得很简单了。该补丁会在 Documentation/ 目录下安装很多说明文件，可以把它们挑出来研究一下。

不同体系结构、不同内核版本使用的 kgdb 由不同的人员维护，为了给需要调试的内核找到合适的补丁，还是在网上搜索一下比较好。

18.9 探测系统

如果对内核调试有丰富的经验的话，那么你会掌握一些诀窍来帮助你更进一步地探测系统从而找到想要的答案。内核调试很有挑战性，即使是一点小的暗示或者技巧都能给你很大的帮助。我们最好把它们联系起来。

18.9.1 用 UID 作为选择条件

如果你开发的是进程相关的部分，有些时候，你可以在提供替代物的同时不打破原有代码的可执行性。这在你重写重要系统调用的时候，或者在你希望进行调试时系统功能依旧健全的情况下非常有用。

举个例子，假设为了加入一个激动人心的新特性，你重写了 `fork()` 系统调用。除非第一次的尝试就完美无缺，否则系统调试就是一场噩梦。如果 `fork()` 系统调用不正常的话，压根就不用指望整个系统还能正常工作。当然，和任何时候一样，希望总是存在的。

一般情况下，只要保留原有的算法而把你的新算法加入到其他位置上，基本就能保证安全。可以利用把用户 id (UID) 作为选择条件来实现这种功能，通过这种选择条件，可以安排到底执行哪种算法：

```
if (current->uid != 7777) {  
    /* 老算法 ... */  
} else {  
    /* 新算法 ... */  
}
```

除了 UID 为 7777 以外，其他所有的用户都用的是老算法。可以创建一个 UID 为 7777 的用户，专门来测试新算法。对于要求很严格的进程相关部分的代码来说，这种方法使得测试变得容易了许多。

18.9.2 使用条件变量

如果代码与进程无关，或者希望有一个针对所有情况都能使用的机制来控制某个特性，可以使用条件变量。这比使用 UID 还来得简单，只需要创建一个全局变量作为一个条件选择开关。如果该变量为零，就使用一个分支上的代码。如果它不为零，就选择另外一个分支。可以通过某种接口提供对这个变量的操控，也可以直接通过调试器进行操控。

18.9.3 使用统计量

有些时候你需要掌握某个特定事件的发生规律。有些时候需要比较多个事件并从中得出规律。通过创建统计量并提供某种机制访问其统计结果，很容易就能满足这种需求。

举个例子，假设我们希望得到 foo 和 bar 的发生频率，那么在某个文件中，当然最好是在定义该事件的那个文件里，定义两个全局变量：

```
unsigned long foo_stat = 0;  
unsigned long bar_stat = 0;
```

每当事件发生的时候，就让相应的变量加 1。然后在觉得合适的地方输出它。比如，可以在 /proc 目录中创建一个文件，还可以新创建一个系统调用。最简单的办法当然还是通过调试器直接访问它们。

注意，这种实现并非是 SMP 安全的。理想的办法是通过原子操作进行实现。但是仅仅对于一个简单的每次加 1 的调试统计量，一般无须搞得这么麻烦。

18.9.4 重复频率限制

为了发现一个错误，开发者们往往在代码的某个部分加入很多错误检查语句（多数对应的都是一些打印语句）。在内核中，有些函数每秒都要被调用很多次。如果你在这样的函数中加入了 printk()，那么系统马上就会被显示调试信息这一个任务压得喘不过气来，很快就什么也干不成了。

有两种相关的技巧可以用来防止此类问题的发生。第一种是重复频率限制，如果某种事件发生的非常频繁，而又需要观察它的整体进展情况，就可以让这种技巧施展身手了。为了避免调试信息发生井喷，可以每隔几秒执行一次打印（或者是其他任何你想完成的操作）。举个例子：

```
static unsigned long prev_jiffy = jiffies;          /* 频率限制 */

if (time_after(jiffies, prev_jiffy + 2*HZ)) {
    prev_jiffy = jiffies;
    printk(KERN_ERR "blah blah blah\n");
}
```

此例中，调试信息最多两秒打印一次。这可以让你的终端不至于被汹涌而至的调试信息洪流充塞，也保证你的系统依旧能用。完全可以根据自己的需要，或低或高地调整这种重复频率。

如果只使用 `printk()`，可以用一个特殊的函数去限制 `printk()` 的调用频率：

```
if (error && printk_ratelimit())
    printk(KERN_DEBUG "error=%d\n", error);
```

如果频率限制生效，那么 `printk_ratelimit()` 返回 0；否则，返回非 0。默认情况下，此函数限制每 5 秒产生一条信息，但是在施加这一条件之前，可以让起始频率为 10 条信息。可以通过 `printk_ratelimit` 和 `printk_ratelimit_burst` `sysctl` 来调整这些参数。

另一种棘手的问题是你如何确认在特定情况下某段代码确实被执行了。与前面的例子不同，你想观察的不是一个实时通知。如果这种通知在被触发一次之后依旧不停地到来，那就比较麻烦了。下面这种技巧针对的就不再是如何限制重复频率了，它要实现的是发生次数限制。

```
static unsigned long limit = 0;

if (limit < 5) {
    limit++;
    printk(KERN_ERR "blah blah blah\n");
}
```

此例中，调试信息输出 5 次就封顶了。5 次之后，打印条件总是不能成立。

不管是上面提到的哪个示例，用到的变量都应该是静态的（`static`），并且应该限制在函数的局部范围以内，这样才能保证变量的值在经历多次函数调用后仍然能够保留下来。

这些例子的代码都不是 SMP 或抢占安全的，不过，只需要用原子操作改造一下就没问题了。不过，对于一个临时的调试检测来说，没必要搞得这么复杂。

18.10 用二分查找法找出引发罪恶的变更

知道 bug 是什么时候引入内核源代码的通常都是很有用的。如果你知道 2.6.33 版中出现了 bug，而能肯定 2.4.29 中没有，那么就能够很容易地对引发这个 bug 的代码变更进行定位。消灭 bug 变得唾手可得——要么取消这个变更，要么对其进行修正。

可是，很多时候并不知道到底是哪个内核版本引入了 bug。你知道当前版本里 bug 是确实存在的，不过，它好像就是存在于当前版本中。只需要花一点点力气，就能找出引发问题的代码变更了。元凶在手，消灭 bug 就指日可待了。

一开始，需要一个可靠的可复制的错误，最好是系统一启动就能查证的 bug。接下来，需要一个能确保没问题的内核（你应该能够找到）。举个例子，你知道几个月前的内核没有这种错误，那么就从那时使用的内核中选取一个。如果发现问题，说明那时就存在了，那就找更早的。找到

不含该 bug 的内核应该不会太难。

接下来需要一个肯定有问题的内核。为了简单起见，应该从已知最早出现该问题的内核开始。

现在，你就可以在问题内核和良好的内核之间使用二分法了。举个例子，假定确保没有问题的内核版本是 2.6.11，有问题的内核版本是 2.6.20。从二者的正中选取一个内核版本，比如说 2.6.15。检查 2.6.15 是否包含此 bug。如果 2.6.15 没有问题，那么就知错误是发生在此版本之后了。所以，再从 2.6.15 开始，在它和 2.6.20 正中选取下一个版本，比如说对 2.6.17 进行检查。如果 2.6.15 有问题，那么错误就可能发生在此版本之前了，那么就该选 2.6.13 作为下一个待查目标了。就这样重复筛选。

最终你肯定能把问题局限在两个相继发行的版本之间——一个包含错误而另外一个不包含。你就能够很容易地对引发这个 bug 的代码变更进行定位。

这种方式比依次对每个版本的内核进行核查要好得多。

18.11 使用 Git 进行二分搜索

Git 源码管理工具提供了一个有用的二分搜索机制。如果你使用 Git 来控制 Linux 源码树的副本，那么 Git 将自动运行二分搜索进程。此外，Git 会在修订版本中进行二分搜索，这样可以找到具体哪次提交的代码引发了 bug。很多 Git 相关的任务比较繁杂，但使用 Git 进行二分搜索并不那么的困难。一开始，你得告诉 Git 你要进行二分搜索：

```
$ git bisect start
```

然后再为 Git 提供一个出现问题的最早内核版本：

```
$ git bisect bad <revision>
```

如果当前的内核版本就是引发 bug 的罪魁祸首，那么就不必提供内核版本：

```
$ git bisect bad
```

然后，还得为 Git 提供一个最新的可正常运行的内核版本：

```
$ git bisect good v2.6.28
```

接下来，Git 将会利用二分搜索法在 Linux 源码树中，自动检测正常的内核版本和有 bug 的内核版本之间哪个版本有隐患。接着再编译、运行以及测试正被检测的版本。如果这个版本一切正常，可以运行下面的命令：

```
$ git bisect good
```

如果这个版本运行有异常——也就是说，如果证明这个给定的内核版本有 bug，可以运行：

```
$ git bisect bad
```

对于每一条命令，Git 将在每一个版本的基础上反复二分搜索源码树，并且返回所查的下一个内核版本。这个过程需要反复执行直到不能再进行二分搜索为止。Git 将最终打印出有问题的版本号。

这本来应该是一个漫长的过程，但是 Git 使得这一过程变得容易起来。如果你已经知道引发 bug 的源（比如，x86 机型的启动代码），你可以指定 git 仅仅在与错误相关的目录列表中去二分搜索提交的补丁。

```
$ git bisect start - arch/x86
```

18.12 当所有的努力都失败时：社区

或许你已经做完了所有你能想到的尝试。你在键盘上呕心沥血了几个小时——实际上，可能是无数日子，答案依旧没有眷顾你。此时，如果 bug 是在 Linux 内核的主流部分中，你可以在内核开发社区中寻求其他开发者的帮助。

你应该向内核邮件列表发送一份电子邮件，对 bug 进行完整而又简洁地描述，你的发现可能会对找到最终的答案起到帮助。毕竟，没人希望 bug 存在。

第 20 章将会重点推荐社区和它最重要的论坛——Linux 内核邮件列表（LKML）。

18.13 小结

本章讨论了内核的调试——调试过程其实是一种寻求实现与目标偏差的行为。我们考察了几种技术：从内核内置的调试架构到调试程序，从记录日志到用 git 二分法查找。因为调试 Linux 内核困难重重，非调试用户程序能比，因此，本章的资料对于试图在内核代码中牛刀小试的任何人都至关重要。

我们将在第 19 章涉及另外的话题：Linux 内核的可移植性。

不要止步！



第 19 章

可移植性

Linux 是一个可移植性非常好的操作系统，它广泛支持许多不同体系结构的计算机。可移植性是指代码从一种体系结构移植到另外一种不同的体系结构上的方便程度。我们都知道 Linux 是可移植的，因为它已经能够在各种不同的体系结构上运行了。但这种可移植性不是凭空得来的——需要在编写可移植代码时就为此付出努力并坚持不懈。现在，这种努力已经开始得到回报了，移植 Linux 到新的系统上就很容易（相对来说）完成。本章中我们将讨论如何编写可移植的代码——编写内核代码和驱动程序时，必须时刻牢记这个问题。

19.1 可移植操作系统

有些操作系统在设计时把可移植性作为头等大事之一，尽可能少地涉及与机器相关的代码。汇编代码用得少之又少，为了支持各种不同类别的体系结构，界面和功能在定义时都尽最大可能地具有普适性和抽象性。这么做最显著的回报就是需要支持新的体系结构时，所需完成的工作要相对容易许多。一些移植性非常高而本身又比较简单的操作系统在支持新的体系结构时，可能只需要为此体系结构编写几百行专门的代码就行了。问题在于，体系结构相关的一些特性往往无法被支持，也不能对特定的机器进行手动优化。选择这种设计，就是利用代码的性能优化能力换取代码的可移植性。Minix、NetBSD 和许多研究用的系统就是这种高度可移植操作系统的实例。

与之相反，还有一种操作系统完全不顾及可移植性，它们尽最大的可能追求代码的性能表现，尽可能多地使用汇编代码，压根就是只为在一种硬件体系结构使用。内核的特性都是围绕硬件提供的特性设计的。因此，将其移植到其他体系结构就等于再重新从头编写一个新的操作系统内核，而且即便进行移植，这种操作系统在其他体系结构上也会不适用。选择这种设计，就是用代码的可移植性换取代码的性能优化能力。这样的系统往往比移植性好的系统更难维护。当然，这种系统对性能的要求不见得比对可移植性系统更强，不过它们还是愿意牺牲可移植性，而不乐意让设计打折扣。DOS 和 Windows 95 便是这种设计方案的最好例证。

Linux 在可移植性这个方面走的是中间路线。差不多所有的接口和核心代码都是独立于硬件体系结构的 C 语言代码。但是，在对性能要求很严格的部分，内核的特性会根据不同的硬件体系进行调整。举例来说，需要快速执行的和底层的代码都与硬件相关并且是用汇编语言写成的。这种实现方式使 Linux 在保持可移植性的同时兼顾对性能的优化。当可移植性妨碍性能发挥的时候，往往性能会被优先考虑。除此之外，代码就一定要保证可移植性。

一般来说，暴露在外的内核接口往往是与硬件体系结构无关的。如果函数的任何部分需要针对特殊的体系结构（无论是出于优化的目的还是作为一种必需的选择）提供支持的时候，这些部

分都会被安置在独立的函数中，等待调用。每种被支持的体系结构都实现了一个与体系结构相关的函数，而且会链接到内核映像之中。

调度程序就是一个好例子。调度程序的主体程序存放在 `kernel/sched.c` 文件中，用 C 语言编写，与体系结构无关。可是，调度程序需要进行的一些工作，比如说切换处理器上下文和切换地址空间等，却不得不依靠相应的体系结构完成。于是，内核用 C 语言编写了函数 `context_switch()` 用于实现进程切换，而在它的内部，则会调用 `switch_to()` 和 `switch_mm()` 分别完成处理器上下文和地址空间的切换。

而对于 Linux 支持的每种体系结构，它们的 `switch_to()` 和 `switch_mm()` 实现都各不相同。所以，当 Linux 需要移植到新的体系结构上的时候，只需要重新编写和提供这样的函数就可以了。

与体系结构相关的代码都存放在 `arch/architecture/` 目录中，`architecture` 是 Linux 支持的体系结构的简称。比如说，Intel x86 体系结构对应的简称是 `x86`（这种体系结构既支持 `x86-32` 又支持 `x86-64`）。与这种体系结构相关的代码都存放在 `arch/x86` 目录下。2.6 系列内核支持的体系结构包括 `alpha`、`arm`、`avr32`、`blackfin`、`cris`、`frv`、`h8300`、`ia64`、`m32r`、`m68k`、`m68knommu`、`mips`、`mn10300`、`parisc`、`powerpc`、`s390`、`sh`、`sparc`、`um`、`x86` 和 `xtensa`。本章稍后给出的表 19-1 是一份更详尽的清单。

19.2 Linux 移植史

当 Linus 最初把 Linux 带到这个无法预测的大千世界的时候，它只能在 i386 上运行。尽管这个操作系统通用性很强，代码也写得不错，可是可移植性在那时算不上是一个关注焦点。实际上，Linus 还一度建议让 Linux 只在 i386 体系结构上驰骋。不过，人们还是在 1993 年开始把 Linux 向 Digital Alpha 体系结构上移植了。Digital Alpha 是一种高性能现代计算机体系结构，它支持 RISC 和 64 位寻址。这与 Linus 最初选的 i386 无疑是天壤之别。虽然如此，最初的这次移植工作最终还是花了将近一年时间，Alpha 机成为了 i386 后第一个被官方支持的体系结构。万事开头难，这次移植的挑战性是最大的，为了提高可移植性，内核中不少代码都被重写了^①。尽管这给整个移植带来了不小的工作量，可是效果是显著的，自此以后，移植变得简单轻松多了。

尽管第一个发行版只支持 Intel i386，但 1.2 版的内核就可以支持 Digital Alpha、Intel x86、MIPS 和 SPARC——虽然支持的不是很完善，而且带些试验性质。

在 2.0 版内核中，加入了对 Motorola 68K 和 PowerPC 的官方支持，而原 1.2 版支持的体系结构也纳入了官方支持的范畴，并且稳定下来。

2.2 版内核加入了对更多体系结构的支持，新增了对 ARMS、IBM S390 和 UltraSPARC 的支持。没过几年，2.4 版内核支持的体系结构就达到了 15 个，像 CRIS、IA_64、64 位 MIPS、HP PA_RISC、64 位 IBM S/390 和 Hitachi SH 都被加进来了。

^① 在内核开发中这很普遍。如果打算做一件事，那么就要把它做好。为了追求完美，内核开发者们是决不会介意重写大段代码的。

当前的 2.6 内核把体系结构的数目进一步提高到了 21 个，有不含 MMU 的 AVR、FR-V 和 Motorola 68k 以及 M32xxx、H8/300、IBM POWER、Xtensa，甚至还提供了用户模式 (Usermode) Linux (一个在 Linux 虚拟机上运行的内核版本)。

每一种体系结构本身就可以支持不同的芯片和机型。像被支持的 ARM 和 PowerPC 等体系结构，它们就可以支持很多不同的芯片和机型。其他的体系结构，比如说 x86 和 SPARC，它们可以支持 32 位和 64 位不同的处理器。所以说，尽管 Linux 移植到了 21 种基本体系结构上，但实际上可以运行它的机器的数目要大得多。

19.3 字长和数据类型

能够由机器一次完成处理的数据称为字。这和我们在文档中用字符 (8 位) 和页 (许多字，通常是 4KB 或 8KB) 来计量数据是相似的。字是指位的整数数目——比如说，1、2、4 或 8 等。但人们说某个机器是多少“位”的时候，他们其实说的就是该机器的字长。比如说，当人们说 Intel i7 是 64 位芯片时，他们的意思是奔腾的字长为 64 位，也就是 8 字节。

处理器通用寄存器 (general-purpose registers, GPR) 的大小和它的字长是相同的。一般来说，对于一个体系结构，它各个部件的宽度 (比如说内存总线) 最少要和它的字长一样大。虽然物理地址空间有时候会比字长小，但虚拟地址空间的大小也等于字长，至少 Linux 支持的体系结构中都是这样的^①。此外，C 语言定义的 long 类型总是对等于机器的字长，而 int 类型有时会比字长小。比如说，Alpha 是 64 位机器，所以它的寄存器、指针和 long 类型都是 64 位长度的，而 int 类型是 32 位的。Alpha 机每一次可以访问和操作一个 64 位长的数据。

字、双字以及混合

有些操作系统和处理器不把它们的标准字长称作字，相反，出于历史原因和某种主观的命名习惯，它们用字来代表一些固定长度的数据类型。比如说，一些系统根据长度把数据划分为字节 (byte, 8 位)、字 (word, 16 位)、双字 (double words, 32 位) 和四字 (quad words 64 位)，而实际上该机是 32 位的。在本书中 (在 Linux 中一般也是这样)，像我们前面所讨论的那样，一个字就代表处理器的字长。

对于支持的每一种体系结构，Linux 都要将 `<asm/types.h>` 中的 `BITS_PER_LONG` 定义为 C long 类型的长度，也就是系统的字长。表 19-1 是 Linux 支持的体系结构和它们的字长的对照表。

一般而言，Linux 对于一种体系结构都会分别实现 32 位和 64 位的不同版本。比如，在 2.6 内核的早期版本中，内核中就同时有 i386 和 x86-64，mips 和 mips64，以及 ppc 和 ppc64。但现在，经过大家的努力，这些体系结构均放在 `arch/` 目录下，每个代码库中既支持 32 位又支持 64 位。

^① 不过实际上可寻址的内存空间也可能会比字长小一些。比如，一个 64 位的体系结构虽然可能会提供 64 位的指针，但可能只会用 48 位来寻址。此外，如果支持 Intel 的 PAE，那么实际的物理内存也有比字长还大的可能。

表 19-1 Linux 支持的体系结构

体系结构	描 述	字 长
alpha	Digital Alpha	64 位
arm	ARM 和增强型 ARM	32 位
avr	AVR	32 位
blackfin	Blackfin	32 位
cris	CRIS	32 位
frv	FR-V	32 位
h8300	H8/300	32 位
ia64	IA-64	64 位
m32r	M32xxx	32 位
m68k	Motorola 68k	32 位
m68knommu	无 MMU 型 M68k	32 位
mips	MIPS	32 位和 64 位
parisc	HP PA-RISC	32 位和 64 位
powerpc	PowerPC	32 位和 64 位
s390	IBM S/390	32 位和 64 位
Sh	Hitachi SH	32 位
Sparc	SPARC	32 位和 64 位
Um	Usermode Linux	32 位和 64 位
x86	x86-32 和 x86-64	32 位和 64 位
xtensa	Xtensa	32 位

C 语言虽然规定了变量的最小长度，但是没有规定变量具体的标准长度，它们可以根据实现变化^①。C 语言的标准数据类型长度随体系结构变化这一特性不断引起争议。好的一面是标准数据类型可以充分利用不同体系结构变化的字长而无须明确定义长度。C 语言中 long 类型的长度就被确定为机器的字长。不好的一面是在编程时不能对标准的 C 数据类型进行大小的假定，没有什么能够保障 int 一定和 long 的长度是相同的^②。

情况其实还会更加复杂，因为用户空间使用的数据类型和内核空间的数据类型不一定要相互关联。sparc64 体系结构就提供了 32 位的用户空间，其中指针、int 和 long 的长度都是 32 位。而在内核空间，它的 int 长度是 32 位，指针和 long 的长度却是 64。没有什么标准来规范这些。

牢记下述准则：

- ANSI C 标准规定，一个 char 的长度一定是 1 字节。
- 尽管没有规定 int 类型的长度是 32 位，但在 Linux 当前所有支持的体系结构中，它都是 32 位的。

① 唯一的例外是 char，它的长度总是 8 位。

② 事实上对于 Linux 支持的 64 位体系结构来说，long 和 int 长度是不同的，int 是 32 位的，而 long 是 64 位的。但对于我们所熟悉的 32 位体系结构而言，两种数据类型都是 32 位的。

- short 类型也类似，在当前所有支持的体系结构中，虽然没有明文规定，但是它都是 16 位的。
- 绝不应该假定指针和 long 的长度，在 Linux 当前支持的体系结构中，它们可以在 32 位和 64 位中变化。
- 由于不同的体系结构 long 的长度不同，决不应该假设 `sizeof(int) = sizeof(long)`。
- 类似地，也不要假设指针和 int 长度相等。

操作系统常用一个简单的助记符来描述此系统中数据类型的大小。比如，64 位的 Windows 系统简称为 LLP64，它说明 long 和指针的长度都是 64 位。64 位的 Linux 系统可简记为 LP64，即 long 和指针都是 64 位。32 位的 Linux 系统简称为 ILP32，即 int、long 和指针的长度均为 32 位。这些助记符可以一目了然地显示出操作系统所提供的字长大小，因为这种方法涉及一种权衡问题。

现在依次来分析 ILP64、LP64 和 LLP64 这三种情况。ILP64 这种操作系统，int、long 和指针的大小都是 64 位。这样的数据长度使得编程变得更加容易，因为 C 语言中主要的数据类型大小是一样的（整型和指针大小的不匹配是编程中常出现的错误）。不过这样也会带来缺点，这种整型比我们平常所需的整型要大很多。在 LP64 操作系统中，程序员可以使用不同大小的整型，但必须注意整型的大小比指针类型要小。对于 LLP64 系统而言，程序员不仅要被迫接受 int 和 long 的大小相同，还要担心整型和指针之间的大小不匹配。大多数程序员都喜欢 LP64 型，即 Linux 所采用的操作系统模型。

19.3.1 不透明类型

不透明数据类型隐藏了它们的内部格式或结构。在 C 语言中，它们就像黑盒一样。支持它们的语言不是很多。作为替代，开发者们利用 typedef 声明一个类型，把它叫做不透明类型，希望其他人别去把它重新转化回对应的那个标准 C 类型。通常开发者们在定义一套特别的接口时才会用到它们。比如说用来保存进程标识符的 pid_t 类型。该类型的实际长度被隐藏起来了——尽管任何人都可以偷偷撩开它的面纱，发现它就是一个 int。如果所有代码都不显式地利用它的长度[⊖]，那么改变时就不会引起什么争议，这种改变确实可能会出现：在老版本的 Unix 系统中，pid_t 的定义是 short 类型。

另外一个不透明数据类型的例子是 atomic_t。在第 10 章中介绍过，它放置的是一个可以进行原子操作的整型值。尽管这种类型就是一个 int，但利用不透明类型可以帮助确保这些数据只在特殊的有关原子操作的函数中才会被使用。不透明类型还帮助我们隐藏了 atomic_t 类型的可用长度，但是该类型也并不总是完整的 32 位，比如在 32 位 SPARC 体系下长度就被限制。

内核还用到了其他一些不透明类型，包括 dev_t、gid_t 和 uid_t 等。

处理不透明类型时的原则是：

- 不要假设该类型的长度。这些类型在某些系统中可能是 32 位，而在其他系统中又可能是 64 位。并且，内核开发者可以任意修改这些类型的大小。
- 不要将该类型转化回其对应的 C 标准类型使用。

⊖ 显式利用长度这里指直接使用 int 类型的长度，比如说在编程时使用 `sizeof(int)` 而不是 `sizeof(pid_t)`。——译者注

- 成为一个大小不可知论者。编程时要保证在该类型实际存储空间和格式发生变化时代码不受影响。

19.3.2 指定数据类型

内核中还有一些数据虽然无须用不透明的类型表示，但它们定义成了指定的数据类型。在中断控制时用到的 flag 参数就是个例子，它应该存放在 unsigned long 类型中。

当存放和处理这些特别的数据时，一定要搞清楚它们对应的类型后再使用。把它们存放在其他（如 unsigned int 等）类型中是一种常见错误。在 32 位机上这没什么问题，可是 64 位机上就会捅娄子了。

19.3.3 长度明确的类型

作为一个程序员，你往往需要在程序中使用长度明确的数据。像操作硬件设备、进行网络通信和操作二进制文件时，通常都必须满足它们明确的内部要求。比如说，一块声卡可能用的是 32 位寄存器，一个网络包有一个 16 位字段，一个可执行文件有 8 位的 cookie。在这些情况下，数据对应的类型应该长度明确。

内核在 <asm/types.h> 中定义了这些长度明确的类型，而该文件又被包含在文件 <linux/types.h> 中。表 19-2 有完整的清单。

表 19-2 长度明确的数据类型

类 型	描 述
s8	带符号字节
u8	无符号字节
s16	带符号 16 位整数
u16	无符号 16 位整数
s32	带符号 32 位整数
u32	无符号 32 位整数
s64	带符号 64 位整数
u64	无符号 64 位整数

其中带符号的变量用得比较少。

这些长度明确的类型大部分都是通过 typedef 对标准的 C 类型进行映射得到的。在一个 64 位机上，它们看起来像：

```
typedef signed char s8;
typedef unsigned char u8;
typedef signed short s16;
typedef unsigned short u16;
typedef signed int s32;
typedef unsigned int u32;
typedef signed long s64;
typedef unsigned long u64;
```

而在 32 位机上，它们可能定义成：

```
typedef signed char s8;  
typedef unsigned char u8;  
typedef signed short s16;  
typedef unsigned short u16;  
typedef signed int s32;  
typedef unsigned int u32;  
typedef signed long long s64;  
typedef unsigned long long u64;
```

上述的这些类型只能在内核内使用，不可以在用户空间出现（比如，在头文件中的某个用户可见结构中出现）。这个限制是为了保护命名空间。不过内核对应这些不可见变量同时也定义了对应的用户可见的变量类型，这些类型与上面类型所不同的是增加了两个下划线前缀。比如，无符号 32 位整型对应的用户空间可见类型就是 `__u32`。该类型除了名字有区别外，其他方面与 `u32` 相同。在内核中你可以任意使用这两个名字，但是如果是用户可见的类型，那必须使用下划线前缀的版本名，防止污染用户空间的命名空间。

19.3.4 char 型的符号问题

C 标准表示 `char` 类型可以带符号也可以不带符号，由具体的编译器、处理器或由它们两者共同决定到底 `char` 是带符号还是不带符号。

大部分体系结构上，`char` 默认是带符号的，它可以自 -128 到 127 之间取值。也有一些例外，比如 ARM 体系结构上，`char` 就是不带符号的，它的取值范围是 0 ~ 255。

举例来说，在默认 `char` 不带符号的情况下，下面的代码实际会把 255 而不是把 -1 赋予 `i`：

```
char i = -1;
```

而另一种机器上，默认 `char` 带符号，就会确切地把 -1 赋予 `i`。如果程序员本意是把 -1 保存在 `i` 中，那么前面的代码就该修改成：

```
signed char i = -1;
```

另外，如果程序员确实希望存储 255，那么代码应该如下：

```
unsigned char = 255;
```

如果在自己的代码中使用了 `char` 类型，那么要保证在带符号和不带符号的情况下代码都没问题。如果能明确要用的是哪一个，就直接声明它。

19.4 数据对齐

对齐是跟数据块在内存中的位置相关的话题。如果一个变量的内存地址正好是它长度的整数倍，它就称作是自然对齐的。举例来说，对于一个 32 位类型的数据，如果它在内存中的地址刚好可以被 4 整除（也就最低两位为 0），那它就是自然对齐的。也就是说，一个大小为 2^n 字节的数据类型，它地址的最低有效位的后 n 位都应该为 0。

一些体系结构对对齐的要求非常严格。通常像 RISC 的系统，载入未对齐的数据会导致处理

器陷入（一种可处理的错误）。还有一些系统可以访问没有对齐的数据，只不过性能会下降。编写可移植性高的代码要避免对齐问题，保证所有的类型都能够自然对齐。

19.4.1 避免对齐引发的问题

编译器通常会通过让所有的数据自然对齐来避免引发对齐问题。实际上，内核开发者在对齐上不用花费太大心思——只有搞 gcc 的那些老兄才应该为此犯愁呢。可是，当程序员使用指针太多，对数据的访问方式超出编译器的预期时，就会引发问题了。

一个数据类型长度较小，它本来是对齐的，如果你用一个指针进行类型转换，并且转换后的类型长度较大，那么通过改指针进行数据访问时就会引发对齐问题（无论如何，某些体系结构会存在这种问题）。也就是说，下面的代码是错误的：

```
char wolf[]="Like a wolf";
char *p = &wolf[1];
unsigned long l = *(unsigned long *)p;
```

这个例子将一个指向 char 型的指针当做指向 unsigned long 型的指针来用，这会引起问题，因为此时会试图从一个并不能被 4 或 8 整除的内存地址上载入 32 位或 64 位的 unsigned long 型数据。

这种复杂的访问可能看起来有些模糊，不过通常就是如此。无论如何，这种错误出现了，所以应该小心。实际编程时错误可能不会像一些例子中那么明显或复杂。

19.4.2 非标准类型的对齐

前面提到了，对于标准数据类型来说，它的地址只要是其长度的整数倍就对齐了。而非标准的（复合的）C 数据类型按照下列原则对齐：

- 对于数组，只要按照基本数据类型进行对齐就可以了，随后的所有元素自然能够对齐。
- 对于联合体，只要它包含的长度最大的数据类型能够对齐就可以了。
- 对于结构体，只要结构体中每个元素能够正确地对齐就可以了。

结构体还要引入填补机制，这会引出下一个问题。

19.4.3 结构体填补

为了保证结构体中每一个成员都能够自然对齐，结构体要被填补。这点确保了当处理器访问结构中一个给定元素时，元素本身是对齐的。举个例子，下面是一个在 32 位机上的结构体：

```
struct animal_struct {
    char dog;                /* 1 字节 */
    unsigned long cat;       /* 4 字节 */
    unsigned short pig;      /* 2 字节 */
    char fox;                /* 1 字节 */
};
```

由于该结构不能准确地满足各个成员自然对齐，所以它在内存中可不是按照原样存放的。编

译器会在内存中创建一个类似下面给出的结构体：

```
struct animal _struct {
    char dog;                /* 1 字节 */
    u8 __pad0[3];            /* 3 字节 */
    unsigned long cat;       /* 4 字节 */
    unsigned short pig;      /* 2 字节 */
    char fox;                /* 1 字节 */
    u8 __pad1;               /* 1 字节 */
};
```

填补的变量都是为了让数据自然对齐而加入的。第一个填充物占用了 3 个字节的空間，保证 cat 可以按照 4 字节对齐。这也自动使其他小的对象都对齐了，因为它们长度都比 cat 要小。第二个（也是最后的）填充是为了填补 struct 本身的大小。额外的这个填补使结构体的长度能够被 4 整除，这样，在由该结构体构成的数组中，每个数组项也就会自然对齐了。

注意，在大部分 32 位系统上，对于任何一个这样的结构体，sizeof(animal_struct) 都会返回 12。C 编译器自动进行填补以保证自然对齐。

通常你可以通过重新排列结构体中的对象来避免填充。这样既可以得到一个较小的结构体，又能保证无须填补它也是自然对齐的。

```
struct animal _struct {
    unsigned long cat;       /* 4 字节 */
    unsigned short pig;      /* 2 字节 */
    char dog;               /* 1 字节 */
    char fox;               /* 1 字节 */
};
```

现在这个结构体只有 8 字节大小了。不过，不是任何时候都可以这样对结构体进行调整的。举个例子，如果该结构体是某个标准的一部分，或者它是现有代码的一部分，那么它的成员次序就已经被定死了，虽然内核（缺少一个正式的 ABI）相比用户空间来说，这种需求要少得多。还有些时候，因为一些原因必须使用某种固定的次序——比如说，为了提高高速缓存的命中率进行优化时设定的变量次序。注意，ANSI C 明确规定不允许编译器改变结构体内成员对象的次序^①——它总是由程序员来决定的。虽然编译器可以帮助你做填充，但是，如果使用 -Wpadded flag 标志，那么将使 gcc 在发现结构体被填充时产生警告。

内核开发者需要注意结构体填补问题，特别是在整体使用时——这是指当需要通过网络发送它们或需要将它们写入文件的时候，因为不同体系结构之间所需要的填补也不尽相同。这也是为什么 C 语言没有提供一个内建的结构体比较操作符的原因之一。结构体内的填充字节中可能会包含垃圾信息，所以在结构体之间进行一字节一字节的比较就不大可能实现了。C 语言的设计者（正确的）感觉到最好还是由程序员自己为不同的情况编写比较函数，这样才能利用到结构体次序信息。

① 如果让编译器随心所欲地改变结构体中各个对象的位置的话，现存的程序大部分都会崩溃。在 C 语言中，函数往往通过在结构体地址上加上偏移量来计算变量的位置。

19.5 字节顺序

字节顺序是指在一个字中各个字节的顺序。处理器在对字取值时既可能将最低有效位所在的字节当做第一个字节（最左边的字节），也可能将其当做最后一个字节（最右边的字节）。如果最高有效位所在的字节放在低字节位置上，其他字节依次放在高字节位置上，那么该字节顺序称作高位优先（big-endian）。如果最低有效位所在的字节放在高字节位置上，其他字节依次放在低字节位置上，那么就称作低位优先（little-endian）。

编写内核代码时不应该假设字节顺序是给定的哪一种（当然，如果你编写的是与体系结构相关的那部分代码就另当别论了）。Linux 内核支持的机器中使用哪一种字节顺序的都有（甚至包括一些可以在启动的时候选择字节顺序的机器），适用性强的代码应该两种字节顺序都支持。

图 19-1 是高位优先字节顺序的一个实例，图 19-2 是低位优先字节顺序的一个实例。

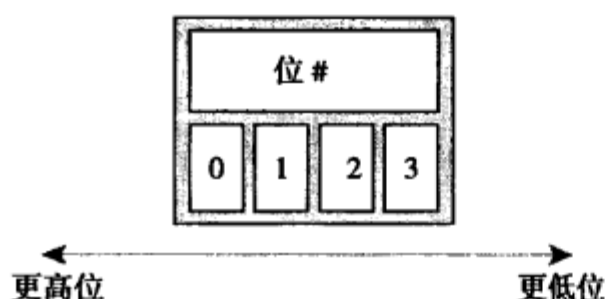


图 19-1 高位优先字节顺序

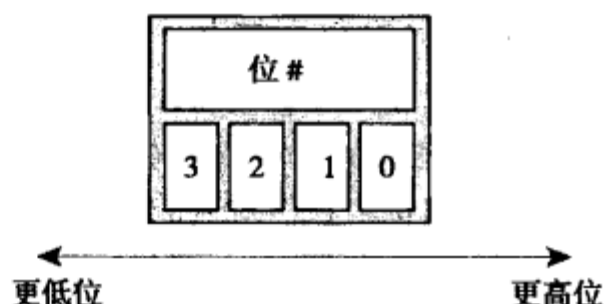


图 19-2 低位优先字节顺序

x86 体系结构，不论 32 位机还是 64 位机，使用的都是低位优先字节顺序。而其他系统大多使用高位优先字节顺序。

让我们看看在实际编程时这些概念有什么意义。让我们考察一下存放在一个 4 字节的整型中的二进制数，它的十进制对应值是 1027：

00000000 00000000 00000100 00000011

在内存中用高位优先和低位优先两种不同字节顺序存放时的比较如表 19-3 所示。

表 19-3 字节顺序比较

地 址	高位优先	低位优先
0	00000000	00000011
1	00000000	00000100
2	00000000	00000000
3	00000011	00000000

注意 使用高位优先的体系结构把最高字节位存放在最小的内存地址上的。这和低位优先形成了鲜明的对照。

最后一个例子，我们提供了如何判断给定的机器使用是高位优先还是低位优先字节顺序的代码：

```
int x = 1;

if (*(char *) &x == 1)
    /* 低位优先 */
```

```
else
    /* 高位优先 */
```

这段代码在用户空间和内核空间都能用。

高位优先和低位优先的历史

高位优先和低位优先源于乔纳森·斯威夫特写于 1726 年的讽刺小说《格列弗游记》。在小说中，虚构的小人国里最重要的政治问题就是应该把鸡蛋从大头敲开还是从小头敲开。那些支持从大头敲开的就是高位优先；而那些支持从小头敲开的，就是低位优先。

高位优先与低位优先的孰优孰差就好像小人国中的政治争论一样，与其说是技术问题，倒不如说是政治问题啦。

对于 Linux 支持的每一种体系结构，相应的内核都会根据机器使用的字节顺序在它的 `<asm/byteorder.h>` 中定义 `__BIG_ENDIAN` 或 `__LITTLE_ENDIAN` 中的一个。

这个头文件还从 `include/linux/byteorder/` 中包含了一组宏命令用于完成字节顺序之间的相互转换。最常用的宏命令有：

```
u32 __cpu_to_be32(u32); /* 把 cpu 字节顺序转换为高位优先字节顺序 */
u32 __cpu_to_le32(u32); /* 把 cpu 字节顺序转换为低位优先字节顺序 */
u32 __be32_to_cpu(u32); /* 把高位优先字节顺序转换为 cpu 字节顺序 */
u32 __le32_to_cpu(u32); /* 把低位优先字节顺序转换为 cpu 字节顺序 */
```

这些转换能够把一种字节顺序变为另一种字节顺序。如果两种字节顺序本来就相同（比如，希望从本地字节顺序转化为高位优先字节顺序，而处理器本身使用的就是高位优先字节顺序），那么宏就什么都不做。否则，它们就进行转换。

19.6 时间

时间测量是另一个内核概念，它随着体系结构甚至内核版本的不同而不同。绝对不要假定时钟中断发生的频率，也就是每秒产生的 jiffies 数目。相反，应该使用 HZ 来正确计量时间。这一点至关重要，因为不但不同的体系结构之间定时中断的频率不同，即使是在同一种体系机构上，两个不同版本的内核之间这种频率也不尽相同。

举个例子，在 x86 系统上，HZ 设定为 100。也就是说，定时中断每秒发生 100 次，也就是每 10ms 一次。可是在 2.6 版以前，x86 上 HZ 定为 1000。而其他体系机构上的数值各不相同：alpha 的 HZ 是 1024 而 ARM 的 HZ 是 100。

绝对不要用 jiffies 直接去和 1000 这样的数值比较，认为这样做大体上不会出问题是要不得的。计量时间的正确方法是乘以或除以 HZ。比如：

```
HZ          /* 1 秒 */
(2*HZ)      /* 2 秒 */
(HZ/2)      /* 半秒 */
(HZ/100)    /* 10ms */
(2*HZ/100)  /* 20ms */
```

HZ 定义在文件 `<asm/param.h>` 中，在前面的第 10 章中曾经讨论过。

19.7 页长度

当处理用页管理的内存时，绝对不要假设页的长度。在 x86-32 下编程的程序员往往错误地认为一页的大小就是 4KB。尽管 x86-32 机器上使用的页确实是 4KB，但是其他不同的体系结构使用的页长度可能不同。实际上有些体系结构还同时支持多种不同长度的页。表 19-4 列举了各种体系结构使用的页的长度。

当处理用页组织管理的内存时，通过 PAGE_SIZE 以字节数来表示页长度。而 PAGE_SHIFT 这个值定义了从最右端屏蔽多少位能够得到该地址对应的页的页号。举例来说，在页长为 4KB 的 x86-32 机上，PAGE_SIZE 为 4096 而 PAGE_SHIFT 为 12。它们都定义于 <asm/page.h> 中。

表 19-4 不同体系结构的页长度

体系结构	PAGE_SHIFT	PAGE_SIZE
alpha	13	8KB
arm	12, 14, 15	4KB, 16KB, 32KB
avr	12	4KB
cris	13	8KB
blackfin	12	16KB
h8300	14	4KB
	12	4KB, 8KB, 16KB, 64KB
m32r	12, 13, 14, 16	4KB
m68k	12	4KB, 8KB
m68knommu	12, 13	4KB
mips	12	4KB
mn10300	12	4KB
parisc	12	4KB
powerpc	12	4KB
s390	12	4KB
sh	12	4KB
sparc	12, 13	4KB, 8KB
um	12	4KB
x86	12	4KB
xtensa	12	4KB

19.8 处理器排序

回忆第 9 章和第 10 章，其中讨论过体系结构对指令序列的排序问题。有些处理器严格限制指令排序，代码指定的所有装载或存储指令都不能被重新排序；而另外一些体系结构对排序要求则很弱，可以自行排序指令序列。

在代码中，如果在对排序要求最弱的体系结构上，要保证指令执行顺序。那么就必须使用诸如 `rmb()` 和 `wmb()` 等恰当的内存屏障来确保处理器以正确顺序提交装载和存储指令。详情请参见第 10 章。

19.9 SMP、内核抢占、高端内存

在讨论可移植性的地方加入有关并发处理、内核抢占和高端内存的部分看起来似乎不太恰当。毕竟，这些都不是会影响到操作系统的硬件之间有所差异的那些特性；恰恰相反，它们都是 Linux 内核本身的一些功能，硬件体系结构根本感知不到它们的存在。但是，它们代表的其实都是可配置的重要选项，而你的代码应该充分考虑到对它们的支持。就是说，只有在编程时就针对 SMP/ 内核抢占 / 高端内存进行了考虑，代码才会无论内核怎样配置，都能身处安全之中。再在前面那些保证可移植性的规范下加上这几条：

- 假设你的代码会在 SMP 系统上运行，要正确地选择和使用锁。
- 假设你的代码会在支持内核抢占的情况下运行，要正确地选择和使用锁和内核抢占语句。
- 假设你的代码会运行在使用高端内存（非永久映射内存）的系统上，必要时使用 `kmap()`。

19.10 小结

要想写出可移植性好、简洁、合适的内核代码，要注意以下两点：

- 编码尽量选取最大公因子：假定任何事情都可能发生，任何潜在的约束也都存在。
- 编码尽量选取最小公约数：不要假定给定的内核特性是可用的，仅仅需要最小的体系结构功能。

编写可移植的代码需要考虑许多问题：字长、数据类型、填充、对齐、字节次序、符号、字节顺序、页大小以及处理器的加载 / 存储排序等。对于绝大多数内核开发来说，可能主要考虑的问题就是保证正确使用数据类型，虽然如此，说不定有朝一日，还是会有些与古老的体系结构有关的问题突然跳出来困扰你。所以说理解移植性的重要性，并且在开发内核过程中时刻注意编写简洁、可移植的代码是非常重要的。



第 20 章

补丁、开发和社区

Linux 的最大优势就是它有一个紧密团结了众多使用者和开发者的社区。社区能帮你检查代码，社区中的专家给你提出忠告，社区中的用户能帮你进行测试，用户还能向你反馈存在的问题。更重要的是，什么样的代码可以加入 Linux 的官方内核树也是由社区做出决定的。因此了解系统到底是怎么运作的就显得尤为重要了。

20.1 社区

如果一定要让 Linux 内核社区在现实世界中找到它的位置，那它也许会叫做内核邮件列表 (Linux Kernel Mailing List) 之家。内核邮件列表 (或者简写成 lkml) 是对内核进行发布、讨论、争辩和打口水仗的主战场。在做任何实际的动作之前，新特性会在此处被讨论，新代码的大部分也会在此处张贴。这个列表每天发布的信息超过 300 条，所以决不适合心血来潮的玩主。任何想踏踏实实研究、认认真真开发内核的人都应该订阅它 (至少要订阅它的摘要或者是它的归档资料)。单单看看这些奇才们使出的一招一式，也能让你受益匪浅了。

你可以通过向 `majordomo@vger.kernel.org` 发送下面的纯文本消息订阅这个邮件列表：

```
subscribe linux-kernel <your@email.address>
```

关于这方面更为详细的信息可以在 <http://vger.kernel.org/> 中找到，此外在 <http://www.tux.org/lkml/>，还有一个专门的 FAQ。

网上还有无数与内核相关或与普通的 Linux 使用相关的资源。<http://kernelnewbies.org/> 是一方适合内核开发初级黑客的乐土——该网站几乎能够满足所有磨刀霍霍向内核的新手的需求。还有两个网站也是不错的资源，包括 <http://www.lwn.net/>，Linux 新闻周刊，它有一个专区报道有关内核的重要新闻；<http://kernelnewbies.org/>，内核直通车，提供关于内核开发一针见血的评论。

20.2 Linux 编码风格

像所有其他大型软件项目一样，Linux 制定了一套编码风格，对代码的格式、风格和布局做出了规定。这么做不是因为 Linux 内核的风格有多么出众 (可能确实还不错) 或是你自己原来的风格有多么拙劣，而是因为保持编码风格的一致有助于提高编程效率。然而对规定编码风格还是存在一些争议，有人认为这其实无关紧要，因为无论如何，最终编译出来的目标码不会受影响。在像内核这样的大型软件项目中，涉及许许多多的开发者，编码的一致性变得至关重要。一致意味着相似和熟悉，也就意味着容易读懂，不含歧义，并且以后的代码仍旧会保持这种风格。这可以让更多的开发者读懂你的代码，也能让你读懂更多其他人编写的代码。在开源项目中，眼球自

然是越多越好。

跟选择一个唯一确定的风格相比，到底选择什么样的风格反而显得不是那么重要了。好在 Linus 早就展示出了该用什么风格，而且绝大部分代码都照这么做了。编码风格的主要规范伴随着 Linus 一贯的幽默，都记录在内核源代码树的 Documentation/CodingStyle 中了。

20.2.1 缩进

缩进风格是用制表位 (Tab) 每次缩进 8 个字符长度。这不是说用 8 个空格缩进就行了。这里的规定很明确，每次缩进通过制表位进行，每个制表位 8 个字符长度。例如：

```
static void get_new_ship(const char *name)
{
    if (!name)
        name = DEFAULT_SHIP_NAME;
    get_new_ship_with_name(name);
}
```

不知为什么，虽然违反它会对可读性带来非常大的冲击，但这个规定还是最容易被人们违反。八个字符长度的缩进能让不同的代码块看起来一目了然，特别是在连续几个小时的开发之后，效果更加明显。当然，随着缩进层数的增加，八字符制表位的左侧可用空间就所剩不多了。这是因为每行最多有 80 个字符（参见 20.2.2 节）。Linus 极其反对这样做，他认为代码不应当复杂、费解到需要两级或者三级缩进。如果真的需要多层缩进，他建议，应当重构你的代码，把复杂的层次关系（为此形成多层缩进）分解为独立的功能。

20.2.2 switch 语句

Switch 语句下属的 case 标记应该缩进到和 switch 声明对齐，这样将有助于减少 8 个字符的 tab 键带来的排版缩进，比如

```
switch (animal) {
case ANIMAL_CAT:
    handle_cats();
    break;
case ANIMAL_WOLF:
    handle_wolves();
    /* fall through */
case ANIMAL_DOG:
    handle_dogs();
    break;
default:
    printk(KERN_WARNING "Unknown animal %d!\n", animal);
}
```

当执行逻辑需要有意地从一个 case 声明尾部进入另外一个 case 声明时，对其进行评注无疑是一个普遍的（良好的）实践经验。如示例中所见。

20.2.3 空格

这一节讨论给符号和关键字加空格，而不涉及在缩进中加空格（这将在后面两节中讨论）。一般来说，Linux 的编码风格规定，空格放在关键字周围，函数名和圆括号之间无空格。例如：

```
if (foo)
while (foo)
for (i = 0; i < NR_CPUS; i++)
switch (foo)
```

相反，函数、宏以及与函数相像的关键字（例如 sizeof、Typeof 以及 alignof）在关键字和圆括号之间没有空格。

```
wake_up_process(task);
size_t nlongs = BITS_TO_LONG(nbits);
int len = sizeof(struct task_struct);
typeof(*p)
__alignof__(struct sockaddr *)
__attribute__((packed))
```

在括号内，如前所示，参数前后也不加空格。例如，下面这是禁止的：

```
int prio = task_prio( task ); /* BAD STYLE! */
```

对于大多数二元或者三元操作符，在操作符的两边加上空格。例如：

```
int sum = a + b;
int product = a * b;
int mod = a % b;
int ret = (bar) ? bar : 0;
return (ret ? 0 : size);
int nr = nr ? : 1; /* allowed shortcut, same as "nr ? nr : 1" */
if (x < y)
if (tsk->flags & PF_SUPERPRIV)
mask = POLLIN | POLLRDNORM;
```

相反，对于大多数一元操作符，在操作符和操作数之间不加空格：

```
if (!foo)
int len = foo.len;
struct work_struct *work = &dwork->work;
foo++;
-bar;
unsigned long inverted = ~mask;
```

在提领运算符的周围加上合适的空格尤为重要。正确的风格是：

```
char *strcpy(char *dest, const char *src)
```

在提领运算符的一边加上空格是不良的风格：

```
char * strcpy(char * dest, const char * src) /* BAD STYLE */
```

把提领运算符放在紧挨类型的地方也是借用 C++ 风格的一种不良作风：

```
char* strcpy(char* dest, const char* src) /* BAD STYLE */
```

20.2.4 花括号

花括号的使用不存在技术上的差异，完全是个人喜好问题，但我们还是必须宣传一致的风格。内核选定的风格是左括号紧跟在语句的最后，与语句在相同的一行。而右括号要新起一行，作为该行的第一个字符。如下例：

```
if (strcmp(buf, "NO_", 3) == 0) {  
    neg = 1;  
    cmp += 3;  
}
```

注意，如果接下来的标识符是相同语句块的一部分，那么右花括号就不单独占一行，而是与那个标识符在同一行，例如：

```
if (ret) {  
    sysctl_sched_rt_period = old_period;  
    sysctl_sched_rt_runtime = old_runtime;  
} else {  
    def_rt_bandwidth.rt_runtime = global_rt_runtime();  
    def_rt_bandwidth.rt_period = ns_to_ktime(global_rt_period());  
}
```

还有，

```
do {  
    percpu_counter_add(&ca->cpustat[idx], val);  
    ca = ca->parent;  
} while (ca);
```

函数不采用这样的书写方式，因为函数不会在内部嵌套定义：

```
unsigned long func(void)  
{  
    /* ... */  
}
```

最后，不需要一定使用括号的语句可以忽略它：

```
if (cnt > 63)  
    cnt = 63;
```

所有这些方法原理都源自 K&R[⊖]。大多数编码风格都遵循 K&R 风格，这是在那本著名的书中所使用的 C 编码风格。

⊖ 《C 语言程序设计（第2版）》由 Brian Kernighan 和 Dennis Ritchie 著，这两位作者简称 K&R，该书是 C 语言的圣经，由 C 语言的发明者和他的同事合著。

20.2.5 每行代码的长度

源代码中要尽可能地保证每行代码长度不超过 80 个字符，因为这样做可使代码最适合在标准的 80×24 的终端上显示。事实上，并不存在一个广泛接受的标准——如果代码行超过 80 应该折到下一行。有些开发者也许根本不理睬代码跨行问题，而是让编辑器以可读的方式处理代码的显示；而有些开发者会手动插入断行符来分割代码行，他们也许会在新行头插入两个 tab 键以便和原先行错开。

类似的，有些开发者会在圆括号内来分行，对齐排列函数参数，比如：

```
static void get_new_parrot(const char *name,
                          unsigned long disposition,
                          unsigned long feather_quality)
```

而另一些开发者虽然也会将参数分行输入，但却不会把它们对齐排列，而是在开头简单的加入两个标准 tab。比如：

```
int find_pirate_flag_by_color(const char *color,
                              const char *name, int len)
```

因为分行没有确定的规则，所以开发者在这点上可采取自由行动。

大多数内核贡献者（包括我在内）更愿意采用前一个例子中的方式：把大于 80 个字符的行进行拆分，尽量让新产生的行与前一行对齐。

20.2.6 命名规范

名称中不允许使用骆驼拼写法（CamelCase）、Studly Caps 或者其他混合的大小写字符。局部变量如果能够清楚地表明它的用途，那么选取 idx 甚至是 i 这样的名称都是可行的。而像 theLoopIndex 这样冗长繁复的名字不在接受之列。匈牙利命名法（在变量名称中加入变量的类别）是不必要的，绝对不允许使用——要知道这里是 C，不是 Java；用的是 Unix，不是 Windows。

而全局变量和函数应该选择包含描述性内容的名称，并且使用小写字母，必要时加上下划线区分单词。给一个全局函数起名为 atty() 会使人迷惑；而像 get_active_tty() 这样就比较容易让人接受了。这里是 Linux，不是 BSD。

20.2.7 函数

根据经验，函数的代码长度不应该超过两屏，局部变量不应超过 10 个。一个函数应该功能单一并且实现精准。将一个函数分解成一些更短小的函数的组合不会带来危害。如果你担心函数调用导致的开销，可以使用 inline 关键字。

20.2.8 注释

代码的注释非常重要，但注释必须按照正确的方式进行。一般情况下，你应该描述的是你的代码要做什么和为什么要做，而不是具体通过什么方式实现的。怎么实现应该由代码本身展现。如果你不是这样做的，那么应该回过头去考虑一下你写的东西了。此外，注释不应该包含谁写了

哪个函数、修改日期和其他那些琐碎而无实际意义的内容。这些信息应该集中在文件最开头的地方。

虽然 gcc 也支持 C++ 风格的注释符号，但内核只使用 C 风格的注释符号。内核中一条注释看起来像是这样：

```
/*
 * get_ship_speed() - return the current speed of the pirate ship
 * We need this to calculate the ship coordinates. As this function can sleep,
 * do not call while holding a spinlock
 */
```

在注释中，重要信息常常以“XXX：”开头，而 bug 通常以“FIXME：”开头，就像：

```
/*
 * FIXME: We assume dog == cat which may not be true in the future
 */
```

内核包含一套自动文档生成工具。它源自 GNOME-doc，略加修改后命名为 Kernel-doc。如果想要生成独立的 HTML 格式文档，运行

```
make htmldocs
```

如果想要 postscript 格式的话，用下列命令：

```
make psdocs
```

你也可以按照特定的格式对你的函数进行注解，这样该工具也可以为你的函数服务：

```
/**
 * find_treasure find 'X marks the spot'
 * @map _treasure map
 * @time - time the treasure was hidden
 *
 * Must call while holding the pirate_ship_lock.
 */
void find_treasure(int map, struct timeval *time)
{
    /* ... */
}
```

有关此方面更多的细节请参看 Documentation/kernel-doc-nano-HOWTO.txt 文件。

20.2.9 typedef

内核开发者们强烈反对使用 typedef 语句。他们的理由是：

- typedef 掩盖了数据的真实类型。
- 由于数据类型隐藏起来了，所以很容易因此而犯错误，比如以传值的方式向栈中推入结构。
- 使用 typedef 往往是因为想要偷懒。[⊖]

⊖ 有些程序员往往是为了少敲打几次键盘而使用 typedef，比如 typedef unsigned char uchar。而这种缩写可能会引发理解和一致性上的问题，所以仅仅出于此目的而使用 typedef 被作者视为懒惰行为。——译者注

无论如何，就算是为了别惹人耻笑吧，尽量少用 typedef。

当然，typedef 也有它施展身手的时候：当需要隐藏变量与体系结构相关的实现细节的时候，当某种类型将来有可能发生变化，而现有程序必须要考虑到向前兼容问题的时候，都需要 typedef。使用 typedef 要谨慎，只有在确实需要的时候再用它；如果仅仅是为了少敲打几下键盘，别使用它。

20.2.10 多用现成的东西

请勿闭门造车。内核本身就提供了字符串操作函数、压缩函数和一个链表接口，所以请使用它们。

不要为了使现存接口更通用化而对它们进行新的封装。你经常会发现，当把一段代码从某个操作系统移植到 Linux 上的时候，表面好像看起来根本没什么问题，可是隐藏在接口下面复杂的函数调用却往往是与它原有的内核相关的。没人愿意面对这些问题，所以请直接使用内核提供的接口。

20.2.11 在源码中减少使用 ifdef

我们不赞成在源码中使用 ifdef 预处理指令。你绝不应该在自己的函数中使用如下的实现方法：

```
...
#ifdef CONFIG_FOO
    foo();
#endif
...
```

相反，应该采取的方法是在 CONFIG_FOO 没定义的时候让 foo() 函数为空。

```
#ifdef CONFIG_FOO
static int foo(void)
{
    /* ... */
}
#else
static inline int foo(void) { }
#endif /*CONFIG_FOO*/
```

这样，你在任何情况下都能调用 foo() 了。让编译器去做这些工作好了。

20.2.12 结构初始化

结构初始化的时候必须在它的成员前加上结构标识符。这种初始化能避免错误地使用其他结构而引发一个初始化错误。它也支持使用忽略值。不幸的是，C99 标准改用了一种丑陋的格式来表示这种标识符，于是 gcc 就再也不支持原来 GNU 风格的标识符了，尽管它看起来确实要更帅一些。结果，内核代码现在必须都要使用新的 C99 标识符格式了，不管它有多难看：

```
struct foo my_foo = {
    .a = INITIAL_A,
```

```
.b      = INITIAL_B,  
};
```

其中 `a` 和 `b` 是结构体 `foo` 的成员，而 `INITIAL_A` 和 `INITIAL_B` 是它们对应的初始值。如果一个字段没有给初始值，那么它就会被设置为 ANSI C 规定的默认值（如指针被设为 `NULL`，整型被设为 `0`，浮点数被设置为 `0.0`）。举例来说，如果 `foo` 结构体还有一个 `int` 型的 `c` 成员，那么上面的初始化语句执行之后 `c` 会被设置为 `0`。

20.2.13 代码的事后修正

即使你得到了一段与内核编码风格风马牛不相及的代码，也不用发愁。只消抬抬手，`indent` 工具就能帮你解决它。`indent` 是一个在大多数 Linux 系统中都能找到的好工具，它可以按照指定的方式对源代码进行格式化。默认情况下它按照不怎么好看的 GNU 编码风格格式化代码。想要用 Linux 内核编码风格，执行下列命令：

```
indent -kr -i8 -ts8 -sob -l80 -ss -bs -psl <file>
```

这样就能调用该工具按内核编码风格对你的代码进行格式化了。此外，还可以通过 `scripts/Lindent` 自动按照所需的格式调用 `indent`。

20.3 管理系统

内核黑客就是那些从事内核开发工作的人。做这些工作有些人是因为钱，有些人是因为嗜好，但几乎所有人都是为了从中找到快乐。所有做出卓越贡献的黑客都能在源代码树根目录上的 `CREDITS` 文件中留名。

内核中几乎每个部分都对应一个维护者。维护者是指一个或几个对内核特定部分负责的人。比如，每个单独的驱动程序都对应一个维护者。每个内核子系统（如网络）也有一个维护者。驱动程序和子系统的维护者也能在源代码树根目录上的 `MAINTAINERS` 文件中找到。

还有一类特殊的维护者称作内核维护者。这些人负责维护的实际上就是代码树本身。以前，由 `Linus` 自己负责维护开发版的内核（乐趣尽在此中），稳定版最开始的一段时间也由他来维护。等到该内核稳定下来了，他就会把火炬传递给最好的内核开发者中的一些人手上，由这些人负责维护该代码树，而 `Linus` 会转身启动下一开发版本的内核开发工作。在 2.6 内核继续保持稳定的“新世界秩序”前提下，`Linus` 仍然维护着 2.6 系列的内核。另外的开发者以严格的“发现 bug-修订 bug”模式维护 2.4 系列内核。

20.4 提交错误报告

如果碰到了 `bug`，最理想的应对无疑是写出修正代码，创建补丁，测试后提交它，这个流程在 20.5 节会详细介绍。当然，也可以报告这个问题，然后让其他人替你解决。

提交一个错误报告最重要的莫过于对问题进行清楚的描述。要讲清楚症状、系统输出信息、完整并经过解码的 `oops`（如果有的话）。更重要的是，你应该尽可能地提供能够准确地重现这个错误的步骤，并提供你的机器的硬件配置基本信息。

然后再来考虑把这个错误报告发送给谁。在内核源代码书的根目录中，MAINTAINERS 文件列举出了每个相关的设备驱动程序和子系统的单独信息——接收关于其所维护的代码的所有问题。如果找不到对此问题感兴趣的人，那么就把它报告给位于 `linux-kernel@vger.kernel.org` 的内核邮件列表。即使你已经找到维护者了，贴一份副本在那里也不会有什么坏处。

文档 REPORTING-BUGS 和 Documentation/oops-tracing.txt 中有更多相关信息。

20.5 补丁

对内核的任何修改都是以补丁的形式发布的，而补丁其实是 GNU diff(1) 程序的一种特定格式的输出，该格式的信息能够被 patch(1) 程序接受。

20.5.1 创建补丁

创建补丁最简单的办法是通过两份内核源代码进行，一份源码，另一份是加进了所修改部分的源代码。一般会给原来的内核代码起名 `linux-x.y.z`（其实就是把源代码包解压缩后所得到的文件夹），而修改过的就起名为 `linux`。然后利用下面的命令通过这两份代码创建补丁：

```
diff -urN linux-x.y.z/ linux/ > my-patch
```

你可以在自己的目录下运行该命令，一般都是在 `home` 目录下，而不是在 `/usr/src/linux` 目录下进行这种操作，所以不一定必须具备超级用户权限。通过 `-u` 参数指定使用特殊的 diff 输出格式。否则得到的 patch 格式怪异，一般人都无法看懂。`-r` 参数保证会遍历所有子目录进行操作，而 `-N` 参数指明做出修改的源代码中所有新加入的文件在 diff 操作时会包含在内。另外，如果想对一个单独的文件进行 diff，你也可以这么做：

```
diff -u linux-x.y.z/some/file linux/some/file > my-patch
```

注意，在你自己代码所在的目录下执行 diff 很重要。这样创建的补丁别人用起来更方便，哪怕他们的目录名字叫 `differ` 也没问题。执行一个这样生成的补丁，只需要在你自己代码树的根目录执行下列命令就可以了：

```
patch -p1 < ../my-patch
```

在这个例子中，补丁的名字叫 `my-patch`，它位于当前目录的上一级目录中。`-p1` 参数用来剥去补丁中头一个目录的名称。这么做的好处是可以在打补丁的时候忽略创建补丁的人的目录命名习惯。

`diffstat` 是一个很有用的工具，它可以列出补丁所引起的变更的统计（加入或移去的代码行）。输出关于补丁的信息，执行：

```
diffstat -p1 my-patch
```

在向 `lkml` 贴出自己的补丁时，附上这份信息往往会很有用。由于 `patch(1)` 会忽略第一个 diff 之前的所有内容，所以你甚至可以在 `patch` 的最前面直接加上简短的说明。

20.5.2 用 Git 创建补丁

如果你用 Git 管理源代码树，你照样需要用 Git 创建补丁——也就是没有必要按部就班地把上述手工的步骤操作一遍，但是你需要忍受 Git 的复杂性。用 Git 创建补丁并不是什么难事，只需要两个过程。首先，你必须是修改者，然后在本地提交你的修改。把修改提交到 Git 树与提交到标准的源代码树并没有什么两样。你根本不需要专门做任何事情去编辑存放在 Git 中的文件。你做出修改后，就需要把所做的修改提交到你的 Git 版本库：

```
git commit -a
```

-a 参数表示提交所有的修改。如果你仅仅想提交某个指定文件的修改，则如下示例：

```
git commit some/file.c
```

但是即使有了 -a 参数，Git 并不立即提交新文件，直到把它们添加到版本库中才提交。要增加一个文件，然后再提交（以及其他所有的修改），则输入如下两条命令：

```
git add some/other/file.c
git commit -a
```

当执行 Git 的 commit 命令时，Git 会要求输入一个更改日志。你应该尽量填写的详细和完整，清楚地解释修改缘由。（我们将在 20.5.3 节里详细介绍修改日志里应该包含什么）你可以针对你的版本库创建多个提交。Git 的设计可谓考虑周全，多个提交甚至可以针对同一文件，每个提交的创建各自独立。当在源码树中有一个（或两个）提交时，可以为每个提交创建一个补丁，可以像 20.5.1 节所描述的那样来处理这个补丁：

```
git format-patch origin
```

对于所有的提交，这样产生的补丁放在你的版本库中而不是原始树中。Git 产生的补丁位于源代码树的根目录中。如果只想为最后第 N 次提交产生补丁，则可以执行下列命令：

```
git format-patch -N
```

例如，下面的命令只为最后一次提交产生一个补丁：

```
git format-patch -1
```

20.5.3 提交补丁

补丁可以按照 20.5.2 节描述的方式创建。如果补丁涉及了某个特定的驱动程序或子系统，应该把它发给 MAINTAINER 中列举的相关部分的维护者。此外，还应该向 Linux 内核邮件列表 linux-kernel@vger.kernel.org 发送一份拷贝。只有在经过广泛的讨论之后，或者是补丁所做的修改很细微并且很容易就能保证正确的时候，才应该向内核维护者（比如说 Linus）提交。

一般包含一份补丁的邮件，它的主题一栏内容应该以 “[PATCH] 简要说明” 的格式写出。邮件的主体部分应该描述所做的改变的技术细节，以及要做这些的原因，越详细越准确越好。在 E-mail 中还要注明补丁对应的内核版本。

内核开发者们都希望能通过邮件阅读补丁，并且能够将其保存为一个单独文件。因此，最好

把补丁直接插入邮件，放在所有信息的最后。还要小心一些，差劲的邮件客户端工具，它们会加入信息或者改变邮件的格式；这会导致补丁出错，从而引起其他开发者的不满。如果你用的邮件客户端工具也有类似表现，就检查一下，看它是否有“Insert Inline”、“Preformat”或类似功能。如果有的话，就用纯文本方式把你的补丁贴到邮件上作为附件，不要对它做什么编码工作。

如果你的补丁很大或者包含对几个不同的逻辑的修改，那么应该将你的补丁分成几块，每块对应一个逻辑。比如你在补丁中引入了一个新的 API，并且同时对几个驱动程序进行了修改以便利用它，那么应该把该补丁一分为二（先是新的 API，然后是对驱动程序的修改），邮件也写成两份。如果任何一个部分需要其他的补丁先行，要明确地注明这一点。

提交之后，保持耐心，等待答复。别因为某些反对言论而灰心——至少你还是得到回应了嘛！和其他人讨论这个问题并且在需要的时候应该提交修正过的新补丁。如果你压根就没听到回声，想想是什么出了问题，然后着手解决它。多请邮件列表和维护者们提出宝贵意见。运气好的话，未来版本的内核发行时，你可能就会看到自己做出的修改了——那可就真的该恭喜你了！

20.6 小结

对于黑客而言，最可贵的品质便是渴望——就如身上痒痒，不抓不快一般的渴望和决心。本书讲述了 Linux 内核的主要部分，讨论了接口、数据结构、算法和原理。它从实践出发，以内在的视角洞悉内核，既可以满足你的好奇心，也可以帮助你开始学习内核。

不过，正如我前面所说，你上路的唯一方法是去自己读、写代码。Linux 社区不但创造了这样的条件，而且很欢迎大家这么做。好了，不管你追求什么，现在就开始去做吧。



参 考 资 料

在这里列出的书籍是对本书内容的补充。注意，本书最好的“课外阅读”参考资料就是内核源代码。作为 Linux 的使用者，我们被赋予无限制地获得全部现代操作系统源代码的权利。别把这当做是理所当然的，不要暴殄天物，要潜心钻研它。

操作系统设计书籍

这些书覆盖了操作系统设计领域，常作为大学教科书。它们包含设计一个功能健全的操作系统所需要的概念、算法、问题和解决方法。我推荐列出的所有书籍，如果非得我选择一本，那么 Deitel 的书既易理解又有趣可读，它是我的首选。

H. Deitel、P. Deitel 以及 D. Choffnes 的《Operating Systems》，Prentice Hall，2003。该书对操作系统的原理进行了透彻讲述，并且列出了很多出色的实例研究，适合把理论付诸于实践。

Tanenbaum, Andrew 的《Modern Operating Systems》，Prentice Hall，2007。该书深刻揭示了标准操作系统设计精髓，其中也不乏讨论许多今天流行的现代操作系统，如 Unix 和 Windows。

Tanenbaum, Andrew 的《Operating Systems : Design and Implementation》，Prentice Hall，2006。该书精辟地介绍了如何设计和实现一个类 Unix 操作系统——Minix。

Silberschatz A.、P. Galvin 和 G. Gagne 的《Operating System Concepts》，John Wiley and Sons，2008。由于该书封面图片看起来和恐龙有关，所以也被称为“恐龙宝典”。这是一本介绍操作系统设计的好书，而且频繁再版，版版经典。

Unix 内核书籍

这些书主要针对 Unix 内核设计与实现，前五本讨论 Unix 特定版本，后两本关注所有 Unix 版本的通用特点。如果你只买其中的两本，那么我坚持推荐最后两本：

Bach, Maurice 的《The Design of the Unix Operating System》，Prentice Hall，1986。这是一本讨论 Unix 系统 V2 版本的优秀书籍。

McKusick M.、K. Bostic、M. Karels 和 J. Quarterman 的《The Design and Implementation of 4.4BSD Operating System》，Addison-Wesley, 1996。操作系统设计者自己编写的讨论设计 4.4BSD 系统的优秀书籍。

McKusick, M. 和 G. Neville-Neil 的《The Design and Implementation of the FreeBSD Operating System》，Addison-Wesley, 2004。这是一本讨论 FreeBSD 5.2 系统设计与实现的优秀书籍。

McDougall, R 和 J. Mauro 的《Solaris Internals: Solaris and OpenSolaris Kernel Architecture》，Prentice Hall，2006。这是一本对 Solaris 内核的核心子系统和算法进行精彩讨论的优秀书籍。

Cooper C. 和 C. Moore 的《HP-UX 11i Internals》，Prentice Hall, 2004。这本书对 HP-UX 和 PA-RISC 体系结构的内幕进行了初探。

Vahalia, Uresh 的《Unix Internals: The New Frontiers》，Prentice Hall, 1995。这是一本关于现代 Unix 系统特色的优秀书籍，主要介绍了线程管理和内核抢占等功能。

Schimmel, Curt 的《UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers》，Addison-Wesley, 1994。该书揭示了在现代体系结构上实现现代 Unix 操作系统可能要面对的种种危险。强烈推荐阅读这本图书。

Linux 内核书籍

以下书籍与本书一样讨论 Linux 内核。有点可惜的是，这方面的好书不多。不过，我推荐阅读下列两本。

Benvenuti, Christian 的《Understanding Linux Network Internals》，O'Reilly and Associates, 2005。这本书对 Linux 网络进行了深入分析。

Corbet, J. 、A. Rubini 和 G. Kroah-Hartman 的《Linux Device Drivers》，O'Reilly and Associates, 2005。这本书讨论了在 2.6 内核上如何编写驱动程序，尤其是针对各种设备重点描述了所支持的编程接口，这是一本经典的介绍编写驱动程序的图书。

关于其他系统的内核书

了解你的敌人——错误和竞争对手——才能处于不败之地。下列书籍讨论了非 Linux 系统的设计与实现。你可以从中获得经验或教训。

Kogan M. 和 H. Deitel 的《The Design of OS/2》，Addison-Wesley, 1996。该书介绍了 OS/2 2.0 的设计。

Singh, Amit 的《Mac OS X Internals : A Systems Approach》，Addison-Wesley Professional, 2006。该书对整个 Mac OS X 系统进行了完整论述，既有深度又有广度。

Solomon, D. 和 M. Russinovich 的《Windows Internals : Covering Windows Server 2008 and Windows Vista》，Microsoft Press, 2009。这是一本介绍非 Unix 操作系统的有趣书籍。

关于 Unix API 的书籍

深入探索 Unix 系统的 API 函数不仅对编写优秀的用户空间应用程序很关键，同时也对理解内核功能大有帮助。

Love, Robert 的《Linux System Programming》，O'Reilly and Associates, 2007。该书作者负责 Linux 的系统级编程，该书涵盖了 Linux 系统调用以及 libc API 等内容，并对 Linux 所具有的一些技巧和方式给予关注。

Stevens W. R. 和 S. Rago 的《Advanced Programming in the UNIX Environment》，Addison-Wesley, 2008。该书是讨论 Unix 系统调用接口的权威书籍。

Stevens W. Richard 的《UNIX Network Programming Volume 1》，Prentice Hall, 2004。这是一本关于 Unix 系统套接字 API 的经典教材。

关于 C 语言编程的书

Linux 内核以及 Linux 系统的其他很多软件都是用 C 语言编写的。下列两本书值得推荐阅读。

Kernighan B. 和 D.Richie 的《The C Programming Language》，Prentice Hall, 1988。这是一本介绍 C 语言编程的权威书籍，由 C 的发明者及其合作者编写。

van der Linden, Peter 的《Expert C Programming》，Prentice Hall, 1994。该书对不易理解的 C 细节给予了难能可贵的讨论，作者的语言风趣幽默。

其他书籍

下面的书籍严格说可能和操作系统无关，但是这些书无疑从某些层面影响了操作系统。

Hofstadter, Douglas 的《G·del, Escher, Bach : An Eternal Golden Braid 》，Basic Books, 1999。这是一本对人类思想进行深刻研究的必备书籍，它的内容覆盖众多主题，其中也包含计算机科学。

Knuth, Donald 的《The Art of Computer Programming, Volume 1》，Adderson-Wesley, 1997。这是一本无价的介绍计算机科学算法的基础书籍，其中介绍了内存管理中的最佳适应算法和最坏适应算法。

Web 站点

Kernel.org。内核代码库的官方站点，同时也容纳了核心内核黑客的大量补丁包。网址是：<http://www.kernel.org>。

Linux Weekly News。出色的新闻站点，对一周 Linux（包括内核）所发生的新闻给予睿智、准确的评论。强烈推荐浏览。网址是：www.lwn.net。

OS 新闻。包括关于操作系统的新闻，也包括原创文章、访谈以及 reviews。www.osnews.com。



本书详细描述了Linux内核的设计与实现。内核代码的编写者、开发者以及程序开发人员都可以通过阅读本书受益，他们可以更好地理解操作系统原理，并将其应用在自己的编码中以提高效率和生产率。

本书详细描述了Linux内核的主要子系统和特点，包括Linux内核的设计、实现和接口，从理论到实践涵盖了Linux内核的方方面面，可以满足读者的各种兴趣和需求。

作者是一位Linux内核核心开发人员，他分享了在开发Linux 2.6内核过程中颇具价值的知识和经验。本书的主题包括进程管理、进程调度、时间管理和定时器、系统调用接口、内存寻址、内存管理和页缓存、VFS、内核同步、移植性相关的问题以及调试技术。同时本书也涵盖了Linux 2.6内核中颇具特色的内容，包括CFS调度程序、抢占式内核、块I/O层以及I/O调度程序。

本书新增内容包括：

- 增加一章专门描述内核数据结构（第6章）；
- 详细描述中断处理程序和下半部机制；
- 扩充虚拟内存和内存分配的内容；
- 调试Linux内核的技巧；
- 内核同步和锁机制的深度描述；
- 提交内核补丁以及参与Linux内核社区的建设性建议。

Linux 内核设计与实现

Linux Kernel Development
Third Edition

原书第3版

客服热线: (010) 88378991, 88361066
购书热线: (010) 68326294, 88379649, 68995259
投稿热线: (010) 88379604
读者信箱: hzsj@hzbook.com

PEARSON

www.pearsonhighered.com



上架指导: 计算机/程序设计

ISBN 978-7-111-33829-1



9 787111 338291

华章网站 <http://www.hzbook.com>

网上购书: www.china-pub.com

封面设计 范华明

定价: 69.00元