

```
foo.gd = alloc_disk(16);  
if (!foo.gd) goto error_no_gendisk;
```

驱动程序首先初始化自旋锁，然后分配一个磁盘描述符。正如在前面的图 14-3 中所看到的，gendisk 结构是块 I/O 子系统中最重要的数据结构，因为它涉及许多其他的数据结构。alloc_disk() 函数也分配一个存放磁盘分区描述符的数组。该函数所需要的参数是数组中 hd_struct 结构的元素个数，16 表示驱动程序可以支持 16 个磁盘，而每个磁盘可以包含 15 个分区（0 分区不使用）。

初始化 gendisk 描述符

接下来，驱动程序初始化 gendisk 描述符的一些字段：

```
foo.gd->private_data = &foo;  
foo.gd->major = FOO_MAJOR;  
foo.gd->first_minor = 0;  
foo.gd->minors = 16;  
set_capacity(foo.gd, foo_disk_capacity_in_sectors);  
strcpy(foo.gd->disk_name, "foo");  
foo.gd->fops = &foo_ops;
```

foo 描述符的地址存放在 gendisk 结构的 private_data 字段中，因此被块 I/O 子系统当作方法调用的低级驱动程序函数可以迅速地查找到驱动程序描述符——如果驱动程序可以并发地处理多个磁盘，那么这种方式可以提高效率。set_capacity() 函数将 capacity 字段初始化为以 512 字节扇区为单位的磁盘大小——这个值也可能在探测硬件并询问磁盘参数时确定。

初始化块设备操作表

gendisk 描述符的 fops 字段被初始化为自定义的块设备方法表的地址（参见本章前面的表 14-4）（注 3）。类似地，设备驱动程序的 foo_ops 表中包含设备驱动程序的特有函数。例如，如果硬件设备支持可移动磁盘，通用块层将调用 media_changed 方法检查自从最后一次安装或打开该块设备以来磁盘是否被更换。通常通过向硬件控制器发送一些低级命令来完成该检查，因此，每个设备驱动程序所实现的 media_changed 方法都是不同的。

类似地，仅当通用块层不知道如何处理 ioctl 命令时才调用 ioctl 方法。例如，当一个 ioctl() 系统调用询问磁盘构造时，也就是磁盘使用的柱面数、磁道数、扇区数以及磁头数时，通常调用该方法。因此，每个设备驱动程序所实现的 ioctl 方法也都是不同的。

注 3： 不应该把块设备方法和块设备文件操作混为一谈，参见本章稍后“打开块设备文件”一节。

分配和初始化请求队列

我们勇敢的设备驱动程序设计者现在将要建立一个请求队列,该队列用于存放等待处理的请求。可以通过如下操作轻松地建立请求队列:

```
foo.gd->rq = blk_init_queue(foo_strategy, &foo.lock);
if (!foo.gd->rq) goto error_no_request_queue;
blk_queue_hardsect_size(foo.gd->rd, foo_hard_sector_size);
blk_queue_max_sectors(foo.gd->rd, foo_max_sectors);
blk_queue_max_hw_segments(foo.gd->rd, foo_max_hw_segments);
blk_queue_max_phys_segments(foo.gd->rd, foo_max_phys_segments);
```

`blk_init_queue()`函数分配一个请求队列描述符并将其中许多字段初始化为缺省值。它接收的参数为设备描述符的自旋锁的地址 (`foo.gd->rq->queue_lock` 字段值) 和设备驱动程序的策略例程 (参见下一节“策略例程”) 的地址 (`foo.gd->rq->request_fn` 字段值)。该函数也初始化 `foo.gd->rq->elevator` 字段, 并强制驱动程序使用缺省的 I/O 调度算法。如果设备驱动程序想要使用其他的调度算法, 可在稍后覆盖 `elevator` 字段的地址。

接下来, 使用几个辅助函数将请求队列描述符的不同字段设为设备驱动程序的特征值 (参考表 14-6 中的类似字段)。

设置中断处理程序

正如第四章的“I/O 中断处理”一节中所介绍的, 设备驱动程序需要为设备注册 IRQ 线。这可以通过如下操作完成:

```
request_irq(foo_irq, foo_interrupt,
            SA_INTERRUPT|SA_SHIRQ, "foo", NULL);
```

`foo_interrupt()` 函数是设备的中断处理程序, 我们将在本章稍后的“中断处理程序”一节中讨论它的一些特性。

注册磁盘

最后, 设备驱动程序的所有数据结构已经准备好了: 初始化阶段的最后一步就是“注册”和激活磁盘。这可以简单地通过执行下面的操作完成:

```
add_disk(foo.gd);
```

`add_disk()` 函数接收 `gendisk` 描述符的地址作为其参数, 主要执行下列操作:

1. 设置 `gd->flags` 的 `GENHD_FL_UP` 标志。
2. 调用 `kobj_map()` 建立设备驱动程序和设备的主设备号 (连同相关范围内的次设备

- 号)之间的连接(参见第十三章的“字符设备驱动程序”一节;注意,在这种情况下, `kobject` 映射域由 `bdev_map` 变量表示)。
3. 注册设备驱动程序模型的 `gendisk` 描述符中的 `kobject` 结构,它作为设备驱动程序处理的一个新设备(例如 `/sys/block/foo`)。
 4. 如果需要,扫描磁盘中的分区表;对于查找到的每个分区,适当地初始化 `foo.gd->part` 数组中相应的 `hd_struct` 描述符。同时注册设备驱动程序模型中的分区(例如 `/sys/block/foo/fool`)。
 5. 注册设备驱动程序模型的请求队列描述符中内嵌的 `kobject` 结构(例如 `/sys/block/foo/queue`)。

一旦 `add_disk()` 返回,设备驱动程序就可以工作了。进行初始化的函数终止;策略例程和中断处理程序开始处理 I/O 调度程序传送给设备驱动程序的每个请求。

策略例程

策略例程是块设备驱动程序的一个函数或一组函数,它与硬件块设备之间相互作用以满足调度队列中所汇集的请求。通过请求队列描述符中的 `request_fn` 方法可以调用策略例程——例如前面一节介绍的 `foo_strategy()` 函数, I/O 调度程序层将请求队列描述符 `q` 的地址传递给该函数。

如前所述,在把新的请求插入到空的请求队列后,策略例程通常才被启动。只要块设备驱动程序被激活,就应该对队列中的所有请求都进行处理,直到队列为空才结束。

策略例程的简单实现如下:对于调度队列中的每个元素,与块设备控制器相互作用共同为请求服务,等待直到数据传送完成,然后把已经服务过的请求从队列中删除,继续处理调度队列中的下一个请求。

这种实现效率并不高。即使假设可以使用 DMA 传送数据,策略例程在等待 I/O 操作完成的过程中也必须自行挂起。也就是说策略例程应该在一个专门的内核线程上执行(我们不想处罚毫不相关的用户进程)。而且,这样的驱动程序也不能支持可以一次处理多个 I/O 数据传送的现代磁盘控制器。

因此,很多块设备驱动程序都采用如下策略:

- 策略例程处理队列中的第一个请求并设置块设备控制器,以便在数据传送完成时可以产生一个中断。然后策略例程就终止。
- 当磁盘控制器产生中断时,中断控制器重新调用策略例程(通常是直接的,有时也通过激活一个工作队列)。策略例程要么为当前请求再启动一次数据传送,要么当

请求的所有数据块已经传送完成时,把该请求从调度队列中删除然后开始处理下一个请求。

请求是由几个 bio 结构组成的,而每个 bio 结构又是由几个段组成的。基本上,块设备驱动程序以以下两种方式使用 DMA:

- 驱动程序建立不同的 DMA 传送方式,为请求的每个 bio 结构中的每个段进行服务。
- 驱动程序建立一种单独的分散 - 聚集 DMA 传送方式,为请求的所有 bio 中的所有段服务。

最后,设备驱动程序策略例程的设计依赖块控制器的特性。每个物理块设备都有不同于其他物理块设备的固有特性(例如,软盘驱动程序把磁道上的块分组为磁道,一次单独的 I/O 操作传送整个磁道),因此对设备驱动程序怎样为每个请求进行服务而做一般假设并没有多大意义。

在我们的例子中, `foo_strategy()` 策略例程应该执行以下操作:

1. 通过调用 I/O 调度程序的辅助函数 `elv_next_request()` 从调度队列中获取当前的请求。如果调度队列为空,就结束这个策略例程:

```
req = elv_next_request(q);
if (!req) return;
```

2. 执行 `blk_fs_request` 宏检查是否设置了请求的 `REQ_CMD` 标志,也即,请求是否包含一个标准的读或写操作:

```
if (!blk_fs_request(req))
    goto handle_special_request;
```

3. 如果块设备控制器支持分散 - 聚集 DMA,那么对磁盘控制器进行编程,以便为整个请求执行数据传送并在传送完成时产生一个中断。`blk_rq_map_sg()` 辅助函数返回一个可以立即被用来启动数据传送的分散 - 聚集链表。

4. 否则,设备驱动程序必须一段一段地传送数据。在这种情形下,策略例程执行 `rq_for_each_bio` 和 `bio_for_each_segment` 两个宏,分别遍历 bio 链表和每个 bio 中的段链表。

```
rq_for_each_bio(bio, rq)
    bio_for_each_segment(bvec, bio, i) {
        /* transfer the i-th segment bvec */
        local_irq_save(flags);
        addr = kmap_atomic(bvec->bv_page, KM_BIO_SRC_IRQ);
        foo_start_dma_transfer(addr+bvec->bv_offset, bvec->bv_len);
        kunmap_atomic(bvec->bv_page, KM_BIO_SRC_IRQ);
        local_irq_restore(flags);
    }
```

如果要传送的数据在高端内存中，那么 `kmap_atomic()` 和 `kunmap_atomic()` 两个函数就是必需的。`foo_start_dma_transfer()` 函数对硬件设备进行编程，以便启动 DMA 数据传送并在 I/O 操作完成时产生一个中断。

5. 返回。

中断处理程序

块设备驱动程序的中断处理程序是在 DMA 数据传送结束时被激活的。它检查是否已经传送完请求的所有数据块。如果是，中断处理程序就调用策略例程处理调度队列中的下一个请求。否则，中断处理程序更新请求描述符的相应字段并调用策略例程处理还没有完成的数据传送。

我们的设备驱动程序 `foo` 的中断处理程序的一个典型片段如下：

```
irqreturn_t foo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct foo_dev_t *p = (struct foo_dev_t *) dev_id;
    struct request_queue *rq = p->gd->rq;
    [...]
    if (!end_that_request_first(rq, uptodate, nr_sectors)) {
        blkdev_dequeue_request(rq);
        end_that_request_last(rq);
    }
    rq->request_fn(rq);
    [...]
    return IRQ_HANDLED;
}
```

`end_that_request_first()` 和 `end_that_request_last()` 两个函数共同承担结束一个请求的任务。

`end_that_request_first()` 函数接收的参数为一个请求描述符、一个指示 DMA 数据传送成功完成的标志以及 DMA 所传送的扇区数（`end_that_request_chunk()` 函数类似，只不过该函数接收的是传送的字节数而不是扇区数）。本质上，它扫描请求中的 bio 结构以及每个 bio 中的段，然后采用如下方式更新请求描述符的字段值：

- 修改 bio 字段使其指向请求中的第一个未完成的 bio 结构。
- 修改未完成 bio 结构的 `bi_idx` 字段使其指向第一个未完成的段。
- 修改未完成段的 `bv_offset` 和 `bv_len` 两个字段使其指定仍需传送的数据。

该函数也在每个已经完成数据传送的 bio 结构上调用 `bio_endio()` 函数。

如果已经传送完请求中的所有数据块，那么 `end_that_request_first()` 返回 0，否则

返回 1。如果返回值是 1，则中断处理程序重新调用策略例程，继续处理该请求。否则，中断处理程序把请求从请求队列中删除（主要由 `blkdev_dequeue_request()` 完成），然后调用 `end_that_request_last()` 辅助函数，并再次调用策略例程处理调度队列中的下一个请求。

`end_that_request_last()` 函数的功能是更新一些磁盘使用统计数，把请求描述符从 I/O 调度程序 `rq->elevator` 的调度队列中删除，唤醒等待请求描述符完成的任一睡眠进程，并释放删除的那个描述符。

打开块设备文件

通过描述打开一个块设备文件时 VFS 所执行的操作，我们将总结本章的内容。

每当一个文件系统被映射到磁盘或分区上时，每当激活一个交换分区时，每当用户态进程向块设备文件发出一个 `open()` 系统调用时，内核都会打开一个块设备文件。在所有情况下，内核本质上执行相同的操作：寻找块设备描述符（如果块设备没有在使用，那么就分配一个新的描述符），为即将开始的数据传送设置文件操作方法。

我们已经在第十三章的“设备文件的 VFS 处理”一节描述了当一个设备文件被打开时，`dentry_open()` 函数是如何定制文件对象的方法。它的 `f_op` 字段设置为表 `def_blk_fops` 的地址，该表的内容如表 14-10 所示。

表 14-10：缺省的块设备文件操作（表 `def_blk_fops`）

方法	用于块设备文件的函数
<code>open</code>	<code>blkdev_open()</code>
<code>release</code>	<code>blkdev_close()</code>
<code>llseek</code>	<code>block_llseek()</code>
<code>read</code>	<code>generic_file_read()</code>
<code>write</code>	<code>blkdev_file_write()</code>
<code>aio_read</code>	<code>generic_file_aio_read()</code>
<code>aio_write</code>	<code>blkdev_file_aio_write()</code>
<code>mmap</code>	<code>generic_file_mmap()</code>
<code>fsync</code>	<code>block_fsync()</code>
<code>ioctl</code>	<code>block_ioctl()</code>
<code>compat_ioctl</code>	<code>compat_blkdev_ioctl()</code>
<code>readv</code>	<code>generic_file_readv()</code>

表 14-10: 缺省的块设备文件操作 (表 def_blk_fops) (续)

方法	用于块设备文件的函数
writev	generic_file_write_nolock()
sendfile	generic_file_sendfile()

我们仅仅考虑 open 方法, 它由 dentry_open() 函数调用。blkdev_open() 接收 inode 和 filp 作为其参数, 它们分别存放了索引节点和文件对象的地址; 该函数本质上执行下列操作:

1. 执行 bd_acquire(inode) 从而获得块设备描述符 bdev 的地址。该函数接收索引节点对象的地址并执行下列主要步骤:
 - a. 检查索引节点对象的 inode->i_bdev 字段是否不为 NULL; 如果是, 表明块设备文件已经打开了, 该字段存放了相应块描述符的地址。在这种情况下, 增加与块设备相关联的 bdev 特殊文件系统的 inode->i_bdev->bd_inode 索引节点的引用计数器的值, 并返回描述符 inode->i_bdev 的地址。
 - b. 块设备文件没有被打开的情况。根据块设备文件的主设备号和次设备号 (参见本章前面的“块设备”一节), 执行 bdget(inode->i_rdev) 获取块设备描述符的地址。如果描述符不存在, bdget() 就分配一个; 但是, 要注意的是描述符可能已经存在, 例如其他块设备文件已经访问了该块设备。
 - c. 将块设备描述符的地址存放在 inode->i_bdev 中, 以便加速将来对相同块设备文件的打开操作。
 - d. 将 inode->i_mapping 字段设置为 bdev 索引节点中相应字段的值。该字段指向地址空间对象, 我们将在第十五章的“address_space 对象”一节中介绍这个对象。
 - e. 把索引节点插入到由 bdev->bd_inodes 确立的块设备描述符的已打开索引节点链表中。
 - f. 返回描述符 bdev 的地址。
2. 将 filp->i_mapping 字段设置为 inode->i_mapping 的值 (参见前面的第 1d 步)。
3. 获取与这个块设备相关的 gendisk 描述符的地址:

```
disk = get_gendisk(bdev->bd_dev, &part);
```

如果被打开的块设备是一个分区, 则返回的索引值存放在本地变量 part 中; 否则, part 为 0。get_gendisk() 函数在 kobject 映射域 bdev_map 上简单地调用

kobj_lookup()来传递设备的主设备号和次设备号（参见本章前面的“注册和初始化设备驱动程序”一节）。

4. 如果 `bdev->bd_openers` 的值不等于 0，表明块设备已经被打开了。检查 `bdev->bd_contains` 字段：
 - a. 如果值等于 `bdev`，那么块设备是一个整盘：调用块设备方法 `bdev->bd_disk->fops->open`（如果定义了），然后检查 `bdev->bd_invalidated` 字段的值，如果需要，调用 `rescan_partitions()` 函数（参见后面的第 6a 步和第 6c 步）。
 - b. 如果不等于 `bdev`，那么块设备是一个分区：增加 `bdev->bd_contains->bd_part_count` 计数器的值。然后跳到第 8 步。
5. 这里块设备是第一次被访问。初始化 `bdev->bd_disk` 为 `gendisk` 描述符的地址 `disk`。
6. 如果块设备是一个整盘（`part` 等于 0），则执行下列子步骤：
 - a. 如果定义了 `disk->fops->open` 块设备方法，就执行它：该方法是由块设备驱动程序定义的定制函数，它执行任何特定的最后一分钟初始化。
 - b. 从 `disk->queue` 请求队列的 `hardsect_size` 字段中获取扇区大小（字节数），使用该值适当地设置 `bdev->bd_block_size` 和 `bdev->bd_inode->i_blkbits` 两个字段。同时用从 `disk->capacity` 中计算来的磁盘大小设置 `bdev->bd_inode->i_size` 字段。
 - c. 如果设置了 `bdev->bd_invalidated` 标志，那么调用 `rescan_partitions()` 扫描分区表并更新分区描述符。该标志是由 `check_disk_change` 块设备方法设置的，仅适用于可移动设备。
7. 否则如果块设备是一个分区，则执行下列子步骤：
 - a. 再次调用 `bdget()` —— 这次是传递 `disk->first_minor` 次设备号 —— 获取整盘的块描述符地址 `whole`。
 - b. 对整盘的块设备描述符重复第 3 步 ~ 第 6 步，如果需要则初始化该描述符。
 - c. 将 `bdev->bd_contains` 设置为整盘描述符的地址。
 - d. 增加 `whole->bd_part_count` 的值从而说明磁盘分区上新的打开操作。
 - e. 用 `disk->part[part-1]` 中的值设置 `bdev->bd_part`，它是分区描述符 `hd_struct` 的地址。同样，执行 `kobject_get(&bdev->bd_part->kobj)` 增加分区引用计数器的值。
 - f. 与第 6b 步中的一样，设置索引节点中表示分区大小和扇区大小的字段。
8. 增加 `bdev->bd_openers` 计数器的值。

9. 如果块设备文件以独占方式被打开（设置了 `filp->f_flags` 中的 `O_EXCL` 标志），那么调用 `bd_claim( bdev, filp)` 设置块设备的持有者（参见本章前面的“块设备”一节）。万一出错——块设备已经有一个拥有者——释放该块设备描述符并返回一个错误码 `-EBUSY`。
10. 返回 0（成功）终止。

`blkdev_open()` 函数一旦终止，`open()` 系统调用如往常一样继续进行。对已打开的文件上将来发出的每个系统调用都将触发一个缺省的块设备文件操作。正如我们将在第十六章中看到的，采用向通用块层提交请求的方式进行块设备的每次数据传送都是高效率的。



页高速缓存

正如在第十二章的“通用文件模型”一节中提到的那样，磁盘高速缓存是一种软件机制，它允许系统把通常存放在磁盘上的一些数据保留在 RAM 中，以便对那些数据的进一步访问不用再访问磁盘而能尽快得到满足。

因为对同一磁盘数据的反复访问频繁发生，所以磁盘高速缓存对系统性能至关重要。与磁盘交互的用户态进程有权反复请求读或写同一磁盘数据。此外，不同的进程可能也需要在不同的时间访问相同的磁盘数据。例如，你可以使用 `cp` 命令拷贝一个文本文件，然后调用你喜欢的编辑器修改它。为了满足你的请求，命令 shell 将创建两个不同的进程，它们在不同的时间访问同一个文件。

我们曾在第十二章提到过其他的磁盘高速缓存：目录项高速缓存和索引节点高速缓存，前者存放的是描述文件系统路径名的目录项对象，而后者存放的是描述磁盘索引节点的索引节点对象。不过要注意，目录项对象和索引结节点对象不只是存放一些磁盘块内容的缓冲区；由此而知，目录项高速缓存和索引节点高速缓存就像磁盘高速缓存那样相当独特。

本章介绍页高速缓存——一种对完整的数据页进行操作的磁盘高速缓存。我们在第一节介绍页高速缓存。然后，我们在“把块存放在页高速缓存中”一节中讨论如何使用页高速缓存快速检索单独的数据块（例如超级块和索引节点），这是提高虚拟文件系统和磁盘文件系统速度的关键特征。接下来，我们在“把脏页写入磁盘”一节描述如何把页高速缓存中的脏页写回到磁盘中。最后，我们在“`sync()`、`fsync()`和`fdatasync()`系统调用”一节中介绍一些系统调用，让用户刷新页高速缓存的内容来更新磁盘内容。

页高速缓存

页高速缓存 (*page cache*) 是 Linux 内核所使用的主要磁盘高速缓存。在绝大多数情况下, 内核在读写磁盘时都引用页高速缓存。新页被追加到页高速缓存以满足用户态进程的读请求。如果页不在高速缓存中, 新页就被加到高速缓存中, 然后用从磁盘读出的数据填充它。如果内存有足够的空闲空间, 就让该页在高速缓存中长期保留, 使其他进程再使用该页时不再访问磁盘。

同样, 在把一页数据写到块设备之前, 内核首先检查对应的页是否已经在高速缓存中; 如果不在, 就要先在其中增加一个新项, 并用要写到磁盘中的数据填充该项。I/O 数据的传送并不是马上开始, 而是要延迟几秒之后才对磁盘进行更新, 从而使进程有机会对要写入磁盘的数据做进一步的修改 (换句话说, 就是内核执行延迟的写操作)。

内核的代码和内核数据结构不必从磁盘读, 也不必写入磁盘 (注 1), 因此, 页高速缓存中的页可能是下面的类型:

- 含有普通文件数据的页。在第十六章我们将描述内核如何处理它们的读、写和内存映射操作。
- 含有目录的页。在第十八章我们将会看到, Linux 采用与普通文件类似的方式操作目录文件。
- 含有直接从块设备文件 (跳过文件系统层) 读出的数据的页。正如我们将在第十六章讨论的那样, 内核处理这种页与处理含有普通文件的页使用相同的函数集合。
- 含有用户态进程数据的页, 但页中的数据已经被交换到磁盘。在第十七章我们将会看到, 内核可能会强行在页高速缓存中保留一些页面, 而这些页面中的数据已经被写到交换区 (可能是普通文件或磁盘分区)。
- 属于特殊文件系统文件的页, 如共享内存的进程间通信 (Interprocess Communication, IPC) 所使用的特殊文件系统 *shm* (参见第十九章)。

正如你所看到的, 页高速缓存中的每个页所包含的数据肯定属于某个文件。这个文件 (或者更准确地说 是文件的索引节点) 就称为页的所有者 (*owner*)。(在第十七章我们会了解到, 含有换出数据的页都属于同一个所有者, 即使它们涉及不同的交换区。)

注 1: 如果要在关机后恢复系统的所有状态 (其实几乎不会出现这种情况), 可以执行“挂起到磁盘”操作 (*hibernation*), 把 RAM 的全部内容保存到交换区, 对此我们不做更多的讨论。

几乎所有的文件读和写操作都依赖于页高速缓存。只有在O_DIRECT标志被置位而进程打开文件的情况下才会出现例外：此时，I/O数据的传送绕过了页高速缓存而使用了进程用户态地址空间的缓冲区（参见第十六章“直接I/O传送”一节）；少数数据库应用软件为了能采用自己的磁盘高速缓存算法而使用了O_DIRECT标志。

内核设计者实现页高速缓存主要为了满足下面两种需要：

- 快速定位含有给定所有者相关数据的特定页。为了尽可能充分发挥页高速缓存的优势，对它应该采用高速的搜索操作。
- 记录在读或写页中的数据时应当如何处理高速缓存中的每个页。例如，从普通文件、块设备文件或交换区读一个数据页必须用不同的实现方式，因此内核必须根据页的所有者选择适当的操作。

页高速缓存中的信息单位显然是一个完整的数据页。在第十八章我们会看到，一个页中包含的磁盘块在物理上不一定是相邻的，所以不能用设备号和块号来识别它，取而代之的是，通过页的所有者和所有者数据中的索引（通常是一个索引节点和在相应文件中的偏移量）来识别页高速缓存中的页。

address_space 对象

页高速缓存的核心数据结构是address_space对象，它是一个嵌入在页所有者的索引节点对象中的数据结构（注2）。高速缓存中的许多页可能属于同一个所有者，从而可能被链接到同一个address_space对象。该对象还在所有者的页和对这些页的操作之间建立起链接关系。

每个页描述符都包括把页链接到页高速缓存的两个字段mapping和index（参见第八章“页描述符”一节）。mapping字段指向拥有页的索引节点的address_space对象，index字段表示在所有者的地址空间中以页大小为单位的偏移量，也就是在所有者的磁盘映像中页中数据的位置。在页高速缓存中查找页时使用这两个字段。

值得庆幸的是，页高速缓存可以包含同一磁盘数据的多个副本。例如，可以用下述方式访问普通文件的同一4KB的数据块：

- 读文件；因此，数据就包含在普通文件的索引节点所拥有的页中。

注2： 页被换出可能会引起缺页异常。在第十七章我们将看到，这些被换出的页拥有不在任何索引节点中的公共address_space对象。

- 从文件所在的设备文件（磁盘分区）读取块，因此，数据就包含在块设备文件的主索引节点所拥有的页中。

因此，两个不同 `address_space` 对象所引用的两个不同的页中出现了相同的磁盘数据。

`address_space` 对象包含如表 15-1 所示的字段。

表 15-1: `address_space` 对象的字段

类型	字段	说明
<code>struct inode *</code>	<code>host</code>	指向拥有该对象的索引节点的指针（如果存在）
<code>struct radix_tree_root</code>	<code>page_tree</code>	表示拥有者页的基树（radix tree）的根
<code>spinlock_t</code>	<code>tree_lock</code>	保护基树的自旋锁
<code>unsigned int</code>	<code>i_mmap_writable</code>	地址空间中共享内存映射的个数
<code>struct prio_tree_root</code>	<code>i_mmap</code>	radix 优先搜索树的根（参见第十七章）
<code>struct list_head</code>	<code>i_mmap_nonlinear</code>	地址空间中非线性内存区的链表
<code>spinlock_t</code>	<code>i_mmap_lock</code>	保护 radix 优先搜索树的自旋锁
<code>unsigned int</code>	<code>truncate_count</code>	截断文件时使用的顺序计数器
<code>unsigned long</code>	<code>nrpages</code>	所有者的页总数
<code>unsigned long</code>	<code>writeback_index</code>	最后一次回写操作所作用的页的索引
<code>struct address_space_operations *</code>	<code>a_ops</code>	对所有者页进行操作的方法
<code>unsigned long</code>	<code>flags</code>	错误位和内存分配器的标志
<code>struct backing_dev_info *</code>	<code>backing_dev_info</code>	指向拥有所有者数据的块设备的 <code>backing_dev_info</code> 的指针
<code>spinlock_t</code>	<code>private_lock</code>	通常是管理 <code>private_list</code> 链表时使用的自旋锁
<code>struct list_head</code>	<code>private_list</code>	通常是与索引节点相关的间接块的脏缓冲区的链表
<code>struct address_space *</code>	<code>assoc_mapping</code>	通常是指向间接块所在块设备的 <code>address_space</code> 对象的指针

如果页高速缓存中页的所有者是一个文件，`address_space` 对象就嵌入在 VFS 索引节点对象的 `i_data` 字段中。索引节点的 `i_mapping` 字段总是指向索引节点的数据页所有者的 `address_space` 对象。`address_space` 对象的 `host` 字段指向其所有者的索引节点对象。

因此，如果页属于一个文件（存放在Ext3文件系统中），那么页的所有者就是文件的索引节点，而且相应的 `address_space` 对象存放在 VFS 索引节点对象的 `i_data` 字段中。索引节点的 `i_mapping` 字段指向同一个索引节点的 `i_data` 字段，而 `address_space` 对象的 `host` 字段也指向这个索引节点。

不过，有些时候情况会更复杂。如果页中包含的数据来自块设备文件，即页含有存放着块设备的“原始”数据，那么就把 `address_space` 对象嵌入到与该块设备相关的特殊文件系统 `bdev` 中文件的“主”索引节点中（块设备描述符的 `bd_inode` 字段引用这个索引节点，参见第十四章“块设备”一节）。因此，块设备文件对应索引节点的 `i_mapping` 字段指向主索引节点中的 `address_space` 对象。相应地，`address_space` 对象的 `host` 字段指向主索引节点。这样，从块设备读取数据的所有页具有相同的 `address_space` 对象，即使这些数据位于不同的块设备文件。

`i_mmap`、`i_mmap_writable`、`i_mmap_nonlinear` 和 `i_mmap_lock` 字段涉及内存映射和反映射，我们将在第十六、十七章讨论这些主题。

`backing_dev_info` 字段指向 `backing_dev_info` 描述符，后者是对所有者的数据所在块设备进行有关描述的数据结构。正如在第十四章“请求队列描述符”一节所描述的，`backing_dev_info` 结构通常嵌入在块设备的请求队列描述符中。

`private_list` 字段是普通链表的首部，文件系统在实现其特定功能时可以随意使用。例如，Ext2 文件系统利用这个链表收集与索引节点相关的“间接”块的脏缓冲区（参见第十八章“数据块寻址”一节）。当刷新操作把索引节点强行写入磁盘时，内核也同时刷新该链表中的所有缓冲区。此外，Ext2 文件系统在 `assoc_mapping` 字段中存放指向间接块所在块设备的 `address_space` 对象，并使用 `assoc_mapping->private_lock` 自旋锁保护多处理器系统中的间接块链表。

`address_space` 对象的关键字段是 `a_ops`，它指向一个类型为 `address_space_operations` 的表，表中定义了对所有者的页进行处理的各种方法。这些方法如表 15-2 所示。

表 15-2: `address_space` 对象的方法

方法	说明
<code>writepage</code>	写操作（从页写到所有者的磁盘映像）
<code>readpage</code>	读操作（从所有者的磁盘映像读到页）
<code>sync_page</code>	如果对所有者的页进行的操作已准备好，则立刻开始 I/O 数据的传输
<code>writepages</code>	把指定数量的所有者脏页写回磁盘
<code>set_page_dirty</code>	把所有者的页设置为脏页

表 15-2: address_space 对象的方法 (续)

方法	说明
readpages	从磁盘中读所有者页的链表
prepare_write	为写操作做准备 (由磁盘文件系统使用)
commit_write	完成写操作 (由磁盘文件系统使用)
bmap	从文件块索引中获取逻辑块号
invalidatepage	使所有者的页无效 (截断文件时使用)
releasepage	由日志文件系统使用以准备释放页
direct_IO	所有者页的直接 I/O 传输 (绕过页高速缓存)

最重要的方法是 readpage、writepage、prepare_write 和 commit_write。我们将在第十六章对它们进行讨论。在绝大多数情况下, 这些方法把所有者的索引节点对象和访问物理设备的低级驱动程序联系起来。例如, 为普通文件的索引节点实现 readpage 方法的函数知道如何确定文件页的对应块在物理磁盘设备上的位置。不过, 我们不必在本章进一步讨论 address_space 的方法。

基树

Linux 支持大到几个 TB 的文件。访问大文件时, 页高速缓存中可能充满太多的文件页, 以至于顺序扫描这些页要消耗大量的时间。为了实现页高速缓存的高效查找, Linux 2.6 采用了大量的搜索树, 其中每个 address_space 对象对应一棵搜索树。

address_space 对象的 page_tree 字段是基树 (radix tree) 的根, 它包含指向所有者的页描述符的指针。给定的页索引表示页在所有者磁盘映像中的位置, 内核能够通过快速搜索操作来确定所需要的页是否在页高速缓存中。当查找所需要的页时, 内核把页索引转换为基树中的路径, 并快速找到页描述符所 (或应当) 在的位置。如果找到, 内核可以从基树获得页描述符, 而且还可以很快确定所找到的页是否是脏页 (也就是应当被刷新到磁盘的页), 以及其数据的 I/O 传送是否正在进行。

基树的每个节点可以有多到 64 个指针指向其他节点或页描述符。底层节点存放指向页描述符的指针 (叶子节点), 而上层的节点存放指向其他节点 (孩子节点) 的指针。每个节点由 radix_tree_node 数据结构表示, 它包括三个字段: slots 是包括 64 个指针的数组, count 是记录节点中非空指针数量的计数器, tags 是二维的标志数组, 在本章稍后 “基树的标记” 一节将对其进行讨论。树根由 radix_tree_root 数据结构表示, 它有三个字段: height 表示树的当前深度 (不包括叶子节点的层数), gfp_mask 指定为新节点请求内存时所用的标志, rnode 指向与树中第一层节点相应的数据结构 radix_tree_node (如果有的话)。

我们来看一个简单的例子。如果树中没有索引大于63，那么树的深度就等于1，因为可能存在的64个叶子可以都存放在第一层的节点中 [如图 15-1 (a) 所示]。不过，如果与索引131相应的新页的描述符肯定存放在页高速缓存中，那么树的深度就增加为2，这样基树就可以查找多达 4 095 个索引 [如图 15-1 (b) 所示]。

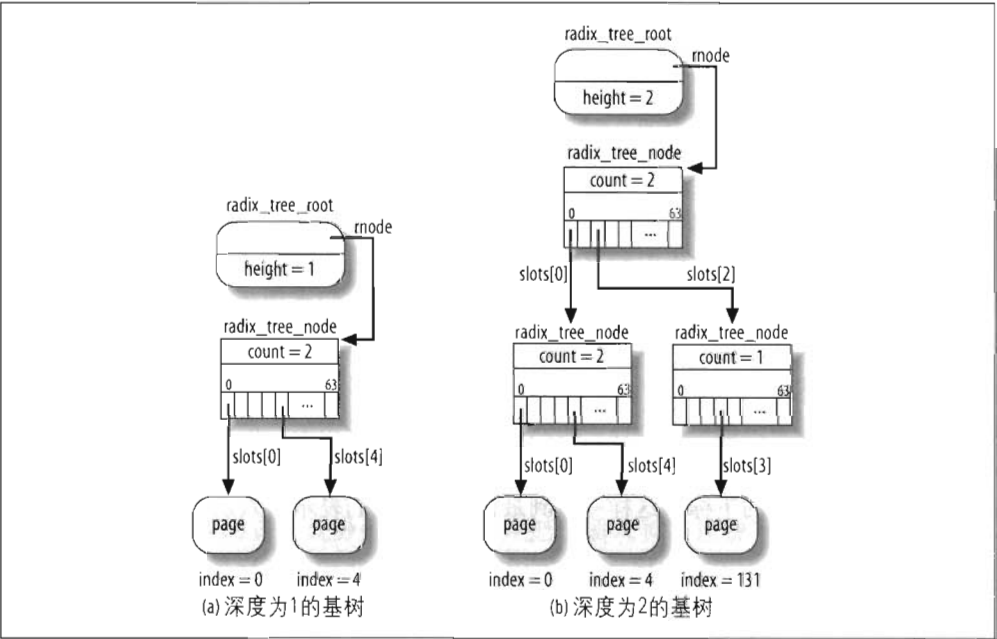


图 15-1：基树的两个例子

表 15-3 显示了页索引的最大值和基于 32 位体系结构的基树中与每个给定深度相应的文件的最大长度。在这里，基树的最大深度是 6，当然系统中的页高速缓存不大可能使用那么大的基树。因为页索引存放在 32 位变量中，当树的深度为 6 时，最高层的节点最多可以有 4 个孩子节点。

表 15-3：每个基树深度对应的最大索引值和文件的最大长度

基树深度	最大索引值	文件的最大长度
0	0	0 字节
1	$2^6 - 1 = 63$	256 KB
2	$2^{12} - 1 = 4\,095$	16 MB
3	$2^{18} - 1 = 262\,143$	1 GB
4	$2^{24} - 1 = 16\,777\,215$	64 GB

表 15-3：每个基树深度对应的最大索引值和文件的最大长度（续）

基树深度	最大索引值	文件的最大长度
5	$2^{30}-1 = 1\,073\,741\,823$	4 TB
6	$2^{32}-1 = 4\,294\,967\,295$	16 TB

回顾一下分页系统是如何利用页表实现线性地址到物理地址转换的，从而理解如何实现页查找。正如第二章“常规分页”一节所讨论的，线性地址最高 20 位分成两个 10 位的字段：第一个字段是页目录中的偏移量，而第二个字段是某个页目录项所指向的页表中的偏移量。

基树中使用类似的方法。页索引相当于线性地址，不过页索引中要考虑的字段的数量依赖于基树的深度。如果基树的深度为 1，就只能表示从 0~63 范围的索引，因此页索引的低 6 位被解释为 slots 数组的下标，每个下标对应第一层的一个节点。如果基树的深度为 2，就可以表示从 0~4095 范围的索引，页索引的低 12 位分成两个 6 位的字段，高位的字段用于表示第一层节点数组的下标，而低位的字段用于表示第二层节点数组的下标。依此类推，如果深度等于 6，页索引的最高两位表示第一层节点数组的下标，接下来的 6 位表示第二层节点数组的下标，这样一直到最低 6 位，它们表示第六层节点数组的下标。

如果基树的最大索引小于应该增加的页的索引，那么内核相应地增加树的深度，基树的中间节点依赖于页索引的值（例子参见图 15-1）。

页高速缓存的处理函数

对页高速缓存操作的基本高级函数有查找、增加和删除页。在以上函数的基础上还有另一个函数确保高速缓存包含指定页的最新版本。

查找页

函数 `find_get_page()` 接收的参数为指向 `address_space` 对象的指针和偏移量。它获取地址空间的自旋锁，并调用 `radix_tree_lookup()` 函数搜索拥有指定偏移量的基树的叶子节点。该函数根据偏移量值中的位依次从树根开始并向下搜索，如上节所述。如果遇到空指针，函数返回 `NULL`；否则，返回叶子节点的地址，也就是所需要的页描述符指针。如果找到了所需要的页，`find_get_page()` 函数就增加该页的使用计数器，释放自旋锁，并返回该页的地址；否则，函数就释放自旋锁并返回 `NULL`。

函数 `find_get_pages()` 与 `find_get_page()` 类似，但它实现在高速缓存中查找一组具有相邻索引的页。它接收的参数是：指向 `address_space` 对象的指针、地址空间中相对

于搜索起始位置的偏移量、所检索到页的最大数量、指向由该函数赋值的页描述符数组的指针。`find_get_pages()`依赖`radix_tree_gang_lookup()`函数实现查找操作，`radix_tree_gang_lookup()`函数为指针数组赋值并返回找到的页数。尽管由于一些页可能不在页高速缓存中而会出现空缺的页索引，但所返回的页还是递增的索引值。

还有另外几个函数实现页高速缓存上的查找操作。例如，`find_lock_page()`函数与`find_get_page()`类似，但它增加返回页的使用计数器，并调用`lock_page()`设置`PG_locked`标志，从而当函数返回时调用者能够以互斥的方式访问返回的页。随后，如果页已经被加锁，`lock_page()`函数就阻塞当前进程。最后，它在`PG_locked`位置位时调用`__wait_on_bit_lock()`函数。后面的函数把当前进程置为`TASK_UNINTERRUPTIBLE`状态，把进程描述符存入等待队列，执行`address_space`对象的`sync_page`方法以取消文件所在块设备的请求队列，最后调用`schedule()`函数来挂起进程，直到把`PG_locked`标志清0。内核使用`unlock_page()`函数对页进行解锁，并唤醒在等待队列上睡眠的进程。

函数`find_trylock_page()`与`find_lock_page()`类似，仅有一点不同，就是`find_trylock_page()`从不阻塞：如果被请求的页已经上锁，函数就返回错误码。最后要说明的是，函数`find_or_create_page()`执行`find_lock_page()`；不过，如果找不到所请求的页，就分配一个新页并把它插入页高速缓存。

增加页

函数`add_to_page_cache()`把一个新页的描述符插入到页高速缓存。它接收的参数有：页描述符的地址`page`、`address_space`对象的地址`mapping`、表示在地址空间内的页索引的值`offset`和为基树分配新节点时所使用的内存分配标志`gfp_mask`。函数执行以下操作：

1. 调用`radix_tree_preload()`函数，它禁用内核抢占，并把一些空的`radix_tree_node`结构赋给每CPU变量`radix_tree_preloads`。`radix_tree_node`结构的分配由slab分配器高速缓存`radix_tree_node_cachep`来完成。如果`radix_tree_preload()`预分配`radix_tree_node`结构不成功，函数`add_to_page_cache()`就终止并返回错误码`-ENOMEM`。否则，如果`radix_tree_preload()`成功地完成预分配，`add_to_page_cache()`函数肯定不会因为缺乏空闲内存或因为文件的大小达到了64GB而无法完成新页描述符的插入。
2. 获取`mapping->tree_lock`自旋锁——注意，`radix_tree_preload()`函数已经禁用了内核抢占。
3. 调用`radix_tree_insert()`在树中插入新节点，该函数执行下述操作：

- a. 调用 `radix_tree_maxindex()` 获得最大索引, 该索引可能被插入具有当前深度的基树; 如果新页的索引不能用当前深度表示, 就调用 `radix_tree_extend()` 通过增加适当数量的节点来增加树的深度 (例如, 对图 15-1 (a) 所示的基树, `radix_tree_extend()` 在它的顶端增加一个节点)。分配新节点是通过执行 `radix_tree_node_alloc()` 函数实现的, 该函数试图从 slab 分配器高速缓存获得 `radix_tree_node` 结构, 如果分配失败, 就从存放在 `radix_tree_preloads` 中的预分配的结构池中获得 `radix_tree_node` 结构。
 - b. 根据页索引的偏移量, 从根节点 (`mapping->page_tree`) 开始遍历树, 直到叶子节点, 如上一节所述。如果需要, 就调用 `radix_tree_node_alloc()` 分配新的中间节点。
 - c. 把页描述符地址存放在对基树所遍历的最后节点的适当位置, 并返回 0。
4. 增加页描述符的使用计数器 `page->_count`。
 5. 由于页是新的, 所以其内容无效: 函数设置页框的 `PG_locked` 标志, 以阻止其他的内核路径并发访问该页。
 6. 用 `mapping` 和 `offset` 参数初始化 `page->mapping` 和 `page->index`。
 7. 递增在地址空间所缓存页的计数器 (`mapping->numpages`)。
 8. 释放地址空间的自旋锁。
 9. 调用 `radix_tree_preload_end()` 重新启用内核抢占。
 10. 返回 0 (成功)。

删除页

函数 `remove_from_page_cache()` 通过下述步骤从页高速缓存中删除页描述符:

1. 获取自旋锁 `page->mapping->tree_lock` 并关中断。
2. 调用 `radix_tree_delete()` 函数从树中删除节点。该函数接收树根的地址 (`page->mapping->page_tree`) 和要删除的页索引作为参数, 并执行下述步骤:
 - a. 如上节所述, 根据页索引从根节点开始遍历树, 直到到达叶子节点。遍历时, 建立 `radix_tree_path` 结构的数组, 描述从根到与要删除的页相应的叶子节点的路径构成。
 - b. 从最后一个节点 (包含指向页描述符的指针) 开始, 对路径数组中的节点开始循环操作。对每个节点, 把指向下一个节点 (或页描述符) 位置数组的元素置为 `NULL`, 并递减 `count` 字段。如果 `count` 变为 0, 就从树中删除节点并把

`radix_tree_node`结构释放给slab分配器高速缓存。然后继续循环处理路径数组中的节点。否则，如果 `count` 不等于0，继续执行下一步。

- c. 返回已经从树中删除的页描述符指针。
3. 把 `page->mapping` 字段置为 `NULL`。
4. 把所缓存页的 `page->mapping->numpages` 计数器的值减1。
5. 释放自旋锁 `page->mapping->tree_lock`，打开中断，函数终止。

更新页

函数 `read_cache_page()` 确保高速缓存中包括最新版本的指定页。它的参数是指向 `address_space` 对象的指针 `mapping`、表示所请求页的偏移量的值 `index`、指向从磁盘读页数据的函数的指针 `filler`（通常是实现地址空间 `readpage` 方法的函数）以及传递给 `filler` 函数的指针 `data`（通常为 `NULL`），下面是对这个函数的简单说明：

1. 调用函数 `find_get_page()` 检查页是否已经在页高速缓存中。
2. 如果页不在页高速缓存中，则执行下述子步骤：
 - a. 调用 `alloc_pages()` 分配一个新页框。
 - b. 调用 `add_to_page_cache()` 在页高速缓存中插入相应的页描述符。
 - c. 调用 `lru_cache_add()` 把页插入该管理区的非活动 LRU 链表中 [参见第十七章“最近最少使用的链表 (LRU)”一节]。
3. 此时，所请求的页已经在页高速缓存中了。调用 `mark_page_accessed()` 函数记录页已经被访问过的事实 [参见第十七章“最近最少使用 (LRU) 链表”一节]。
4. 如果页不是最新的 (`PG_uptodate` 标志为0)，就调用 `filler` 函数从磁盘读该页。
5. 返回页描述符的地址。

基树的标记

前面我们曾强调，页高速缓存不仅允许内核快速获得含有块设备中指定数据的页，还允许内核从高速缓存中快速获得给定状态的页。

例如，我们假设内核必须从高速缓存获得属于指定所有者的所有页和脏页（即其内容还没有写回磁盘）。存放在页描述符中的 `PG_dirty` 标志表示页是否是脏的，但是，如果绝大多数页都不是脏页，遍历整个基树以顺序访问所有叶子节点（页描述符）的操作就太慢了。

相反,为了能快速搜索脏页,基树中的每个中间节点都包含一个针对每个孩子节点(或叶子节点)的脏标记,当有且只有至少有一个孩子节点的脏标记被置位时这个标记被设置。最底层节点的脏标记通常是页描述符的 PG_dirty 标志的副本。通过这种方式,当内核遍历基树搜索脏页时,就可以跳过脏标记为0的中间结点的所有子树:中间结点的脏标记为0说明其子树中的所有页描述符都不是脏的。

同样的想法应用到了 PG_writeback 标志,该标志表示页正在被写回磁盘。这样,为基树的每个结点引入两个页描述符的标志: PG_dirty 和 PG_writeback (参见第八章“页描述符”一节)。每个结点的 tags 字段中有两个64位的数组来存放这两个标志。tags[0] (PAGECACHE_TAG_DIRTY) 数组是脏标记,而 tags[1] (PAGECACHE_TAG_WRITEBACK) 数组是写回标记。

设置页高速缓存中页的 PG_dirty 或 PG_writeback 标志时调用函数 radix_tree_tag_set(), 它作用于三个参数:基树的根、页的索引以及要设置的标记的类型 (PAGECACHE_TAG_DIRTY 或 PAGECACHE_TAG_WRITEBACK)。函数从树根开始并向下搜索到与指定索引对应的叶子结点,对于从根通往叶子路径上的每一个节点,函数利用指向路径中下一个结点的指针设置标记。然后,函数返回页描述符的地址。结果是,从根结点到叶子结点的路径中的所有结点都以适当的方式被加上了标记。

清除页高速缓存中页的 PG_dirty 或 PG_writeback 标志时调用函数 radix_tree_tag_clear(), 它的参数与函数 radix_tree_tag_set() 的参数相同。函数从树根开始并向下到叶子结点,建立描述路径的 radix_tree_path 结构的数组。然后,函数从叶子结点到根结点向后进行操作:清除底层结点的标记,然后检查是否结点数数组中所有标记都被清0,如果是,函数把上层父结点的相应标记清0,并如此继续上述操作。最后,函数返回页描述符的地址。

从基树删除页描述符时,必须更新从根结点到叶子结点的路径中结点的相应标记。函数 radix_tree_delete() 可以正确地完成这个工作(尽管我们在上一节没有提到这一点)。而函数 radix_tree_insert() 不更新标记,因为插入基树的所有页描述符的 PG_dirty 和 PG_writeback 标志都被认为是清零的。如果需要,内核可以随后调用函数 radix_tree_tag_set()。

函数 radix_tree_tagged() 利用树的所有结点的标志数组来测试基树是否至少包括一个指定状态的页。函数通过执行下面的代码轻松地完成这一任务 (root 是指向基树的 radix_tree_root 结构的指针, tag 是要测试的标记):

```
for (idx = 0; idx < 2; idx++) {
    if (root->rnode->tags[tag][idx])
        return 1;
}
return 0;
```

因为可能假设基树所有结点的标记都正确地更新过，所以 `radix_tree_tagged()` 函数只需要检查第一层的标记。使用该函数的一个例子是：确定一个包含脏页的索引节点是否要写回磁盘。注意，函数在每次循环时要测试在无符号长整型的32个标志中，是否有被设置的标志。

函数 `find_get_pages_tag()` 和 `find_get_pages()` 类似，只有一点不同，就是前者返回的只是那些用 `tag` 参数标记的页。正如我们将来在“把脏页写入磁盘”一节所见的，该函数对快速找到一个索引节点的所有脏页是非常关键的。

把块存放在页高速缓存中

我们在第十四章“块设备的处理”一节已经看到，VFS（映射层）和各种文件系统以叫做“块”的逻辑单位组织磁盘数据。

在Linux内核的旧版本中，主要有两种不同的磁盘高速缓存：页高速缓存和缓冲区高速缓存，前者用来存放访问磁盘文件内容时生成的磁盘数据页，后者把通过VFS（管理磁盘文件系统）访问的块的内容保留在内存中。

从2.4.10的稳定版本开始，缓冲区高速缓存其实就不存在了。事实上，由于效率的原因，不再单独分配块缓冲区；相反，把它们存放在叫做“缓冲区页”的专门页中，而缓冲区页保存在页高速缓存中。

缓冲区页在形式上就是与称做“缓冲区首部”的附加描述符相关的数据页，其主要目的是快速确定页中的一个块在磁盘中的地址。实际上，页高速缓存内的页中的一大块数据在磁盘上的地址不一定是相邻的。

块缓冲区和缓冲区首部

每个块缓冲区都有 `buffer_head` 类型的缓冲区首部描述符。该描述符包含内核必须了解的、有关如何处理块的所有信息。因此，在对所有块操作之前，内核检查缓冲区首部。缓冲区首部的字段在表 15-4 中列出。

表 15-4：缓冲区首部的字段

类型	字段	说明
unsigned long	b_state	缓冲区状态标志
struct buffer_head *	b_this_page	指向缓冲区页的链表中的下一个元素的指针

表 15-4: 缓冲区首部的字段 (续)

类型	字段	说明
struct page *	b_page	指向拥有该块的缓冲区页的描述符的指针
atomic_t	b_count	块使用计数器
u32	b_size	块大小
sector_t	b_blocknr	与块设备相关的块号 (逻辑块号)
char *	b_data	块在缓冲区页内的位置
struct block_device *	b_bdev	指向块设备描述符的指针
bh_end_io_t *	b_end_io	I/O 完成方法
void *	b_private	指向 I/O 完成方法数据的指针
struct list_head	b_assoc_buffers	为与某个索引节点相关的间接块的链表提供的指针 (参见本章前面“address_space 对象”一节)

缓冲区首部的两个字段编码表示块的磁盘地址: b_bdev 字段表示包含块的块设备 (参见第十四章“块设备”一节), 通常是磁盘或分区; 而 b_blocknr 字段存放逻辑块号, 即块在磁盘或分区中的编号。

b_data 字段表示块缓冲区在缓冲区页中的位置。实际上, 这个位置的编号依赖于页是否在高端内存。如果页在高端内存, 则 b_data 字段存放的是块缓冲区相对于页的起始位置的偏移量, 否则, b_data 存放的是块缓冲区的线性地址。

b_state 字段可以存放几个标志。其中一些标志是通用的, 把它们列在表 15-5 中。每个文件系统还可以定义自己的私有缓冲区首部标志。

表 15-5: 缓冲区首部的通用标志

标志	说明
BH_Uptodate	缓冲区包含有效数据时被置位
BH_Dirty	如果缓冲区脏就置位 (表示缓冲区中的数据必须写回块设备)
BH_Lock	如果缓冲区加锁就置位, 通常发生在缓冲区进行磁盘传输时
BH_Req	如果已经为初始化缓冲区而请求数据传输就置位
BH_Mapped	如果缓冲区被映射到磁盘就置位, 即: 如果相应的缓冲区首部的 b_bdev 和 b_blocknr 是有效的就置位
BH_New	如果相应的块刚被分配而还没有被访问过就置位

表 15-5: 缓冲区首部的通用标志 (续)

标志	说明
BH_Async_Read	如果在异步地读缓冲区就置位
BH_Async_Write	如果在异步地写缓冲区就置位
BH_Delay	如果还没有在磁盘上分配缓冲区就置位
BH_Boundary	如果两个相邻的块在其中一个提交之后不再相邻就置位
BH_Write_EIO	如果写块时出现 I/O 错误就置位
BH_Ordered	如果必须严格地把块写到在它之前提交的块的后面就置位 (用于日志文件系统)
BH_Eopnotsupp	如果块设备的驱动程序不支持所请求的操作就置位

管理缓冲区首部

缓冲区首部有它们自己的 slab 分配器高速缓存, 其描述符 `kmem_cache_s` 存在变量 `bh_cachep` 中。 `alloc_buffer_head()` 和 `free_buffer_head()` 函数分别用于获取和释放缓冲区首部。

缓冲区首部的 `b_count` 字段是相应的块缓冲区的引用计数器。在每次对块缓冲区进行操作之前递增计数器并在操作之后递减它。除了周期性地检查保存在页高速缓存中的块缓冲区之外, 当空闲内存变得很少时也要对它进行检查, 只有引用计数器等于 0 的块缓冲区才可以被回收 (参见第十七章)。

当内核控制路径希望访问块缓冲区时, 应该先递增引用计数器。确定块在页高速缓存中的位置的函数 (`__getblk()`, 参见本章稍后“在页高速缓存中搜索块”一节) 自动完成这项工作, 因此, 高层函数通常不增加块缓冲区的引用计数器。

当内核控制路径停止访问块缓冲区时, 应该调用 `__brelse()` 或 `__bforget()` 递减相应的引用计数器。这两个函数之间的不同是 `__bforget()` 还从间接块链表 (缓冲区首部的 `b_assoc_buffers` 字段, 参见前面的“块缓冲区和缓冲区首部”一节) 中删除块, 并把该缓冲区标记为干净的, 因此强制内核忽略对缓冲区所做的任何修改, 但实际上缓冲区依然必须被写回磁盘。

缓冲区页

只要内核必须单独地访问一个块, 就要涉及存放块缓冲区的缓冲区页, 并检查相应的缓冲区首部。

下面是内核创建缓冲区页的两种普通情况：

- 当读或写的文件页在磁盘块中不相邻时。发生这种情况是因为文件系统为文件分配了非连续的块，或因为文件有“洞”（参见第十八章“文件的洞”一节）。
- 当访问一个单独的磁盘块时（例如，当读超级块或索引节点块时）。

在第一种情况下，把缓冲区页的描述符插入普通文件的基树；保存好缓冲区首部，因为其中存有重要的信息，即存有数据在磁盘上位置的块设备和逻辑块号。在第十六章我们将了解内核如何利用这种类型的缓冲区页。

在第二种情况下，把缓冲区页的描述符插入基树，树根是与块设备相关的特殊 *bdev* 文件系统中索引节点的 *address_space* 对象（参见本章前面“*address_space* 对象”一节）。这种缓冲区页必须满足很强的约束条件，就是所有的块缓冲区涉及的块必须是在块设备上相邻存放的。

这种情况的一个应用实例是：如果虚拟文件系统要读大小为 1024 个字节的索引节点块（包含给定文件的索引节点）。内核并不是只分配一个单独的缓冲区，而是必须分配一个整页，从而存放四个缓冲区；这些缓冲区将存放块设备上相邻的 4 块数据，其中包括所请求的索引节点块。

本章我们将重点讨论第二种类型的缓冲区页，即所谓的块设备缓冲区页（有时简称为块设备页）。

在一个缓冲区页内的所有块缓冲区大小必须相同，因此，在 80x86 体系结构上，根据块的大小，一个缓冲区页可以包括 1~8 个缓冲区。

如果一个页作为缓冲区页使用，那么与它的块缓冲区相关的所有缓冲区首部都被收集在一个单向循环链表中。缓冲区页描述符的 *private* 字段指向页中第一个块的缓冲区首部（注 3）；每个缓冲区首部存放在 *b_this_page* 字段中，该字段是指向链表中下一个缓冲区首部的指针。此外，每个缓冲区首部还把缓冲区页描述符的地址存放在 *b_page* 字段中。图 15-2 显示了一个缓冲区页，其中包含四个块缓冲区和对应的缓冲区首部。

注 3： 由于 *private* 字段包含有效数据，而且页的 *PG_private* 标志被设置，因此，如果页中包含磁盘数据并且设置了 *PG_private* 标志，该页就是一个缓冲区页。注意，尽管如此，其他与块 I/O 子系统无关的内核组件也因为别的用途而使用 *private* 和 *PG_private* 字段。

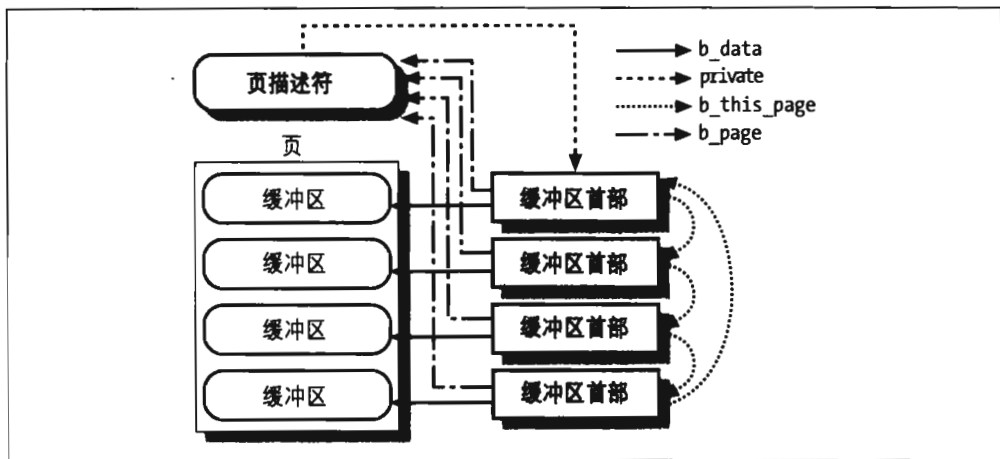


图 15-2：一个缓冲区页，其中包含四个块缓冲区和对应的缓冲区首部

分配块设备缓冲区页

当内核发现指定块的缓冲区所在的页不在页高速缓存中时，就分配一个新的块设备缓冲区页（参见本章稍后“在页高速缓存中搜索块”一节）。特别是，对块的查找操作会由于下述原因而失败：

1. 包含数据块的页不在块设备的基树中：这种情况下，必须把新页的描述符加到基树中。
2. 包含数据块的页在块设备的基树中，但这个页不是缓冲区页：在这种情况下，必须分配新的缓冲区首部，并将它链接到所属的页，从而把它变成块设备缓冲区页。
3. 包含数据块的缓冲区页在块设备的基树中，但页中块的大小与所请求的块大小不相同：这种情况下，必须释放旧的缓冲区首部，分配经过重新赋值的缓冲区首部并将它链接到所属的页。

内核调用函数 `grow_buffers()` 把块设备缓冲区页添加到页高速缓存中，该函数接收三个标识块的参数：

- `block_device` 描述符的地址 `bdev`。
- 逻辑块号 `block`（块在块设备中的位置）。
- 块大小 `size`。

该函数本质上执行下列操作：

1. 计算数据页在所请求块的块设备中的偏移量 `index`。
2. 如果需要, 就调用 `grow_dev_page()` 创建新的块设备缓冲区页。该函数依次执行下列子步骤:
 - a. 调用函数 `find_or_create_page()`, 传递给它的参数有: 块设备的 `address_space` 对象(`bdev->bd_inode->i_mapping`)、页偏移 `index` 以及 `GFP_NOFS` 标志。正如在前面“页高速缓存的处理函数”一节所描述的, `find_or_create_page()` 在页高速缓存中搜索需要的页, 如果需要, 就把新页插入高速缓存。
 - b. 此时, 所请求的页已经在页高速缓存中, 而且函数获得了它的描述符地址。函数检查它的 `PG_private` 标志; 如果为空, 说明页还不是一个缓冲区页 (没有相关的缓冲区首部), 就跳到第 2e 步。
 - c. 页已经是缓冲区页。从页描述符的 `private` 字段获得第一个缓冲区首部的地址 `bh`, 并检查块大小 `bh->size` 是否等于所请求的块大小; 如果大小相等, 在页高速缓存中找到的页就是有效的缓冲区页, 因此跳到第 2g 步。
 - d. 如果页中块的大小有错误, 就调用 `try_to_free_buffers()` (参见下一节) 释放缓冲区页的上一个缓冲区首部。
 - e. 调用函数 `alloc_page_buffers()` 根据页中所请求的块大小分配缓冲区首部, 并把它们插入由 `b_this_page` 字段实现的单向循环链表。此外, 函数用页描述符的地址初始化缓冲区首部的 `b_page` 字段, 用块缓冲区在页内的线性地址或偏移量初始化 `b_data` 字段。
 - f. 在字段 `private` 中存放第一个缓冲区首部的地址, 把 `PG_private` 字段置位, 并递增页的使用计数器 (页中的块缓冲区被算作一个页用户)。
 - g. 调用 `init_page_buffers()` 函数初始化连接到页的缓冲区首部的字段 `b_bdev`、`b_blocknr` 和 `b_bstate`。因为所有的块在磁盘上都是相邻的, 因此逻辑块号是连续的, 而且很容易从块得出。
 - h. 返回页描述符地址。
3. 为页解锁 (函数 `find_or_create_page()` 曾为页加了锁)。
4. 递减页的使用计数器 (函数 `find_or_create_page()` 曾递增了计数器)。
5. 返回 1 (成功)。

释放块设备缓冲区页

就像我们在第十七章将要了解的那样, 当内核试图获得更多的空闲内存时, 就释放块设

备缓冲区页。显然，不可能释放有脏缓冲区或上锁的缓冲区的页。内核调用函数 `try_to_release_page()` 释放缓冲区页，该函数接收页描述符的地址 `page`，并执行下述步骤（注4）：

1. 如果设置了页的 `PG_writeback` 标志，则返回 0（因为正在把页写回磁盘，所以不可能释放该页）。
2. 如果已经定义了块设备 `address_space` 对象的 `releasepage` 方法，就调用它（通常没有为块设备定义的 `releasepage` 方法）。
3. 调用函数 `try_to_free_buffers()` 并返回它的错误代码。

函数 `try_to_free_buffers()` 依次扫描链接到缓冲区页的缓冲区首部，它本质上执行下列操作：

1. 检查页中所有缓冲区的缓冲区首部的标志。如果有些缓冲区首部的 `BH_Dirty` 或 `BH_Locked` 标志被置位，说明函数不可能释放这些缓冲区，所以函数终止并返回 0（失败）。
2. 如果缓冲区首部在间接缓冲区的链表中（参见本章前面“块缓冲区和缓冲区首部”一节），该函数就从链表中删除它。
3. 清除页描述符的 `PG_private` 标记，把 `private` 字段设置为 `NULL`，并递减页的使用计数器。
4. 清除页的 `PG_dirty` 标记。
5. 反复调用 `free_buffer_head()`，以释放页的所有缓冲区首部。
6. 返回 1（成功）。

在页高速缓存中搜索块

当内核需要读或写一个单独的物理设备块时（例如一个超级块），必须检查所请求的块缓冲区是否已经在页高速缓存中。在页高速缓存中搜索指定的块缓冲区（由块设备描述符的地址 `bdev` 和逻辑块号 `nr` 表示）的过程分成三个步骤：

1. 获取一个指针，让它指向包含指定块的块设备的 `address_space` 对象（`bdev->bd_inode->i_mapping`）。

注4： 还可以对普通文件所拥有的缓冲区页调用 `try-to-release-page()` 函数。

2. 获得设备的块大小 (`bdev->bd_block_size`), 并计算包含指定块的页索引。这需要在逻辑块号上进行位移操作。例如, 如果块的大小是 1024 字节, 每个缓冲区页包含四个块缓冲区, 那么页的索引是 `nr/4`。
3. 在块设备的基树中搜索缓冲区页。获得页描述符之后, 内核访问缓冲区首部, 它描述了页中块缓冲区的状态。

不过, 实现的细节要更为复杂。为了提高系统性能, 内核维持一个小磁盘高速缓存数组 `bh_lrus` (每个 CPU 对应一个数组元素), 即所谓的最近最少使用 (LRU) 块高速缓存。每个磁盘高速缓存有 8 个指针, 指向被指定 CPU 最近访问过的缓冲区首部。对每个 CPU 数组的元素排序, 使指向最后被使用过的那个缓冲区首部的指针索引为 0。相同的缓冲区首部可能出现在几个 CPU 数组中 (但是同一个 CPU 数组中不会有相同的缓冲区首部)。在 LRU 块高速缓存中每出现一次缓冲区首部, 该缓冲区首部的使用计数器 `b_count` 就加 1。

__find_get_block() 函数

函数 `__find_get_block()` 的参数有: `block_device` 描述符地址 `bdev`、块号 `block` 和块大小 `size`。函数返回页高速缓存中的块缓冲区对应的缓冲区首部的地址; 如果不存在指定的块, 就返回 `NULL`。该函数本质上执行下面的操作:

1. 检查执行 CPU 的 LRU 块高速缓存数组中是否有一个缓冲区首部, 其 `b_bdev`、`b_blocknr` 和 `b_size` 字段分别等于 `bdev`、`block` 和 `size`。
2. 如果缓冲区首部在 LRU 块高速缓存中, 就刷新数组中的元素, 以便让指针指在第一个位置 (索引为 0) 刚找到的缓冲区首部, 递增它的 `b_count` 字段, 并跳转到第 8 步。
3. 如果缓冲区首部不在 LRU 块高速缓存中, 根据块号和块大小得到与块设备相关的页的索引:

```
index = block >> (PAGE_SHIFT - bdev->bd_inode->i_blkbits)
```

4. 调用 `find_get_page()` 确定存有所请求的块缓冲区的缓冲区页的描述符在页高速缓存中的位置。该函数传递的参数有: 指向块设备的 `address_space` 对象的指针 (`bdev->bd_inode->i_mapping`) 和页索引。页索引用于确定存有所请求的块缓冲区的缓冲区页的描述符在页高速缓存中的位置。如果高速缓存中没有这样的页, 就返回 `NULL` (失败)。
5. 此时, 函数已经得到了缓冲区页描述符的地址: 它扫描链接到缓冲区页的缓冲区首部链表, 查找逻辑块号等于 `block` 的块。

6. 递减页描述符的 count 字段 (find_get_page() 曾递增它的值)。
7. 把 LRU 块高速缓存中的所有元素向下移动一个位置, 并把指向所请求块的缓冲区首部的指针插入到第一个位置。如果一个缓冲区首部已经不在 LRU 块高速缓存中, 就递减它的引用计数器 b_count。
8. 如果需要, 就调用 mark_page_accessed() 把缓冲区页移至适当的 LRU 链表中 [参见第十七章 “最近最少使用 (LRU) 链表” 一节]。
9. 返回缓冲区首部指针。

__getblk() 函数

函数 __getblk() 与 __find_get_block() 接收相同的参数, 也就是 block_device 描述符的地址 bdev、块号 block 和块大小 size, 并返回与缓冲区对应的缓冲区首部的地址。即使块根本不存在, 该函数也不会失败, __getblk() 友好地分配块设备缓冲区页并返回将要描述块的缓冲区首部的指针。注意, __getblk() 返回的块缓冲区不必存有效数据——缓冲区首部的 BH_Uptodate 标志可能被清 0。

函数 __getblk() 本质上执行下面的步骤:

1. 调用 __find_get_block() 检查块是否已经在页高速缓存中。如果找到块, 则函数返回其缓冲区首部的地址。
2. 否则, 调用 grow_buffers() 为所请求的页分配一个新的缓冲区页 (参见本章前面 “分配块设备缓冲区页” 一节)。
3. 如果 grow_buffers() 分配这样的页时失败, __getblk() 试图通过调用函数 free_more_memory() 回收一部分内存 (参见第十七章)。
4. 跳转到第 1 步。

__bread() 函数

函数 __bread() 接收与 __getblk() 相同的参数, 即 block_device 描述符的地址 bdev、块号 block 和块大小 size, 并返回与缓冲区对应的缓冲区首部的地址。与 __getblk() 相反的是, 如果需要的话, 在返回缓冲区首部之前函数 __bread() 从磁盘读块。函数 __bread() 执行下述步骤:

1. 调用 __getblk() 在页高速缓存中查找与所请求的块相关的缓冲区页, 并获得指向相应的缓冲区首部的指针。
2. 如果块已经在页高速缓存中并包含有效数据 (BH_Uptodate 标志被置位), 就返回缓冲区首部的地址。

3. 否则, 递增缓冲区首部的引用计数器。
4. 把 `end_buffer_read_sync()` 的地址赋给 `b_end_io` 字段 (参见下一节)。
5. 调用 `submit_bh()` 把缓冲区首部传送到通用块层 (参见下一节)。
6. 调用 `wait_on_buffer()` 把当前进程插入等待队列, 直到 I/O 操作完成, 即直到缓冲区首部的 `BH_Lock` 标志被清 0。
7. 返回缓冲区首部的地址。

向通用块层提交缓冲区首部

一对 `submit_bh()` 和 `ll_rw_block()` 函数, 允许内核对缓冲区首部描述的一个或多个缓冲区进行 I/O 数据传送。

`submit_bh()` 函数

内核利用 `submit_bh()` 函数向通用块层传递一个缓冲区首部, 并由此请求传输一个数据块。它的参数是数据传输的方向 (本质上就是 `READ` 或 `WRITE`) 和指向描述块缓冲区的缓冲区首部的指针 `bh`。

`submit_bh()` 函数假设缓冲区首部已经被彻底初始化; 尤其是, 必须正确地为 `b_bdev`、`b_blocknr` 和 `b_size` 字段赋值以标识包含所请求数据的磁盘上的块。如果块缓冲区在块设备缓冲区页中, 就由 `__find_get_block()` 完成对缓冲区首部的初始化, 就像在上一节所描述的。不过, 我们将在下一章看到, 还可以对普通文件所有的缓冲区页中的块调用 `submit_bh()`。

`submit_bh()` 函数只是一个起连接作用的函数, 它根据缓冲区首部的内容创建一个 `bio` 请求, 并随后调用 `generic_make_request()` (参见第十四章“提交请求”一节)。函数执行的主要步骤如下:

1. 设置缓冲区首部的 `BH_Req` 标志以表示块至少被访问过一次。此外, 如果数据传输的方向是 `WRITE`, 就将 `BH_Write_EIO` 标志清 0。
2. 调用 `bio_alloc()` 分配一个新的 `bio` 描述符 (参见第十四章“`bio` 结构”一节)。
3. 根据缓冲区首部的内容初始化 `bio` 描述符的字段:
 - a. 把块中的第一个扇区的号 (`bh->b_blocknr*bh->b_size/512`) 赋给 `bi_sector` 字段。
 - b. 把块设备描述符的地址 (`bh->b_bdev`) 赋给 `bi_bdev` 字段。

- c. 把块大小 (`bh->b_size`) 赋给 `bi_size` 字段。
 - d. 初始化 `bi_io_vec` 数组的第一个元素 以使该段对应于块缓冲区: 把 `bh->b_page` 赋给 `bi_io_vec[0].bv_page`, 把 `bh->b_size` 赋给 `bi_io_vec[0].bv_len`, 并把块缓冲区在页中的偏移量 `bh->b_data` 赋给 `bi_io_vec[0].bv_offset`。
 - e. 把 `bi_vcnt` 置为 1 (只有一个涉及 bio 的段), 并把 `bi_idx` 置为 0 (将要传输的是当前段)。
 - f. 把 `end_bio_bh_io_sync()` 的地址赋给 `bi_end_io` 字段, 并把缓冲区首部的地址赋给 `bi_private` 字段; 数据传输结束时调用函数 (见下面)。
4. 递增 bio 的引用计数器 (它变为 2)。
 5. 调用 `submit_bio()`, 把 `bi_rw` 标志设置为数据传输的方向, 更新每 CPU 变量 `page_states` 以表示读和写的扇区数, 并对 bio 描述符调用 `generic_make_request()` 函数。
 6. 递减 bio 的使用计数器; 因为 bio 描述符现在已经被插入 I/O 调度程序的队列, 所以没有释放 bio 描述符。
 7. 返回 0 (成功)。

当针对 bio 上的 I/O 数据传输终止的时候, 内核执行 `bi_end_io` 方法, 具体来说执行 `end_bio_bh_io_sync()` 函数。后者本质上从 bio 的 `bi_private` 字段获取缓冲区首部的地址, 然后调用缓冲区首部 (在调用 `submit_bh()` 之前已为它正确赋值) 的方法 `b_end_io`, 最后调用 `bio_put()` 释放 bio 结构。

ll_rw_block()函数

有些时候内核必须立刻触发几个数据块的数据传输, 这些数据块不一定物理上相邻。`ll_rw_block()` 函数接收的参数有数据传输的方向 (本质上就是 READ 或 WRITE)、要传输的数据块的块号以及指向块缓冲区所对应的缓冲区首部的指针数组。该函数在所有缓冲区首部上进行循环, 每次循环执行下面的操作:

1. 检查并设置缓冲区首部的 `BH_Lock` 标志; 如果缓冲区已经被锁住, 而另外一个内核控制路径已经激活了数据传输, 就不处理这个缓冲区, 而跳转到第 9 步。
2. 把缓冲区首部的使用计数器 `b_count` 加 1。
3. 如果数据传输的方向是 WRITE, 就让缓冲区首部的方法 `b_end_io` 指向函数 `end_buffer_write_sync()` 的地址, 否则让 `b_end_io` 指向 `end_buffer_read_sync()` 函数的地址。

4. 如果数据传输的方向是WRITE, 就检查并清除缓冲区首部的BH_Dirty标志。如果该标志没有置位, 就不必把块写入磁盘, 因此跳转到第7步。
5. 如果数据传输的方向是READ或READA(向前读), 检查缓冲区首部的BH_Uptodate标志是否被置位; 如果是, 就不必从磁盘读块, 因此跳转到第7步。
6. 此时必须读或写数据块: 调用submit_bh()函数把缓冲区首部传递到通用块层, 然后跳转到第9步。
7. 通过清除BH_Lock标志为缓冲区首部解锁, 然后唤醒所有等待块解锁的进程。
8. 递减缓冲区首部的b_count字段。
9. 如果数组中还有其他的缓冲区首部要处理, 就选择下一个缓冲区首部并跳转回到第1步, 否则, 就结束。

注意, 如果函数ll_rw_block()把缓冲区首部传递到通用块层, 而留下加了锁的缓冲区和增加了的引用计数器, 这样, 在完成数据传输之前就不可能访问该缓冲区, 也不可能释放这个缓冲区。当块的数据传送结束时, 内核执行缓冲区首部的b_end_io方法。假设没有I/O错误, end_buffer_write_sync()和end_buffer_read_sync()函数只是简单地把缓冲区首部的BH_Uptodate字段置位, 为缓冲区解锁, 并递减它的引用计数器。

把脏页写入磁盘

正如我们所了解的, 内核不断用包含块设备数据的页填充页高速缓存。只要进程修改了数据, 相应的页就被标记为脏页, 即把它的PG_dirty标志置位。

Unix系统允许把脏缓冲区写入块设备的操作延迟执行, 因为这种策略可以显著地提高系统的性能。对高速缓存中的页的几次写操作可能只需对相应的磁盘块进行一次缓慢的物理更新就可以满足。此外, 写操作没有读操作那么紧迫, 因为进程通常是不会由于延迟写而挂起, 而大部分情况都因为延迟读而挂起。正是由于延迟写, 使得任一物理块设备平均为读请求提供的服务将多于写请求。

一个脏页可能直到最后一刻(即直到系统关闭时)都一直逗留在主存中。然而, 从延迟写策略的局限性来看, 它有两个主要的缺点:

- 如果发生了硬件错误或电源掉电的情况, 那么就无法再获得RAM的内容, 因此, 从系统启动以来对文件进行的很多修改就丢失了。
- 页高速缓存的大小(由此存放它所需的RAM的大小)就可能要很大——至少要与所访问块设备的大小相同。

因此，在下列条件下把脏页刷新（写入）到磁盘：

- 页高速缓存变得太满，但还需要更多的页，或者脏页的数量已经太多。
- 自从页变成脏页以来已过去太长时间。
- 进程请求对块设备或者特定文件任何待定的变化都进行刷新。通过调用 `sync()`、`fsync()` 或 `fdatasync()` 系统调用来实现（参见本章稍后“`sync()`、`fsync()` 和 `fdatasync()` 系统调用”一节）。

缓冲区页的引入使问题更加复杂。与每个缓冲区页相关的缓冲区首部使内核能够了解每个独立块缓冲区的状态。如果至少有一个缓冲区首部的 `BH_Dirty` 标志被置位，就应该设置相应缓冲区页的 `PG_dirty` 标志。当内核选择要刷新的缓冲区页时，它扫描相应的缓冲区首部，并只把脏块的内容有效地写到磁盘。一旦内核把缓冲区的所有脏页刷新到磁盘，就把页的 `PG_dirty` 标记清 0。

pdflush 内核线程

早期版本的 Linux 使用 *bdflush* 内核线程系统地扫描页高速缓存以搜索要刷新的脏页，并且使用另一个内核线程 *kupdate* 来保证所有的页不会“脏”太长的时间。Linux 2.6 用一组通用内核线程 *pdflush* 代替上述两个线程。

这些内核线程结构灵活，它们作用于两个参数：一个指向线程要执行的函数的指针和一个函数要用的参数。系统中 *pdflush* 内核线程的数量是要动态调整的：*pdflush* 线程太少时就创建，太多时就杀死。因为这些内核线程所执行的函数可以阻塞，所以创建多个而不是一个 *pdflush* 内核线程可以改善系统性能。

根据下面的原则控制 *pdflush* 线程的产生和消亡：

- 必须有至少两个，最多八个 *pdflush* 内核线程。
- 如果到最近的 1s 期间没有空闲 *pdflush*，就应该创建新的 *pdflush*。
- 如果最近一次 *pdflush* 变为空闲的时间超过了 1s，就应该删除一个 *pdflush*。

所有的 *pdflush* 内核线程都有 `pdflush_work` 描述符（如表 15-6 所示）。空闲 *pdflush* 内核线程的描述符都集中在 `pdflush_list` 链表中；在多处理器系统中，`pdflush_lock` 自旋锁保护该链表不会被并发访问。`nr_pdflush_threads` 变量（注 5）存放 *pdflush* 内核线程（空闲的或忙的）的总数。最后，`last_empty_jifs` 变量存放 *pdflush* 线程的 `pdflush_list` 链表变为空的时间（以 jiffies 表示）。

注 5： 可以从文件 `/proc/sys/vm/nr_pdflush_threads` 中读出这个变量的值。

表 15-6: `pdflush_work` 描述符的字段

类型	字段	说明
<code>struct task_struct</code>	<code>who</code>	指向内核线程描述符的指针
<code>void(*) (unsigned long)</code>	<code>fn</code>	内核线程所执行的回调函数
<code>unsigned long</code>	<code>arg0</code>	给回调函数的参数
<code>struct list head</code>	<code>list</code>	<code>pdflush_list</code> 链表的链接
<code>unsigned long</code>	<code>when_i_went_to_sleep</code>	当内核线程可用时的时间(以jiffies表示)

所有 *pdflush* 内核线程都执行函数 `__pdflush()`，它本质上循环执行一直到内核线程死亡。我们不妨假设 *pdflush* 内核线程是空闲的，而进程正在 `TASK_INTERRUPTIBLE` 状态睡眠。一旦内核线程被唤醒，`__pdflush()` 就访问其 `pdflush_work` 描述符，并执行字段 `fn` 中的回调函数，把 `arg0` 字段中的参数传递给该函数。当回调函数结束时，`__pdflush()` 检查 `last_empty_jifs` 变量的值：如果不存在空闲 *pdflush* 内核线程的时间已经超过 1s，而且 *pdflush* 内核线程的数量不到 8 个，函数 `__pdflush()` 就创建另外一个内核线程。相反，如果 `pdflush_list` 链表中的最后一项对应的 *pdflush* 内核线程空闲时间超过了 1s，而且系统中有两个以上的 *pdflush* 内核线程，函数 `__pdflush()` 就终止：就像在第三章“内核线程”一节所描述的，相应的内核线程执行 `_exit()` 系统调用，并因此而被撤消。否则，如果系统中 *pdflush* 内核线程不多于两个，`__pdflush()` 就把内核线程的 `pdflush_work` 描述符重新插入到 `pdflush_list` 链表中，并使内核线程睡眠。

`pdflush_operation()` 函数用来激活空闲的 *pdflush* 内核线程。该函数作用于两个参数：一个指针 `fn`，指向必须执行的函数；以及参数 `arg0`。函数执行下面的步骤：

1. 从 `pdflush_list` 链表中获取 `pdf` 指针，它指向空闲 *pdflush* 内核线程的 `pdflush_work` 描述符。如果链表为空，就返回 -1。如果链表中仅剩一个元素，就把 `jiffies` 的值赋给变量 `last_empty_jifs`。
2. 把参数 `fn` 和 `arg0` 分别赋给 `pdf->fn` 和 `pdf->arg0`。
3. 调用 `wake_up_process()` 唤醒空闲的 *pdflush* 内核线程，即 `pdf->who`。

把哪些工作委托给 *Pdflush* 内核线程来完成呢？其中一些工作与脏数据的刷新相关。尤其是，*pdflush* 通常执行下面的回调函数之一：

- `background_writeout()`：系统地扫描页高速缓存以搜索要刷新的脏页（参见下一节“搜索要刷新的脏页”）。

- `wb_kupdate()`：检查页高速缓存中是否有“脏”了很长时间的页（参见本章稍后“回写陈旧的脏页”一节）。

搜索要刷新的脏页

所有的基树都可能要有要刷新的脏页。为了得到所有这些页，就要彻底搜索与在磁盘上有映像的索引节点相应的所有 `address_space` 对象。由于页高速缓存可能有大量的页，如果用一个单独的执行流来扫描整个高速缓存，会令 CPU 和磁盘长时间繁忙。因此，Linux 使用一种复杂的机制把对页高速缓存的扫描划分为几个执行流。

`wakeup_bdflush()` 函数接收页高速缓存中应该刷新的脏页数量作为参数，0 值表示高速缓存中的所有脏页都应该写回磁盘。该函数调用 `pdflush_operation()` 唤醒 `pdflush` 内核线程（参见上一节），并委托它执行回调函数 `background_writeout()`，后者有效地从页高速缓存获得指定数量的脏页，并把它们写回磁盘。

当内存不足或用户显式地请求刷新操作时执行 `wakeup_bdflush()` 函数。特别是在下述情况下会调用该函数：

- 用户态进程发出 `sync()` 系统调用（参见本章稍后“`sync()`、`fsync()` 和 `fdatasync()` 系统调用”一节）
- `grow_buffers()` 函数分配一个新缓冲区页时失败（参见前面“分配块设备缓冲区页”一节）
- 页框回收算法调用 `free_more_memory()` 或 `try_to_free_pages()`（参见第十七章）
- `mempool_alloc()` 函数分配一个新的内存池元素时失败（参见第八章“内存池”一节）

此外，执行 `background_writeout()` 回调函数的 `pdflush` 内核线程是由满足以下两个条件的进程唤醒的：一是对页高速缓存中的页内容进行了修改，二是引起脏页部分增加到超过某个脏背景阈值（*background threshold*）。背景阈值通常设置为系统中所有页的 10%，不过可以通过修改文件 `/proc/sys/vm/dirty_background_ratio` 来调整这个值。

`background_writeout()` 函数依赖于作为双向通信设备的 `writeback_control` 结构：一方面，它告诉辅助函数 `writeback_inodes()` 要做什么；另一方面，它保存写回磁盘的页的数量的统计值。下面是这个结构最重要的字段：

`sync_mode`

表示同步模式：`WB_SYNC_ALL` 表示如果遇到一个上锁的索引节点，必须等待而不能略过它；`WB_SYNC_HOLD` 表示把上锁的索引节点放入稍后涉及的链表中；`WB_SYNC_NONE` 表示简单地略过上锁的索引节点。

bdi

如果不为空，就指向 backing_dev_info 结构。此时，只有属于基本块设备的脏页将会被刷新。

older_than_this

如果不为空，就表示应该略过比指定值还新的索引节点。

nr_to_write

当前执行流中仍然要写的脏页的数量。

nonblocking

如果这个标志被置位，就不能阻塞进程。

background_writeout() 函数只作用于一个参数 nr_pages，表示应该刷新到磁盘的最少页数。它本质上执行下述步骤：

1. 从每CPU变量page_state中读当前页高速缓存中页和脏页的数量。如果脏页所占的比例低于给定的阈值，而且已经至少有nr_pages页被刷新到磁盘，该函数就终止。这个阈值通常大约是系统中总页数的40%，可以通过写文件 */proc/sys/vm/dirty_ratio* 来调整这个值。
2. 调用 writeback_inodes() 尝试写 1024 个脏页（见下面）。
3. 检查有效写过的页的数量，并减少需要写的页的个数。
4. 如果已经写过的页少于 1024 页，或略过了一些页，则可能块设备的请求队列处于拥塞状态：此时，background_writeout() 函数使当前进程在特定的等待队列上睡眠 100ms，或使当前进程睡眠到队列变得不拥塞。
5. 返回到第 1 步。

writeback_inodes() 函数只作用于一个参数，就是指针 wbc，它指向 writeback_control 描述符。该描述符的 nr_to_write 字段存有要刷新到磁盘的页数。函数返回时，该字段存有要刷新到磁盘的剩余页数，如果一切顺利，则该字段的值被赋为 0。

我们假设 writeback_inodes() 函数被调用的条件为：指针 wbc->bdi 和 wbc->older_than_this 被置为 NULL，WB_SYNC_NONE 同步模式和 wbc->nonblocking 标志置位（这些值都由 background_writeout() 函数设置）。函数 writeback_inodes() 扫描在 super_blocks 变量中建立的超级块链表（参见第十二章“超级块对象”一节）。当遍历完整个链表或刷新的页数达到预期数量时，就停止扫描。对每个超级块 sb，函数执行下述步骤：

1. 检查 `sb->s_dirty` 或 `sb->s_io` 链表是否为空：第一个链表集中了超级块的脏索引节点，而第二个链表集中了等待被传输到磁盘的索引节点（见下面）。如果两个链表都为空，说明相应文件系统的索引节点没有脏页，因此函数处理链表中的下一个超级块。
2. 此时，超级块有脏索引节点。对超级块 `sb` 调用 `sync_sb_inodes()`，该函数执行下面的操作：
 - a. 把 `sb->s_dirty` 的所有索引节点插入 `sb->s_io` 指向的链表，并清空脏索引节点链表。
 - b. 从 `sb->s_io` 获得下一个索引节点的指针。如果该链表为空，就返回。
 - c. 如果 `sync_sb_inodes()` 函数开始执行后，索引节点变为脏节点，就略过这个索引节点的脏页并返回。注意，`sb->s_io` 链表中可能残留一些脏索引节点。
 - d. 如果当前进程是 `pdflush` 内核线程，`sync_sb_inodes()` 就检查运行在另一个 CPU 上的 `pdflush` 内核线程是否已经试图刷新这个块设备文件的脏页。这是通过一个原子测试和对索引节点的 `backing_dev_info` 的 `BDI_pdflush` 标志的设置操作来完成的。本质上，它对同一个请求队列上有多个 `pdflush` 内核线程是毫无意义的（参见本章前面“`pdflush` 内核线程”一节）。
 - e. 把索引节点的引用计数器加 1。
 - f. 调用 `__writeback_single_inode()` 回写与所选择的索引节点相关的脏缓冲区：
 - (1) 如果索引节点被锁定，就把它移到脏索引节点链表中（`inode->i_sb->s_dirty`）并返回 0。（因为我们假定 `wbc->sync_mode` 字段不等于 `WB_SYNC_ALL`，所以函数不会因为等待索引节点解锁而阻塞。）
 - (2) 使用索引节点地址空间的 `writepages` 方法，或者在没有这个方法的情况下使用 `mpage_writepages()` 函数来写 `wbc->nr_to_write` 个脏页。该函数调用 `find_get_pages_tag()` 函数快速获得索引节点地址空间的所有脏页（参见本章前面“基树的标记”一节），细节将在下一章描述。
 - (3) 如果索引节点是脏的，就用超级块的 `write_inode` 方法把索引节点写到磁盘。实现该方法的函数通常依靠 `submit_bh()` 来传输一个数据块（参见本章前面“向通用块层提交缓冲区首部”一节）。
 - (4) 检查索引节点的状态。如果索引节点还有脏页，就把索引节点移回 `sb->s_dirty` 链表；如果索引节点引用计数器为 0，就把索引节点移到 `inode_unused` 链表中；否则就把索引节点移到 `inode_in_use` 链表中。（参见第十二章“索引节点对象”一节）。

- (5) 返回在第 2f(2)步所调用的函数的错误代码。
- g. 回到 `sync_sb_inodes()` 函数中。如果当前进程是 *pdflush* 内核线程，就把在第 2d 步设置的 `BDI_pdflush` 标志清 0。
 - h. 如果略过了刚处理的索引节点中的一些页，那么该索引节点包括锁定的缓冲区：把 `sb->s_io` 链表中的所有剩余索引节点移回到 `sb->s_dirty` 链表中，以后将重新处理它们。
 - i. 把索引节点的引用计数器减 1。
 - j. 如果 `wbc->nr_to_write` 大于 0，则回到第 2b 步搜索同一个超级块的其他脏索引节点。否则，`sync_sb_inodes()` 函数终止。
3. 回到 `writeback_inodes()` 函数中。如果 `wbc->nr_to_write` 大于 0，就跳转到第 1 步，并继续处理全局链表中的下一个超级块。否则，就返回。

回写陈旧的脏页

如前所述，内核试图避免当一些页很久没有被刷新时发生饥饿危险。因此，脏页在保留一定时间后，内核就显式地开始进行 I/O 数据的传输，把脏页的内容写到磁盘。

回写陈旧脏页的工作委托给了被定期唤醒的 *pdflush* 内核线程。在内核初始化期间，`page_writeback_init()` 函数建立 `wb_timer` 动态定时器，以便定时器的到期时间发生在 `dirty_writeback_centisecs` 文件中所规定的几百分之一秒之后（通常是 500 分之一秒，不过可以通过修改 `/proc/sys/vm/dirty_writeback_centisecs` 文件调整这个值）。定时器函数 `wb_timer_fn()` 本质上调用 `pdflush_operation()` 函数，传递给它的参数是回调函数 `wb_kupdate()` 的地址。

`wb_kupdate()` 函数遍历页高速缓存搜索陈旧的脏索引节点，它执行下面的步骤：

1. 调用 `sync_supers()` 函数把脏的超级块写到磁盘中（参见下一节）。虽然这与页高速缓存中的页刷新没有很密切的关系，但对 `sync_supers()` 的调用确保了任何超级块脏的时间通常不会超过 5s。
2. 把当前时间减 30s 所对应的值（用 jiffies 表示）的指针存放在 `writeback_control` 描述符的 `older_than_this` 字段中。允许一个页保持脏状态的最长时间是 30s。
3. 根据每 CPU 变量 `page_state` 确定当前在页高速缓存中脏页的大概数量。
4. 反复调用 `writeback_inodes()`，直到写入磁盘的页数等于上一步所确定的值，或直到把所有保持脏状态时间超过 30s 的页都写到磁盘。如果在循环的过程中一些请求队列变得拥塞，函数就可能去睡眠。

5. 用 `mod_timer()` 重新启动 `wb_timer` 动态定时器：一旦从调用该函数开始经历过文件 `dirty_writeback_centisecs` 中规定的几百分之一秒时间后，定时器到期（或者如果本次执行的时间太长，就从现在开始 1s 后到期）。

sync()、fsync()和fdatasync()系统调用

在本节我们简要介绍用户应用程序把脏缓冲区刷新到磁盘会用到的三个系统调用：

`sync()`

允许进程把所有的脏缓冲区刷新到磁盘。

`fsync()`

允许进程把属于特定打开文件的所有块刷新到磁盘。

`fdatasync()`

与 `fsync()` 非常相似，但不刷新文件的索引节点块。

sync ()系统调用

`sync()` 系统调用的服务例程 `sys_sync()` 调用一系列辅助函数：

```
wakeup_bdflush(0);
sync_inodes(0);
sync_supers();
sync_filesystems(0);
sync_filesystems(1);
sync_inodes(1);
```

正如上一节所描述的，`wakeup_bdflush()` 启动 *pdflush* 内核线程，把页高速缓存中的所有脏页刷新到磁盘。

`sync_inodes()` 函数扫描超级块的链表以搜索要刷新的脏索引节点；它作用于参数 `wait`，该参数表示在执行完刷新之前函数是否必须等待。函数扫描当前已安装的所有文件系统的超级块；对于每个包含脏索引节点的超级块，`sync_inodes()` 首先调用 `sync_sb_inodes()` 刷新相应的脏页（我们在前面“搜索要刷新的脏页”一节曾对该函数进行过说明），然后调用 `sync_blockdev()` 显式刷新该超级块所在块设备的脏缓冲区页。这一步之所以能完成是因为许多磁盘文件系统的 `write_inode` 超级块方法仅仅把磁盘索引节点对应的块缓冲区标记为“脏”；函数 `sync_blockdev()` 确保把 `sync_sb_inodes()` 所完成的更新有效地写到磁盘。

函数 `sync_supers()` 把脏超级块写到磁盘，如果需要，也可以使用适当的 `write_super` 超级块操作。最后，`sync_filesystems()` 为所有可写的文件系统执行 `sync_fs` 超级块

方法。该方法只不过是提供给文件系统的一个“钩子”，在需要对每个同步执行一些特殊操作时使用，只有像 Ext3(参见第十八章)这样的日志文件系统使用这个方法。

注意，`sync_inodes()`和`sync_filesystems()`都是被调用两次，一次是参数`wait`等于0时，另一次是`wait`等于1时。这样做的目的是：首先，它们把未上锁的索引节点快速刷新到磁盘；其次，它们等待所有上锁的索引节点被解锁，然后把它们逐个地写到磁盘。

`fsync ()`和`fdatasync ()` 系统调用

系统调用`fsync()`强制内核把文件描述符参数`fd`所指定文件的所有脏缓冲区写到磁盘中（如果需要，还包括存有索引节点的缓冲区）。相应的服务例程获得文件对象的地址，并随后调用`fsync`方法。通常这个方法以调用函数`__writeback_single_inode()`结束，该函数把与被选中的索引节点相关的脏页和索引节点本身都写回磁盘（参见本章前面“搜索要刷新的脏页”一节）。

系统调用`fdatasync()`与`fsync()`非常相似，但是它只把包含文件数据而不是那些包含索引节点信息的缓冲区写到磁盘。由于Linux 2.6没有提供专门的`fdatasync()`文件方法，该系统调用使用`fsync`方法，因此与`fsync()`是相同的。



第十六章

访问文件

访问基于磁盘的文件是一种复杂的活动，既涉及 VFS 抽象层（第十二章）、块设备的处理（第十四章），也涉及磁盘高速缓存的使用（第十五章）。本章介绍内核如何使用这些技术实现文件的读及写。本章所涵盖的主题既应用于磁盘文件系统的普通文件，也应用于块设备文件，将这两种文件系统都简单地统称为“文件”。

本章所介绍的内容是调用了读或写方法之后（如第十二章中所描述）系统所处的阶段。在这里，我们会说明每个读操作最终是如何把所需要的数据传递给用户态进程的，以及每个写操作最终又是如何把数据标志为“就绪”以传送到磁盘上的。其他传送过程是使用第十四章和第十五章中描述的技术来处理的。

访问文件的模式有多种。我们在本章考虑如下几种情况：

规范模式

规范模式下文件打开后，标志 `O_SYNC` 与 `O_DIRECT` 清 0，而且它的内容是由系统调用 `read()` 和 `write()` 来存取。系统调用 `read()` 将阻塞调用进程，直到数据被拷贝进用户态地址空间（内核允许返回的字节数少于要求的字节数）。但系统调用 `write()` 不同，它在数据被拷贝到页高速缓存（延迟写）后就马上结束。这会在“读写文件”这一节详细阐述。

同步模式

同步模式下文件打开后，标志 `O_SYNC` 置 1 或稍后由系统调用 `fcntl()` 对其置 1。这个标志只影响写操作（读操作总是会阻塞），它将阻塞调用进程，直到数据被有效地写入磁盘。这也会在“读写文件”这一节详细阐述。

内存映射模式

内存映射模式下文件打开后,应用程序发出系统调用`mmap()`将文件映射到内存中。因此,文件就成为RAM中的一个字节数组,应用程序就可以直接访问数组元素,而不需用系统调用`read()`、`write()`或`lseek()`。这将在“内存映射”这一节详细阐述。

直接 I/O 模式

直接 I/O 模式下文件打开后,标志`O_DIRECT`置1。任何读写操作都将数据在用户态地址空间与磁盘间直接传送而不通过页高速缓存。这将在“直接 I/O 传送”这一节详细阐述。(标志`O_SYNC`和`O_DIRECT`的值可以有四种组合。)

异步模式

异步模式下,文件的访问可以有两种方法,即通过一组 POSIX API 或 Linux 特有的系统调用来实现。所谓异步模式就是数据传输请求并不阻塞调用进程,而是在后台执行,同时应用程序继续它的正常运行。这将在“异步 I/O”这一节详细阐述。

读写文件

在第十二章的“`read()`和`write()`系统调用”一节中已经说明了`read()`和`write()`系统调用是如何实现的。相应的服务例程最终会调用文件对象的`read`和`write`方法,这两个方法可能依赖文件系统。对磁盘文件系统来说,这些方法能够确定正被访问的数据所在物理块的位置,并激活块设备驱动程序开始数据传送。

读文件是基于页的,内核总是一次传送几个完整的数据页。如果进程发出`read()`系统调用来读取一些字节,而这些数据还不在RAM中,那么,内核就要分配一个新页框,并使用文件的适当部分来填充这个页,把该页加入页高速缓存,最后把所请求的字节拷贝到进程地址空间中。对于大部分文件系统来说,从文件中读取一个数据页就等同于在磁盘上查找所请求的数据存放在哪些块上。只要这个过程完成了,内核就可以通过向通用块层提交适当的I/O操作来填充这些页。事实上,大多数磁盘文件系统的`read`方法是由名为`generic_file_read()`的通用函数实现的。

对基于磁盘的文件来说,写操作的处理相当复杂,因为文件大小可以改变,因此内核可能会分配磁盘上的一些物理块。当然,这个过程到底如何实现要取决于文件系统的类型。不过,很多磁盘文件系统是通过通用函数`generic_file_write()`实现它们的`write`方法的。这样的文件系统如Ext2、System V/Coherent/Xenix及Minix。另一方面,还有几个文件系统(如日志文件系统和网络文件系统)通过自定义的函数实现它们的`write`方法。

从文件中读取数据

让我们讨论一下 `generic_file_read()` 函数，该函数实现了几乎所有磁盘文件系统中的普通文件及任何块设备文件的 `read` 方法。该函数作用于以下参数：

<code>filp</code>	文件对象的地址
<code>buf</code>	用户态线性区的线性地址，从文件中读出的数据必须存放在这里
<code>count</code>	要读取的字符个数
<code>ppos</code>	指向一个变量的指针，该变量存放读操作开始处的文件偏移量（通常为 <code>filp</code> 文件对象的 <code>f_pos</code> 字段）

第一步，函数初始化两个描述符。第一个描述符存放在类型为 `iovec` 的局部变量 `local_iov` 中；它包含用户态缓冲区的地址（`buf`）与长度（`count`），该缓冲区用来存放待读文件中的数据。第二个描述符存放在类型为 `kiocb` 的局部变量 `kiocb` 中；它用来跟踪正在运行的同步和异步 I/O 操作的完成状态。`kiocb` 描述符的主要字段描述如表 16-1 所示。

表 16-1: `kiocb` 描述符的主要字段

类型	字段	说明
<code>struct list_head</code>	<code>ki_run_list</code>	以后要重新操作的 I/O 链表指针
<code>long</code>	<code>ki_flags</code>	<code>kiocb</code> 描述符的标志
<code>int</code>	<code>ki_users</code>	<code>kiocb</code> 描述符的引用计数器
<code>unsigned int</code>	<code>ki_key</code>	异步 I/O 操作标识符，同步 I/O 操作标识符为 <code>KIOCB_SYNC_KEY</code> (<code>0xffffffff</code>)
<code>struct file *</code>	<code>ki_filp</code>	与正在进行的 I/O 操作相关的文件对象指针
<code>struct kioctx *</code>	<code>ki_ctx</code>	异步 I/O 环境描述符指针（参见本章后面的“异步 I/O”一节）
<code>int (*)</code> (<code>struct kiocb *</code> , <code>struct io_event *</code>)	<code>ki_cancel</code>	当取消异步 I/O 操作时所调用的方法
<code>ssize_t (*)</code> (<code>struct kiocb *</code>)	<code>ki_retry</code>	当重试异步 I/O 操作时所调用的方法

表 16-1: kiocb 描述符的主要字段 (续)

类型	字段	说明
void (*) (struct kiocb *)	ki_dtor	当清除 kiocb 描述符时所调用的方法
struct list_head	ki_list	在异步操作环境下, 当前进行的 I/O 操作链表的指针
union	ki_obj	对于同步操作, 它是指向发出该操作的进程描述符的指针; 对于异步操作, 它是指向用户态数据结构 iocb 的指针
__u64	ki_user_data	给用户态进程返回的值
loff_t	ki_pos	正在进行 I/O 操作的当前文件位置
unsigned short	ki_opcode	操作类型(read、write 或 sync)
size_t	ki_nbytes	被传输的字节数
char *	ki_buf	用户态缓冲区的当前位置
size_t	ki_left	待传输的字节数
wait_queue_t	ki_wait	异步 I/O 操作等待队列
void *	private	由文件系统层自由使用

函数 `generic_file_read()` 通过执行宏 `init_sync_kiocb` 来初始化描述符 `kiocb`, 并设置一个同步操作对象的有关字段。具体地说就是, 该宏设置 `ki_key` 字段为 `KIOCB_SYNC_KEY`、`ki_filp` 字段为 `filp`、`ki_obj` 字段为 `current`。

然后, `generic_file_read()` 调用 `__generic_file_aio_read()` 并将刚填完的 `iovec` 和 `kiocb` 描述符地址传给它。后面这个函数返回一个值, 这个值通常就是从文件有效读入的字节数。`generic_file_read()` 返回值后结束。

函数 `__generic_file_aio_read()` 是所有文件系统实现同步和异步读操作所使用的通用例程。该函数接受四个参数: `kiocb` 描述符的地址 `iocb`、`iovec` 描述符数组的地址 `iov`、数组的长度和存放文件当前指针的一个变量的地址 `ppos`。`iovec` 描述符数组被函数 `generic_file_read()` 调用时只有一个元素, 该元素描述待接收数据的用户态缓冲区(注 1)。

注 1: `read()` 系统调用的一个叫做 `readv()` 的变体允许应用程序定义多个用户态缓冲区, 从文件读出的数据分散存放在其中; `__generic_file_aio_read()` 函数也实现这种功能。下面我们要假设从文件读出的数据将只拷贝到一个用户态缓冲区, 不过, 可以想象, 使用多个缓冲区虽然简单, 但需要执行更多的步骤。

我们现在来说明函数 `_generic_file_aio_read()` 的操作。为简单起见，我们只针对最常见的情形，即对页高速缓存文件的系统调用 `read()` 所引发的同步操作。本章后面我们会阐述该函数执行的其他情形。同样，我们不讨论如何对错误和异常的处理。

该函数执行的步骤如下：

1. 调用 `access_ok()` 来检查 `iovec` 描述符所描述的用户态缓冲区是否有效。因为起始地址和长度已经从 `sys_read()` 服务例程得到，因此在使用前需要对它们进行检查（参见第十章“验证参数”一节）。如果参数无效，则返回错误代码 `-EFAULT`。
2. 建立一个读操作描述符，也就是一个 `read_descriptor_t` 类型的数据结构。该结构存放与单个用户态缓冲相关的文件读操作的当前状态。该描述符的字段参见表16-2。
3. 调用函数 `do_generic_file_read()`，传送给它文件对象指针 `filp`、文件偏移量指针 `ppos`、刚分配的读操作描述符的地址和函数 `file_read_actor()` 的地址（后面还会阐述）。
4. 返回拷贝到用户态缓冲区的字节数，即 `read_descriptor_t` 数据结构中 `written` 字段的值。

表 16-2: 读操作描述符的字段

类型	字段	说明
size_t	written	已经拷贝到用户态缓冲区的字节数
size_t	count	待传送的字节数
char *	buf	在用户态缓冲区中的当前位置
int	error	读操作的错误码（0 表示无错误）

函数 `do_generic_file_read()` 从磁盘读入所请求的页并把它们拷贝到用户态缓冲区。具体执行如下步骤：

1. 获得要读取的文件对应的 `address_space` 对象，它的地址存放在 `filp->f_mapping`。
2. 获得地址空间对象的所有者，即索引节点对象，它将拥有填充了文件数据的页面。它的地址存放在 `address_space` 对象的 `host` 字段中。如果所读文件是块设备文件，那么所有者就不是由 `filp->f_dentry->d_inode` 所指向的索引节点对象，而是 `bdev` 特殊文件系统中的索引节点对象。
3. 把文件看作细分的数据页（每页 4096 字节），并从文件指针 `*ppos` 导出第一个请求字节所在页的逻辑号，即地址空间中的页索引，并把它存放在 `index` 局部变量中。也把第一个请求字节在页内的偏移量存放在 `offset` 局部变量中。

4. 开始一个循环来读入包含请求字节的所有页，要读数据的字节数存放在 `read_descriptor_t` 描述符的 `count` 字段中。在一次单独的循环期间，函数通过执行下列的子步骤来传送一个数据页：
 - a. 如果 `index*4096+offset` 超过存放在索引节点对象的 `i_size` 字段中的文件大小，则从循环退出，并跳到第 5 步。
 - b. 调用 `cond_resched()` 来检查当前进程的标志 `TIF_NEED_RESCHED`。如果该标志置位，则调用函数 `schedule()`。
 - c. 如果有预读的页，则调用 `page_cache_readahead()` 读入这些页面。我们在后面“文件的预读”一节讨论预读。
 - d. 调用 `find_get_page()`，并传入指向 `address_space` 对象的指针及索引值作为参数；它将查找页高速缓存以找到包含所请求数据的页描述符（如果有的话）。
 - e. 如果 `find_get_page()` 返回 `NULL` 指针，则所请求的页不在页高速缓存中。如果这样，它将执行如下步骤：
 - (1) 调用 `handle_ra_miss()` 来调整预读系统的参数。
 - (2) 分配一个新页。
 - (3) 调用 `add_to_page_cache()` 插入该新页描述符到页高速缓存中。记住该函数将新页的 `PG_locked` 标志置位。
 - (4) 调用 `lru_cache_add()` 插入新页描述符到 LRU 链表（参见第十七章）。
 - (5) 跳到第 4j 步，开始读文件数据。
 - f. 如果函数已运行至此，说明页已经位于页高速缓存中。检查标志 `PG_uptodate`，如果置位，则页所存数据是最新的，因此无需从磁盘读数据。跳到第 4m 步。
 - g. 页中的数据是无效的，因此必须从磁盘读取。函数通过调用 `lock_page()` 函数获取对页的互斥访问。正如第十五章“页高速缓存的处理函数”一节中所描述的，如果 `PG_locked` 已经置位，则 `lock_page()` 阻塞当前进程直到标志被清 0。
 - h. 现在页已由当前进程锁定。然而，另一个进程也许会在上一步之前已从页高速缓存中删除该页，那么，它就要检查页描述符的 `mapping` 字段是否为 `NULL`。在这种情形下，它将调用 `unlock_page()` 来解锁页，减少它的引用计数（`find_get_page()` 增加计数），并跳回第 4a 步来重读同一页。
 - i. 如果函数已运行至此，说明页已被锁定且在页高速缓存中。再次检查标志 `PG_uptodate`，因为另一个内核控制路径可能已经完成第 4f 步和第 4g 步的必要读操作。如果标志置位，则调用 `unlock_page()` 并跳至第 4m 来跳过读操作。

- j. 现在真正的I/O操作可以开始了,调用文件的address_space对象之readpage方法。相应的函数会负责激活磁盘到页之间的I/O数据传输。我们以后再讨论该函数对普通文件与块设备文件都会做些什么。
- k. 如果标志PG_uptodate还没有置位,则它会等待直到调用lock_page()函数后页被有效读入。该页在第4g步中锁定,一旦读操作完成就被解锁。因此当前进程在I/O数据传输完成时才停止睡眠。
- l. 如果index超出文件包含的页数(该数是通过将inode对象的i_size字段的值除以4096得到的),那么它将减少页的引用计数器,并跳出循环至第5步。这种情况发生在这个正被本进程读的文件同时有其他进程正在删减它的时候。
- m. 将应被拷入用户态缓冲区的页中的字节数存放在局部变量nr中。这个值应该等于页的大小(4096字节),除非offset非0(这只发生在读请求书的首尾页时)或请求数据不全在该文件中。
- n. 调用mark_page_accessed()将标志PG_referenced或PG_active置位,从而表示该页正被访问并且不应该被换出(参见第十七章)。如果同一文件(或它的一部分)在do_generic_file_read()的后续执行中要读几次,那么这个步骤只在第一次读时执行。
- o. 现在到了把页中的数据拷贝到用户态缓冲区的时候了。为了这么做,do_generic_file_read()调用file_read_actor()函数,该函数的地址作为参数传递。file_read_actor()执行下列步骤:
 - (1) 调用kmap(),该函数为处于高端内存中的页建立永久的内核映射(参见第八章“高端内存页框的内核映射”一节)。
 - (2) 调用__copy_to_user(),该函数把页中的数据拷贝到用户态地址空间(参见第十章“访问进程地址空间”一节)。注意,这个操作在访问用户态地址空间时如果有缺页异常将会阻塞进程。
 - (3) 调用kunmap()来释放页的任一永久内核映射。
 - (4) 更新read_descriptor_t描述符的count、written和buf字段。
- p. 根据传入用户态缓冲区的有效字节数来更新局部变量index和count。一般情况下,如果页的最后一个字节已拷贝到用户态缓冲区,那么index的值加1而offset的值清0;否则,index的值不变而offset的值被设为已拷贝到用户态缓冲区的字节数。
- q. 减少页描述符的引用计数器。
- r. 如果read_descriptor_t描述符的count字段不为0,那么文件中还有其他数据要读,跳至第4a步继续循环来读文件中的下一页数据。

5. 所有请求的或者说可以读到的数据已读完。函数更新预读数据结构 `filp->f_ra` 来标记数据已被顺序从文件读入（参见下一节“文件的预读”）。
6. 把 `index*4096+offset` 值赋给 `*ppos`，从而保存以后调用 `read()` 和 `write()` 进行顺序访问的位置。
7. 调用 `update_atime()` 把当前时间存放在文件的索引节点对象的 `i_atime` 字段中，并把它标记为脏后返回。

普通文件的 `readpage` 方法

我们从前一节看到，`do_generic_file_read()` 反复使用 `readpage` 方法把一个个页从磁盘读到内存中。

`address_space` 对象的 `readpage` 方法存放的是函数地址，这种函数有效地激活从物理磁盘到页高速缓存的 I/O 数据传送。对于普通文件，这个字段通常指向调用 `mpage_readpage()` 函数的封装函数。例如，Ext3 文件系统的 `readpage` 方法由下列函数实现：

```
int ext3_readpage(struct file *file, struct page *page)
{
    return mpage_readpage(page, ext3_get_block);
}
```

需要封装函数是因为 `mpage_readpage()` 函数接收的参数为待填充页的页描述符 `page` 及有助于 `mpage_readpage()` 找到正确块的函数的地址 `get_block`。封装函数依赖文件系统并因此能提供适当的函数来得到块。这个函数把相对于文件开始位置的块号转换为相对于磁盘分区中块位置的逻辑块号（例子参见第十八章）。当然，后一个参数依赖于普通文件所在文件系统的类型；在前面的例子中，这个参数就是 `ext3_get_block()` 函数的地址。所传递的 `get_block` 函数总是用缓冲区首部来存放有关重要信息，如块设备（`b_dev` 字段）、设备上请求数据的位置（`b_blocknr` 字段）和块状态（`b_state` 字段）。

函数 `mpage_readpage()` 在从磁盘读入一页时可选择两种不同的策略。如果包含请求数据的块在磁盘上是连续的，那么函数就用单个 `bio` 描述符向通用块层发出读 I/O 操作。而如果不连续，函数就对页上的每一块用不同的 `bio` 描述符来读。`get_block` 函数依赖于文件系统，它的一个重要作用就是：确定文件中的下一块在磁盘上是否也是下一块。

具体地说，`mpage_readpage()` 函数执行下列步骤：

1. 检查页描述符的 `PG_private` 字段：如果置位，则该页是缓冲区页，也就是该页与描述组成该页的块的缓冲区首部链表相关（参见第十五章“把块存放在页高速缓存

中”一节)。这意味着该页过去已从磁盘读入过,而且页中的块在磁盘上不是相邻的。跳到第 11 步,用一次读一块的方式读该页。

2. 得到块的大小(存放在 `page->mapping->host->i_blkbits` 索引节点字段),然后计算出访问该页的所有块所需要的两个值,即页中的块数及页中第一块的文件块号,也就是相对于文件起始位置页中第一块的索引。
3. 对于页中的每一块,调用依赖于文件系统的 `get_block` 函数,作为参数传递过去,得到逻辑块号,即相对于磁盘或分区开始位置的块索引。页中所有块的逻辑块号存放在一个本地数组中。
4. 在执行上一步的同时,检查可能发生的异常条件。具体有这几种情况:当一些块在磁盘上不相邻时,或某块落入“文件洞”内(参见第十八章的“文件的洞”一节)时,或一个块缓冲区已经由 `get_block` 函数写入时。那么跳到第 11 步,用一次读一块的方式读该页。
5. 如果函数运行至此,说明页中的所有块在磁盘上是相邻的。然而,它可能是文件中的最后一页,因此页中的一些块可能在磁盘上没有映像。如果这样的话,它将页中相应的块缓冲区填上 0;如果不是这样,它将页描述符的标志 `PG_mappedtodisk` 置位。
6. 调用 `bio_alloc()` 分配包含单一段的一个新 `bio` 描述符,并且分别用块设备描述符地址和页中第一个块的逻辑块号来初始化 `bi_bdev` 字段和 `bi_sector` 字段。这两个信息已在上面的第 3 步中得到。
7. 用页的起始地址、所读数据的首字节偏移量(0)和所读的字节总数设置 `bio` 段的 `bio_vec` 描述符。
8. 将 `mpage_end_io_read()` 函数的地址赋给 `bio->bi_end_io` 字段(见下面)。
9. 调用 `submit_bio()`,它将用数据传输的方向设定 `bi_rw` 标志,更新每 CPU 变量 `page_states` 来跟踪所读扇区数,并在 `bio` 描述符上调用 `generic_make_request()` 函数(参见第十四章的“向 I/O 调度程序发出请求”一节)。
10. 返回 0 (成功)。
11. 如果函数跳到这里,则页中含有的块在磁盘上不连续。如果页是最新的(`PG_uptodate` 置位),函数就调用 `unlock_page()` 来对该页解锁;否则调用 `block_read_full_page()` 用一次读一块的方式读该页(见下面)。
12. 返回 0 (成功)。

函数 `mpage_end_io_read()` 是 `bio` 的完成方法,一旦 I/O 数据传输结束它就开始执行。假定没有 I/O 错误,该函数将页描述符的标志 `PG_uptodate` 置位,调用 `unlock_page()` 来对该页解锁并唤醒任何因为该事件而睡眠的进程,然后调用 `bio_put()` 来清除 `bio` 描述符。

块设备文件的 readpage 方法

在第十三章“设备文件的 VFS 处理”一节和第十四章的“打开块设备文件”一节中，我们讨论了内核如何处理请求以打开块设备文件。我们还看到 `init_special_inode()` 函数如何建立设备的索引节点及 `blkdev_open()` 如何完成其打开阶段。

在 `bdev` 特殊文件系统中，块设备使用 `address_space` 对象，该对象存放在对应块设备索引节点的 `i_data` 字段。不像普通文件（在 `address_space` 对象中它的 `readpage` 方法依赖于文件所属的文件系统的类型），块设备文件的 `readpage` 方法总是相同的。它是由 `blkdev_readpage()` 函数实现的，该函数调用 `block_read_full_page()`：

```
int blkdev_readpage(struct file * file, struct * page page)
{
    return block_read_full_page(page, blkdev_get_block);
}
```

正如你看到的，这个函数又是一个封装函数，这里是 `block_read_full_page()` 函数的封装函数。这一次，第二个参数也指向一个函数，该函数把相对于文件开始处的文件块号转换为相对于块设备开始处的逻辑块号。不过，对于块设备文件来说，这两个数是一致的，因此，`blkdev_get_block()` 函数执行下列步骤：

1. 检查页中第一个块的块号是否超过块设备的最后一块的索引值（存放在 `bdev->bd_inode->i_size` 中的块设备大小除以存放在 `bdev->bd_block_size` 中的块大小得到该索引值；`bdev` 指向块设备描述符）。如果超过，那么对于写操作它返回 `-EIO`，而对于读操作它返回 0。（超出块设备读也是不允许的，但不返回错误代码。内核可以对块设备的最后数据试着发出读请求，而得到的缓冲区页只被部分映射）。
2. 设置缓冲区首部的 `b_dev` 字段为 `b_dev`。
3. 设置缓冲区首部的 `b_blocknr` 字段为文件块号，它将被作为参数传给本函数。
4. 把缓冲区首部的 `BH_Mapped` 标志置位，以表明缓冲区首部的 `b_dev` 和 `b_blocknr` 字段是有效的。

函数 `block_read_full_page()` 以一次读一块的方式读一页数据。正如我们已看到的，当读块设备文件和磁盘上块不相邻的普通文件时都使用该函数。它执行如下步骤：

1. 检查页描述符的标志 `PG_private`，如果置位，则该页与描述组成该页的块的缓冲区首部链表相关（参见第十五章的“把块存放在页高速缓存中”一节）；否则，调用 `create_empty_buffers()` 来为该页所含所有块缓冲区分配缓冲区首部。页中第一个缓冲区的缓冲区首部地址存放在 `page->private` 字段中。每个缓冲区首部的 `b_this_page` 字段指向该页中下一个缓冲区的缓冲区首部。

2. 从相对于页的文件偏移量 (page->index 字段) 计算出页中第一块的文件块号。
3. 对该页中每个缓冲区的缓冲区首部, 执行如下子步骤:
 - a. 如果标志 BH_Uptodate 置位, 则跳过该缓冲区继续处理该页的下一个缓冲区。
 - b. 如果标志 BH_Mapped 未置位, 并且该块未超出文件尾, 则调用依赖于文件系统的 get_block 函数, 该函数的地址已被作为参数得到。对于普通文件, 该函数在文件系统的磁盘数据结构中查找, 得到相对于磁盘或分区开始处的缓冲区逻辑块号。对于块设备文件, 不同的是该函数把文件块号当作逻辑块号。对这两种情形, 函数都将逻辑块号存放在相应缓冲区首部的 b_blocknr 字段中, 并将标志 BH_Mapped 置位 (注 2)。
 - c. 再检查标志 BH_Uptodate, 因为依赖于文件系统的 get_block 函数可能已触发块 I/O 操作而更新了缓冲区。如果 BH_Uptodate 置位, 则继续处理该页的下一个缓冲区。
 - d. 将缓冲区首部的地址存放在局部数组 arr 中, 继续该页的下一个缓冲区。
4. 假如上一步中没遇到“文件洞”, 则将该页的标志 PG_mappedtodisk 置位。
5. 现在局部变量 arr 中存放了一些缓冲区首部的地址, 与其对应的缓冲区的内容不是最新的。如果数组为空, 那么页中的所有缓冲区都是有效的, 因此, 该函数设置页描述符的 PG_uptodate 标志, 调用 unlock_page() 对该页解锁并返回。
6. 局部数组 arr 非空。对数组中的每个缓冲区首部, block_read_full_page() 执行下列子步骤:
 - a. 将 BH_Lock 标志置位。该标志一旦置位, 函数将一直等到该缓冲区释放。
 - b. 将缓冲区首部的 b_end_io 字段设为 end_buffer_async_read() 函数的地址 (见下面), 并将缓冲区首部的 BH_Async_Read 标志置位。
7. 对局部数组 arr 中的每个缓冲区首部调用 submit_bh(), 将操作类型设为 READ。就像我们在前面看到的那样, 该函数触发了相应块的 I/O 数据传输。
8. 返回 0。

注 2: 访问普通文件时, 如果一个数据块处于“文件洞”中, get_block 函数就可能找不到这个块 (参见第十八章“文件的洞”一节)。此时, 函数用 0 填充这个块缓冲区并设置缓冲区首部的 BH_Uptodate 标志。

函数 `end_buffer_async_read()` 是缓冲区首部的完成方法。对块缓冲区的 I/O 数据传输一结束，它就执行。假定没有 I/O 错误，函数将缓冲区首部的 `BH_Uptodate` 标志置位而将 `BH_Async_Read` 标志清 0。那么，函数就得到包含块缓冲区的缓冲区页描述符（它的地址存放在缓冲区首部的 `b_page` 字段中），同时检查是否页中所有块是最新的；如果是，函数将该页的 `PG_uptodate` 标志置位并调用 `unlock_page()`。

文件的预读

很多磁盘的访问都是顺序的。我们在第十八章会看到，普通文件以相邻扇区成组存放在磁盘上，因此很少移动磁头就可以快速检索到文件。当程序读或拷贝一个文件时，它通常从第一个字节到最后一个字节顺序地访问文件。因此，在处理进程对同一文件的一系列读请求时，可以从磁盘上很多相邻的扇区读取。

预读 (*read-ahead*) 是一种技术，这种技术在于在实际请求前读普通文件或块设备文件的几个相邻的数据页。在大多数情况下，预读能极大地提高磁盘的性能，因为预读使磁盘控制器处理较少的命令，其中的每条命令都涉及一大组相邻的扇区。此外，预读还能提高系统的响应能力。顺序读取文件的进程通常不需要等待请求的数据，因为请求的数据已经在 RAM 中了。

但是，预读对于随机访问的文件是没有用的；在这种情况下，预读实际上是有害的，因为它用无用的信息浪费了页高速缓存的空间。因此，当内核确定出最近所进行的 I/O 访问与前一次 I/O 访问不是顺序的时就减少或停止预读。

文件的预读需要更复杂的算法，这是由于以下几个原因：

- 由于数据是逐页进行读取的，因此预读算法不必考虑页内偏移量，只要考虑所访问的页在文件内部的位置就可以了。
- 只要进程持续地顺序访问一个文件，预读就会逐渐增加。
- 当前的访问与上一次访问不是顺序的时（随机访问），预读就会逐渐减少乃至禁止。
- 当一个进程重复地访问同一页（即只使用文件的很小一部分）时，或者当几乎所有的页都已在页高速缓存内时，预读就必须停止。
- 低级 I/O 设备驱动程序必须在合适的时候激活，这样当将来进程需要时，页已传送完毕。

如果请求的第一页紧跟上次访问所请求的最后一页，那么相对于上次的文件访问，内核把文件的这次访问看作是顺序的。

当访问给定文件时，预读算法使用两个页面集，各自对应文件的一个连续区域。这两个页面集分别叫做当前窗（*current window*）和预读窗（*ahead window*）。

当前窗内的页是进程请求的页和内核预读的页，且位于页高速缓存内（当前窗内的页不必是最新的，因为 I/O 数据传输仍可能在运行中）。当前窗包含进程顺序访问的最后一页，且可能有内核预读但进程未请求的页。

预读窗内的页紧接着当前窗内的页，它们是内核正在预读的页。预读窗内的页都不是进程请求的，但内核假定进程会迟早请求。

当内核认为是顺序访问而且第一页在当前窗内时，它就检查是否建立了预读窗。如果没有，内核创建一个预读窗并触发相应页的读操作。理想情况下，进程继续从当前窗请求页，同时预读窗的页则正在传送。当进程请求的页在预读窗，那么预读窗就成为当前窗。

预读算法使用的主要数据结构是 `file_ra_state` 描述符，它的字段见表 16-3。每个文件对象在它的 `f_ra` 字段中存放这样的一个描述符。

表 16-3: `file_ra_state` 描述符的字段

类型	字段	说明
unsigned long	start	当前窗内第一页的索引
unsigned long	size	当前窗内的页数（当临时禁止预读时为 -1，0 表示当前窗空）
unsigned long	flags	控制预读的一些标志
unsigned long	cache_hit	连续高速缓存命中数（进程请求的页同时又在页高速缓存内）
unsigned long	prev_page	进程请求的最后一页的索引
unsigned long	ahead_start	预读窗内第一页的索引
unsigned long	ahead_size	预读窗的页数（0 表示预读窗口空）
unsigned long	ra_pages	预读窗的最大页数（0 表示预读窗永久禁止）
unsigned long	mmap_hit	预读命中计数器（用于内存映射文件）
unsigned long	mmap_miss	预读失败计数器（用于内存映射文件）

当一个文件被打开时，在它的 `file_ra_state` 描述符中，除了 `prev_page` 和 `ra_pages` 这两个字段，其他的所有字段都置为 0。

`prev_page` 字段存放着进程在上一次读操作中所请求页的最后一页的索引。它的初值是 -1。

ra_pages字段表示当前窗的最大页数,即对该文件允许的最大预读量。该字段的初始值(缺省值)存放在该文件所在块设备的backing_dev_info描述符中(参见第十四章的“请求队列描述符”一节)。一个应用可以修改一个打开文件的ra_pages字段从而调整预读算法;具体的实现方法是调用posix_fadvise()系统调用,并传给它命令POSIX_FADV_NORMAL(设最大预读量为缺省值,通常是32页)、POSIX_FADV_SEQUENTIAL(设最大预读量为缺省值的两倍)和POSIX_FADV_RANDOM(最大预读量为0,从而永久禁止预读)。

flags字段内有两个重要的字段RA_FLAG_MISS和RA_FLAG_INCACHE。如果已被预读的页不在页高速缓存内(可能的原因是内核为了释放内存而加以收回了,参见第十七章),则第一个标志置位,这时候下一个要创建的预读窗大小将被缩小。当内核确定进程请求的最后256页都在页高速缓存内时(连续高速缓存命中数存放在ra->cache_hit字段中),第二个标志置位,这时内核认为所有的页都已在页高速缓存内,进而关闭预读。

何时执行预读算法?这有下列几种情形:

- 当内核用用户态请求来读文件数据的页时。这一事件触发page_cache_readahead()函数的调用(参见本章前面“从文件中读取数据”一节有关do_generic_file_read()描述的第4c步)。
- 当内核为文件内存映射分配一页时(参见本章后面“内存映射的请求调页”一节中的filemap_nopage()函数,它再次调用page_cache_readahead()函数)。
- 当用户态应用执行readahead()系统调用时,它会对某个文件描述符显式触发某预读活动。
- 当用户态应用使用POSIX_FADV_NOREUSE或POSIX_FADV_WILLNEED命令执行posix_fadvise()系统调用时,它会通知内核,某个范围的文件页不久将要被访问。
- 当用户态应用使用MADV_WILLNEED命令执行madvise()系统调用时,它会通知内核,某个文件内存映射区域中的给定范围的文件页不久将要被访问。

page_cache_readahead() 函数

page_cache_readahead()函数处理没有被特殊系统调用显式触发的所有预读操作。它填写当前窗和预读窗,根据预读命中数更新当前窗和预读窗的大小,也就是根据过去对文件访问预读策略的成功程度来调整。

当内核必须满足对某个文件一页或多页的读请求时,函数就被调用,该函数有下面五个参数:

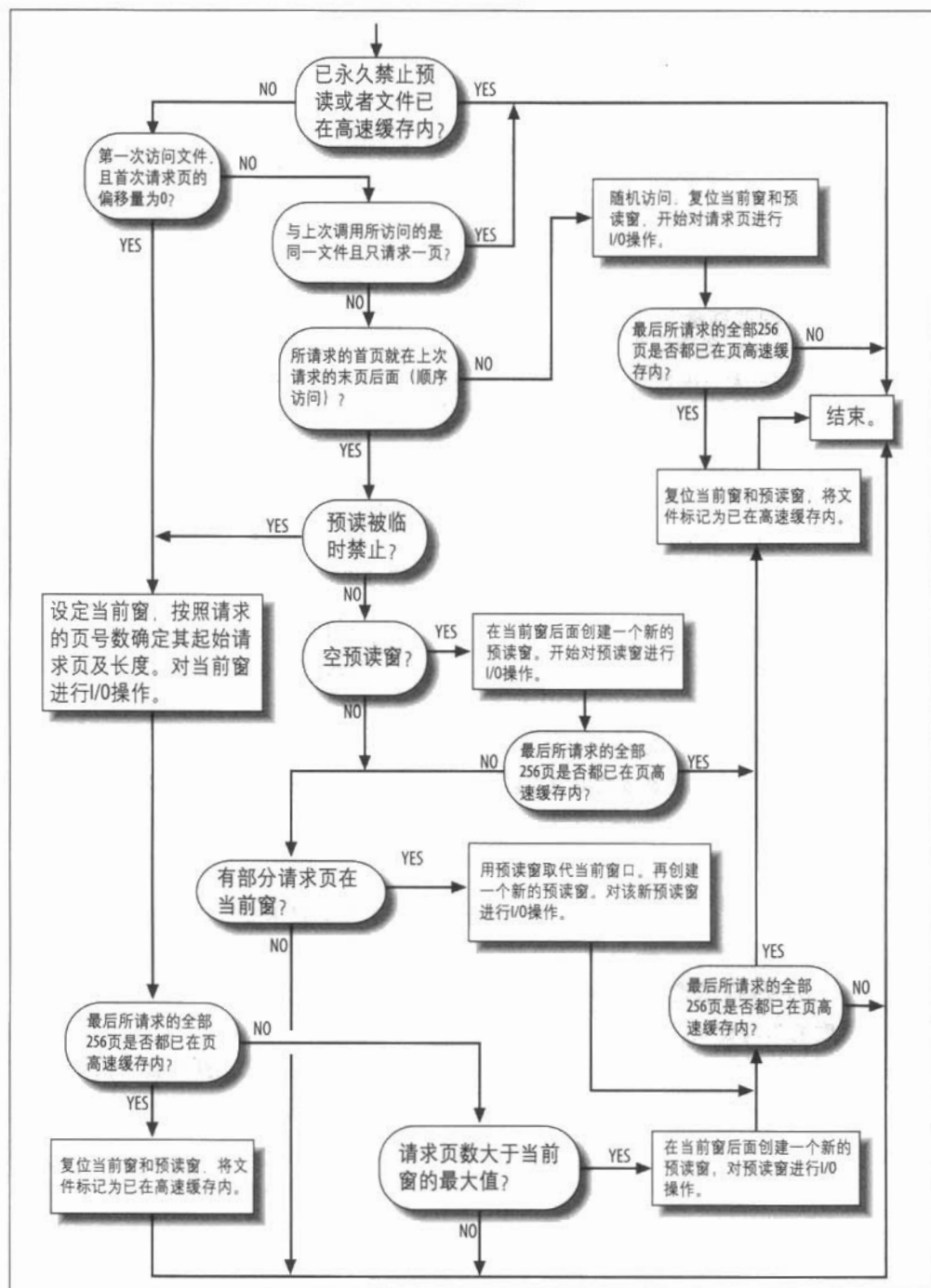
mapping	描述页所有者的 address_space 对象指针
ra	包含该页的文件 file_ra_state 描述符指针
filp	文件对象地址
offset	文件内页的偏移量
req_size	要完成当前读操作还需要读的页数（注3）

图 16-1 是 `page_cache_readahead()` 的流程图。该函数基本上作用于 `file_ra_state` 描述符的字段，因此，尽管流程图中的行为描述不很正规，你还是能很容易地确定函数执行的实际步骤。例如，为了检查请求页是否与刚读的页相同，函数检查 `ra->prev_page` 字段的值和 `offset` 参数的值是否一致（见前面的表 16-3）。

当进程第一次访问一个文件，并且其第一个请求页是文件中偏移量为 0 的页时，函数假定进程要进行顺序访问。那么，函数从第一页创建一个新的当前窗。初始当前窗的长度（总是为 2 的幂）与进程第一个读操作所请求的页数有一定的联系。请求页数越大，当前窗越大，一直到最大值，最大值存放在 `ra->ra_pages` 字段。反之，当进程第一次访问文件，但其第一个请求页在文件中的偏移量不为 0 时，函数假定进程不是执行顺序读。那么，函数暂时禁止预读（`ra->size` 字段设为 -1）。但是当预读暂时被禁止而函数又认为需要顺序访问时，将建立一个新的当前窗。

如果预读窗不存在，一旦函数认为在当前窗内进程执行了顺序读，则预读窗将被建立。预读窗总是从当前窗的最后一页开始。但它的长度与当前窗的长度相关：如果 `RA_FLAG_MISS` 标志置位，则预读窗长度是当前窗长度减 2，小于 4 时设为 4；否则，预读窗长度是当前窗长度的 4 倍或 2 倍。如果进程继续顺序访问文件，最终预读窗成为新的当前窗，新的预读窗被创建。这样，随着进程顺序地读文件，预读会大大地增强。

注 3：实际上，如果读操作要读的页数大于预读窗的最大尺寸，就会多次调用 `page_cache_readahead()` 函数。因此，`req_size` 参数可能比完成读操作还需要读的页数小。

图 16-1: 函数 `page_cache_readahead()` 的流程图

一旦函数认识到对文件的访问相对于上一次不是顺序的，当前窗与预读窗就被清空，预读被暂时禁止。当进程的读操作相对于上一次文件访问为顺序时，预读将重新开始。

每次 `page_cache_readahead()` 创建一个新窗，它就开始对所包含页的读操作。为了读一大组页，函数 `page_cache_readahead()` 调用 `blockable_page_cache_readahead()`。为减少内核开销，后面这个函数采用下面灵活的方法：

- 如果服务于块设备的请求队列是读拥塞的，就不进行读操作。
- 将要读的页与页高速缓存进行比较，如果该页已在页高速缓存内，跳过即可。
- 在从磁盘进行读之前，读请求所需的全部页框是一次性分配的。如果不能一次性得到全部页框，预读操作就只在可以得到的页上进行。而且把预读推迟至所有页框都得到时再进行并没有多大意义。
- 只要可能，通过使用多段 `bio` 描述符向通用块层发出读操作（参见第十四章“段”一节）。这通过 `address_space` 对象专用的 `readpages` 方法实现（假如已定义）；如果没有定义，就通过反复调用 `readpage` 方法来实现。`readpage` 方法在前面“从文件中读取数据”一节中对于单段情形有详细描述，但稍作修改就可以很容易地将其用于多段情形。

`handle_ra_miss()` 函数

在某些情况下，预读策略似乎不是十分有效，内核就必须修正预读参数。让我们考虑本章前面“从文件中读取数据”一节中描述的 `do_generic_file_read()` 函数。在第 4c 步中调用函数 `page_cache_readahead()`。图 16-1 中展示了两种情形：请求页在当前窗或预读窗表明它已经被预先读入了，或者还没有，则调用 `blockable_page_cache_readahead()` 来读入。在这两种情形下，函数 `do_generic_file_read()` 应该在第 4d 步中就在页高速缓存中找到了该页，如果没有，就表示该页框已被收回算法从高速缓存中删除。在这种情形下，`do_generic_file_read()` 调用 `handle_ra_miss()` 函数，这个函数会通过将 `RA_FLAG_MISS` 标志置位与 `RA_FLAG_INCACHE` 标志清 0 来调整预读算法。

写入文件

回想一下，`write()` 系统调用涉及把数据从调用进程的用户态地址空间中移动到内核数据结构中，然后再移动到磁盘上。文件对象的 `write` 方法允许每种文件类型都定义一个专用的写操作。在 Linux 2.6 中，每个磁盘文件系统的 `write` 方法都是一个过程，该过程主要标识写操作所涉及的磁盘块，把数据从用户态地址空间拷贝到页高速缓存的某些页中，然后把这页中的缓冲区标记成脏。

许多文件系统（包括Ext2或JFS）通过 `generic_file_write()` 函数来实现文件对象的 `write` 方法。它有如下参数：

`file`

文件对象指针

`buf`

用户态地址空间中的地址，必须从这个地址获取要写入文件的字符

`count`

要写入的字符个数

`ppos`

存放文件偏移量的变量地址，必须从这个偏移量处开始写入

该函数执行以下操作：

1. 初始化 `iovec` 类型的一个局部变量，它包含用户态缓冲区的地址与长度（参见本章前面“从文件读取数据”一节中对 `generic_file_read()` 函数的描述）。
2. 确定所写文件索引节点对象的地址 `inode (file->f_mapping->host)` 和获得信号量 (`inode->i_sem`)。有了这个信号量，一次只能有一个进程对某个文件发出 `write()` 系统调用。
3. 调用宏 `init_sync_kiocrb` 初始化 `kiocrb` 类型的局部变量。就像本章前面“从文件读取数据”一节中描述的那样，该宏将 `ki_key` 字段设置为 `KIOCB_SYNC_KEY`（同步 I/O 操作）、`ki_filp` 字段设置为 `filp`、`ki_obj` 字段设置为 `current`。
4. 调用 `_generic_file_aio_write_nolock()` 函数（见下面）将涉及的页标记为脏，并传递相应的参数：`iovec` 和 `kiocrb` 类型的局部变量地址、用户态缓冲区的段数（这里只有一个）和 `ppos`。
5. 释放 `inode->i_sem` 信号量。
6. 检查文件的 `O_SYNC` 标志、索引节点的 `S_SYNC` 标志及超级块的 `MS_SYNCHRONOUS` 标志。如果至少一个标志置位，则调用函数 `sync_page_range()` 来强制内核将页高速缓存中第 4 步涉及的所有页刷新，阻塞当前进程直到 I/O 数据传输结束。然后依次地，`sync_page_range()` 先执行 `address_space` 对象的 `writpages` 方法（如果有定义）或 `mpage_writpages()` 函数来开始这些脏页的 I/O 传输（参见本章后面“将脏页写到磁盘”一节），然后调用 `generic_osync_inode()` 将索引节点和相关的缓冲区刷新到磁盘，最后调用 `wait_on_page_bit()` 挂起当前进程一直到全部所刷新页的 `PG_writeback` 标志清 0。

7. 将 `__generic_file_aio_write_nolock()` 函数的返回值返回，通常是写入的有效字节数。

函数 `__generic_file_aio_write_nolock()` 接收四个参数：`kiocb` 描述符的地址 `iocb`、`iovec` 描述符数组的地址 `iov`、该数组的长度以及存放文件当前指针的变量的地址 `ppos`。当被 `generic_file_write()` 调用时，`iovec` 描述符数组只有一个元素，该元素描述待写数据的用户态缓冲区（注4）。

我们现在来解释 `__generic_file_aio_write_nolock()` 函数的行为。为简单起见，我们只讨论最常见的情形，即对有页高速缓存的文件进行 `write()` 系统调用的一般情况。我们在本章后面会讨论该函数在其他情况下的行为。我们不讨论如何处理错误和异常条件。

该函数执行如下步骤：

1. 调用 `access_ok()` 确定 `iovec` 描述符所描述的用户态缓冲区是有效的（起始地址和长度已从服务例程 `sys_write()` 得到，因此使用前必须对其检查。参见第十章“验证参数”一节）。如果参数无效，则返回错误 `-EFAULT`。
2. 确定待写文件（`file->f_mapping->host`）索引节点对象的地址 `inode`。记住：如果文件是一个块设备文件，这就是一个 `bdev` 特殊文件系统的索引节点（参见第十四章）。
3. 将文件（`file->f_mapping->backing_dev_info`）的 `backing_dev_info` 描述符的地址设为 `current->backing_dev_info`。实际上，即使相应请求队列是拥塞的，这个设置也会允许当前进程写回由 `file->f_mapping` 拥有的脏页（参见第十七章）。
4. 如果 `file->flags` 的 `O_APPEND` 标志置位而且文件是普通文件（非块设备文件），它将 `*ppos` 设为文件尾，从而新数据将都追加到文件的后面。
5. 对文件大小进行几次检查。比如，写操作不能把一个普通文件增大到超过每用户的上限或文件系统的上限，每用户上限存放在 `current->signal->rlim[RLIMIT_FSIZE]`（参见第三章“进程资源限制”一节），文件系统上限存放在 `inode->i_sb->s_maxbytes`。另外，如果文件不是“大型文件”（当 `file->f_flags` 的 `O_LARGEFILE` 标志清0时），那么它的大小不能超出2GB。如果没有设定所述限制，它就减少待写字节数。

注4：系统调用 `write()` 的一个叫做 `writew()` 的变体允许应用程序定义多个用户态缓冲区，从中可以获取待写入文件的数据。函数 `generic_file_aio_write_nolock()` 也具有这种功能。下面我们假设将从一个用户缓冲区取数据，不过我们可以想象，使用多个缓冲区虽然简单，但需要执行更多的步骤。

6. 如果设定, 则将文件的 `suid` 标志清 0, 而且如果是可执行文件的话就将 `sgid` 标志也清 0 (参见第一章“访问权限和文件模式”一节)。我们并不要用户能修改 `setuid` 文件。
7. 将当前时间存放在 `inode->mtime` 字段 (文件写操作的最新时间) 中, 也存放在 `inode->ctime` 字段 (修改索引节点的最新时间) 中, 而且将索引节点对象标记为脏。
8. 开始循环以更新写操作中涉及的所有文件页。在每次循环期间, 执行下列子步骤:
 - a. 调用 `find_lock_page()` 在页高速缓存中搜索该页 (参见第十五章“页高速缓存的处理函数”一节)。如果函数找到了该页, 则增加引用计数并将 `PG_locked` 标志置位。
 - b. 如果该页不在页高速缓存中, 则分配一个新页框并调用 `add_to_page_cache()` 在页高速缓存内插入此页。正如第十五章“页高速缓存的处理函数”一节所述的那样, 这个函数也会增加引用计数并将 `PG_locked` 标志置位。另外函数还在内存管理区的非活动链表中插入一页 (参见第十七章)。
 - c. 调用索引节点 (`file->f_mapping`) 中 `address_space` 对象的 `prepare_write` 方法。对应的函数会为该页分配和初始化缓冲区首部。我们在后面的章节中再讨论该函数对于普通文件和块设备文件做些什么。
 - d. 如果缓冲区在高端内存中, 则建立用户态缓冲区的内核映射 (参见第八章的“高端内存页框的内核映射”一节), 然后它调用 `__copy_from_user()` 把用户态缓冲区中的字符拷贝到页中, 并且释放内核映射。
 - e. 调用索引节点 (`file->f_mapping`) 中 `address_space` 对象的 `commit_write` 方法。对应的函数把基础缓冲区标记为脏, 以便随后把它们写到磁盘。我们在后面两节讨论该函数对于普通文件和块设备文件做些什么。
 - f. 调用 `unlock_page()` 清 `PG_locked` 标志, 并唤醒等待该页的任何进程。
 - g. 调用 `mark_page_accessed()` 来为内存回收算法更新页状态 [参见第十七章“最近最少使用 (LRU) 链表”一节]。
 - h. 减少页引用计数来撤销第 8a 或 8b 步中的增加值。
 - i. 在这一步, 还有另一页被标记为脏, 它检查页高速缓存中脏页比例是否超过一个固定的阈值 (通常为系统中页的 40%)。如果这样, 则调用 `writeback_inodes()` 来刷新几十页到磁盘 (参见第十五章的“搜索要刷新的脏页”一节)。

- j. 调用 `cond_resched()` 来检查当前进程的 `TIF_NEED_RESCHED` 标志。如果该标志置位，则调用 `schedule()` 函数。
- 9. 现在，在写操作中所涉及的文件的所有页都已处理。更新 `*ppos` 的值，让它正好指向最后一个被写入的字符之后的位置。
- 10. 设置 `current->backing_dev_info` 为 `NULL`（参见第 3 步）。
- 11. 返回写入文件的有效字符数后结束。

普通文件的 `prepare_write` 和 `commit_write` 方法

`address_space` 对象的 `prepare_write` 和 `commit_write` 方法专用于由 `generic_file_write()` 实现的通用写操作，这个函数适用于普通文件和块设备文件。对文件的受写操作影响的每一页，调用一次这两个方法。

每个磁盘文件系统都定义了自己的 `prepare_write` 方法。与读操作类似，这个方法只不过是普通函数的一个封装函数。例如，`Ext2` 文件系统通过下列函数实现 `prepare_write` 方法：

```
int ext2_prepare_write(struct file *file, struct page *page, unsigned
from, unsigned to)
{
    return block_prepare_write(page, from, to, ext2_get_block);
}
```

在前面“从文件读取数据”一节已经提到 `ext2_get_block()` 函数；它把相对于文件的块号转换为逻辑块号（表示数据在物理块设备上的位置）。

`block_prepare_write()` 函数通过执行下列步骤为文件页的缓冲区和缓冲区首部做准备：

1. 检查某页是否是一个缓冲区页（如果是则 `PG_Private` 标志置位）；如果该标志清 0，则调用 `create_empty_buffers()` 为页中所有的缓冲区分配缓冲区首部（参见第十五章“缓冲区页”一节）。
2. 对与页中包含的缓冲区对应的每个缓冲区首部，及受写操作影响的每个缓冲区首部，执行下列操作：
 - a. 如果 `BH_New` 标志置位，则将它清 0（参见下面）。
 - b. 如果 `BH_New` 标志已清 0，则函数执行下列子步骤：
 - (1) 调用依赖于文件系统的函数，该函数的地址 `get_block` 以参数形式传递过来。查看这个文件系统磁盘数据结构并查找缓冲区的逻辑块号（相对于磁

盘分区的起始位置而不是普通文件的起始位置)。与文件系统相关的函数把这个数存放在对应缓冲区首部的`b_blocknr`字段,并设置它的`BH_Mapped`标志。与文件系统相关的函数可能为文件分配一个新的物理块(例如,如果访问的块掉进普通文件的一个“洞”中,参见第十八章“文件的洞”一节)。在这种情况下,设置`BH_New`标志。

- (2) 检查`BH_New`标志的值;如果它被置位,则调用`unmap_underlying_metadata()`来检查页高速缓存内的某个块设备缓冲区页是否包含指向磁盘同一块的一个缓冲区(注5)。该函数实际上调用`__find_get_block()`在页高速缓存内查找一个旧块(参见第十五章“在页高速缓存中搜索块”一节)。如果找到一块,函数将`BH_Dirty`标志清0并等待直到该缓冲区的I/O数据传输完毕。此外,如果写操作不对整个缓冲区进行重写,则用0填充未写区域。然后考虑页中的下一个缓冲区。
 - c. 如果写操作不对整个缓冲区进行重写且它的`BH_Delay`和`BH_Uptodate`标志未置位(也就是说,已在磁盘文件系统数据结构中分配了块,但是RAM中的缓冲区并没有有效的数据映像),函数对该块调用`ll_rw_block()`从磁盘读取它的内容(参见第十五章“向通用块层提交缓冲区首部”一节)。
3. 阻塞当前进程,直到在第2c步触发的所有读操作全部完成。
 4. 返回0。

一旦`prepare_write`方法返回,`generic_file_write()`函数就用存放在用户态地址空间中的数据更新页。接下来,调用`address_space`对象的`commit_write`方法。这个方法由`generic_commit_write()`函数实现,几乎适用于所有非日志型磁盘文件系统。

`generic_commit_write()`函数执行下列步骤:

1. 调用`__block_commit_write()`函数,然后依次执行如下步骤:
 - a. 考虑页中受写操作影响的所有缓冲区;对于其中的每个缓冲区,将对应缓冲区首部的`BH_Uptodate`和`BH_Dirty`标志置位。
 - b. 标记相应索引节点为脏,正如第十五章“搜索要刷新的脏页”一节中所述,这需要将索引节点加入超级块脏的索引节点链表。
 - c. 如果缓冲区页中的所有缓冲区是最新的,则将`PG_uptodate`标志置位。

注5: 尽管可能性不大,但在一个用户直接向块设备文件写数据块时,还是会出现这种情况,从而越过文件系统。

- d. 将页的 PG_dirty 标志置位，并在基树中将页标记成脏（参见第十五章“基树”一节）。
2. 检查写操作是否将文件增大。如果增大，则更新文件索引节点对象的 i_size 字段。
3. 返回 0。

块设备文件的 prepare_write 和 commit_write 方法

写入块设备文件的操作非常类似于对普通文件的相应操作。事实上，块设备文件的 address_space 对象的 prepare_write 方法通常是由下列函数实现的：

```
int blkdev_prepare_write(struct file *file, struct page *page, unsigned
from, unsigned to)
{
    return block_prepare_write(page, from, to, blkdev_get_block);
}
```

你可以看到，这个函数只不过是前一节讨论过的 block_prepare_write() 函数的封装函数。当然，唯一的差异是第二个参数，它是一个指向函数的指针，该函数必须把相对于文件开始处的文件块号转换为相对于块设备开始处的逻辑块号。回想一下，对于块设备文件来说，这两个数是一致的（参见前面“从文件读取数据”一节中对 blkdev_get_block() 函数的讨论）。

用于块设备文件的 commit_write 方法是由下列简单的封装函数实现的：

```
int blkdev_commit_write(struct file *file, struct page *page, unsigned
from, unsigned to)
{
    return block_commit_write(page, from, to);
}
```

正如你所看到的，用于块设备的 commit_write 方法与用于普通文件的 commit_write 方法本质上做同样的事情（我们在前一节描述了 block_commit_write() 函数）。唯一的差异是这个方法不检查写操作是否扩大了文件，你根本不可能在块设备文件的末尾追加字符来扩大它。

将脏页写到磁盘

系统调用 write() 的作用就是修改页高速缓存内一些页的内容，如果页高速缓存内没有所要的页则分配并追加这些页。某些情况下（例如文件带 O_SYNC 标志打开），I/O 数据传输立即启动（参见本章前面“写入文件”一节中 generic_file_write() 函数的第 6 步）。但是通常 I/O 数据传输是延迟进行的，这在第十五章的“把脏页写入磁盘”一节中描述过。

当内核要有效启动 I/O 数据传输时，就要调用文件 `address_space` 对象的 `writpages` 方法，它在基树中寻找脏页，并把它们刷新到磁盘。例如 Ext2 文件系统通过下面的函数实现 `writpages` 方法：

```
int ext2_writpages(struct address_space *mapping, struct
writeback_control *wbc)
{
    return mpage_writpages(mapping, wbc, ext2_get_block);
}
```

你可以看到，该函数是通用 `mpage_writpages()` 的一个简单的封装函数。事实上，若文件系统没有定义 `writpages` 方法，内核则直接调用 `mpage_writpages()` 并把 `NULL` 传给第三个参数。`ext2_get_block()` 函数在前面“从文件读取数据”一节中已讲到过，这是一个依赖于文件系统的函数，它将文件块号转换成逻辑块号。

`writeback_control` 数据结构是一个描述符，它控制 `writeback` 写回操作如何执行，我们在第十五章“搜索要被刷新的脏页”一节中已有描述。

`mpage_writpages()` 函数执行下列步骤：

1. 如果请求队列写阻塞，但进程不希望阻塞，则不向磁盘写任何页就返回。
2. 确定文件的首页，如果 `writeback_control` 描述符给定一个文件内的初始位置，函数将把它转换成页索引。否则，如果 `writeback_control` 描述符指定进程无需等待 I/O 数据传输结束，它将 `mapping->writeback_index` 的值设为初始页索引（即从上一个写回操作的最后一页开始扫描）。最后，如果进程必须等待 I/O 数据传输完毕，则从文件的第一页开始扫描。
3. 调用 `find_get_pages_tag()` 在页高速缓存中查找脏页描述符（参见第十五章“基树的标记”一节）。
4. 对上一步得到的每个页描述符，执行如下步骤：
 - a. 调用 `lock_page()` 来锁定该页。
 - b. 确认页是有效的并在页高速缓存内（因为另一个内核控制路径可能已在第 3 步与第 4a 步间作用于该页）。
 - c. 检查页的 `PG_writeback` 标志。如果置位，表明页已被刷新到磁盘。如果进程必须等待 I/O 数据传输完毕，则调用 `wait_on_page_bit()` 在 `PG_writeback` 清 0 之前一直阻塞当前进程；当函数结束时，以前运行的任何 `writeback` 操作都被终止。否则，如果进程无需等待，它将检查 `PG_dirty` 标志：如果 `PG_dirty` 标志现已清 0，则正在运行的写回操作将处理该页，将它解锁并跳回第 4a 步继续下一页。

- d. 如果 `get_block` 的参数是 `NULL` (没有定义 `writpage` 方法), 它将调用文件 `address_space` 对象的 `mapping->writepage` 方法将页刷新到磁盘。否则, 如果 `get_block` 的参数不是 `NULL`, 它就调用 `mpage_writepage()` 函数。详见第 8 步。
5. 调用 `cond_resched()` 来检查当前进程的 `TIF_NEED_RESCHED` 标志, 如果该标志置位就调用 `schedule()` 函数。
6. 如果函数没有扫描完给定范围内的所有页, 或者写到磁盘的有效页数小于 `writeback_control` 描述符中原先的给定值, 那么跳回第 3 步。
7. 如果 `writeback_control` 描述符没有给定文件内的初始位置, 它将最后一个扫描页的索引值赋给 `mapping->writeback_index` 字段。
8. 如果在第 4d 步中调用了 `mpage_writepage()` 函数, 而且返回了 `bio` 描述符地址, 那么调用 `mpage_bio_submit()` (见下面)。

像 Ext2 这样的典型文件系统所实现的 `writpage` 方法是一个通用的 `block_write_full_page()` 函数的封装函数, 并将依赖于文件系统的 `get_block` 函数的地址作为参数传给它。就像本章前面“从文件读取数据”一节描述的 `block_read_full_page()` 一样, `block_write_full_page()` 函数也依次执行: 分配页缓冲区首部 (如果还不在于缓冲区页中), 对每页调用 `submit_bh()` 函数来指定写操作。就块设备文件而言, 就用 `block_write_full_page()` 的封装函数 `blkdev_writpage()` 实现 `writpage` 方法。

许多非日志型文件系统依赖于 `mpage_writepage()` 函数而不是自定义的 `writpage` 方法。这样能改善性能, 因为 `mpage_writepage()` 函数进行 I/O 传输时, 在同一个 `bio` 描述符中聚集尽可能多的页。这就使得块设备驱动程序能利用现代硬盘控制器的 DMA 分散-聚集能力。

长话短说, `mpage_writepage()` 函数将检查: 待写页包含的块在磁盘上是否不相邻, 该页是否包含文件洞, 页上的某块是否没有脏或不是最新的。如果以上情况至少一条成立, 函数就像上面那样仍然用依赖于文件系统的 `writpage` 方法。否则, 将页追加为 `bio` 描述符的一段。 `bio` 描述符的地址将作为参数被传给函数, 如果该地址为 `NULL`, `mpage_writepage()` 将初始化一个新的 `bio` 描述符并将地址返回给调用函数, 调用函数转而在未来调用 `mpage_writepage()` 时再将该地址传回来。这样, 同一个 `bio` 可以加载几个页。如果 `bio` 中某页与上一个加载页不相邻, `mpage_writepage()` 就调用 `mpage_bio_submit()` 开始该 `bio` 的 I/O 数据传输, 并为该页分配一个新的 `bio`。

`mpage_bio_submit()` 函数将 `bio` 的 `bi_end_io` 方法设为 `mpage_end_io_write()` 的地址; 然后调用 `submit_bio()` 开始传输 (参见第十五章“向通用块层提交缓冲区首部”

一节)。一旦数据传输成功结束,完成函数 `mpage_end_io_write()` 就唤醒那些等待页传输结束的进程,并清除 `bio` 描述符。

内存映射

正如我们在第九章的“线性区”一节中已经介绍过的一样,一个线性区可以和磁盘文件系统的普通文件的某一部分或者块设备文件相关联。这就意味着内核把对区线性中页内某个字节的访问转换成对文件中相应字节的操作。这种技术称为内存映射 (*memory mapping*)。

有两种类型的内存映射:

共享型

在线性区页上的任何写操作都会修改磁盘上的文件,而且,如果进程对共享映射中的一个页进行写,那么这种修改对于其他映射了这同一文件的所有进程来说都是可见的。

私有型

当进程创建的映射只是为读文件,而不是写文件时才会使用此种映射。出于这种目的,私有映射的效率要比共享映射的效率更高。但是对私有映射页的任何写操作都会使内核停止映射该文件中的页。因此,写操作既不会改变磁盘上的文件,对访问相同文件的其他进程也不可见。但是私有内存映射中还没有被进程改变的页面因为其他进程进行的文件更新而更新。

进程可以发出一个 `mmap()` 系统调用来创建一个新的内存映射(参见本章后面的“创建内存映射”一节)。程序员必须指定一个 `MAP_SHARED` 标志或 `MAP_PRIVATE` 标志作为这个系统调用的参数。正如你可以猜到的那样,前一种情况下,映射是共享的,而后一种情况下,映射是私有的。一旦创建了这种映射,进程就可以从这个新线性区的内存单元读取数据,也就等价于读取了文件中存放的数据。如果这个内存映射是共享的,那么进程可以通过对相同的内存单元进行写而达到修改相应文件的目的。为了撤消或者缩小一个内存映射,进程可以使用 `munmap()` 系统调用(参见后面的“撤消内存映射”一节)。

作为一条通用规则,如果一个内存映射是共享的,相应的线性区就设置了 `VM_SHARED` 标志;如果一个内存映射是私有的,那么相应的线性区就清除了 `VM_SHARED` 标志。正如我们在后面会看到的一样,对于只读共享内存映射来说,有一个特例不符合本规则。

内存映射的数据结构

内存映射可以用下列数据结构的组合来表示:

- 与所映射的文件相关的索引节点对象
- 所映射文件的 `address_space` 对象
- 不同进程对一个文件进行不同映射所使用的文件对象
- 对文件进行每一不同映射所使用的 `vm_area_struct` 描述符
- 对文件进行映射的线性区所分配的每个页框所对应的页描述符

图 16-2 说明了这些数据结构是如何链接在一起的。图的左边给出了标识文件的索引节点。每个索引节点对象的 `i_mapping` 字段指向文件的 `address_space` 对象。每个 `address_space` 对象的 `page_tree` 字段又指向该地址空间的页的基树（参见第十五章“基树”一节），而 `i_mmap` 字段指向第二棵树，叫做 radix 优先级搜索树（priority search tree, PST），这种树由地址空间的线性区组成。PST 的主要作用是为了执行“反向映射”，这是为了快速标识共享一页的所有进程。我们将在下一章中详细讨论 PST，因为它们用于页框回收。对同一文件的文件对象和索引节点之间链接的建立是通过 `f_mapping` 字段达到的。

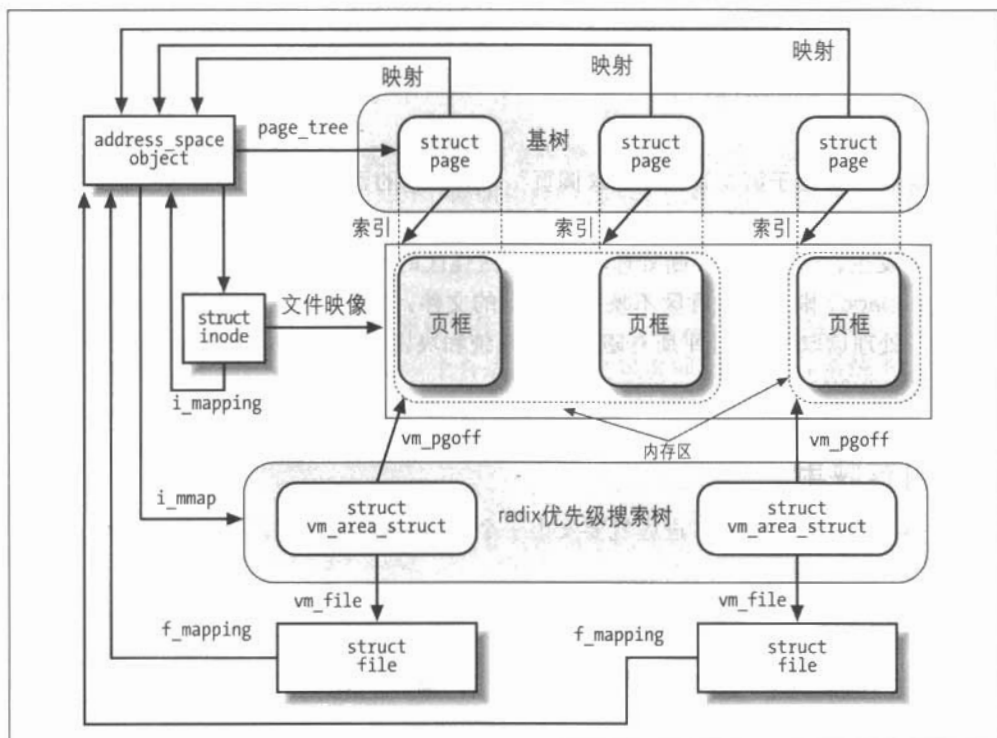


图 16-2: 文件内存映射的数据结构

每个线性区描述符都有一个 `vm_file` 字段，与所映射文件的文件对象链接（如果该字段为 `NULL`，则线性区没有用于内存映射）。第一个映射单元的位置存放在线性区描述符的 `vm_pgoff` 字段，它表示以页大小为单位的偏移量。所映射的文件那部分的长度就是线性区的大小，这可以从 `vm_start` 和 `vm_end` 字段计算出来。

共享内存映射的页通常都包含在页高速缓存中；私有内存映射的页只要还没有被修改，也都包含在页高速缓存中。当进程试图修改一个私有内存映射的页时，内核就把该页框进行复制，并在进程页表中用复制的页来替换原来的页框，这是第八章中介绍的写时复制机制的应用之一。虽然原来的页框还仍然在页高速缓存中，但不再属于这个内存映射，这是由于被复制的页框替换了原来的页框。依次类推，这个复制的页框不会被插入到页高速缓存中，因为其中所包含的数据不再是磁盘上表示那个文件的有效数据。

图16-2还显示了包含在页高速缓存中的几个指向内存映射文件的页的页描述符。注意图中的第一个线性区有三页，但是只为它分配了两个页框；猜想一下，大概是拥有这个线性区的进程从没有访问过第三页。

对每个不同的文件系统，内核提供了几个钩子（hook）函数来定制其内存映射机制。内存映射实现的核心委托给文件对象的 `mmap` 方法。对于大多数磁盘文件系统和块设备文件，这个方法是由叫做 `generic_file_mmap()` 的通用函数实现的，该函数将在下一节进行描述。

文件内存映射依赖于第九章的“请求调页”一节描述的请求调页机制。事实上，一个新建立的内存映射就是一个不包含任何页的线性区。当进程引用线性区中的一个地址时，缺页异常发生，缺页异常中断处理程序检查线性区的 `nopage` 方法是否被定义。如果没有定义 `nopage`，则说明线性区不映射磁盘上的文件；否则，进行映射，这个方法通过访问块设备处理读取的页。几乎所有磁盘文件系统和块设备文件都通过 `filemap_nopage()` 函数实现 `nopage` 方法。

创建内存映射

要创建一个新的内存映射，进程就要发出一个 `mmap()` 系统调用，并向该函数传递以下参数：

- 文件描述符，标识要映射的文件。
- 文件内的偏移量，指定要映射的文件部分的第一个字符。
- 要映射的文件部分的长度。

- 一组标志。进程必须显式地设置MAP_SHARED标志或MAP_PRIVATE标志来指定所请求的内存映射的种类（注6）。
- 一组权限，指定对线性区进行访问的一种或者多种权限：读访问（PROT_READ）、写访问（PROT_WRITE）或执行访问（PROT_EXEC）。
- 一个可选的线性地址，内核把该地址作为新线性区应该从哪里开始的一个线索。如果指定了MAP_FIXED标志，且内核不能从指定的线性地址开始分配新线性区，那么这个系统调用失败。

mmap()系统调用返回新线性区中第一个单元位置的线性地址。为了兼容起见，在80 × 86体系结构中，内核在系统调用表中为mmap()保留两个表项：一个在索引90处，一个在索引192处。前一个表项对应于old_mmap()服务例程（由老的C库使用），而后一个表项对应于sys_mmap2()服务例程（由新近的C库使用）。这两个服务例程仅在如何传递系统调用的第6个参数时有所差异。这两个服务例程都调用do_mmap_pgoff()函数（参见第九章“分配线性地址区间”一节）。我们现在就详细介绍当创建对文件进行映射的线性区时执行的步骤。我们所讨论的是do_mmap_pgoff()的file参数（文件对象指针）非空的情形。为清楚起见，我们要列举描述do_mmap_pgoff()的步骤，并指出在新条件下执行的其他步骤。

步骤1

检查是否为要映射的文件定义了mmap文件操作。如果没有，就返回一个错误码。文件操作表中的mmap值为NULL说明相应的文件不能被映射（例如，因为这是一个目录）。

步骤2

函数get_unmapped_area()调用文件对象的get_unmapped_area方法，如果已定义，就为文件的内存映射分配一个合适的线性地址区间。磁盘文件系统不会定义这个方法，那么像第九章“线性区的处理”一节描述的那样，get_unmapped_area()函数就调用内存描述符的get_unmapped_area方法。

步骤3

除了进行正常的一致性检查之外，还要对所请求的内存映射的种类（存放在mmap()

注6： 进程可以设置MAP_ANONYMOUS标志来指定新线性区是匿名的，也就是说，与任何基于磁盘的文件都无关（参见第九章的“请求调页”一节）。进程也可以创建具有MAP_SHARED和MAP_ANONYMOUS标志的线性区。在这种情况下，线性区在tmpfs文件系统中映射一个特殊的文件（参见第十九章的“IPC共享内存”一节），这个文件可以由创建进程的所有后代来访问。

系统调用的 `flags` 参数中) 与在打开文件时所指定的标志 (存放在 `file` → `f-mode` 字段中) 进行比较。根据这两个消息源, 执行以下的检查:

- 如果请求一个共享可写的内存映射, 就检查文件是为写入而打开的, 而不是以追加模式打开的 (`open()` 系统调用的 `O_APPEND` 标志)。
- 如果请求一个共享内存映射, 就检查文件上没有强制锁 (参见第十二章中的“文件加锁”一节)。
- 对于任何种类的内存映射, 都要检查文件是为读操作而打开的。

如果以上这些条件都不能满足, 就返回一个错误码。

另外, 当初始化非线性区描述符的 `vm_flags` 字段时, 要根据文件的访问权限和所请求的内存映射的种类设置 `VM_READ`、`VM_WRITE`、`VM_EXEC`、`VM_SHARED`、`VM_MAYREAD`、`VM_MAYWRITE`、`VM_MAYEXEC` 和 `VM_MAYSHARE` 标志 (参见第九章“线性区访问权限”一节)。最佳情况下, 对于不可写共享内存映射, 标志 `VM_SHARED` 和 `VM_MAYWRITE` 清 0。可以这样处理是因为不允许进程写入这个线性区的页, 因此, 这种映射的处理就与私有映射的处理相同。但是, 内核实际上允许共享该文件的其他进程读这个线性区中的页。

步骤 4

用文件对象的地址初始化线性区描述符的 `vm_file` 字段, 并增加文件的引用计数器。对映射的文件调用 `mmap` 方法, 将文件对象地址和线性区描述符地址作为参数传给它。对于大多数文件系统, 该方法由 `generic_file_mmap()` 实现, 它执行下列步骤:

- a. 将当前时间赋给文件索引节点对象的 `i_atime` 字段, 并将该索引节点标记为脏。
- b. 用 `generic_file_vm_ops` 表的地址初始化线性区描述符的 `vm_ops` 字段。在这个表中的方法, 除了 `nopage` 和 `populate` 方法外, 其他所有都为空。`nopage` 方法由 `filemap_nopage()` 实现, 而 `populate` 方法由 `filemap_populate()` 实现 (参见本章后面的“非线性内存映射”一节)。

步骤 5

增加文件索引节点对象 `i_writecount` 字段的值, 该字段就是写进程的引用计数器。

撤消内存映射

当进程准备撤消一个内存映射时, 就调用 `munmap()`; 该系统调用还可用于减少每种内存区的大小。给它传递的参数如下:

- 要删除的线性地址区间中第一个单元的地址。
- 要删除的线性地址区间的长度。

该系统调用的 `sys_munmap()` 服务例程实际上是调用 `do_munmap()` 函数，该函数在第九章的“释放线性地址区间”一节已有描述。注意，不需要将待撤销可写共享内存映射中的页刷新到磁盘。实际上，因为这些页仍然在页高速缓存内，因此继续起磁盘高速缓存的作用。

内存映射的请求调页

出于效率的原因，内存映射创建之后并没有立即把页框分配给它，而是尽可能向后推迟到不能再推迟——也就是说，当进程试图对其中的一页进行寻址时，就产生一个“缺页”异常。

我们在第九章中的“缺页异常处理程序”一节中已经看到，内核是如何验证缺页所在的地址是否包含在某个进程的线性区中的。如果是这样，那么内核就检查这个地址所对应的页表项，如果表项为空就调用 `do_no_page()` 函数（参见第九章的“请求调页”一节）。

`do_no_page()` 函数执行对请求调页的所有类型都通用的操作，例如分配页框和更新页表。它还检查所涉及的线性区是否定义了 `nopage` 方法。在第九章的“请求调页”一节中，我们已经介绍了这个方法没有定义的情况（匿名线性区）。现在我们讨论当 `nopage` 方法被定义时，`do_no_page()` 所执行的主要操作：

1. 调用 `nopage` 方法，它返回包含所请求页的页框的地址。
2. 如果进程试图对页进行写入而该内存映射是私有的，则通过把刚读取的页拷贝一份并把它插入页的非活动链表中避免进一步的“写时复制”异常（参见第十七章）。如果私有内存映射区域还没有一个包含新页的被动匿名线性区（slave anonymous memory region），它要么追加一个新的被动匿名线性区，要么增大现有的（参见第九章的“线性区”一节）。在下面的步骤中，该函数使用新页而不是 `nopage` 方法返回的页，所以后者不会被用户态进程修改。
3. 如果某个其他进程删改或作废了该页（`address_space` 描述符的 `truncate_count` 字段就是用于这种检查的），函数将跳回第 1 步，尝试再次获得该页。
4. 增加进程内存描述符的 `rss` 字段，表示一个新页框已分配给进程。
5. 用新页框的地址以及线性区的 `vm_page_prot` 字段中所包含的页访问权来设置缺页所在的地址对应的页表项。
6. 如果进程试图对这个页进行写入，则把页表项的 `Read/Write` 和 `Dirty` 位强制置为

1. 在这种情况下, 或者把这个页框互斥地分配给进程, 或者让页成为共享; 在这两种情况下, 都应该允许对这个页进行写入。

请求调页算法的核心在于线性区的 `nopage` 方法。一般来说, 该方法必须返回进程所访问页所在的页框地址。其实现依赖于页所在线性区的种类。

在处理对磁盘文件进行映射的线性区时, `nopage` 方法必须首先在页高速缓存中查找所请求的页。如果没有找到相应的页, 这个方法就必须将其从磁盘上读入。大部分文件系统都是使用 `filemap_nopage()` 函数来实现 `nopage` 方法的, 该函数接收三个参数:

`area`

所请求页所在线性区的描述符地址。

`address`

所请求页的线性地址。

`type`

存放函数侦测到的缺页类型 (`VM_FAULT_MAJOR` 或 `VM_FAULT_MINOR`) 的变量的指针。

`filemap_nopage()` 函数执行以下步骤:

1. 从 `area->vm_file` 字段得到文件对象地址 `file`; 从 `file->f_mapping` 得到 `address_space` 对象地址; 从 `address_space` 对象的 `host` 字段得到索引节点对象地址。
2. 用 `area` 的 `vm_star` 和 `vm_pgoff` 字段来确定从 `address` 开始的页对应的数据在文件中的偏移量。
3. 检查文件偏移量是否大于文件大小。如果是, 就返回 `NULL`, 这就意味着分配新页失败, 除非缺页是由调试程序通过 `ptrace()` 系统调用跟踪另一个进程引起的, 我们打算讨论这种特殊情况。
4. 如果线性区的 `VM_RAND_READ` 标志置位 (见下面), 我们假定进程以随机方式读内存映射中的页, 那么它忽略预读, 跳到第 10 步。
5. 如果线性区的 `VM_SEQ_READ` 标志置位 (见下面), 我们假定进程以严格顺序方式读内存映射中的页, 那么它调用 `page_cache_readahead()` 从缺页处开始预读 (参见本章前面“文件的预读”一节)。
6. 调用 `find_get_page()`, 在页高速缓存内寻找由 `address_space` 对象和文件偏移量标识的页。如果没找到这样的页, 跳到第 11 步。

7. 如果函数运行至此,说明没在页高速缓存内找到页,检查内存区的VM_SEQ_READ标志:
 - a. 如果标志置位,内核将强行预读线性区中的页,从而预读算法失败,它就调用handle_ra_miss()来调整预读参数(参见本章前面“文件的预读”一节),并跳到第10步。
 - b. 否则,如果标志未置位,将文件file_ra_state描述符中的mmap_miss计数器加1。如果失败数远大于命中数(存放在mmap_hit计数器内),它将忽略预读,跳到第10步。
8. 如果预读没有永久禁止(file_ra_state描述符的ra_pages字段大于0),它将调用do_page_cache_readahead(),读入围绕请求页的一组页。
9. 调用find_get_page()来检查请求页是否在页高速缓存中,如果在,则跳到第11步。
10. 调用page_cache_read()。这个函数检查请求页是否在页高速缓存中,如果不在,则分配一个新页框,把它追加到页高速缓存,执行mapping->a_ops->readpage方法,安排一个I/O操作从磁盘读入该页内容。
11. 调用grab_swap_token()函数,尽可能为当前进程分配一个交换标记(参见第十七章“交换标记”一节)。
12. 请求页现已在页高速缓存内,将文件file_ra_state描述符的mmap_hit计数器加1。
13. 如果页不是最新的(标志PG_uptodate flag未置位),就调用lock_page()锁定该页,执行mapping->a_ops->readpage方法来触发I/O数据传输,调用wait_on_page_bit()后睡眠,一直等到该页被解锁,就是说等到数据传输完成。
14. 调用mark_page_accessed()来标记请求页为访问过(参见下一章)。
15. 如果在页高速缓存内找到该页的最新版,将*type设为VM_FAULT_MINOR,否则设为VM_FAULT_MAJOR。
16. 返回请求页地址。

用户态进程可以通过madvise()系统调用来调整filemap_nopage()函数的预读行为。MADV_RANDOM命令将线性区的VM_RAND_READ标志置位,从而指定以随机方式访问线性区的页。MADV_SEQUENTIAL命令将线性区的VM_SEQ_READ标志置位,从而指定以严格顺序方式访问页。最后,MADV_NORMAL命令将复位VM_RAND_READ和VM_SEQ_READ标志,从而指定以不确定的顺序访问页。

把内存映射的脏页刷新到磁盘

进程可以使用msync()系统调用把属于共享内存映射的脏页刷新到磁盘。这个系统调用

所接收的参数为：一个线性地址区间的起始地址、区间的长度以及具有下列含义的一组标志。

MS_SYNC

要求这个系统调用挂起进程，直到 I/O 操作完成为止。在这种方式中，调用进程就可以假设当系统调用完成时，这个内存映射中的所有页已经被刷新到磁盘。

MS_ASYNC (对 MS_SYNC 的补充)

要求系统调用立即返回，而不用挂起调用进程。

MS_INVALIDATE

要求系统调用使同一文件的其他内存映射无效（没有真正实现，因为在 Linux 中无用）。

对线性地址区间中所包含的每个线性区，`sys_msync()` 服务例程都调用 `msync_interval()`。后者依次执行以下操作：

1. 如果线性区描述符的 `vm_file` 字段为 `NULL`，或者如果 `VM_SHARED` 标志清 0，就返回 0（说明这个线性区不是文件的可写共享内存映射）。
2. 调用 `filemap_sync()` 函数，该函数扫描包含在线性区中的线性地址区间所对应的页表项。对于找到的每个页，重设对应页表项的 `Dirty` 标志，调用 `flush_tlb_page()` 刷新相应的转换后援缓冲器（translation lookaside buffer, TLB）。然后设置页描述符中的 `PG-dirty` 标志，把页标记为脏。
3. 如果 `MS_ASYNC` 标志置位，它就返回。因此，`MS_ASYNC` 标志的实际作用就是将线性区的页标志 `PG_dirty` 置位。该系统调用并没有实际开始 I/O 数据传输。
4. 如果函数运行至此，则 `MS_SYNC` 标志置位，因此函数必须将内存区的页刷新到磁盘，而且，当前进程必须睡眠一直到所有 I/O 数据传输结束。为做到这一点，函数要得到文件索引节点的信号量 `i_sem`。
5. 调用 `filemap_fdatawrite()` 函数，该函数接收的参数为文件的 `address_space` 对象的地址。该函数必须用 `WB_SYNC_ALL` 同步模式建立一个 `writeback_control` 描述符，而且要检查地址空间是否有内置的 `writpages` 方法。如果有，则调用这个函数后返回。如果没有，就执行 `mpage_writpages()` 函数（参见本章前面“将脏页写到磁盘”一节）。
6. 检查文件对象的 `fsync` 方法是否已定义，如果是，就执行它。对于普通文件，这个方法限制自己把文件的索引节点对象刷新到磁盘。然而，对于块设备文件，这个方法调用 `sync_blockdev()`，它会激活该设备所有脏缓冲区的 I/O 数据传输。

7. 执行 `filemap_fdatawait()` 函数。在第十五章的“基树的标记”一节中讲过，页高速缓存中的基树标识了所有通过 `PAGECACHE_TAG_WRITEBACK` 标记正在往磁盘写的页。函数快速地扫描覆盖给定线性地址区间的这一部分基树来寻找 `PG_writeback` 标志置位的页。函数调用 `wait_on_page_bit()` 使其中每一页睡眠，一直到 `PG_writeback` 标志清 0，也就是等到正在进行的该页的 I/O 数据传输结束。
8. 释放文件的信号量 `i_sem` 并返回。

非线性内存映射

对普通文件，Linux 2.6 内核还提供一个访问方法，即非线性内存映射。非线性内存映射基本上还是前面所述的文件内存映射，但它的内存页映射的并不是文件的顺序页，而是每一内存页都映射的是文件数据的随机页（任意页）。

当然，一个用户态应用每次针对同一文件的不同 4096 字节部分重复调用 `mmap()` 系统调用，也可以得到同样的结果。然而，因为每个映射需要一个独立的线性区，所以这种方法对于大文件的非线性映射是非常低效的。

为了实现非线性映射，内核使用了另外的一些数据结构。首先，线性区描述符的 `VM_NONLINEAR` 标志用于表示线性区存在一个非线性映射。给定文件的所有非线性映射线性区描述符都存放在一个双向循环链表，该链表根植于 `address_space` 对象的 `i_mmap_nonlinear` 字段。

为创建一个非线性内存映射，用户态进程首先以 `mmap()` 系统调用创建一个常规的共享内存映射。应用然后调用 `remap_file_pages()` 来重新映射内存映射中的一些页。该系统调用的 `sys_remap_file_pages()` 服务例程有下面几个参数：

```
start
    调用进程共享文件内存映射区域内的线性地址
size
    文件重新映射部分的字节数
prot
    未用(必须为 0)
pgoff
    待映射文件初始页的页索引
flags
    控制非线性映射的标志
```

该服务例程用线性地址 `start`、页索引 `pgoff` 和映射尺寸 `size` 所确定的文件数据部分进行重新映射。如果线性区非共享或不能容纳要映射的所有页，则系统调用失败并返回错误码。实际上，该服务例程把线性区插入文件的 `i_mmap_nonlinear` 链表，并调用该线性区的 `populate` 方法。

对于所有普通文件，`populate` 方法是由 `filemap_populate()` 函数实现的。它执行以下步骤：

1. 检查 `remap_file_pages()` 系统调用的 `flags` 参数中 `MAP_NONBLOCK` 标志是否清 0。如果是，则调用 `do_page_cache_readahead()` 预读待映射文件的页。
2. 对重新映射的每一页，执行下列步骤：
 - a. 检查页描述符是否已在页高速缓存内，如果不在且 `MAP_NONBLOCK` 未置位，那从磁盘读入该页。
 - b. 如果页描述符在页高速缓存内，它将更新对应线性地址的页表项来指向该页框，并更新线性区描述符的页引用计数器。
 - c. 否则，如果没有在页高速缓存内找到该页描述符，它将文件页的偏移量存放在该线性地址对应的页表项的最高 32 位，并将页表项的 `Present` 位清 0、`Dirty` 位置位。

正如第九章“请求调页”一节所述，当处理请求调页错误时，`handle_pte_fault()` 函数检查页表项的 `Present` 和 `Dirty` 位。如果它们的值对应一个非线性内存映射，则 `handle_pte_fault()` 调用 `do_file_page()` 函数，从页表项的高位中取出所请求文件页的索引，然后，`do_file_page()` 函数调用线性区的 `populate` 方法从磁盘读入页并更新页表项本身。

因为非线性内存映射的内存页是按照相对于文件开始处的页索引存放在页高速缓存中，而不是按照相对于线性区开始处的索引存放的，所以非线性内存映射刷新到磁盘的方式与线性内存映射是一样的（参见本章前面“把内存映射的脏页刷新到磁盘”一节）。

直接 I/O 传送

我们已经看到，在 Linux 2.6 版本中，通过文件系统与通过引用基本块设备文件上的块，甚至与通过建立文件内存映射访问一个普通文件之间没有什么本质的差异。但是，还是有一些非常复杂的程序（自缓存的应用程序，*self-caching application*）更愿意具有控制 I/O 数据传送机制的全部权力。例如，考虑高性能数据库服务器：它们大都实现了自己的高速缓存机制，以充分挖掘对数据库独特的查询机制。对于这些类型的程序，内核页高速缓存毫无帮助；相反，因为以下原因它可能是有害的：

- 很多页框浪费在复制已在 RAM 中的磁盘数据上（在用户级磁盘高速缓存中）。
- 处理页高速缓存和预读的多余指令降低了 `read()` 和 `write()` 系统调用的执行效率，也降低了与文件内存映射相关的分页操作。
- `read()` 和 `write()` 系统调用不是在磁盘和用户存储器之间直接传送数据，而是分两次传送：在磁盘和内核缓冲区之间和在内核缓冲区与用户存储器之间。

因为必须通过中断和直接内存访问（DMA）处理块硬件设备，而且这只能在内核态完成，因此，最终需要某种内核支持来实现自缓存的应用程序。

Linux 提供了绕过页高速缓存的简单方法：直接 I/O 传送。在每次 I/O 直接传送中，内核对磁盘控制器进行编程，以便在自缓存的应用程序的用户态地址空间中的页与磁盘之间直接传送数据。

我们知道，任何数据传送都是异步进行的。当数据传送正在进行时，内核可能切换当前进程，CPU 可能返回到用户态，产生数据传送的进程的页可能被交换出去，等等。这对于普通 I/O 数据传送没有什么影响，因为它们涉及磁盘高速缓存中的页，磁盘高速缓存由内核拥有，不能被换出去，并且对内核态的所有进程都是可见的。

另一方面，直接 I/O 传送应当在给定进程的用户态地址空间的页内移动数据。内核必须当心这些页是由内核态的任一进程访问的，当数据传送正在进行时不能把它们交换出去。让我们看看这是如何实现的。

当自缓存的应用程序要直接访问文件时，它以 `O_DIRECT` 标志置位的方式打开文件（参见第十二章“`open()` 系统调用”一节）。在运行 `open()` 系统调用时，`dentry_open()` 函数检查打开文件的 `address_space` 对象是否有已实现的 `direct_IO` 方法，如果没有则返回错误码。对一个已打开的文件也可以由 `fcntl()` 系统调用的 `F_SETFL` 命令把 `O_DIRECT` 置位。

让我们首先看第一种情况，这里自缓存应用程序对一个以 `O_DIRECT` 标志置位的方式打开的文件调用 `read()` 系统调用。在本章前面“从文件中读取数据”一节提到过，文件的 `read` 方法通常是由 `generic_file_read()` 函数实现的，它初始化 `iovec` 和 `kiocb` 描述符并调用 `__generic_file_aio_read()`。后面这个函数检查 `iovec` 描述符描述的用户态缓冲区是否有效，然后检查文件的 `O_DIRECT` 标志是否置位。当被 `read()` 调用时，该函数执行的代码段实际上等效于下面的代码：

```
if (filp->f_flags & O_DIRECT) {
    if (count == 0 || *ppos > filp->f_mapping->host->i_size)
        return 0;
    retval = generic_file_direct_IO(READ, iocb, iov, *ppos, 1);
    if (retval > 0)
```

```
        *ppos += retval;  
        file_accessed(filp);  
        return retval;  
    }  
}
```

函数检查文件指针的当前值、文件大小与请求的字符数，然后调用 `generic_file_direct_IO()` 函数，传给它 `READ` 操作类型、`iocb` 描述符、`iovec` 描述符、文件指针的当前值以及 `io_vec` 描述符中指定的用户态缓冲区号 (1)。当 `generic_file_direct_IO()` 结束时，`_generic_file_aio_read()` 更新文件指针，设置对文件索引节点的访问时间戳，然后返回。

对一个以 `O_DIRECT` 标志置位打开的文件调用 `write()` 系统调用时，情况类似。在本章前面“写入文件”一节讲到过，文件的 `write` 方法就是调用 `generic_file_aio_write_nolock()`。该函数检查 `O_DIRECT` 标志是否置位，如果置位，则调用 `generic_file_direct_IO()` 函数，而这次限定的是 `WRITE` 操作类型。

`generic_file_direct_IO()` 函数有以下参数：

`rw`

操作类型: `READ` 或 `WRITE`

`iocb`

`kiocb` 描述符指针 (参见表 16-1)

`iov`

`iovec` 描述符数组指针 (参见本章前面“从文件中读取数据”一节)

`offset`

文件偏移量

`nr_segs`

`iov` 数组中 `iovec` 描述符数

`generic_file_direct_IO()` 函数的执行步骤如下：

1. 从 `kiocb` 描述符的 `ki_filp` 字段得到文件对象的地址 `file`，从 `file->f_mapping` 字段得到 `address_space` 对象的地址 `mapping`。
2. 如果操作类型为 `WRITE`，而且一个或多个进程已创建了与文件的某个部分关联的内存映射，那么它调用 `unmap_mapping_range()` 取消该文件所有页的内存映射。如果任何取消映射的页所对应的页表项，其 `Dirty` 位置位，则该函数也确保它在页高速缓存内的相应页被标记为脏。
3. 如果根植于 `mapping` 的基树不为空 (`mapping->npages` 大于 0)，则调用

`filemap_fdatawrite()`和`filemap_fdatawait()`函数刷新所有脏页到磁盘，并等待I/O操作结束（参见本章前面“把内存映射的脏页刷新到磁盘”一节）。（即使自缓存应用程序是直接访问文件的，系统中还可能有通过页高速缓存访问文件的其他应用程序。为了避免数据的丢失，在启动直接I/O传送之前，磁盘映像要与页高速缓存进行同步）。

4. 调用mapping地址空间的`direct_IO`方法（参见下面的段落）。
5. 如果操作类型为`WRITE`，则调用`invalidate_inode_pages2()`扫描mapping基树中的所有页并释放它们。该函数同时也清空指向这些页的用户态页表项。

大多数情况下，`direct_IO`方法都是`__blockdev_direct_IO()`函数的封装函数。这个函数相当复杂，它调用大量的辅助数据结构和函数，但是实际上它所执行的操作与本章所描述的操作一样：对存放在相应块中要读或写的数据进行拆分，确定数据在磁盘上的位置，并添加一个或多个用于描述要进行的I/O操作的bio描述符。当然，数据将被直接从iovec数组中iovec描述符确定的用户态缓冲区读写。调用`submit_bio()`函数将bio描述符提交给通用块层（参见第十五章“向通用块层提交缓冲区首部”一节）。通常情况下，`__blockdev_direct_IO()`函数并不立即返回，而是等所有的直接I/O传送都已完成才返回；因此，一旦`read()`或`write()`系统调用返回，自缓存应用程序就可以安全地访问含有文件数据的缓冲区。

异步 I/O

POSIX 1003.1 标准为异步方式访问文件定义了一套库函数（如表 16-4 所示）。“异步”实际上就是：当用户态进程调用库函数读写文件时，一旦读写操作进入队列函数就结束，甚至有可能真正的I/O数据传输还没有开始。这样调用进程可以在数据正在传输时继续自己的运行。

表 16-4：异步 I/O 的 POSIX 库函数

函数	说明
<code>aio_read()</code>	从文件异步读数据
<code>aio_write()</code>	向文件异步写数据
<code>aio_fsync()</code>	请求刷新所有正在运行的异步 I/O 操作（不阻塞）
<code>aio_error()</code>	获得正在运行的异步 I/O 操作的错误代码
<code>aio_return()</code>	获得一个已完成异步 I/O 操作的返回码
<code>aio_cancel()</code>	取消正在运行的异步 I/O 操作
<code>aio_suspend()</code>	挂起进程直到至少其中一个正在运行的异步 I/O 操作完成

使用异步 I/O 很简单, 应用程序还是通过 `open()` 系统调用打开文件, 然后用描述请求操作的信息填充 `struct aiocb` 类型的控制块。 `struct aiocb` 控制块最常用的字段有:

`aio_fildes`

文件的文件描述符 (由 `open()` 系统调用返回)

`aio_buf`

文件数据的用户态缓冲区

`aio_nbytes`

待传输的字节数

`aio_offset`

读写操作在文件中的起始位置 (与“同步”文件指针无关)

最后, 应用程序将控制块地址传给 `aio_read()` 或 `aio_write()`。一旦请求的 I/O 数据传输已由系统库或内核送进队列, 这两个函数就结束。应用程序稍后可以调用 `aio_error()` 检查正在运行的 I/O 操作的状态。如数据传输仍在进行当中, 则返回 `EINPROGRESS`; 如果成功完成, 则返回 0; 如果失败, 则返回一个错误码。`aio_return()` 函数返回已完成异步 I/O 操作的有效读写字节数; 或者如果失败, 返回 -1。

Linux 2.6 中的异步 I/O

异步 I/O 可以由系统库实现而完全不需要内核支持。实际上 `aio_read()` 或 `aio_write()` 库函数克隆当前进程, 让子进程调用同步的 `read()` 或 `write()` 系统调用, 然后父进程结束 `aio_read()` 或 `aio_write()` 函数并继续程序的执行。因此, 它不用等待由子进程启动的同步操作完成。但是, 这个“穷人版” POSIX 函数比内核层实现的异步 I/O 要慢很多。

Linux 2.6 内核版运用一组系统调用实现异步 I/O。但在 Linux 2.6.11 中, 这个功能还在实现中, 异步 I/O 只能用于打开 `O_DIRECT` 标志置位的文件 (见上一节)。表 16-5 列出了异步 I/O 的系统调用。

表 16-5: 异步 I/O 的 Linux 系统调用

系统调用	说明
<code>io_setup()</code>	为当前进程初始化一个异步环境
<code>io_submit()</code>	提交一个或多个异步 I/O 操作
<code>io_getevents()</code>	获得正在运行的异步 I/O 操作的完成状态
<code>io_cancel()</code>	取消一个正在运行的异步 I/O 操作
<code>io_destroy()</code>	删除当前进程的异步环境

异步 I/O 环境

如果一个用户态进程调用 `io_submit()` 系统调用开始异步 I/O 操作，那么它必须预先创建一个异步 I/O 环境。

基本上，一个异步 I/O 环境（简称 AIO 环境）就是一组数据结构，这个数据结构用于跟踪进程请求的异步 I/O 操作的运行情况。每个 AIO 环境与一个 `kiocx` 对象关联，它存放了与该环境有关的所有信息。一个应用可以创建多个 AIO 环境。一个给定进程的所有的 `kiocx` 描述符存放在一个单向链表中，该链表位于内存描述符的 `iocx_list` 字段（参见第九章中的表 9-2）。

我们不再详细讨论 `kiocx` 对象。但是我们应当注意一个 `kiocx` 对象使用的重要的数据结构：AIO 环。

AIO 环是用户态进程中地址空间的内存缓冲区，它也可以由内核态的所有进程访问。`kiocx` 对象中的 `ring_info.mmap_base` 和 `ring_info.mmap_size` 字段分别存放 AIO 环的用户态起始地址和长度。`ring_info.ring_pages` 字段则存放有一个数组指针，该数组存放所有含 AIO 环的页框的描述符。

AIO 环实际上是一个环形缓冲区，内核用它来写正运行的异步 I/O 操作的完成报告。AIO 环的第一个字节有一个首部（`struct aio_ring` 数据结构），后面的所有字节是 `io_event` 数据结构，每个都表示一个已完成的异步 I/O 操作。因为 AIO 环的页映射至进程的用户态地址空间，应用可以直接检查正运行的异步 I/O 操作的情况，从而避免使用相对较慢的系统调用。

`io_setup()` 系统调用为调用进程创建一个新的 AIO 环境。它有两个参数：正在运行的异步 I/O 操作的最大数目（这将确定 AIO 环的大小）和一个存放环境句柄的变量指针。这个句柄也是 AIO 环的基地址。`sys_io_setup()` 服务例程实际上是调用 `do_mmap()` 为进程分配一个存放 AIO 环的新匿名线性区（参见第九章“分配线性地址区间”一节），然后创建和初始化描述该 AIO 环境的 `kiocx` 对象。

相反地，`io_destroy()` 系统调用删除 AIO 环境，还删除含有对应 AIO 环的匿名线性区。这个系统调用阻塞当前进程直到所有正在运行的异步 I/O 操作结束。

提交异步 I/O 操作

为开始异步 I/O 操作，应用要调用 `io_submit()` 系统调用。该系统调用有三个参数：

`ctx_id`

由 `io_setup()`（标识 AIO 环境）返回的句柄

iocbpp

iocb 类型描述符的指针数组的地址，其中描述符的每项描述一个异步 I/O 操作

nr

iocbpp 指向的数组长度

iocb 数据结构与 POSIX aiocb 描述符有同样的字段 aio_fildes、aio_buf、aio_nbytes、aio_offset，另外还有 aio_lio_opcode 字段存放请求操作的类型（典型地有：read、write 或 sync）。

sys_io_submit() 服务例程执行下列步骤：

1. 验证 iocb 描述符数组的有效性。
2. 在内存描述符的 ioctx_list 字段所对应的链表中查找 ctx_id 句柄对应的 kiocb 对象。
3. 对数组中的每一个 iocb 描述符，执行下列子步骤：
 - a. 获得 aio_fildes 字段中的文件描述符对应的文件对象地址。
 - b. 为该 I/O 操作分配和初始化一个新的 kiocb 描述符。
 - c. 检查 AIO 环中是否有空闲位置来存放操作的完成情况。
 - d. 根据操作类型设置 kiocb 描述符的 ki_retry 方法（见下面）。
 - e. 执行 aio_run_iocb() 函数，它实际上调用 ki_retry 方法为相应的异步 I/O 操作启动数据传输。如果 ki_retry 方法返回 -EIOCBRETRY，则表示异步 I/O 操作已提交但还没有完全成功：稍后在这个 kiocb 上，aio_run_iocb() 函数会被再次调用（见下面）；否则，调用 aio_complete()，为异步 I/O 操作在 AIO 环境的环中追加完成事件。

如果异步 I/O 操作是一个读请求，那么对应 kiocb 描述符的 ki_retry 方法是由 aio_pread() 实现的。该函数实际上执行的是文件对象的 aio_read 方法，然后按照 aio_read 方法的返回值更新 kiocb 描述符的 ki_buf 和 ki_left 字段（参见本章前面的表 16-1）。最后 aio_pread() 返回从文件读入的有效字节数，或者，如果函数确定请求的字节没有传输完，则返回 -EIOCBRETRY。对于大部分文件系统，文件对象的 aio_read 方法就是调用 __generic_file_aio_read() 函数。假如文件的 O_DIRECT 标志置位，函数就调用 generic_file_direct_IO() 函数，这在上一节描述过。但在这种情况下，__blockdev_direct_IO() 函数不是阻塞当前进程使之等待 I/O 数据传输完毕，而是立即返回。因为异步 I/O 操作仍在运行，aio_run_iocb() 会被再次调用，而这一次

的调用者是 aio_wq 工作队列的 aio 内核线程。kiocb 描述符跟踪 I/O 数据传输的运行。终于所有数据传输完毕，将完成结果追加到 AIO 环。

类似地，如果异步 I/O 操作是一个写请求，那么对应 kiocb 描述符的 ki_retry 方法是由 aio_pwrite() 实现的。该函数实际上执行的是文件对象的 aio_write 方法，然后按照 aio_write 方法的返回值更新 kiocb 描述符的 ki_buf 和 ki_left 字段（参见本章前面的表 16-1）。最后 aio_pwrite() 返回写入文件的有效字节数，或者，如果函数确定请求的字节没有完全传输完，则返回 -EIOCBRETRY。对于大部分文件系统，文件对象的 aio_write 方法就是调用 generic_file_aio_write_nolock() 函数。假如文件的 O_DIRECT 标志置位，跟上面一样，函数就调用 generic_file_direct_IO() 函数。

第十七章



回收页框

在前面的章节中，我们说明了内核如何通过记录空闲和占用的页框来处理动态内存。我们还讨论了用户态的每个进程怎样拥有自己的线性地址空间，又是怎样以每次一页的方式得到所请求的内存，以及如何在万不得已时才把页框分配给进程。最后，我们也讨论了如何使用动态内存实现内存与磁盘高速缓存。

在本章，我们通过讨论页框的回收完成对虚拟内存子系统的描述。在第一节“页框回收算法”中，我们阐述了内核回收页框的原因与策略；然后在“反向映射”一节做一个技术补充，介绍了内核使用的一种数据结构，借助这个结构，内核可以快速定位指向同一个页框的所有页表项；而“PFRA 实现”一节则介绍 Linux 使用的页框回收算法；最后一节“交换”，几乎可以自成一章，这一节讲述了交换子系统，它是将匿名页（并非文件的映射数据）保存到磁盘的内核部件。

页框回收算法

Linux 中有一点很有意思，在为用户态进程与内核分配动态内存时，所作的检查是马马虎虎的。

比如，对单个用户所创建进程的 RAM 使用总量并不作严格检查（第三章的“进程资源限制”一节提到的限制只针对单个进程）；对内核使用的许多磁盘高速缓存和内存高速缓存大小也同样不作限制。

减少控制是一种设计选择，这使内核以最好的可行方式使用可用的 RAM。当系统负载较低时，RAM 的大部分由磁盘高速缓存占用，很少正在运行的进程可以从中获益。但是，

当系统负载增加时, RAM的大部分则由进程页占用, 高速缓存就会缩小从而给后来的进程让出空间。

我们在前面的章节中看到, 内存及磁盘高速缓存抓取了那么多的页框但从未释放任何页框。这是合理的, 因为高速缓存系统并不知道进程是否(什么时候)会重新使用某些缓存的数据, 因此不能确定高速缓存的哪些部分应该释放。此外, 正是有了第九章描述的请求调页机制, 只要用户态进程继续执行, 它们就能获得页框; 然而, 请求调页没有办法强制进程释放不再使用的页框。

因此, 迟早所有空闲内存将被分配给进程和高速缓存。Linux内核的页框回收算法(*page frame reclaiming algorithm, PFRA*)采取从用户态进程和内核高速缓存“窃取”页框的办法补充伙伴系统的空闲块列表。

实际上, 在用完所有空闲内存之前, 就必须执行页框回收算法。否则, 内核很可能陷入一种内存请求的僵局中, 并导致系统崩溃。也就是说, 要释放一个页框, 内核就必须把页框的数据写入磁盘; 但是, 为了完成这一操作, 内核却要请求另一个页框(例如, 为I/O数据传送分配缓冲区首部)。因为不存在空闲页框, 因此, 不可能释放页框。

页框回收算法的目标之一就是保存最少的空闲页框池以便内核可以安全地从“内存紧缺”的情形中恢复过来。

选择目标页

页框回收算法(PFRA)的目标就是获得页框并使之空闲。显然, PFRA选取的页框肯定不是空闲的, 即这些页框原本不在伙伴系统的任何一个free_area数组中(参见第八章的“伙伴系统算法”一节)。

PFRA按照页框所含内容, 以不同的方式处理页框。我们将它们区分成: 不可回收页、可交换页、可同步页和可丢弃页, 如表17-1所示。

表17-1: PFRA处理的页框类型

页类型	说明	回收操作
不可回收页	空闲页(包含在伙伴系统列表中)	(不允许也无需回收)
	保留页(PG_reserved标志置位)	
	内核动态分配页	
	进程内核态堆栈页	
	临时锁定页(PG_locked标志置位)	
	内存锁定页(在线性区中且VM_LOCKED标志置位)	

表 17-1: PFRA处理的页框类型 (续)

页类型	说明	回收操作
可交换页	用户态地址空间的匿名页 tmpfs 文件系统的映射页 (如 IPC 共享内存的页)	将页的内容保存在交换区
可同步页	用户态地址空间的映射页 存有磁盘文件数据且在页高速缓存中的页 块设备缓冲区页 某些磁盘高速缓存的页 (如索引节点高速缓存)	必要时, 与磁盘映像同步这些页
可丢弃页	内存高速缓存中的未使用页 (如 slab 分配器高速缓存) 目录项高速缓存的未使用页	无需操作

在表 17-1 中, 所谓“映射页”是指该页映射了一个文件的某个部分。比如, 属于文件内存映射的用户态地址空间中所有页都是映射页, 页高速缓存中的任何其他页也是映射页。映射页差不多都是可同步的: 为回收页框, 内核必须检查页是否为脏, 而且必要时将页的内容写到相应的磁盘文件中。

相反, 所谓的“匿名页”是指它属于一个进程的某匿名线性区 (例如, 进程的用户态堆和堆栈中的所有页为匿名页)。为回收页框, 内核必须将页中内容保存到一个专门的磁盘分区或磁盘文件, 叫做“交换区” (参见后面“交换”一节)。因此, 所有匿名页都是可交换的。

通常, 特殊文件系统中的页是不可回收的。唯一的例外是 *tmpfs* 特殊文件系统的页, 它可以被保存在交换区后被回收。在第十九章中我们将看到 *tmpfs* 特殊文件系统用于 IPC 共享内存机制。

当 PFRA 必须回收属于某进程用户态地址空间的页框时, 它必须考虑页框是否为共享的。共享页框属于多个用户态地址空间, 而非共享页框属于单个用户态地址空间。注意, 非共享页框可能属于几个轻量级进程, 这些进程使用同一个内存描述符。

当进程创建子进程时, 就建立了共享页框。正如第九章“写时复制”一节所述, 子进程页表都从父进程中复制过来的, 父子进程因此共享同一个页框。共享页框的另一个常见情形是: 一个或多个进程以共享内存映射的方式访问同一个文件 (参见第十六章的“内存映射”一节) (注 1)。

注 1: 应该注意的是, 尽管如此, 当一个单独的进程通过共享内存映射访问文件时, 相应的页对 PFRA 来说却是非共享的。同样, PFRA 可能把属于私有内存映射的页作为共享页 (例如: 两个文件读同一个文件的某个部分, 但都不修改这部分的数据所在页的内容)。

PFRA 设计

尽管很容易确定回收内存的候选页（粗略地说，任何属于磁盘和内存高速缓存的页，以及属于进程用户态地址空间的页），但是选择合适的目标页可能是内核设计中最精巧的问题。

事实上，对处理虚拟内存子系统的开发者来说，其最难的工作在于找到一种合适的算法，这种算法既能确保台式计算机有可接受的性能（在这种计算机上内存的需要是相当有限的，而对系统响应的要求则是十分严格的），也能确保像大型数据库服务器那样的高级计算机有可接受的性能（在这种计算机上对内存的需要则巨大无比）。

不幸的是，找到一种较佳的页框回收算法是一种相当经验性的工作，很少有理论的支持。这种情形类似于对决定进程动态优先级的因素进行评估：主要目的是调整参数以达到较好的性能，不要问太多的为什么。通常情况下，这仅仅是“让我们试一试这种方法，看看会发生什么……”这么回事。这种经验主义方法的负面效果就是代码变化快。因此我们无法保证：在你阅读本章时，这里讲的 Linux 2.6.11 使用的内存回收算法与 Linux 2.6 的最新版本中所使用的内存回收算法完全一致。但是，这里所讲的一般原则和主要的启发式准则还会继续使用。

一叶障目，不见泰山。因此，让我们先看看 PFRA 采用的几个总的原则，这些原则包含在本章后面介绍的几个函数中。

首先释放“无害”页

在进程用户态地址空间的页回收之前，必须先回收没有被任何进程使用的磁盘与内存高速缓存中的页。实际上，回收磁盘与内存高速缓存的页框并不需要修改任何页表项。我们在本章后面“最近最少使用（LRU）链表”一节会看到，在使用“交换倾向因子（swap tendency factor）”后，这个准则可以做出一些调整。

将用户态进程的所有页定为可回收页

除了锁定页，PFRA 必须能够窃得任何用户态进程页，包括匿名页。这样，睡眠较长时间的进程将逐渐失去所有页框。

同时取消引用一个共享页框的所有页表项的映射，就可以回收该共享页框

当 PFRA 要释放几个进程共享的页框时，它就清空引用该页框的所有页表项，然后回收该页框。

只回收“未用”页

使用简化的最近最少使用（Least Recently Used, LRU）置换算法，PFRA 将页分

为“在用 (*in_use*)”与“未用 (*unused*)”(注2)。如果某页很长时间没有被访问,那么它将来被访问的可能性较小,就可以将它看作未用;另一方面,如果某页最近被访问过,那么它将来被访问的可能性较大,就必须将它看作在用。PFRA只回收未用页。这就是第二章中“硬件高速缓存”一节所讲局部性原则的另一个应用。

LRU算法的主要思想就是用一个计数器来存放RAM中每一页的页年龄,即上一次访问该页到现在已经过的时间。这个计数器可使PFRA只回收任何进程的最旧页。一些计算机平台提供较为成熟的LRU算法(注3)。不幸的是,80x86处理器不提供这样的硬件功能,因此Linux内核不能依赖页计数器记录每页的页年龄。为解决这一问题, Linux使用每个页表项中的访问标志位(*Accessed*),在页被访问时,该标志位由硬件自动置位;而且,页年龄由页描述符在链表(两个不同的链表之一)中的位置来表示[参见本章后面“最近最少使用(LRU)链表”一节]。

因此,页框回收算法是几种启发式方法的混合:

- 谨慎选择检查高速缓存的顺序。
- 基于页年龄的变化排序(在释放最近访问的页之前,应当释放最近最少使用的页)。
- 区别对待不同状态的页(例如,不脏的页与脏页之间,最好把前者换出,因为前者不必写磁盘)。

反向映射

正如上一节所述,PFRA的目标之一是能释放共享页框。为达到这个目的, Linux 2.6内核能够快速定位指向同一页框的所有页表项。这个过程就叫做反向映射(*reverse mapping*)。

反向映射方法的简单解决之道,就是在页描述符中引入附加字段,从而将某页描述符所确定的页框中对应的所有页表项联接起来。但是,保持更新这样的链表将会大大增加系统开销,因此,就有更成熟的方法设计出来了。Linux 2.6就有叫做“面向对象的反向映射”的技术。实际上,对任何可回收的用户态页,内核保留系统中该页所在所有线性区(“对象”)的反向链接,每个线性区描述符存放一个指针指向一个内存描述符,而该内存描述符又包含一个指针指向一个页全局目录(*Page Global Directory*)。因此,这

注2: 还可以认为PFRA是“曾经使用过的”算法,它的思想来源于T.Johnson和D.Shasha在1994年提出的2Q缓冲区管理置换算法。

注3: 例如,一些大型机的CPU自动更新每个页表项中表示相应页年龄的计数器。

些反向链接使得PFRA能够检索引用某页的所有页表项。因为线性区描述符比页描述符少，所以更新共享页的反向链接就比较省时间。我们来看看这一方法是如何实现的。

首先，PFRA必须要确定待回收页是共享的或是非共享的，以及是映射页或是匿名页。为做到这一点，内核要查看页描述符的两个字段：`_mapcount`和`mapping`。

`_mapcount`字段存放引用页框的页表项数目。计数器的起始值为-1，这表示没有页表项引用该页框；如果值为0，就表示页是非共享的；而如果值大于0，则表示页是共享的。`page_mapcount`函数接收页描述符地址，返回值为`_mapcount+1`（这样，如返回值为1，表明是某个进程的用户态地址空间中存放的一个非共享页）。

页描述符的`mapping`字段用于确定页是映射的或匿名的。说明如下：

- 如果`mapping`字段空，则该页属于交换高速缓存（参见本章后面“交换高速缓存”一节）。
- 如果`mapping`字段非空，且最低位是1，表示该页为匿名页；同时`mapping`字段中存放的是指向`anon_vma`描述符的指针（参见下一节“匿名页的反向映射”）。
- 如果`mapping`字段非空，且最低位是0，表示该页为映射页；同时`mapping`字段指向对应文件的`address_space`对象（参见第十五章的“`address_space`对象”一节）。

Linux的`address_space`对象在RAM中是对齐的，所以其起始地址是4的倍数。因此其`mapping`字段的最低位可以用作一个标志位来表示该字段的指针是指向`address_space`对象还是`anon_vma`描述符。这是一个不规范的编程技巧，但内核要使用大量的页描述符，所以这些数据结构必须尽可能的小。`PageAnon()`函数接收页描述符地址作为参数，如果`mapping`字段的最低位置位，则函数返回1；否则返回0。

`try_to_unmap()`函数接收页描述符指针作为参数，尝试清空所有引用该页描述符对应页框的页表项。如果从页表项中成功清除所有对该页框的应用，函数返回`SWAP_SUCCESS`(0)；如果有些引用不能清除，函数返回`SWAP_AGAIN`(1)；如果出错，函数返回`SWAP_FAIL`(2)。这个函数很短：

```
int try_to_unmap(struct page *page)
{
    int ret;
    if (PageAnon(page))
        ret = try_to_unmap_anon(page);
    else
        ret = try_to_unmap_file(page);
    if (!page_mapped(page))
        ret = SWAP_SUCCESS;
    return ret;
}
```

函数 `try_to_unmap_anon()` 和 `try_to_unmap_file()` 分别处理匿名页和映射页。后面会对这两个函数加以说明。

匿名页的反向映射

匿名页经常是由几个进程共享的。最为常见的情形是：创建新进程，这在第九章中“写时复制”一节里已有描述，父进程的所有页框，包括匿名页，同时也分配给子进程。另外（但不常见），进程创建线性区时使用两个标志 `MAP_ANONYMOUS` 和 `MAP_SHARED`，表明这个区域内的页将由该进程后面的子进程共享。

将引用同一个页框的所有匿名页链接起来的策略非常简单，即将该页框所在的匿名线性区存放在一个双向循环链表中。要注意的是：即使一个匿名线性区存有不同的页，也始终只有一个反向映射链表用于该区域中的所有页框。

当为一个匿名线性区分配第一页时，内核创建一个新的 `anon_vma` 数据结构，它只有两个字段：`lock` 和 `head`。`lock` 字段是竞争条件下保护链表的自旋锁；`head` 字段是线性区描述符双向循环链表的头部。然后，内核将匿名线性区的 `vm_area_struct` 描述符插入 `anon_vma` 链表。为实现这个目的，`vm_area_struct` 数据结构中包含有对应该链表的两个字段：`anon_vma_node` 和 `anon_vma`。`anon_vma_node` 字段存放指向链表中前一个和后一个元素的指针，而 `anon_vma` 字段指向 `anon_vma` 数据结构。最后，按前面所述，内核将 `anon_vma` 数据结构的地址存放在匿名页描述符的 `mapping` 字段。如图 17-1 所示。

当已被一个进程引用的页框插入另一个进程的页表项时（例如调用 `fork()` 系统调用时，参见第三章中“`clone()`、`fork()` 及 `vfork()` 系统调用”一节），内核只是将第二个进程的匿名线性区插入 `anon_vma` 数据结构的双向循环链表，而第一个进程线性区的 `anon_vma` 字段指向该 `anon_vma` 数据结构。因此每个 `anon_vma` 链表通常包含不同进程的线性区（注 4）。

如图 17-1 所示，借助 `anon_vma` 链表，内核可以快速定位引用同一匿名页框的所有页表项。实际上，每个区域描述符在 `vm_mfn` 字段中存放内存描述符地址，而该内存描述符又有一个 `pgd` 字段，其中存有进程的页全局目录。这样，页表项就可以从匿名页的起始线性地址得到，而该线性地址可以由线性区描述符以及页描述符的 `index` 字段得到。

注 4： `anon_vma` 的链表还可以包括同一个进程的几个相邻的匿名线性区。通常在系统调用 `mprotect()` 把一个匿名线性区划分成两个或更多个线性区时就会出现这种情况。

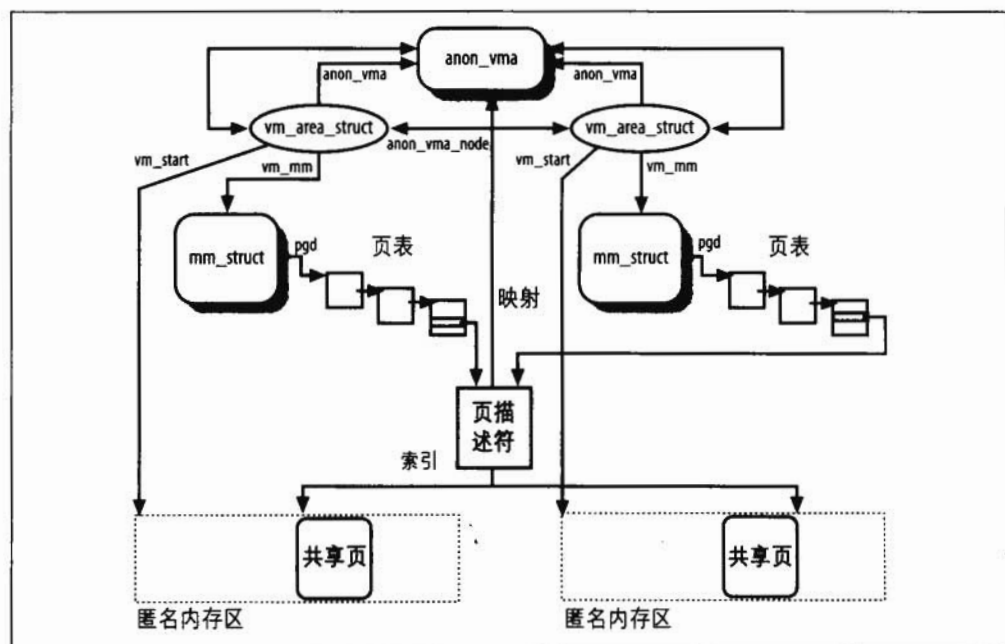


图 17-1: 匿名页的面向对象反向映射

try_to_unmap_anon()函数

当回收匿名页框时，PFRA 必须扫描 `anon_vma` 链表中的所有线性区，仔细检查是否每个区域都存有一个匿名页，而其对应的页框就是目标页框。这一工作就是通过 `try_to_unmap_anon()` 函数实现的，它接收目标页框描述符作为参数，执行的主要步骤如下：

1. 获得 `anon_vma` 数据结构的自旋锁，页描述符的 `mapping` 字段指向该数据结构。
2. 扫描线性区描述符的 `anon_vma` 链表。对该链表中的每一个 `vma` 线性区描述符，调用 `try_to_unmap_one()` 函数，传给它参数 `vma` 和页描述符（参见下面）。如果由于某种原因返回值为 `SWAP_FAIL`，或如果页描述符的 `_mapcount` 字段表明已找到所有引用该页框的页表项，那么停止扫描，而不用扫描到链表底部。
3. 释放第 1 步得到的自旋锁。
4. 返回最后调用 `try_to_unmap_one()` 函数得到的值：`SWAP_AGAIN`（部分成功）或 `SWAP_FAIL`（失败）。

try_to_unmap_one()函数

try_to_unmap_one()函数由try_to_unmap_anon()和try_to_unmap_file()重复调用。它有两个参数：page和vma。page是一个指向目标页描述符的指针；而vma是指向线性区描述符的指针。该函数执行的主要步骤如下：

1. 计算出待回收页的线性地址，所依据的参数有：线性区的起始线性地址（vma->vm_start）、被映射文件的线性区偏移量（vma->vm_pgoff）和被映射文件内的页偏移量（page->index）。对于匿名页，vma->vm_pgoff字段是0或者vm_start/PAGE_SIZE；相应地，page->index字段是区域内的页索引或是页的线性地址除以PAGE_SIZE。
2. 如果目标页是匿名页，则检查页的线性地址是否在线性区内。如果不是，则结束并返回SWAP_AGAIN（在介绍匿名页的反向映射时，我们讲过anon_vma链表可能存有不包含目标页的线性区）。
3. 从vma->vm_mm得到内存描述符地址，并获得保护页表的自旋锁vma->vm_mm->page_table_lock。
4. 成功调用pgd_offset()、pud_offset()、pmd_offset()和pte_offset_map()以获得对应目标页线性地址的页表项地址。
5. 执行一些检查来验证目标页可有效回收。下面的检查步骤中，如果任何一步失败，函数跳到第12步，结束并返回一个有关的错误码：SWAP_AGAIN或SWAP_FAIL。
 - a. 检查指向目标页的页表项。如果不成功，则函数返回SWAP_AGAIN。这可能在以下几种情形下发生：
 - 指向页框的页表项与COW关联，而vma标识的匿名线性区仍然属于原页框的anon_vma链表。
 - mremap()系统调用可重新映射线性区，并通过直接修改页表项将页移到用户态地址空间。这种特殊情况下，因为页描述符的index字段不能用于确定页的实际线性地址，所以面向对象的反向映射就不能使用了。
 - 文件内存映射是非线性的（参见第十六章的“非线性内存映射”一节）。
 - b. 验证线性区不是锁定（VM_LOCKED）或保留（VM_RESERVED）的。如果有锁定（VM_LOCKED）或保留情况之一出现，函数就返回SWAP_FAIL。
 - c. 验证页表项中的访问标志位（Accessed）被清0。如果没有，该函数将它清0，并返回SWAP_FAIL。访问标志位位置位表示页在用，因此不能被回收。
 - d. 检查页是否属于交换高速缓存（参见本章后面“交换高速缓存”一节），且此时

它正由 `get_user_pages()` 处理（参见第九章的“分配线性地址区间”一节）。在这种情形下，为避免恶性竞争条件，函数返回 `SWAP_FAIL`。

6. 页可以被回收。如果页表项的 `Dirty` 标志位置位，则将页的 `PG_dirty` 标志置位。
7. 清空页表项，刷新相应的 TLB。
8. 如果是匿名页，函数将换出页（swapped-out page）标识符插入页表项，以便将来访问时将该页换入（参见本章后面“交换”一节）。而且，递减存放在内存描述符 `anon_rss` 字段中的匿名页计数器。
9. 递减存放在内存描述符 `rss` 字段中的页框计数器。
10. 递减页描述符的 `_mapcount` 字段，因为对用户态页表项中页框的引用已被删除。
11. 递减存放在页描述符 `_count` 字段中的页框使用计数器。如果计数器变为负数，则从活动或非活动链表中删除页描述符[参见本章后面“最近最少使用 (LRU) 链表”一节]，而且调用 `free_hot_page()` 释放页框（参见第八章的“每 CPU 页框高速缓存”一节）。
12. 调用 `pte_unmap()` 释放临时内核映射，因为第 4 步中的 `pte_offset_map()` 可能分配了一个这样的映射（参见第八章的“高端内存页框的内核映射”一节）。
13. 释放第 3 步中获得的自旋锁 `vma->vm_mm->page_table_lock`。
14. 返回相应的错误码（成功时返回 `SWAP_AGAIN`）。

映射页的反向映射

与匿名页相比，映射页的面向对象反向映射所基于的思想很简单：我们总是可以获得指向一个给定页框的页表项，方法就是访问相应映射页所在的线性区描述符。因此，反向映射的关键就是一个精巧的数据结构，这个数据结构可以存放与给定页框有关的所有线性区描述符。

我们在上一节看到，匿名线性区描述符存放在双向循环链表中。获得引用给定页框的所有页表项，就是对该链表中的元素进行线性扫描。共享匿名页框的数量不是很大，因此这个方法工作得很好。

与匿名页相反，映射页经常是共享的，这是因为不同的进程常会共享同一个程序代码。例如，几乎所有进程都会共享包含标准 C 库代码的页（参见第二十章的“库”一节）。因此，Linux 2.6 依靠叫做“优先搜索树（priority search tree）”的结构来快速定位引用同一页框的所有线性区。

每个文件对应一个优先搜索树。它存放在 `address_space` 对象的 `i_mmap` 字段中，该 `address_space` 对象包含在文件的索引节点对象中。因为映射页描述符的 `mapping` 字段指向 `address_space` 对象，所以总是能够快速检索搜索树的根。

优先搜索树

Linux 2.6 使用的优先搜索树 (PST) 是基于 Edward McCreight 于 1985 年提出的一种数据结构，用于表示一组相互重叠的区间。McCreight 树是一个堆和对称搜索树的混合体，且用于对一个区间集进行查询。例如，“在一个给定区间内有哪些区间？”和“哪些区间与给定区间相交？”这种查询所花的时间与树的高度和结果区间的数量成正比。

PST 中的每一个区间相当于一个树的节点，它由基索引 (*radix index*) 和堆索引 (*heap index*) 两个索引来标识。基索引表示区间的起始点而堆索引表示终点。PST 实际上是一个依赖于基索引的搜索树，并附加一个类堆属性，即一个节点的堆索引不会小于其子节点的堆索引。

Linux 优先搜索树与 McCreight 数据结构的不同有两个重要方面：第一，Linux 树不总是对称的（对称算法要耗费很多的系统空间和执行时间）；第二，Linux 树被修改成存放线性区而不是线性区间。

每个线性区可以被看成是文件页的一个区间，并由在文件中的起始位置（基索引）和终点位置（堆索引）所确定。但是，线性区通常是从同一页开始（通常从页索引 0 开始）。不幸的是，McCreight 的原数据结构不能存放起始位置完全一样的区间。补充解决方案是：除了基索引和堆索引，PST 的每个节点附带一个大小索引 (*size index*)。该大小索引的值为线性区大小（页数）减 1。该大小索引使搜索程序能够区分同一起始文件位置的不同线性区。

然而，大小索引会大大增加不同的节点数，会使 PST 溢出。特别是，当有很多节点具有相同的基索引但堆索引不同时，PST 就无法全部容下它们。为了解决这个问题，PST 可以包括溢出子树 (*overflow subtree*)，该子树以 PST 的叶为根，且包含具有相同基索引的节点。

此外，不同进程拥有的线性区可能是映射了相同文件的相同部分（如上面提及的标准 C 库）。在这种情况下，对应这些区域的所有节点具有相同的基索引、堆索引和大小索引。当必须在 PST 中插入一个与现存某个节点具有相同索引值（基索引、堆索引和大小索引都相同）的线性区时，内核将该线性区描述符插入一个以原 PST 节点为根的双向循环列表。

图 17-2 所示是一个简单的优先搜索树。在图的左侧，我们看到有七个线性区覆盖着一个文件的前六页。每个区间都标有基索引、堆索引和大小索引。在图的右侧，则是对应的

PST。注意，子节点的堆索引都不大于相应父节点的堆索引。而且我们可以看到，任意一个节点的左子节点基索引也都不大于右子节点基索引，如果基索引相等，则按照大小索引排序。让我们假定：PFRA 搜索包含某页（索引为 5）的全部线性区。搜索算法从根（0, 5, 5）开始，因为相应区间包含该页，那么这就是得到的第一个线性区。然后算法搜索根的左子节点（0, 4, 4），比较堆索引（4）和页索引，因为堆索引较小，所以区间不包括该页。而且，有了 PST 的类堆属性，该节点的所有子节点都不包括该页。因此，算法直接跳到根的右子节点（2, 3, 5），其相应区间包含该页，因此得到这个区间。然后，算法搜索子节点（1, 2, 3）和（2, 0, 2），但它们都不包含该页。

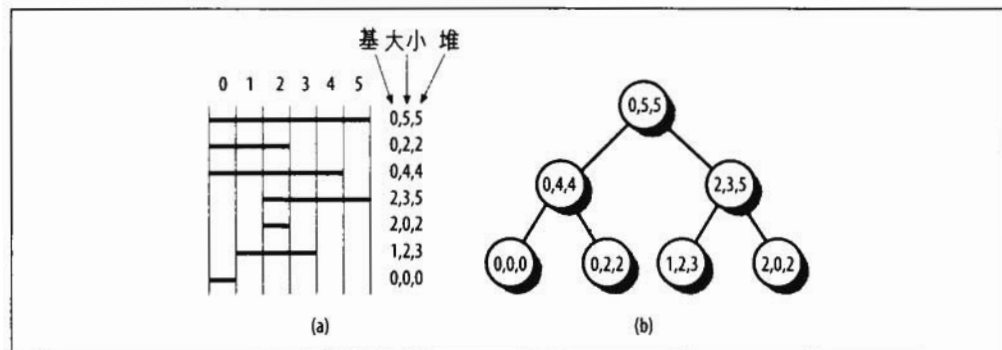


图 17-2: 简单的优先搜索树

因篇幅有限，我们对实现 Linux PST 的数据结构与函数无法作详尽阐述。我们只讨论由 `prio_tree_node` 数据结构表示的一个 PST 节点。该数据结构在每个线性区描述符的 `shared.prio_tree_node` 字段中。`shared.vm_set` 数据结构作为 `shared.prio_tree_node` 的替代品，可以用来将线性区描述符插入一个 PST 节点的链表副本。可以用 `vma_prio_tree_insert()` 和 `vma_prio_tree_remove()` 函数分别插入和删除 PST 节点。两个函数的参数都是线性区描述符地址与 PST 根地址。对 PST 的搜索可调用 `vma_prio_tree_foreach` 宏来实现，该宏循环搜索所有线性区描述符，这些描述符在给定范围的线性地址中包含至少一页。

try_to_unmap_file() 函数

`try_to_unmap_file()` 函数由 `try_to_unmap()` 调用，并执行映射页的反向映射。当为线性内存映射时，该函数就很容易描述（参见第十六章的“内存映射”一节）。这种情况下，它执行的步骤如下：

1. 获得 `page->mapping->i_mmap_lock` 自旋锁。
2. 对搜索树应用 `vma_prio_tree_foreach()` 宏，搜索树的根存放在 `page->mapping`

->i_mmap 字段。对宏发现的每个 vm_area_struct 描述符，函数调用 try_to_unmap_one()，尝试对该页所在的线性区页表项清 0（参见前面“匿名页的反向映射”一节）。如果由于某种原因，返回 SWAP_FAIL，或者如果页描述符的 _mapcount 字段表明引用该页框的所有页表项都已找到，则搜索过程马上结束。

3. 释放 page->mapping->i_mmap_lock 自旋锁。
4. 根据所有的页表项清 0 与否，返回 SWAP_AGAIN 或 SWAP_FAIL。

如果映射是非线性的（参见第十六章的“非线性内存映射”一节），那么 try_to_unmap_one() 函数可能无法清 0 某些页表项，这是因为页描述符的 index 字段（该字段存放文件中页的位置）不再对应线性区中的页位置，try_to_unmap_one() 函数就无法确定页的线性地址，也就无法得到页表项地址。

唯一的解决方法是对文件非线性线性区的穷尽搜索。双向链表以文件的所有非线性线性区的描述符所在的 page->mapping 文件的 address-space 对象的 i_mmap_nonlinear 字段为根。对每个这样的线性区，try_to_unmap_file() 函数调用 try_to_unmap_cluster()，而 try_to_unmap_cluster() 函数会扫描该线性区线性地址所对应的所有页表项，并尝试将它们清 0。

因为搜索可能很费时，所以执行有限扫描，而且通过试探法决定扫描线性区的哪一部分，vma_area_struct 描述符的 vm_private_data 字段存有当前扫描的当前指针。因此，try_to_unmap_file() 函数在某些情况下可能会找不到待停止映射的页。出现这种情况时，try_to_unmap() 函数发现页仍然是映射的，那么返回 SWAP_AGAIN 而不是 SWAP_SUCCESS。

PFRA 实现

页框回收算法必须处理多种属于用户态进程、磁盘高速缓存和内存高速缓存的页，而且必须遵照几条试探法准则。因此，PFRA 有很多函数也就不奇怪了。图 17-3 列出了 PFRA 的主要函数，箭头表示函数调用。例如，try_to_free_pages() 函数调用 shrink_caches()、shrink_slab() 和 out_of_memory() 三个函数。

正如你所看到的，PFRA 有几个入口（entry point）。实际上，页框回收算法的执行有三种基本情形：

内存紧缺回收

内核发现内存紧缺

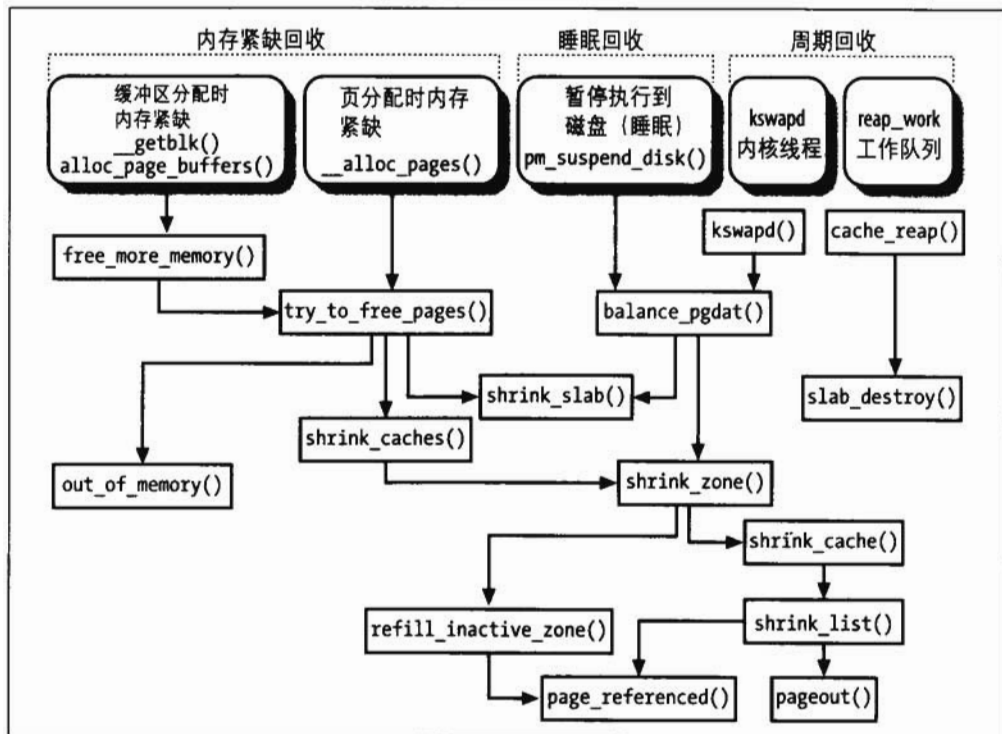


图 17-3: PFRA 的主要函数

睡眠回收

在进入 suspend-to-disk 状态时，内核必须释放内存（我们不再进一步讨论这种情形）

周期回收

必要时，周期性激活内核线程执行内存回收算法

内存紧缺回收在下列几种情形下激活：

- grow_buffers() 函数（由 __getblk() 调用）无法获得新的缓冲区页（参见第十五章的“在页高速缓存中搜索块”一节）。
- alloc_page_buffers() 函数（由 create_empty_buffers() 调用）无法获得页临时缓冲区首部（参见第十六章的“读写文件”一节）。
- __alloc_pages() 函数无法在给定的内存管理区（memory zone）中分配一组连续页框（参见第八章中“伙伴系统算法”一节）。

周期回收由下面两种不同的内核线程激活：

- *kswapd* 内核线程，它检查某个内存管理区中空闲页框数是否已低于 *pages_high* 值的标高（参见后面的“周期回收”一节）。
- *events* 内核线程，它是预定义工作队列的工作者线程（参见第四章的“工作队列”一节）；PFRA 周期性地调度预定义工作队列中的一个任务执行，从而回收 slab 分配器处理的位于内存高速缓存中的所有空闲 slab（参见第八章的“slab 分配器”一节）。

我们现在详细讨论页框回收算法的各个部分，也包括图 17-3 中的所有函数。

最近最少使用（LRU）链表

属于进程用户态地址空间或页高速缓存的所有页被分成两组：活动链表与非活动链表。它们被统称为 LRU 链表。前面一个链表用于存放最近被访问过的页；后面的则存放有一段时间没有被访问过的页。显然，页必须从非活动链表中窃取。

页的活动链表和非活动链表是页框回收算法的核心数据结构。这两个双向链表的头分别存放在每个 zone 描述符（参见第八章的“内存管理区”一节）的 *active_list* 和 *inactive_list* 字段，而该描述符的 *nr_active* 和 *nr_inactive* 字段表示存放在两个链表中的页数。最后，*lru_lock* 字段是一个自旋锁，保护两个链表免受 SMP 系统上的并发访问。

如果页属于 LRU 链表，则设置页描述符中的 *PG_lru* 标志。此外，如果页属于活动链表，则设置 *PG_active* 标志，而如果页属于非活动链表，则清除 *PG_active* 标志。页描述符的 *lru* 字段存放指向 LRU 链表中下一个元素和前一个元素的指针。

另外有几个辅助函数处理 LRU 链表：

`add_page_to_active_list()`

将页加入管理区的活动链表头部并递增管理区描述符的 *nr_active* 字段。

`add_page_to_inactive_list()`

将页加入管理区的非活动链表头部并递增管理区描述符的 *nr_inactive* 字段。

`del_page_from_active_list()`

从管理区的活动链表中删除页并递减管理区描述符的 *nr_active* 字段。

`del_page_from_inactive_list()`

从管理区的非活动链表中删除页并递减管理区描述符的 *nr_inactive* 字段。

`del_page_from_lru()`

检查页的 *PG_active* 标志。依据检查结果，将页从活动或非活动链表中删除，递减

管理区描述符的 `nr_active` 或 `nr_inactive` 字段, 且如有必要, 将 `PG_active` 标志清 0。

`activate_page()`

检查 `PG_active` 标志, 如果未置位 (页在非活动链表中), 将页移到活动列表中, 依次调用 `del_page_from_inactive_list()` 和 `add_page_to_active_list()`, 最后将 `PG_active` 标志置位。在移动页之前, 获得管理区的 `lru_lock` 自旋锁。

`lru_cache_add()`

如果页不在 LRU 链表中, 将 `PG_lru` 标志置位, 得到管理区的 `lru_lock` 自旋锁, 调用 `add_page_to_inactive_list()` 把页插入管理区的非活动链表。

`lru_cache_add_active()`

如果页不在 LRU 链表中, 将 `PG_lru` 和 `PG_active` 标志置位, 得到管理区的 `lru_lock` 自旋锁, 调用 `add_page_to_active_list()` 把页插入管理区的活动链表。

事实上, 最后两个函数, `lru_cache_add()` 和 `lru_cache_add_active()` 稍有些复杂。这两个函数实际上并没有立刻把页移到 LRU, 而是在 `pagevec` 类型的临时数据结构中聚集这些页, 每个结构可以存放多达 14 个页描述符指针。只有当一个 `pagevec` 结构写满了, 页才真正被移到 LRU 链表中。这种机制可以改善系统性能, 这是因为只当 LRU 链表实际修改后才获得 LRU 自旋锁。

在 LRU 链表之间移动页

PFRA 把最近访问过的页集中放在活动链表中, 以便当查找要回收的页框时不扫描这些页。相反, PFRA 把很长时间没有访问的页集中放在非活动链表中。当然, 应该根据页是否正被访问, 把页从非活动链表移到活动链表或者退回。

显然, 两个状态 (“活动” 和 “非活动”) 是不足以描述所有可能的访问模式的。例如, 假定日志进程每隔 1 小时把一些数据写入一个页中。尽管这个页是 “不活动的” 已经很长时间, 但是访问使它变为 “活动的”, 因此即使这一页在整整 1 小时内没有被访问, 也不回收相应的页框。当然, 对这种问题并没有通用的解决方法, 因为 PFRA 没有办法预测用户态进程的行为; 不过, 页不应该在每次单独的访问中就改变自己的状态似乎是合理的。

在页描述符中的 `PG_referenced` 标志用来把一个页从非活动链表移到活动链表所需的访问次数加倍; 也把一个页从活动链表移到非活动链表所需的 “丢失访问” 次数加倍 (见下面)。例如, 假定在非活动链表中的一个页其 `PG_referenced` 标志置为 0。第一次访问把这个标志置为 1, 但是这一页仍然留在非活动链表中。第二次对该页访问时发现这

一标志被设置, 因此, 把页移到活动链表。但是, 如果第一次访问之后在给定的时间间隔内第二次访问没有发生, 那么页框回收算法就可能重置 PG_referenced 标志。

如图17-4所示, PFRA使用 mark_page_accessed()、page_referenced() 和 refill_inactive_zone() 函数在 LRU 链表之间移动页。在图中, 包含有页的 LRU 链表由 PG_active 标志的状态表示。

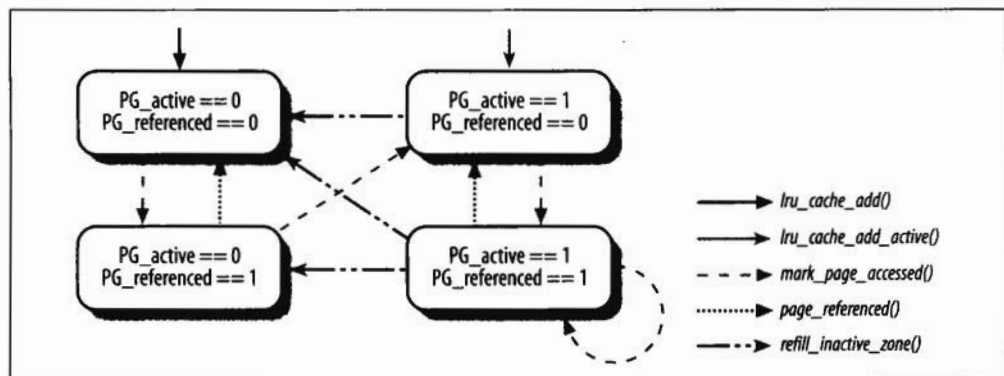


图 17-4: 在 LRU 链表之间移动页

mark_page_accessed() 函数

当内核必须把一个页标记为访问过时, 就调用 mark_page_accessed() 函数。每当内核决定一个页是被用户态进程、文件系统层还是设备驱动程序引用时, 这种情况就会发生。例如, 在下列情况下调用 mark_page_accessed():

- 当按需装入进程的一个匿名页时 (由 do_anonymous_page() 函数执行; 参见第九章“请求调页”一节)。
- 当按需装入内存映射文件的一个页时 (由 filemap_nopage() 函数执行; 参见第十六章“内存映射的请求调页”一节)。
- 当按需装入 IPC 共享内存区的一个页时 (由 shmem_nopage() 函数执行; 参见第十九章“IPC 共享内存”一节)。
- 当从文件读取数据页时 (由 do_generic_file_read() 函数执行; 参见第十六章“从文件中读取数据”一节)。
- 当换入一个页时 (由 do_swap_page() 函数执行; 参见后面的“换入页”一节)。
- 当在页高速缓存中搜索一个缓冲区页时 (参见第十五章“在页高速缓存中搜索块”一节中介绍的 __find_get_block() 函数)。

mark_page_accessed()函数执行下列代码片段：

```
if (!PageActive(page) && PageReferenced(page) && PageLRU(page)) {
    activate_page(page);
    ClearPageReferenced(page);
} else if (!PageReferenced(page))
    SetPageReferenced(page);
```

如图 17-4 所示，该函数调用前，只有当 PG_referenced 标志置位，它才把页从非活动链表移到活动链表。

page_referenced()函数

PFRA 扫描一页调用一次 page_referenced()函数，如果 PG_referenced 标志或页表项中的某些 Accessed 标志位置位，则该函数返回 1，否则返回 0。该函数首先检查页描述符的 PG_referenced 标志。如果标志置位则清 0。然后使用面向对象的反向映射方法，对引用该页的所有用户态页表项中的 Accessed 标志位进行检查并清 0。为此，函数用到三个辅助函数：page_referenced_anon()、page_referenced_file()和 page_referenced_one()，这与本章前面“反向映射”一节中的 try_to_unmap_xxx()函数类似。page_referenced()函数还会用到交换标记 (swap token，参见本章后面“交换标记”一节)。

从活动链表到非活动链表移动页不是由 page_referenced()函数，而是由 refill_inactive_zone()函数实施的。实际上，refill_inactive_zone()函数除此之外还有其他很多功能，因此我们要进行深入的讨论。

refill_inactive_zone()函数

如图 17-3 所示，refill_inactive_zone()函数由 shrink_zone()调用，而 shrink_zone()函数对页高速缓存和用户态地址空间进行页回收（参见本章后面“内存紧缺回收”一节）。此函数有两个参数：zone 和 sc。指针 zone 指向一个内存管理区描述符，指针 sc 指向一个 scan_control 结构。PFRA 广泛使用 scan_control 这个数据结构，该结构存放着回收操作执行时的有关信息。表 17-2 中列出了它的字段。

表 17-2：scan_control 描述符的字段

类型	字段	说明
unsigned long	nr_to_scan	活动链表中待扫描的目标页数
unsigned long	nr_scanned	当前迭代中扫描过的非活动页数
unsigned long	nr_reclaimed	当前迭代中回收的页数
unsigned long	nr_mapped	用户态地址空间引用的页数

表 17-2: scan_control 描述符的字段 (续)

类型	字段	说明
int	nr_to_reclaim	待回收的目标页数
unsigned int	priority	扫描优先级, 范围从 12 到 0, 低优先级表示扫描更多的页
unsigned int	gfp_mask	调用进程传来的 GFP 掩码
int	may_writepage	如果置位, 则允许将脏页写到磁盘 (只针对便携情形)

refill_inactive_zone() 函数的工作至关重要, 因为, 从活动链表将页移到非活动链表就意味着页迟早要被 PFRA 捕获。如果函数的掠夺性过强, 就会有过多的页从活动链表被移动到非活动链表。因此, PFRA 就会回收大量的页框, 系统性能会受到影响。反过来, 如果函数太懒惰, 就没有足够的未用页来补充非活动链表, PFRA 就不能回收内存。为此, 该函数可以调整自己的行为: 开始时, 对每次调用, 扫描非活动链表中少量的页, 但是当 PFRA 很难回收内存时, refill_inactive_zone() 在每次调用时就逐渐增加扫描的活动页数。scan_control 数据结构中 priority 字段的值控制该函数的行为 (低值表示更紧迫的优先级)。

还有一个试探法可以调整 refill_inactive_zone() 函数行为。LRU 链表中有两类页: 属于用户态地址空间的页、不属于任何用户态进程且在页高速缓存中的页。如前所述, PFRA 倾向于压缩页高速缓存, 而将用户态进程的页留在 RAM 中。然而, 每一种策略中都没有一个固定的黄金法则保证系统的高性能, 所以 refill_inactive_zone() 函数使用交换倾向 (swap tendency) 经验值, 由它确定函数是移动所有的页还是只移动不属于用户态地址空间的页 (注 5)。函数按如下公式计算交换倾向值:

$$\text{交换倾向值} = \text{映射比率} / 2 + \text{负荷值} + \text{交换值}$$

映射比率 (mapped ratio) 是用户态地址空间所有内存管理区的页 (sc->nr_mapped) 占所有可分配页框数的百分比。mapped_ratio 的值大表示动态内存大部分用于用户态进程, 而值小则表示大部分用于页高速缓存。

负荷值 (distress) 用于表示 PFRA 在管理区中回收页框的效率。其依据是前一次 PFRA

注 5: “交换倾向”这一表述可能引起误解, 因为用户态地址空间的页可以是“可交换的”、“可同步的”以及“可丢弃的”。不过, 交换倾向值确实控制了 PFRA 实现的交换量, 因为几乎所有可交换的页都属于用户态地址空间。

运行时管理区的扫描优先级，这个优先级存放在管理区描述符的 `prev_priority` 字段。负荷值与管理区前一次优先级的对应关系如下：

管理区前一次优先级	12...7	6	5	4	3	2	1	0
负荷值	0	1	3	6	12	25	50	100

最后，交换值 (*swappiness*) 是一个用户定义常数，通常为60。系统管理员可以在 `/proc/sys/vm/swappiness` 文件内修改这个值，或用相应的 `sysctl()` 系统调用调整这个值。

只有当管理区交换倾向值大于等于100时，页才从进程地址空间回收。那么当系统管理员将交换值设为0时，PFRA 就不会从用户态地址空间回收页，除非管理区的前一次优先级为0（这不大可能发生）。如果系统管理员将交换值设为100，那么PFRA 每次调用该函数时都会从用户态地址空间回收页。

下面是 `refill_inactive_zone()` 函数执行步骤的一个简要说明：

1. 调用 `lru_add_drain()`，把仍留在 `pagevec` 数据结构中的所有页移入活动与非活动链表。
2. 获得 `zone->lru_lock` 自旋锁。
3. 对 `zone->active_list` 中的页进行首次扫描，从链表的底部开始向上，一直执行下去，直到链表为空或 `sc->nr_to_scan` 的页扫描完毕。在这一次循环中每扫描一页，函数就将引用计数器加1，从 `zone->active_list` 中删除页描述符，把它放在临时局部链表 `l_hold` 中。但是如果页框引用计数器是0，则把该页放回活动链表。实际上，引用计数器为0的页框一定属于管理区的伙伴系统，但释放页框时，首先递减使用计数器，然后将页框从LRU链表删除并插入伙伴系统链表。因此在一个很小的时间段，PFRA 可能会发现LRU链表中的空闲页。
4. 把已扫描的活动页数追加到 `zone->pages_scanned`。
5. 从 `zone->nr_active` 中减去移入局部链表 `l_hold` 中的页数。
6. 释放 `zone->lru_lock` 自旋锁。
7. 计算交换倾向值（见上面）。
8. 对局部链表 `l_hold` 中的页运行第二次循环。这次循环的目的是：把其中的页分到两个子链表 `l_active` 和 `l_inactive` 中。属于某个进程用户态地址空间的页（即 `page->mapcount` 为非负数的页）被加入 `l_active` 的条件是：交换倾向值小于100，或者是匿名页但又没有激活的交换区，或者应用于该页的 `page_referenced()` 函

- 数返回正数（正数表示该页最近被访问过）。而任何其他情形下，页被加入 `l_inactive` 链表（注6）。
9. 获得 `zone->lru_lock` 自旋锁。
 10. 对局部链表 `l_inactive` 中的页执行第三次循环。把页移入 `zone->inactive_list` 链表，更新 `zone->nr_inactive` 字段，同时递减被移页框的使用计数器，从而抵消第3步中增加的值。
 11. 对局部链表 `l_active` 中的页执行第四次也是最后一次循环。把页移入 `zone->active_list` 链表，更新 `zone->nr_active` 字段，同时递减被移页框的使用计数器，从而抵消第3步中增加的值。
 12. 释放自旋锁 `zone->lru_lock` 并返回。

注意，`refill_inactive_zone()` 只检查用户态地址空间页的 `PG_referenced` 标志（见第8步）。相反的情况是，页在活动链表的底部，也就是较长时间以前被访问过，那么不大可能会在近期被访问。另外，如果页属于某个用户态进程且最近被使用过，那么函数也不会将页从活动链表删除。

内存紧缺回收

当内存分配失败时激活内存紧缺回收。在图 17-3 中，在分配 VFS 缓冲区或缓冲区首部时，内核调用 `free_more_memory()`；而当从伙伴系统分配一个或多个页框时，调用 `try_to_free_pages()`。

`free_more_memory()` 函数

`free_more_memory()` 函数执行如下操作：

1. 调用 `wakeup_bdflush()` 唤醒一个 `pdflush` 内核线程，并触发页高速缓存中 1024 个脏页的写操作（参见第十五章的“`pdflush` 内核线程”一节）。写脏页到磁盘的操作将最终使包含缓冲区、缓冲区首部和其他 VFS 数据结构的页框成为可释放的。
2. 调用 `sched_yield()` 系统调用的服务例程，为 `pdflush` 内核线程提供执行机会。
3. 对系统的所有内存节点，启动一个循环[参见第八章的“非一致内存访问(NUMA)”

注6： 注意，不属于任何用户态地址空间的页被移动到非活动链表中，但是由于它的 `PG_referenced` 标志没有清0，所以对该页的第一次访问导致函数 `mark_page_accessed()` 把该页移回活动链表中（参见图 17-4）。

一节]。对每一个节点，调用 `try_to_free_pages()` 函数，传给它的参数是一个“紧缺”内存管理区链表（在 80x86 体系结构中是 `ZONE_DMA` 和 `ZONE_NORMAL`，参见第八章的“内存管理区”一节）。

`try_to_free_pages()` 函数

`try_to_free_pages()` 函数接收如下三个参数：

`zones`

要回收的页所在的内存管理区链表（参见第八章的“内存管理区”一节）。

`gfp_mask`

用于失败的内存分配的一组分配标志（参见第八章的“分区页框分配器”一节）。

`order`

没有使用。

该函数的目标就是通过重复调用 `shrink_caches()` 和 `shrink_slab()` 函数释放至少 32 个页框，每次调用后优先级会比前一次提高。有关的辅助函数可以获得 `scan_control` 类型描述符中的优先级，以及正在进行的扫描操作的其他参数（见前面的表 17-2）。最低的、也是初始的优先级是 12，而最高的、也是最终的优先级是 0。如果 `try_to_free_pages()` 没能在某次（共 13 次）调用 `shrink_caches()` 和 `shrink_slab()` 函数时成功回收至少 32 个页框，PFRA 就要黔驴技穷了。最后一招：删除一个进程，释放它的所有页框。这一操作由 `out_of_memory()` 函数执行（参见本章后面“内存不足删除程序”一节）。

该函数主要执行如下步骤：

1. 分配和初始化一个 `scan_control` 描述符，具体说就是把分配掩码 `gfp_mask` 存入 `gfp_mask` 字段。
2. 对 `zones` 链表中的每个管理区，将管理区描述符的 `temp_priority` 字段设为初始优先级 12，而且计算管理区 LRU 链表中的总页数。
3. 从优先级 12 到 0，执行最多 13 次的循环，每次迭代执行如下子步骤：
 - a. 更新 `scan_control` 描述符的一些字段。具体地，把用户态进程的总页数存入 `nr_mapped` 字段，把本次迭代的当前优先级存入 `priority` 字段。而且将 `nr_scanned` 和 `nr_reclaimed` 字段设为 0。
 - b. 调用 `shrink_caches()`，传给它 `zones` 链表和 `scan_control` 描述符地址作为参数。这个函数扫描管理区的非活动页（见下面）。

- c. 调用 `shrink_slab()` 从可压缩内核高速缓存中回收页（参见后面“回收可压缩磁盘高速缓存的页”一节）。
 - d. 如果 `current->reclaim_state` 非空，则将 slab 分配器高速缓存中回收的页数（该数存放在一个由进程描述符字段指向的小型数据结构中）追加到 `scan_control` 描述符的 `nr_reclaimed` 字段。在调用 `try_to_free_pages()` 函数之前，`_alloc_pages()` 函数建立 `current->reclaim_state` 字段，并在结束后马上清除该字段（不可思议的是，`free_more_memory()` 不设置这个字段）。
 - e. 如果已达目标（`scan_control` 描述符的 `nr_reclaimed` 字段大于等于 32），则跳出循环到第 4 步。
 - f. 如果未达目标，但已扫描完成至少 49 页，函数则调用 `wakeup_bdflood()` 激活 `pdflush` 内核线程，并将页高速缓存中的一些脏页写入磁盘（参见第十五章的“搜索要刷新的脏页”一节）。
 - g. 如果函数已完成 4 次迭代而又未达目标，则调用 `blk_congestion_wait()` 挂起进程，一直到没有拥塞的 WRITE 请求队列或 100ms 超时已过（参见第十四章的“请求描述符”一节）。
- 4. 把每个管理区描述符的 `prev_priority` 字段设为上一次调用 `shrink_caches()` 使用的优先级，该值存放在管理区描述符的 `temp_priority` 字段。
 - 5. 如果成功回收则返回 1，否则返回 0。

shrink_caches() 函数

`shrink_caches()` 函数由 `try_to_free_pages()` 调用，它有两个参数：内存管理区链表 `zones` 和 `scan_control` 描述符地址 `sc`。

该函数的目的只是对 `zones` 链表中的每个管理区调用 `shrink_zone()` 函数。但对给定管理区调用 `shrink_zone()` 之前，`shrink_caches()` 函数用 `sc->priority` 字段的值更新管理区描述符的 `temp_priority` 字段，这就是扫描操作的当前优先级。而且如果 PFRA 的上一次调用优先级高于当前优先级，即这个管理区进行页框回收变得更难了，那么 `shrink_caches()` 把当前优先级拷贝到管理区描述符的 `prev_priority`。最后，如果管理区描述符中的 `all_unreclaimable` 标志置位，且当前优先级小于 12，则 `shrink_caches()` 不调用 `shrink_zone()`，也就是说，在 `try_to_free_pages()` 的第一迭代中不调用 `shrink_caches()`。当 PFRA 确定一个管理区都是不可回收页，扫描该管理区的页纯粹是浪费时间时，则将 `all_unreclaimable` 标志置位。

shrink_zone()函数

shrink_zone()函数有两个参数：zone和sc。zone是指向struct_zone描述符的指针；sc是指向scan_control描述符的指针。该函数的目标是从管理区非活动链表回收32页。它每次在更大的一段管理区非活动链表上重复调用辅助函数shrink_cache()，以期达到目标。而且shrink_zone()重复调用refill_inactive_zone()函数来补充管理区非活动链表[参见前面“最近最少使用(LRU)链表”一节]。

管理区描述符的nr_scan_active和nr_scan_inactive字段在这里起到很重要的作用。为提高效率，函数每批处理32页。因此如果函数在低优先级运行（对应sc->priority的高值），且某个LRU链表中没有足够的页，函数就跳过对这个链表的扫描。但因此跳过的活动或不活动页数就分别存放在nr_scan_active或nr_scan_inactive中，这样函数下次执行时再处理这些跳过的页。

shrink_zone()函数的具体执行步骤如下：

1. 递增zone->nr_scan_active，增量是活动链表(zone->nr_active)的一小部分。实际增量取决于当前优先级，其范围是：zone->nr_active/2¹²到zone->nr_active/2⁰（即管理区内的总活动页数）。
2. 递增zone->nr_scan_inactive，增量是非活动链表(zone->nr_inactive)的一小部分。实际增量取决于当前优先级，其范围是：zone->nr_inactive/2¹²到zone->nr_inactive。
3. 如果zone->nr_scan_active字段大于等于32，函数就把该值赋给局部变量nr_active，并把该字段设为0，否则把nr_active设为0。
4. 如果zone->nr_scan_inactive字段大于等于32，函数就把该值赋给局部变量nr_inactive，并把该字段设为0，否则把nr_inactive设为0。
5. 设定scan_control描述符的sc->nr_to_reclaim字段为32。
6. 如果nr_active和nr_inactive都为0，则无事可做，函数结束。这不常见，用户态进程没有被分配到任何页时才可能出现这种情形。
7. 如果nr_active为正，则补充管理区非活动链表：

```
sc->nr_to_scan = min(nr_active, 32)
nr_active -= sc->nr_to_scan
refill_inactive_zone(zone, sc)
```

8. 如果nr_inactive为正，则尝试从非活动链表回收最多32页：

```
sc->nr_to_scan = min(nr_inactive, 32)
nr_inactive -= sc->nr_to_scan
shrink_cache(zone, sc)
```

9. 如果 `shrink_zone()` 成功回收 32 页（现在 `sc->nr_to_reclaim` 小于等于 0），则结束；否则，跳回第 6 步。

`shrink_cache()` 函数

`shrink_cache()` 函数又是一个辅助函数，它的主要目的就是管理区非活动链表取出一组页，把它们放入一个临时链表，然后调用 `shrink_list()` 函数对这个链表中的每一页进行有效的页框回收操作。`shrink_cache()` 函数的参数与 `shrink_zones()` 一样，都是 `zone` 和 `sc`，执行的主要步骤如下：

1. 调用 `lru_add_drain()`，把仍然在 `pagevec` 数据结构中的页移入活动与非活动链表[参见本章前面“最近最少使用 (LRU) 链表”一节]。
2. 获得 `zone->lru_lock` 自旋锁。
3. 处理非活动链表中的页（最多 32 页），对于每一页，函数递增使用计数器；检查该页是否不会被释放到伙伴系统（参见 `refill_inactive_zone()` 的第 3 步的讨论），把页从管理区非活动链表移入一个局部链表。
4. 把 `zone->nr_inactive` 计数器的值减去从非活动链表中删除的页数。
5. 递增 `zone->pages_scanned` 计数器的值，增量为在非活动链表中有效检查的页数。
6. 释放 `zone->lru_lock` 自旋锁。
7. 调用 `shrink_list()` 函数，传给它上面第 3 步中搜集的页（在局部链表中）。下面将详细讨论（你一定很期盼的讨论）。
8. 把 `sc->nr_to_reclaim` 字段的值减去由 `shrink_list()` 实际回收的页数。
9. 再次获取 `zone->lru_lock` 自旋锁。
10. 把局部链表中 `shrink_list()` 没有成功释放的页放回非活动或活动链表。注意，`shrink_list()` 有可能置位 `PG_active` 标志，从而将某页标记为活动页。这一操作使用 `pagevec` 数据结构对一组页进行处理[参见本章前面“最近最少使用 (LRU) 链表”一节]。
11. 如果函数扫描的页数至少是 `sc->nr_to_scan`，且如果没有成功回收目标页数（即 `sc->nr_to_reclaim` 仍然大于 0），则跳回第 3 步。
12. 释放 `zone->lru_lock` 自旋锁并结束。

`shrink_list()` 函数

我们现在讨论页框回收算法的核心部分。从 `try_to_free_pages()` 到 `shrink_cache()` 函数，前面所述这些函数的目的就是找到一组适合回收的候选页。`shrink_list()` 函数则从

参数 `page_list` 链表中尝试回收这些页，该函数的第二个参数 `sc` 是指向 `scan_control` 描述符的指针。当 `shrink_list()` 返回时，`page_list` 链表中剩下的是无法回收的页。

函数执行步骤如下：

1. 如果当前进程的 `need_resched` 字段置位，则调用 `schedule()`。
2. 执行一个循环，处理 `page_list` 链表中的每一页。对其中每个元素，从链表中删除页描述符并尝试回收该页框。如果由于某种原因页框不能释放，则把该页描述符插入一个局部链表。
3. 现在 `page_list` 已空，函数再把页描述符从局部链表移回 `page_list` 链表。
4. 递增 `sc->nr_reclaimed` 字段，增量为第 2 步中回收的页数，并返回这个数。

当然，`shrink_list()` 函数尝试回收页框的代码确实很有意思。图 17-5 是这段代码的流程图。

`shrink_list()` 处理的每个页框只可能有三种结果：

- 调用 `free_cold_page()` 函数，把页释放到管理区伙伴系统（参见第八章中“每 CPU 页框高速缓存”一节），因此被有效回收。
- 页没有被回收，因此被重新插入 `page_list` 链表。但是 `shrink_list()` 假设不久还能回收该页。因此函数让页描述符的 `PG_active` 标志保持清 0，这样页将被放回内存管理区的非活动链表（参见前面 `shrink_cache()` 函数描述的第 9 步）。这种情况对应于图 17-5 中标为“INACTIVE”的小方框。
- 页没有被回收，因此被重新插入 `page_list` 链表。但是，或是页正被使用，或是 `shrink_list()` 假设近期无法回收该页。函数将页描述符的 `PG_active` 标志置位，这样页将被放回内存管理区的活动链表。这种情况对应于图 17-5 中标为“ACTIVE”的小方框。

`shrink_list()` 函数不会去回收锁定页（`PG_locked` 置位）与写回页（`PG_writeback` 置位）。`shrink_list()` 调用 `page_referenced()` 函数检查该页是否最近被引用过，参见本章前面“最近最少使用（LRU）链表”一节中的描述。

要回收匿名页，就必须把它加入交换高速缓存，那么就必须在交换区为它保留一个新页槽（slot）。参见本章后面“交换”一节的详细讨论。

如果页在某个进程的用户态地址空间（页描述符的 `_mapcount` 字段大于等于 0），则 `shrink_list()` 调用 `try_to_unmap()` 寻找引用该页框的所有页表项（参见本章前面“反向映射”一节）。当然，只有当这个函数返回 `SWAP_SUCCESS` 时，回收才可继续。

如果是脏页，则写回磁盘前不能回收。为此，`shrink_list()`使用`pageout()`函数（后面会加以说明）。只有当`pageout()`不必进行写操作或写操作不久将结束时，回收才可继续。

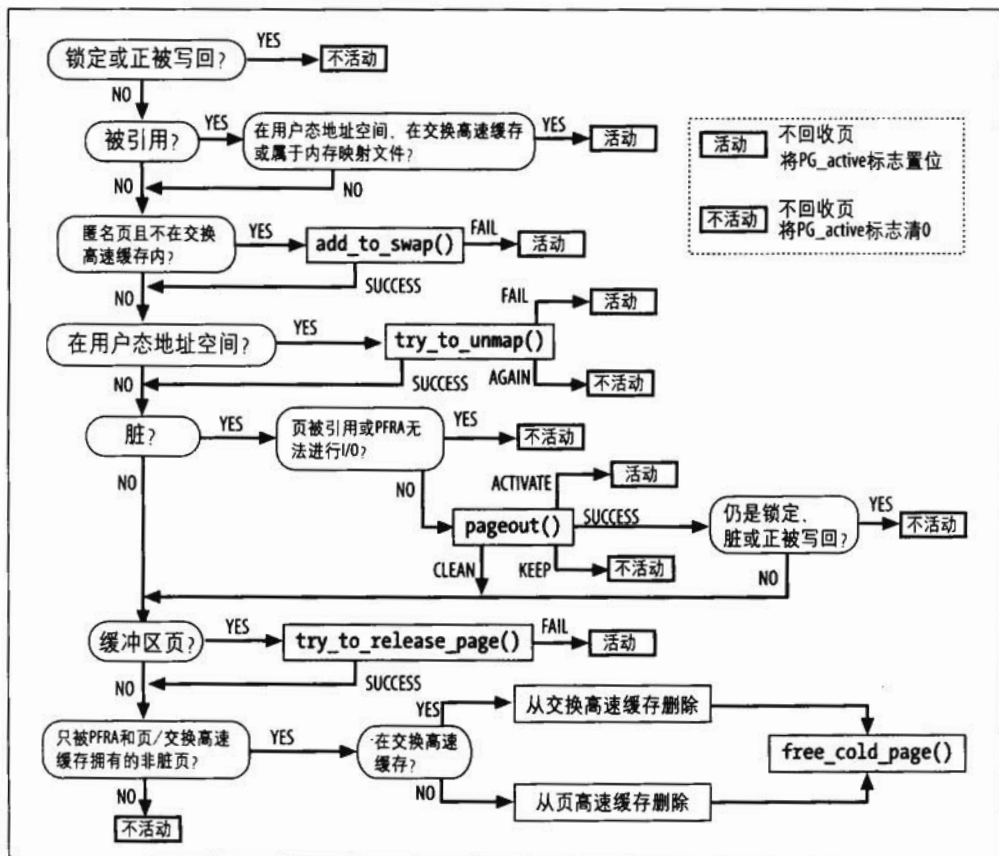


图 17-5: `shrink_list()` 函数的页框回收流程

如果页包含 VFS 缓冲区，则 `shrink_list()` 调用 `try_to_release_page()` 释放关联的缓冲区首部（参见第十五章中“释放块设备缓冲区页”一节）。

最后，如果一切顺利，`shrink_list()` 就检查页的引用计数器。若等于 2，那么这两个拥有者就是：页高速缓存（如果是匿名页，则为交换高速缓存）和 PFRA 自己（`shrink_cache()` 函数中第 3 步中会递增引用计数器，参见前面）。这种情况下，如果页仍然不为脏，则页可以回收。为此，首先根据页描述符的 `PG_swapcache` 标志的值，从页高速缓存或交换高速缓存删除该页，然后，执行函数 `free_cold_page()`。

pageout()函数

当一个脏页必须写回磁盘时，`shrink_list()`调用`pageout()`函数。函数执行的主要步骤如下：

1. 检查页存放在页高速缓存还是交换高速缓存中（参见本章后面“交换高速缓存”一节）。进一步检查该页是否由页高速缓存（或交换高速缓存）与PFRA拥有。如果检查失败，则返回`PAGE_KEEP`（如果没有被`shrink_list()`回收，则写页到磁盘就没有意义了）。
2. 检查`address_space`对象的`writepage`方法是否已定义。如果没有，则返回`PAGE_ACTIVATE`。
3. 检查当前进程是否可以向块设备（与`address_space`对象对应）请求队列发出写请求。实际上，`kswapd`和`pdflush`内核线程总会发出写请求；而普通进程只有在请求队列不拥塞的情况下才能发出写请求，除非`current->backing_dev_info`字段指向块设备的`backing_dev_info`数据结构（参见第十六章“写入文件”一节中`generic_file_aio_write_nolock()`函数描述的第3步）。
4. 检查是否仍然是脏页。如果不是则返回`PAGE_CLEAN`。
5. 建立一个`writeback_control`描述符，调用`address_space`对象的`writepage`方法以启动一个写回操作（参见第十六章中“将脏页写到磁盘”一节）。
6. 如果`writepage`方法返回错误码，则函数返回`PAGE_ACTIVATE`。
7. 返回`PAGE_SUCCESS`。

回收可压缩磁盘高速缓存的页

我们从前面的章节中知道，内核在页高速缓存之外还使用其他磁盘高速缓存，例如，目录项高速缓存与索引节点高速缓存（参见第十二章“目录项高速缓存”）。当要回收其中的页框时，PFRA就必须检查这些磁盘高速缓存是否可压缩。

PFRA处理的每个磁盘高速缓存在初始化时必须注册一个`shrinker`函数。`shrinker`函数有两个参数：待回收页框数和一组GFP分配标志。函数按照要求从磁盘高速缓存回收页，然后返回仍然留在高速缓存内的可回收页数。

`set_shrinker()`函数向PFRA注册一个`shrinker`函数。该函数分配一个`shrinker`类型的描述符，在该描述符中存放`shrinker`函数的地址，然后把描述符插入一个全局链表，该链表存放在`shrinker_list`全局变量中。`set_shrinker()`函数还初始化`shrinker`描述符的`seeks`字段，通俗地说，这个字段表示：在高速缓存中的元素一旦被删除，那么重建一个所需的代价。

在 Linux 2.6.11 中，向 PFRA 注册的磁盘高速缓存很少。除了目录项高速缓存和索引节点高速缓存之外，注册 shrinker 函数的只有磁盘限额层、文件系统元信息块高速缓存（主要用于文件系统扩展属性）和 XFS 日志文件系统。

从可压缩磁盘高速缓存回收页的 PFRA 函数叫作 shrink_slab()（函数名有点误导，因为该函数与 slab 分配器高速缓存没什么关系）。它由 try_to_free_pages()（在前面“内存紧缺回收”一节中有描述）和 balance_pgdat() 调用（在后面的“周期回收”一节会有描述）。

对于从可压缩磁盘高速缓存回收的代价与及从 LRU 链表回收的代价（由 shrink_list() 执行）之间，shrink_slab() 函数试图作出一种权衡。实际上，函数扫描 shrinker 描述符的链表，调用这些 shrinker 函数并得到磁盘高速缓存中总的可回收页数。然后，函数再一次扫描 shrinker 描述符的链表，对于每个可压缩磁盘高速缓存，函数推算出待回收页框数。推算考虑的因素有：磁盘高速缓存中总的可回收页数、在磁盘高速缓存中重建一页的相关代价、LRU 链表中的页数。然后再调用 shrinker 函数尝试回收一组页（至少 128 页）。

因篇幅所限，我们只简单讨论目录项高速缓存和索引节点高速缓存的 shrinker 函数。

从目录项高速缓存回收页框

shrink_dcache_memory() 函数是目录项高速缓存的 shrinker 函数。它搜索高速缓存中的未用目录项对象，即没有被任何进程引用的目录项对象，然后将它们释放（参见第十二章的“目录项对象”一节）。

由于目录项高速缓存对象是通过 slab 分配器分配的，因此 shrink_dcache_memory() 函数可能导致一些 slab 变成空闲的，这样有些页框就可以被 cache_reap() 回收（参见本章后面“周期回收”一节）。此外，目录项高速缓存起索引节点高速缓存控制器的作用，因此，当一个目录项对象被释放时，存放相应索引节点对象的页就可以变为未用，而最终被释放。

shrink_dcache_memory() 函数接收两个参数：待回收页框数和 GFP 掩码。一开始，它检查 GFP 掩码中的 __GFP_FS 标志位是否清 0，如果是则返回 -1，因为释放目录项可能触发基于磁盘文件系统的操作。通过调用 prune_dcache()，就可以有效地进行页框回收。该函数扫描未用目录项链表（该链表的头部存放在 dentry_unused 变量中），一直到获得请求数量的释放对象或整个链表扫描完毕。对每个最近未被引用的对象，函数执行如下步骤：

1. 把目录项对象从目录项散列表、从其父目录中的目录项对象链表、从拥有者索引节点的目录项对象链表中删除。

2. 调用 `d_iput` 目录项方法（如果定义）或者 `iput()` 函数减少目录项的索引节点的引用计数器。
3. 调用目录项对象的 `d_release` 方法（如果定义）。
4. 调用 `call_rcu()` 函数以注册一个会删除目录项对象的回调函数[参见第五章“读-拷贝-更新（RCU）”一节]，该回调函数又调用 `kmem_cache_free()` 把对象释放给 slab 分配器（参见第八章“从高速缓存中释放 slab”一节）。
5. 减少父目录的引用计数器。

最后，依据仍然留在目录项高速缓存中的未用目录项数，`shrink_dcache_memory()` 返回一个值。更准确地说，返回值是未用目录项数乘以 100 除以 `sysctl_vfs_cache_pressure` 全局变量的值。该变量的系统默认值是 100，因此返回值实际就是未用目录项数。但是通过修改文件 `/proc/sys/vm/vfs_cache_pressure` 或通过有关的 `sysctl()` 系统调用，系统管理员可以改变这个变量值。把值改为小于 100，则使 `shrink_slab()` 从目录项高速缓存（与索引节点高速缓存，见下一节）回收的页少于从 LRU 链表中回收的页。反之，如把值改为大于 100，则使 `shrink_slab()` 从目录项高速缓存回收的页多于从 LRU 链表中回收的页。

从索引节点高速缓存回收页框

`shrink_ichache_memory()` 函数被调用来从索引节点高速缓存删除未用索引节点对象。“未用”就是指索引节点不再有一个控制目录项对象。这个函数非常类似于刚描述的 `shrink_dcache_memory()`。它检查 `gfp_mask` 参数的 `__GFP_FS` 位，然后调用 `prune_ichache()`，最后与前面一样，依据仍然留在索引节点高速缓存中的未用索引节点数和 `sysctl_vfs_cache_pressure` 变量的值，返回一个值。

`prune_ichache()` 函数又扫描 `inode_unused` 链表（参见第十二章“索引节点对象”一节）。要释放一个索引节点，函数必须释放与该索引节点关联的任何私有缓冲区，它使页高速缓存内（引用该索引节点的）不再使用的干净页框无效，然后通过调用 `clear_inode()` 和 `destroy_inode()` 函数来删除索引节点对象。

周期回收

PFRA 用两种机制进行周期回收：`kswapd` 内核线程和 `cache_reap` 函数。前者调用 `shrink_zone()` 和 `shrink_slab()` 从 LRU 链表中回收页，后者则被周期性地调用以便从 slab 分配器中回收未用的 slab。

kswapd 内核线程

kswapd 内核线程是激活内存回收的另外一种机制。为什么还需要这个内核线程呢？当空闲内存变得紧缺并且发出另一个内存分配请求时，调用 `try_to_free_pages()` 还不够吗？

遗憾的是，实际情形并非如此。有些内存分配请求是由中断和异常处理程序执行的，它们不会阻塞等待释放页框的当前进程；还有，有些内存分配请求是由已经获得对临界资源互斥访问权限，因此就不能激活 I/O 数据传送的内核控制路径实现的。在极少情况下，所有的内存分配请求都是由这种内核控制路径完成的，因此内核将永远不能释放空闲内存。

kswapd 利用机器空闲的时间保持内存空闲也对系统性能有良好的影响，进程因此能很快获得自己的页。

每个内存节点对应各自的 *kswapd* 内核线程[参见第八章中“非一致内存访问 (NUMA)”一节]。每个这样的线程通常睡眠在等待队列中，该等待队列以节点描述符的 `kswapd_wait` 字段为头部。但是，如果 `_alloc_pages()` 发现所有适合内存分配的内存管理区包含的空闲页框数低于“警告”阈值（一个依据内存管理区描述符的 `pages_low` 和 `protection` 字段推算出来的值）时，那么相应内存节点的 *kswapd* 内核线程被激活（参见第八章“管理区分配器”一节）。从本质上说，为了避免更多紧张的“内存紧缺”的情形，内核才开始回收页框。

正如第八章“保留的页框池”一节所述，每个管理区描述符还包括字段 `pages_min` 和 `pages_high`。前者表示必须保留的最小空闲页框数阈值；后者表示“安全”空闲页框数阈值，即空闲页框数大于该阈值时，应该停止页框回收。

kswapd 内核线程执行 `kswapd()` 函数。内核线程被初始化的内容是：把线程绑定到访问内存节点的 CPU；再把 `reclaim_state` 描述符地址存入进程描述符的 `current->reclaim_state` 字段（参见本章前面 `try_to_free_pages()` 函数的描述中的第 3d 步）；把 `current->flags` 字段的 `PF_MEMALLOC` 和 `PF_KSWAP` 标志置位，其含义是进程将回收内存，运行时允许使用全部可用空闲内存。每当 *kswapd* 内核线程被唤醒，`kswapd()` 函数执行下列主要操作：

1. 调用 `finish_wait()` 从节点的 `kswapd_wait` 等待队列删除内核线程（参见第三章中“如何组织进程”一节）。
2. 调用 `balance_pgdat()` 对 *kswapd* 的内存节点进行内存回收（见下面）。
3. 调用 `prepare_to_wait()` 把进程设成 `TASK_INTERRUPTIBLE` 状态，并让它在节点的 `kswapd_wait` 等待队列中睡眠。

4. 调用 `schedule()` 让 CPU 处理一些其他可运行进程。

`balance_pgdat()` 函数又执行下面的主要步骤：

1. 建立 `scan_control` 描述符（参见本章前面的表 17-2）。
2. 把内存节点的每个管理区描述符中的 `temp_priority` 字段设为 12（最低优先级）。
3. 执行一个循环，从 12 到 0 最多 13 次迭代。每次迭代执行下列子步骤：
 - a. 扫描内存管理区，寻找空闲页框数不足的最高管理区（从 `ZONE_DMA` 到 `ZONE_HIGHMEM`）。由 `zone_watermark_ok()` 函数进行每次的检测（参见第八章中“管理区分配器”一节的描述）。如果所有管理区都有大量空闲页框，则跳到第 4 步。
 - b. 对一部分管理区再一次进行扫描，范围是从 `ZONE_DMA` 到第 3a 步找到的管理区。对每个管理区，必要时用当前优先级更新管理区描述符的 `prev_priority` 字段，且连续调用 `shrink_zone()` 以回收管理区中的页（参见前面“内存紧缺回收”一节）。然后，调用 `shrink_slab()` 从可压缩磁盘高速缓存回收页（参见前面“回收可压缩磁盘高速缓存的页”一节）。
 - c. 如果已有至少 32 页被回收，则跳出循环至第 4 步。
4. 用各自 `temp_priority` 字段的值更新每个管理区描述符的 `prev_priority` 字段。
5. 如果仍有“内存紧缺”管理区存在，且如果进程的 `need_resched` 字段置位，则调用 `schedule()`。当再一次执行时，跳到第 1 步。
6. 返回回收的页数。

cache_reap()函数

PFRA 还必须回收 slab 分配器高速缓存的页（参见第八章“内存区管理”一节）。为此，它使用 `cache_reap()` 函数，该函数周期性（差不多每两秒一次）地在预定事件工作队列（参见第四章“工作队列”一节）中被调度。它的地址存放在每 CPU 变量 `reap_work` 的 `func` 字段，该变量为 `work_struct` 类型。

`cache_reap()` 函数主要执行如下步骤：

1. 尝试获得 `cache_chain_sem` 信号量，该信号量保护 slab 高速缓存描述符链表。如果信号量已取得，就调用 `schedule_delayed_work()` 去调度该函数的下一次执行，然后结束。
2. 否则，扫描存放在 `cache_chain` 链表中的 `kmem_cache_t` 描述符。对找到的每一个高速缓存描述符，函数执行以下步骤：

- a. 如果高速缓存描述符的 `SLAB_NO_REAP` 标志置位, 则页框回收被禁止, 因此处理链表中的下一个高速缓存。
 - b. 清空局部 slab 高速缓存 (参见第八章的“空闲 slab 对象的本地高速缓存”一节), 则会有新的 slab 被释放。
 - c. 每个高速缓存都有“收割时间 (reap time)”, 该值存放在高速缓存描述符中 `kmem_list3` 结构的 `next_reap` 字段 (参见第八章的“高速缓存描述符”一节)。如果 `jiffies` 值仍然小于 `next_reap`, 则继续处理链表中的下一个高速缓存。
 - d. 把存放在 `next_reap` 字段的下一次“收割时间”设为: 从现时起的 4s。
 - e. 在多处理器系统中, 函数清空 slab 共享高速缓存 (参见第八章中“空闲 slab 对象的本地高速缓存”一节), 那么会有新的 slab 被释放。
 - f. 如有新的 slab 最近被加入高速缓存, 即高速缓存描述符中 `kmem_list3` 结构的 `free_touched` 标志置位, 那么跳过这个高速缓存, 继续处理链表中的下一个高速缓存。
 - g. 根据经验公式计算要释放的 slab 数量。基本上, 这个数取决于高速缓存中空闲对象数的上限和能装入单个 slab 的对象数。
 - h. 对高速缓存空闲 slab 链表中的每个 slab, 重复调用 `slab_destroy()`, 一直到链表为空或者已回收目标数量的空闲 slab。
 - i. 调用 `cond_resched()` 检查当前进程的 `TIF_NEED_RESCHED` 标志, 如果该标志置位, 则调用 `schedule()`。
3. 释放 `cache_chain_sem` 信号量。
 4. 调用 `schedule_delayed_work()` 去调度该函数的下一次执行, 然后结束。

内存不足删除程序

尽管 PFRA 尽量保留一定的空闲页框数, 但虚拟内存子系统的压力可能变得很高, 以至于所有可用内存耗尽。这很快会造成系统内的所有工作冻结。为满足一些紧迫请求, 内核总试图释放内存, 但是无法成功, 这是因为交换区已满且所有磁盘高速缓存已被压缩。因此, 没有进程可以继续执行, 也就没有进程会释放它所拥有的页框。

为应对这种突发情况, PFRA 使用所谓的内存不足 (*out of memory*, OOM) 删除程序, 该程序选择系统中的一个进程, 强行删除它并释放页框。OOM 删除程序就像是外科大夫, 为挽救一个人的生命而进行截肢。失去手脚当然是坏事, 但这是不得已而为之。

当空闲内存十分紧缺且 PFRA 又无法成功回收任何页时, `__alloc_pages()` 调用 `out_of_`

`memory()` 函数 (参见第八章中“管理区分配器”一节)。函数调用 `select_bad_process()` 在现有进程中选择一个“牺牲品”，然后调用 `oom_kill_process()` 删除该进程。

当然，`select_bad_process()` 并不是随机挑选进程的。被选进程应满足下列条件：

- 它必须拥有大量页框，从而可以释放出大量内存（为应对“子母弹”进程，函数计算母进程所属所有子进程的内存占用总量）。
- 删除它只损失少量工作成果（删除一个工作了几个小时或几天的批处理进程就不是个好主意）。
- 它应具有较低的静态优先级，用户通常给不太重要的进程赋予较低的优先级。
- 它不应是有 `root` 特权的进程，特权进程的工作通常比较重要。
- 它不应直接访问硬件块设备（如 X Window 服务器），因为硬件不能处在一个无法预知的状态。
- 它不能是 `swapper`（进程 0）、`init`（进程 1）和任何其他内核线程。

`select_bad_process()` 函数扫描系统中的每一个进程，根据以上准则用经验公式计算一个值，这个值表示选择这个进程的有利程度，然后返回最有利的被选进程描述符的地址。`out_of_memory()` 函数再调用 `oom_kill_process()` 并发出死亡信号（通常是 `SIGKILL`，参见第十一章），该信号发给该进程的一个子进程，或如果做不到，就发给该进程本身。`oom_kill_process()` 同时也删除与被选进程共享内存描述符的所有克隆进程。

交换标记

在阅读本章时，你可能认识到 Linux VM 子系统的代码太复杂，尤其是 PFRA，以致于无法预测任意负荷下它的行为。而且在有些情形下，VM 子系统表现出了一些病态行为。交换失效 (*swap thrashing*) 现象就是其中一例：当系统内存不足时，PFRA 全力把页写入磁盘以释放内存并从一些进程窃取相应的页框；而同时，这些进程要继续执行，也全力访问它们的页。因此内核把 PFRA 刚释放的页框又分配给这些进程，并从磁盘读回其内容。其结果就是页被无休止地写入磁盘并且再从磁盘读回。大部分的时间耗在访问磁盘上，从而没有进程能实质性地运行下去。

为减少交换失效的发生，一种由 Jiang 和 Zhang 在 2004 年提出的技术在内核版本 2.6.9 中得到实现。即把所谓的交换标记 (*swap token*) 赋给系统中的单个进程，该标记可以使该进程免于页框回收，所以进程可以实质性地运行，而且即使内存十分稀少，也有希望运行至结束。

交换标记的具体实现形式是 `swap_token_mm` 内存描述符指针。当进程拥有交换标记时，`swap_token_mm` 被设为进程内存描述符的地址。

页框回收算法的免除以如此简洁的方式实现了。我们在“最近最少使用 (LRU) 链表”一节看到，只当最近没有被引用时，一页才可从活动链表移入非活动链表。`page_referenced()` 函数进行这一检查。如果该页属于一个线性区，该区域所在进程拥有交换标记，那么该函数认可这个交换标记并返回 1 (被引用)。实际上，交换标记在几种情况下不予考虑：PFRA 代表一个拥有交换标记的进程运行，以及 PFRA 达到页框回收的最难优先级 (0 级)。

`grab_swap_token()` 函数决定是否将交换标记赋给当前进程。对每个主缺页 (major page fault) 调用该函数，这只有两种情形：

- 当 `filemap_nopage()` 函数发现请求页不在页高速缓存中时 (参见第十六章中“内存映射的请求调页”一节)。
- 当 `do_swap_page()` 函数从交换区读入一个新页时 (参见本章后面“换入页”一节)。

`grab_swap_token()` 函数在分配交换标记之前要进行一些检查，具体地说，就是要满足下列条件才可赋予交换标记：

- 上次调用 `grab_swap_token()` 后，至少已过了 2s。
- 在上一次调用 `grab_swap_token()` 后，当前拥有交换标记的进程没再提出主缺页，或该进程拥有交换标记的时间超出 `swap_token_default_timeout` 个节拍。
- 当前进程最近没有获得过交换标记。

交换标记的持有时间最好长一些，甚至以分钟为单位，因为其目标就是允许进程完成其执行。在 Linux 2.6.11 中，交换标记的持有时间默认值很小，即一个节拍。但是，通过编辑 `/proc/sys/vm/swap_token_default_timeout` 文件或发出相应的 `sysctl()` 系统调用，系统管理员可以修改 `swap_token_default_timeout` 变量的值。

当删除一个进程时，内核检查该进程是否拥有交换标记。如果是则放开它。这由 `mmapput()` 函数实现 (参见第九章的“内存描述符”一节)。

交换

交换 (swapping) 用来为非映射页在磁盘上提供备份。从前面的讨论我们知道有三类页必须由交换子系统处理：

- 属于进程匿名线性区（例如，用户态堆栈和堆）的页。
- 属于进程私有内存映射的脏页。
- 属于 IPC 共享内存区的页（参见第十九章的“IPC 共享内存”一节）。

就像请求调页，交换对于程序必须是透明的。换句话说，不需要在代码中嵌入与交换有关的特别指令。为了理解这是如何实现的，回想一下第二章的“常规分页”一节，我们知道每个页表项包含一个 Present 标志。内核利用这个标志来通知属于某个进程地址空间的页已被换出。在这个标志之外，Linux 还利用页表项中的其他位存放换出页标识符 (swapped-out page identifier)。该标识符用于编码换出页在磁盘上的位置。当缺页异常发生时，相应的异常处理程序可以检测到该页不在 RAM 中，然后调用函数从磁盘换入该缺页。

交换子系统的主要功能总结如下：

- 在磁盘上建立交换区 (swap area)，用于存放没有磁盘映像的页。
- 管理交换区空间。当需求发生时，分配与释放页槽 (page slot)。
- 提供函数用于从 RAM 中把页换出 (swap out) 到交换区或从交换区换入 (swap in) 到 RAM 中。
- 利用页表项 (现已被换出的换出页表项) 中的换出页标识符跟踪数据在交换区中的位置。

总之，交换是页框回收的一个最高级特性。如果我们要确保进程的所有页框都能被 PFRA 随意回收，而不仅仅是回收有磁盘映像的页，那么就必须使用交换。当然，你可以用 *swapon* 命令关闭交换，但此时随着磁盘系统负载增加，很快会发生磁盘系统瘫痪。

我们还需指出，交换可以用来扩展内存地址空间，使之被用户态进程有效地使用。事实上，一个大交换区可允许内核运行几个大需求量的应用，它们的内存总需求量超过系统中安装的物理内存量。但是，就性能而言，RAM 的仿真还是比不上 RAM 本身。进程对当前换出页的每一次访问，与对 RAM 中页的访问比起来，要慢几个数量级。简而言之，如果性能重要，那么交换仅仅作为最后一个方案；为了解决不断增长的计算需求增加 RAM 芯片的容量仍然是一个最好的方法。

交换区

从内存中换出的页存放在交换区 (swap area) 中，交换区的实现可以使用自己的磁盘分区，也可以使用包含在大型分区中的文件。可以定义几种不同的交换区，最大个数由 MAX_SWAPFILES 宏（通常被设置成 32）确定。

如果有多个交换区,就允许系统管理员把大的交换空间分布在几个磁盘上,以使硬件可以并发操作这些交换区;这样处理还允许在系统运行时不用重新启动系统就可以扩大交换空间的大小。

每个交换区都由一组页槽 (*page slot*) 组成,也就是说,由一组 4096 字节大小的块组成,每块中包含一个换出的页。交换区的第一个页槽用来永久存放有关交换区的信息,其格式由 *swap_header* 联合体(由两个结构 *info* 和 *magic* 组成)来描述。*magic* 结构提供了一个字符串,用来把磁盘某部分明确地标记成交换区,它只含有一个字段 *magic.magic*,这个字段含有一个 10 字符的“magic”字符串。*magic* 结构从根本上允许内核明确地把一个文件或分区标记成交换区,这个字符串的内容就是“SWAPSPACE2”。该字段通常位于第一个页槽的末尾。

info 结构包括以下字段:

bootbits

交换算法不使用该字段。该字段对应于交换区的第一个 1024 字节,可以存放分区数据、磁盘标签等等。

version

交换算法的版本。

last_page

可有效使用的最后一个页槽。

nr_badpages

有缺陷的页槽的个数。

padding[125]

填充字节。

badpages[1]

一共 637 个数字,用来指定有缺陷页槽的位置。

创建与激活交换区

只要系统是打开的,存放在交换区中的数据就是有意义的。当系统被关闭时,所有的进程都被杀死,因此,进程存放在交换区中的数据也被丢弃。基于这个原因,交换区包含很少的控制信息,实际上包含交换区类型和有缺陷页槽的链表。这种控制信息很容易存放在一个单独的 4KB 页中。

通常,系统管理员在创建 Linux 系统中的其他分区时都创建一个交换分区,然后使用 *mkswap* 命令把这个磁盘区设置成一个新的交换区。该命令对刚才介绍的第一个页槽中

的字段进行初始化。由于磁盘中可能会有些坏块，这个程序还可以对其他所有的页槽进行检查从而确定有缺陷页槽的位置。但是执行 `mkswap` 命令会把交换区设置成非激活的状态。每个交换区都可以在系统启动时在脚本文件中被激活，也可以在系统运行之后动态激活。

每个交换区由一个或多个交换子区 (*swap extent*) 组成，每个交换子区由一个 `swap_extent` 描述符表示，每个子区对应一组页（更准确地说，是一组页槽），它们在磁盘上是物理相邻的。`swap_extent` 描述符由下面这几部分组成：交换区的子区首页索引、子区的页数和子区的起始磁盘扇区号。当激活交换区自身的同时，组成交换区的有序子区链表也被创建。存放在磁盘分区中的交换区只有一个子区；但是，存放在普通文件中的交换区则可能有多个子区，这是因为文件系统有可能没把该文件全部分配在磁盘的一组连续块中。

如何在交换区中分布页

当换出时，内核尽力把换出的页存放在相邻的页槽中，从而减少在访问交换区时磁盘的寻道时间，这是高效交换算法的一个重要因素。

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

交换区描述符

每个活动的交换区在内存中都有自己的 `swap_info_struct` 描述符，其字段如表17-3所示。

表17-3：交换区描述符的字段

类型	字段	说明
unsigned int	flags	交换区标志
spinlock_t	sdev_lock	保护交换区的自旋锁
struct file *	swap_file	指针，指向存放交换区的普通文件或设备文件的文件对象
struct block_device *	bdev	存放交换区的块设备描述符
struct list head	extent_list	组成交换区的子区链表的头部

表 17-3: 交换区描述符的字段 (续)

类型	字段	说明
int	nr_extents	组成交换区的子区数量
struct swap_extent *	curr_swap_extent	指向最近使用的子区描述符的指针
unsigned int	old_block_size	存放交换区的磁盘分区自然块大小
unsigned short *	swap_map	指向计数器数组的指针, 交换区的每个页槽对应一个数组元素
unsigned int	lowest_bit	在搜索一个空闲页槽时要扫描的第一个页槽
unsigned int	highest_bit	在搜索一个空闲页槽时要扫描的最后一个页槽
unsigned int	cluster_next	在搜索一个空闲页槽时要扫描的下一个页槽
unsigned int	cluster_nr	在从头重新开始扫描之前空闲页槽的分配次数
int	prio	交换区优先级
int	pages	可用页槽的个数
unsigned long	max	交换区的大小, 以页为单位
unsigned long	inuse_pages	交换区内已用页槽数
int	next	指向下一个交换区描述符的指针

flags 字段包括三个重叠的子字段:

SWP_USED

如果交换区是活动的, 该值就是 1; 如果交换区不是活动的, 该值就是 0。

SWP_WRITEOK

如果可以写入交换区, 该值就是 1; 如果交换区只读, 该值就是 0 (可以是活动的或不是活动的)。

SWP_ACTIVE

这个两位的字段实际上是 SWP_USED 和 SWP_WRITEOK 的组合。如果前面两个标志置位, 那么 SWP_ACTIVE 标志置位。

swap_map 字段指向一个计数器数组, 交换区的每个页槽对应一个元素。如果计数器值等于 0, 那么这个页槽就是空闲的; 如果计数器为正数, 那么换出页就填充了这个页槽。实际上, 页槽计数器的值就表示共享换出页的进程数。如果计数器的值为 SWAP_MAP_MAX (等于 32767), 那么存放在这个页槽中的页就是“永久”的, 并且不能从相应的页槽中

删除。如果计数器的值是 `SWAP_MAP_BAD` (等于 32768), 那么就认为这个页槽是有缺陷的, 也就是不可用的 (注 7)。

`prio` 字段是一个有符号的整数, 表示交换子系统依据这个值考虑每个交换区的次序。

`sdev_lock` 字段是一个自旋锁, 它防止 SMP 系统上对交换区数据结构 (主要是交换描述符) 的并发访问。

`swap_info` 数组包括 `MAX_SWAPFILES` 个交换区描述符。只有那些设置了 `SWP_USED` 标志的交换区才被使用, 因为它们是活动区域。图 17-6 说明了 `swap_info` 数组、一个交换区和相应的计数器数组的情况。

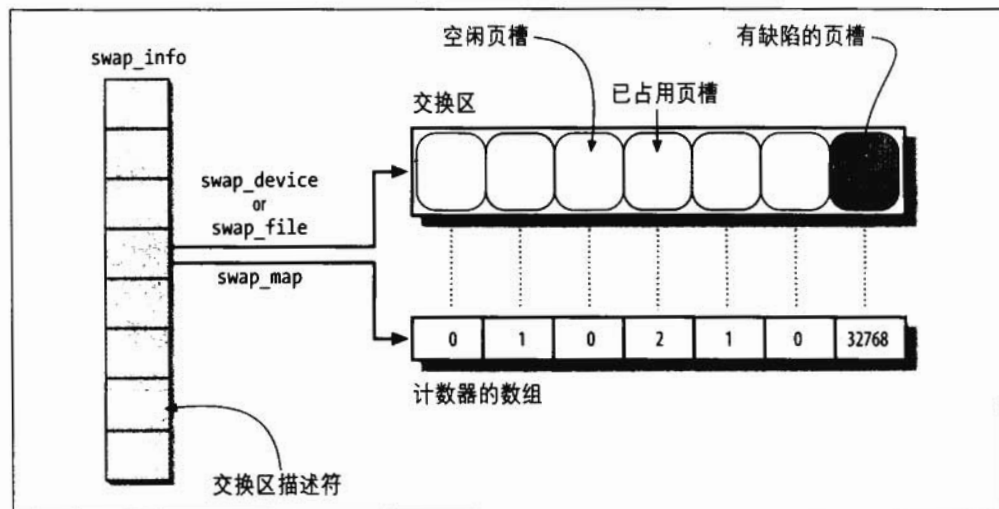


图 17-6: 交换区数据结构

`nr_swapfiles` 变量存放数组中包含或已包含所使用交换区描述符的最后一个元素的索引。这个变量有些名不符实, 它并没有包含活动交换区的个数。

活动交换区描述符也被插入按交换区优先级排序的链表中。该链表是通过交换区描述符的 `next` 字段实现的, `next` 字段存放的是 `swap_info` 数组中下一个描述符的索引。该字段作为索引的这种用法与我们已经见过的很多名为 `next` 字段的用法有所不同, 后者通常都是指针。

注 7: “永久”页槽防止 `swap_map` 计数器溢出。没有这些“永久”页槽, 如果有效的页槽被多次引用, 它们就会变得“有缺陷”, 从而导致数据丢失。但是, 谁也不真正期望一个页槽计数器达到 32768。这仅仅是一条权宜之计。

前31个最高位不全等于0，最后一位等于0

该页现在被换出。

最低位等于1

该页包含在RAM中。

注意，交换区的最大值由表示页槽的可用位数决定。在80x86体系结构上，有24位可用，这就限制了交换区的大小为 2^{24} 个页槽（也就是64GB）。

由于一个页可以属于几个进程的地址空间（参见前面的“反向映射”一节），所以它可能从一个进程的地址空间中被换出，但是仍旧保留在主存中；因此可能把同一个页换出多次。当然，一个页在物理上只被换出并存储一次，但是后来每次试图换出该页都会增加swap_map计数器的值。

在试图换出一个已经换出的页时就会调用swap_duplicate()函数。该函数只是验证以参数传递的换出页标识符是否有效，并增加相应的swap_map计数器的值。更确切地说，该函数执行以下操作：

1. 使用swp_type和swp_offset宏从参数中提取出交换区号type和页槽索引offset。
2. 检查交换区是否被激活；如果不是，则返回0（无效的标识符）。
3. 检查页槽是否有效且是否不为空闲（swap_map计数器大于0且小于SWAP_MAP_BAD）；如果不是，则返回0（无效的标识符）。
4. 否则，换出页的标识符确定出一个有效页的位置。如果页槽的swap_map计数器还没有达到SWAP_MAP_MAX，则增加它的值。
5. 返回1（有效的标识符）。

激活和禁用交换区

一旦交换区被初始化，超级用户（或者更确切地说是任何具有CAP_SYS_ADMIN权能的用户，有关内容将在第二十章中的“进程的信任状和权能”一节中介绍）就可以分别使用swapon和swapoff程序激活和禁用交换区。这两个程序分别使用了swapon()和swapoff()系统调用，我们将简要介绍相应的服务例程。

sys_swapon()服务例程

sys_swapon()服务例程接收如下参数：

specialfile

这个参数指向设备文件（或分区）的路径名（在用户态地址空间），或指向实现交换区的普通文件的路径名。

swap_flags

这个参数由一个单独的 SWAP_FLAG_PREFER 位加上交换区优先级的 31 位组成（只有在 SWAP_FLAG_PREFER 位置位时，优先级位才有意义）。

sys_swapon() 函数对创建交换区时放入第一个页槽中的 swap_header 联合体字段进行检查。其执行的主要步骤有：

1. 检查当前进程是否具有 CAP_SYS_ADMIN 权能。
2. 在交换区描述符 swap_info 数组的前 nr_swapfiles 个元素中查找 SWP_USED 标志为 0（即对应的交换区不是活动的）的第一个描述符。如果找到一个不活动交换区，则跳到第 4 步。
3. 新交换区数组索引等于 nr_swapfiles；它检查保留给交换区索引的位数是否足够用于编码新索引。如果不够，则返回错误代码；如果足够，就将 nr_swapfiles 的值加 1。
4. 找到未用交换区索引：它初始化这个描述符的字段，即把 flags 置为 SWP_USED，把 lowest_bit 和 highest_bit 置为 0。
5. 如果 swap_flags 参数为新交换区指定了优先级，则设置描述符的 prio 字段。否则，就把所有活动交换区中最低的优先级减 1 后赋给这个字段（这样就假设最后一个被激活的交换区在最慢的块设备上）。如果没有其他交换区是活动的，就把该字段设置成 -1。
6. 从用户态地址空间复制由 specialfile 参数所指向的字符串。
7. 调用 filp_open() 打开由 specialfile 参数指定的文件（参见第十二章的“open() 系统调用”一节）。
8. 把 filp_open() 返回的文件对象地址存放在交换区描述符的 swap_file 字段。
9. 检查 swap_info 中其他的活动交换区，以确认该交换区还未被激活。具体就是，检查交换区描述符的 swap_file->f_mapping 字段中存放的 address_space 对象地址。如果交换区已被激活，则返回错误码。
10. 如果 specialfile 参数标识一个块设备文件，则执行下列子步骤：
 - a. 调用 bd_claim() 把交换子系统设置成块设备的占有者（参见第十四章的“块设备”一节）。如果块设备已有一个占有者，则返回错误码。

- b. 把 `block_device` 描述符地址存入交换区描述符的 `bdev` 字段。
 - c. 把设备的当前块大小存放在交换区描述符的 `old_block_size` 字段, 然后把设备的块大小设成 4096 字节 (即页的大小)。
11. 如果 `specialfile` 参数标识一个普通文件, 则执行下列子步骤:
- a. 检查文件索引节点 `i_flags` 字段中的 `S_SWAPFILE` 字段。如果该标志置位, 说明文件已被用作交换区, 返回错误码。
 - b. 把该文件所在块设备的描述符地址存入交换区描述符的 `bdev` 字段。
12. 读入存放在交换区页槽 0 中的 `swap_header` 描述符。为达到这个目的, 它调用 `read_cache_page()`, 并传入参数: 由 `swap_file->f_mapping` 指向的 `address_space` 对象、页索引 0、文件 `readpage` 方法的地址 (存放在 `swap_file->f_mapping->a_ops->readpage`) 和指向文件对象 `swap_file` 的指针。然后等待直到页被读入内存。
13. 检查交换区中第一页的最后 10 个字符中的魔术字符串是否等于 “SWAPSPACE2”。如果不是, 就返回一个错误码。
14. 根据存放在 `swap_header` 联合体的 `info.last_page` 字段中的交换区的大小, 初始化交换区描述符的 `lowest_bit` 和 `highest_bit` 字段。
15. 调用 `vmalloc()` 来创建与新交换区相关的计数器数组, 并把它地址存放在交换描述符的 `swap_map` 字段中。还要根据 `swap_header` 联合体的 `info.bad_pages` 字段中存放的有缺陷的页槽链表把这个数组的元素初始化成 0 或 `SWAP_MAP_BAD`。
16. 通过访问第一个页槽中的 `info.last_page` 和 `info.nr_badpages` 字段计算可用页槽的个数, 并把它存入交换区描述符的 `pages` 字段。而且把交换区中的总页数赋给 `max` 字段。
17. 为新交换区建立子区链表 `extent_list` (如果交换区建立在磁盘分区上, 则只有一个子区), 并相应地设定交换区描述符的 `nr_extents` 和 `curr_swap_extent` 字段。
18. 把交换区描述符的 `flags` 字段设为 `SWP_ACTIVE`。
19. 更新 `nr_good_pages`、`nr_swap_pages` 和 `total_swap_pages` 三个全局变量。
20. 把新交换区描述符插入 `swap_list` 变量所指向的链表中。
21. 返回 0 (成功)。

sys_swapoff() 服务例程

`sys_swapoff()` 服务例程使 `specialfile` 参数所指定的交换区无效。`sys_swapoff()` 比 `sys_swapon()` 复杂得多, 也更加耗时, 因为使之无效的这个分区现在可能仍然还包含

几个进程的页。因此，强制该函数扫描交换区并把所有现有的页都换入。由于每个换入操作都需要一个新的页框，因此如果现在没有空闲页框，这个操作就可能失败。在这种情况下，该函数就返回一个错误码。所有这些操作都是通过执行以下主要步骤实现的：

1. 验证当前进程是否具有 `CAP_SYS_ADMIN` 权能。
2. 拷贝内核空间中 `specialfile` 所指向的字符串。
3. 调用 `filp_open()`，打开 `specialfile` 参数确定的文件。与往常一样，该函数返回文件对象的地址。
4. 扫描交换区描述符链表 `swap_list`，比较由 `filp_open()` 返回的文件对象地址与活动交换区描述符的 `swap_file` 字段中的地址，如果不一致，说明传给函数的是一个无效参数，则返回一个错误码。
5. 调用 `cap_vm_enough_memory()`，检查是否有足够的空闲页框把交换区上存放的所有页换入。如果不够，交换区就不能禁用，然后释放文件对象，返回错误码。这只是个粗略的检查，但可使内核免于许多无用的磁盘操作。当执行这项检查时，`cap_vm_enough_memory()` 要考虑由 `slab` 高速缓存分配且 `SLAB_RECLAIM_ACCOUNT` 标志置位的页框（参见第八章中“`slab` 分配器与分区页框分配器的接口”一节），这样的页（被认为是可回收的这些页）的数量存放在 `slab_reclaim_pages` 变量中。
6. 从 `swap_list` 链表中删除该交换区描述符。
7. 从 `nr_swap_pages` 和 `total_swap_pages` 的值中减去存放在交换区描述符的 `pages` 字段中的值。
8. 把交换区描述符 `flags` 字段中的 `SWP_WRITEOK` 标志清 0。这可禁止 PFRA 向交换区换出更多的页。
9. 调用 `try_to_unuse()` 函数（见下面）强制把这个交换区中剩余的所有页都移到 RAM 中，并相应地修改使用这些页的进程的页表。当执行该函数时，当前进程（即运行 `swaponoff` 的进程）的 `PF_SWAPOFF` 标志置位。该标志置位只有一个结果：如页框严重不足，`select_bad_process()` 函数就会被强制选择并删除该进程（参见本章前面“内存不足删除程序”一节）。
10. 一直等到交换区所在的块设备驱动器被卸载（参见第十四章“激活块设备驱动程序”一节）。这样在交换区被禁用之前，`try_to_unuse()` 发出的读请求会被驱动器处理。
11. 如果在分配所有请求的页框时 `try_to_unuse()` 函数失败，那么就不能禁用这个交换区。因此，`sys_swapoff()` 执行下列子步骤：
 - a. 把这个交换区描述符重新插入 `swap_list` 链表，并把它的 `flags` 字段置为 `SWP_WRITEOK`。

- b. 把交换区描述符中 `pages` 字段的值加到 `nr_swap_pages` 和 `total_swap_pages` 变量以恢复其原值。
 - c. 调用 `filp_close()` 关闭在第 3 步中打开的文件（参见第十二章“`close()` 系统调用”一节），并返回错误码。
12. 否则，所有已用的页槽都已经被成功传送到 RAM 中。因此，执行下列子步骤：
- a. 释放存有 `swap_map` 数组和子区描述符的内存区域。
 - b. 如果交换区存放在磁盘分区，则把块大小恢复到原值，该原值存放在交换区描述符的 `old_block_size` 字段。而且，调用 `bd_release()` 函数，使交换子系统不再占有该块设备（参见 `sys_swapon()` 函数第 10a 步的描述）。
 - c. 如果交换区存放在普通文件中，则把文件索引节点的 `S_SWAPFILE` 标志清 0。
 - d. 调用 `filp_close()` 两次，第一次针对 `swap_file` 文件对象，第二次针对第 3 步中 `filp_open()` 返回的对象。
 - e. 返回 0（成功）。

try_to_unuse() 函数

`try_to_unuse()` 函数使用一个索引参数，该参数标识待清空的交换区。该函数换入页并更新已换出页的进程的所有页表。因此，该函数从 `init_mm` 内存描述符（用作标记）开始，访问所有内核线程和进程的地址空间。这是一个相当耗时的函数，通常以开中断运行。因此，与其他进程的同步也是关键的。

`try_to_unuse()` 函数扫描交换区的 `swap_map` 数组。当它找到一个“在用”页槽时，首先换入其中的页，然后开始查找引用该页的进程。这两个操作的顺序对避免竞争条件是至关重要的。当 I/O 数据传送正在进行时，页被加锁，因此没有进程可以访问它。一旦 I/O 数据传送完成，页又被 `try_to_unuse()` 加锁，以使它不会被另一个内核控制路径再次换出。因为每个进程在开始进行换入或换出操作之前查找页高速缓存，所以这也可以避免竞争条件（参见后面“交换高速缓存”一节）。最后，由 `try_to_unuse()` 所考虑的交换区被标记为不可写（`SWP_WRITEOK` 标志被清 0），因此，没有进程可以对这个交换区的页槽执行换出。

但是，可能强迫 `try_to_unuse()` 对交换区引用计数器的 `swap_map` 数组扫描几次。这是因为对换出页引用的线性区可能在一次扫描中消失，而在随后又出现在进程链表中。

例如，回想 `do_munmap()` 函数的描述（在第九章“释放线性地址区间”一节）：只要进程释放一个线性地址区间，`do_munmap()` 就从进程链表中删除所有受影响线性地址所在的线性区；随后，该函数把只是部分解除映射的那部分线性区重新插入进程链表中。

`do_munmap()`还要负责释放属于已释放线性地址区间的换出页；但是，如果换出的页属于重新插入进程链表的线性区，则最好不要释放它们。

因此，`try_to_unuse()`对引用给定页槽的进程进行查找时可能失败，因为相应的线性区暂时没有包含在进程链表中。为了处理这种情况，`try_to_unuse()`一直对`swap_map`数组进行扫描，直到所有的引用计数器都变为空。引用了换出页的“神出鬼没”的线性区最终会重新出现在进程链表中，因此，`try_to_unuse()`终将会成功释放所有页槽。

让我们现在来描述`try_to_unuse()`所执行的主要操作。传递给它的参数为交换区`swap_map`数组的引用计数器，该函数在这个引用计数器上执行连续循环。如果当前进程接收到一个信号，则循环会中断，函数返回错误码。对于数组中的每个引用计数器，`try_to_unuse()`执行下列步骤：

1. 如果计数器等于0（没有页存放在这里）或者等于`SWAP_MAP_BAD`，则对下一个页槽继续处理。
2. 否则，调用`read_swap_cache_async()`函数（参见本章后面“换入页”一节）换入该页。这包括分配一个新页框（如果必要），用存放在页槽中的数据填充新页框并把这个页存放在交换高速缓存。
3. 等待，直到用磁盘中的数据适当地更新了这个新页，然后锁住它。
4. 当正在执行前一步时，进程有可能被挂起。因此，还要检查这个页槽的引用计数器是否变为空，如果是，说明这个交换页可能被另一个内核控制路径释放，然后继续处理下一个页槽。
5. 对于以`init_mm`为头部的双向链表（参见第九章“内存描述符”一节）中的每个内存描述符，调用`unuse_process()`。这个耗时的函数扫描拥有内存描述符的进程的所有页表项，并用这个新页框的物理地址替换页表中每个出现的换出页标识符。为了反映这种移动，还要把`swap_map`数组中的页槽计数器减1（除非计数器等于`SWAP_MAP_MAX`），并增加这个页框的引用计数器。
6. 调用`shmem_unuse()`检查换出的页是否用于IPC共享内存资源，并适当地处理那种情况（参见第十九章“IPC共享内存”一节）。
7. 检查页的引用计数器。如果它的值等于`SWAP_MAP_MAX`，则页槽是“永久的”。为了释放它，则把引用计数器强制置为1。
8. 交换高速缓存可能也拥有该页（它对引用计数器的值起作用）。如果页属于交换高速缓存，就调用`swap_writepage()`函数把页的内容刷新到磁盘（如果页为脏），调用`delete_from_swap_cache()`从交换高速缓存删去页，并把页的引用计数减1。

9. 设置页描述符的 `PG_dirty` 标志, 并打开页框的锁, 递减它的引用计数器 (取消第 5 步的增量)。
10. 检查当前进程的 `need_resched` 字段; 如果它被设置, 则调用 `schedule()` 放弃 CPU。禁用交换区是一件冗长的工作, 内核必须保证系统中的其他进程仍然继续执行。只要这个进程再次被调度程序选中, `try_to_unuse()` 函数就从这一步继续执行。
11. 继续到下一个页槽, 从第 1 步开始。

`try_to_unuse()` 继续执行, 直到 `swap_map` 数组中的每个引用计数器都为空。回想一下, 即使这个函数已经开始检查下一个页槽, 但是前一个页槽的引用计数器有可能仍然为正。事实上, 一个“神出鬼没”的进程可能还在引用这个页, 典型的原因是某些线性区已经被临时从第 5 步所扫描的进程链表中删除。`try_to_unuse()` 最终会捕获到每个引用。但是, 在此期间, 页不再位于交换高速缓存, 它的锁被打开, 并且页的一个拷贝仍然包含在要禁用的交换区的页槽中。

一般会认为这种情形可能导致数据丢失。例如, 假定某个“神出鬼没”的进程访问页槽, 并开始换入其中的页。因为页不再位于交换高速缓存, 因此, 进程用从磁盘读取的数据填充一个新的页框。但是, 这个页框可能不同于与“神出鬼没”进程共享页的那些进程曾经拥有的页框。

当禁用交换区时这个问题不会发生, 因为只有在换出的页属于私有匿名内存映射时 (注 8), 对“神出鬼没”进程的干涉才会发生。在这种情况下, 使用第九章描述的“写时复制”机制来处理页框, 所以, 把不同的页框分配给引用了同一页的进程是完全合法的。但是, `try_to_unuse()` 函数将页标记为“脏” (第 9 步)。否则, `shrink_list()` 函数可能随后从某个进程的页表中删除这一页, 而并不把它保存在另一个交换区中 (参见后面“换出页”一节)。

分配和释放页槽

正如我们将在后面看到的那样, 在释放内存时, 内核要在很短的时间内把很多页都交换出去。因此尽力把这些页存放在相邻的页槽中非常重要, 这样就减少了在访问交换区时磁盘的寻道时间。

搜索空闲页槽的第一种方法可以选择下列两种既简单而又有些极端的策略之一:

- 总是从交换区的开头开始。这种方法在换出操作过程中可能会增加平均寻道时间, 因为空闲页槽可能已经被弄得凌乱不堪。

注 8: 事实上, 页可能属于 IPC 共享内存区, 第十九章将讨论这种情况。

- 总是从最后一个已分配的页槽开始。如果交换区的大部分空间都是空闲的（这是最通常的情况），那么这种方法在换入操作过程中会增加平均寻道时间，因为所占用的为数不多的页槽可能是零散存放的。

Linux 采用了一种混合的方法。除非发生以下这些条件之一，否则 Linux 总是从最后一个已分配的页槽开始查找。

- 已经到达交换区的末尾。
- 上次从交换区的开头重新分配之后，已经分配了 `SWAPFILE_CLUSTER`（通常是 256）个空闲页槽。

`swap_info_struct` 描述符的 `cluster_nr` 字段存放已分配的空闲页槽数。当函数从交换区的开头重新分配时该字段被重置为 0。`cluster_next` 字段存放在下一次分配时要检查的第一个页槽的索引（注 9）。

为了加速对空闲页槽的搜索，内核要保证每个交换区描述符的 `lowest_bit` 和 `highest_bit` 字段是最新的。这两个字段定义了第一个和最后一个可能为空的页槽，换言之，所有低于 `lowest_bit` 和高于 `highest_bit` 的页槽都被认为已经分配过。

scan_swap_map() 函数

`scan_swap_map()` 函数用来在给定的交换区中查找一个空闲页槽。该函数只作用于一个参数，该参数指向交换区描述符并返回一个空闲页槽的索引。如果交换区不含有任何空闲页槽，就返回 0。该函数执行以下步骤：

1. 首先试图使用当前的簇。如果交换区描述符的 `cluster_nr` 字段是正数，就从 `cluster_next` 索引处的元素开始对计数器的 `swap_map` 数组进行扫描，查找一个空项。如果找到一个空项，就减少 `cluster_nr` 字段的值并转到第 4 步。
2. 如果执行到这儿，那么，或者 `cluster_nr` 字段为空，或者从 `cluster_next` 开始搜索后没有在 `swap_map` 数组中找到空项。现在就应该开始第二阶段的混合查找。把 `cluster_nr` 重新初始化成 `SWAPFILE_CLUSTER`，并从 `lowest_bit` 索引处开始重新扫描这个数组，以便试图找到有 `SWAPFILE_CLUSTER` 个空闲页槽的一个组。如果找到这样的一个组，就转到第 4 步。
3. 不存在 `SWAPFILE_CLUSTER` 个空闲页槽的组。从 `lowest_bit` 索引处开始重新开始扫描这个数组，以便试图找到一个单独的空闲页槽。如果没有找到空项，就把

注 9：正如你可能已经注意到的一样，Linux 数据结构的名字并不都是恰当的。在这种情况下，内核并不会真正“聚簇（cluster）”交换区的页槽。

lowest_bit 字段置为数组的最大索引, highest_bit 字段置为 0, 并返回 0 (交换区已满)。

4. 已经找到空项。把 1 放在空项中, 减少 nr_swap_pages 的值, 如果需要就修改 lowest_bit 和 highest_bit 字段, 把 inuse_page 字段的值加 1, 并把 cluster_next 字段设置成刚才分配的页槽的索引加 1。
5. 返回刚才分配的页槽的索引。

get_swap_page() 函数

get_swap_page() 函数通过搜索所有活动的交换区来查找一个空闲页槽。它返回一个新近分配页槽的换出页标识符, 如果所有的交换区都填满, 就返回 0, 该函数要考虑活动交换区的不同优先级。

该函数需要经过两遍扫描, 以便在容易发现页槽时节约运行时间。第一遍是部分的, 只适用于只有相同优先级的交换区。该函数以轮询的方式在这种交换区中查找一个空闲页槽。如果没有找到空闲页槽, 就从交换区链表的起始位置开始进行第二遍扫描。在第二遍扫描中, 要对所有的交换区都进行检查。更确切地说, 该函数执行以下步骤:

1. 如果 nr_swap_pages 为空或者如果没有活动的交换区, 就返回 0。
2. 首先考虑 swap_list.next 所指向的交换区 (回想一下, 交换区链表是按照优先级从高到低的顺序排列的)。
3. 如果交换区是活动的, 就调用 scan_swap_map() 来获得一个空闲页槽。如果 scan_swap_map() 函数返回一个页槽索引, 该函数的任务基本上就完成了, 但是它还要准备下一次被调用。因此, 如果下一个交换区的优先级和这个交换区的优先级相同 (即轮询使用这些交换区), 该函数就把 swap_list.next 修改成指向交换区链表中的下一个交换区。如果下一个交换区的优先级和当前交换区的优先级不同, 该函数就把 swap_list.next 设置成交换区链表中的第一个交换区 (这样下一次搜索操作就会从优先级最高的交换区开始)。该函数最终返回刚才分配的页槽所对应的换出页标识符。
4. 或者交换区是不可写的, 或者交换区中没有空闲页槽。如果交换区链表中的下一个交换区的优先级和当前交换区的优先级相同, 就把下一个交换区设置成当前交换区并跳转到第 3 步。
5. 此时, 交换区链表中的下一个交换区的优先级小于前一个交换区的优先级。下一步操作取决于该函数正在进行哪一遍扫描。
 - a. 如果这是第一遍 (局部) 扫描, 就考虑链表中的第一个交换区并跳转到第 3 步, 这样就开始第二遍扫描。

- b. 否则, 就检查交换区链表中是否有下一个元素。如果有, 就考虑这个元素并跳转到第 3 步。
6. 此时, 第二遍对链表的扫描已经完成, 并没有发现空闲页槽, 返回 0。

swap_free()函数

当换入页时, 调用 `swap_free()` 函数以对相应的 `swap_map` 计数器进行减 1 操作 (参见表 17-3)。当相应的计数器达到 0 时, 由于页槽的标识符不再包含在任何页表项中, 因此页槽就变成空闲。但是, 我们将在后面“交换高速缓存”一节看到, 交换高速缓存也记入页槽拥有者的个数。

该函数只作用于一个参数 `entry`, `entry` 表示换出页标识符。函数执行以下步骤:

1. 从 `entry` 参数导出交换区索引和页槽索引 `offset`, 并获得交换区描述符的地址。
2. 检查交换区是否是活动的。如果不是, 就立即返回。
3. 如果正在释放的页槽对应的 `swap_map` 计数器小于 `SWAP_MAP_MAX`, 就减少这个计数器的值。回想一下, 值为 `SWAP_MAP_MAX` 的项都被认为是永久的 (不可删除的)。
4. 如果 `swap_map` 计数器变成 0, 就增加 `nr_swap_pages` 的值, 减少 `inuse_pages` 字段的值, 如果需要就修改这个交换区描述符的 `lowest_bit` 和 `highest_bit` 字段。

交换高速缓存

向交换区来回传送页面会引发很多竞争条件, 具体地说, 交换子系统必须仔细处理下面的情形:

多重换入

两个进程可能同时要换入同一个共享匿名页。

同时换入换出

一个进程可能换入正由 PFRA 换出的页。

交换高速缓存 (*swap cache*) 的引入就是为了解决这类同步问题。关键的原则是, 没有检查交换高速缓存是否已包括了所涉及的页, 就不能进行换入或换出操作。有了交换高速缓存, 涉及同一页的并发交换操作总是作用于同一个页框的。因此, 内核可以安全地依赖页描述符的 `PG_locked` 标志, 以避免任何竞争条件。

考虑一下共享同一换出页的两个进程这种情形。当第一个进程试图访问页时, 内核开始换入页操作, 第一步就是检查页框是否在交换高速缓存中, 我们假定页框不在交换高速缓存中, 那么内核就分配一个新页框并把它插入交换高速缓存, 然后开始 I/O 操作, 从

交换区读入页的数据；同时，第二个进程访问该共享匿名页，与上面相同，内核开始换入操作，检查涉及的页框是否在交换高速缓存中。现在页框是在交换高速缓存，因此内核只是访问页框描述符，在 `PG_locked` 标志清 0 之前（即 I/O 数据传输完毕之前），让当前进程睡眠。

当换入换出操作同时出现时，交换高速缓存起着至关重要的作用。在本章前面“内存紧缺回收”一节我们描述过，`shrink_list()` 函数要开始换出一个匿名页，就必须当 `try_to_unmap()` 从进程（所有拥有该页的进程）的用户态页表中成功删除了该页后才可以。但是当换出的写操作还在执行的时候，这些进程中可能有某个进程要访问该页，而产生换入操作。

在写入磁盘前，待换出页由 `shrink_list()` 存放在交换高速缓存。考虑页 P 由两个进程（A 和 B）共享。最初，两个进程的页表项都引用该页框，该页有两个拥有者，如图 17-8 (a) 所示。当 PFRA 选择回收页时，`shrink_list()` 把页框插入交换高速缓存。如图 17-8 (b) 所示，现在页框有三个拥有者，而交换区中的页槽只被交换高速缓存引用。然后 PFRA 调用 `try_to_unmap()` 从这两个进程的页表项中删除对该页框的引用。一旦这个函数结束，该页框就只有交换高速缓存引用它，而引用页槽的有这两个进程和交换高速缓存，如图 17-8 (c) 所示。假定：当页中的数据写入磁盘时，进程 B 访问该页，即它要用该页内部的线性地址访问内存单元。那么，缺页异常处理程序发现页框在交换高速缓存，并把物理地址放回进程 B 的页表项，如图 17-8 (d) 所示。相反地，如果换出操作结束，而没有并发换入操作，`shrink_list()` 函数则从交换高速缓存删除该页框并把它释放到伙伴系统，如图 17-8 (e) 所示。

你可以认为交换高速缓存是一个临时区域，该区域存有正在被换入或换出的匿名页描述符。当换入或换出结束时（对于共享匿名页，换入换出操作必须对共享该页的所有进程进行），匿名页描述符就可以从交换高速缓存删除（注 10）。

交换高速缓存的实现

交换高速缓存由页高速缓存数据结构和过程实现（在第十五章的“页高速缓存”一节中有描述）。回想一下，页高速缓存的核心就是一组基树，借助基树，算法就可以从 `address_space` 对象地址（即该页的拥有者）和偏移量值推算出页描述符的地址。

在交换高速缓存中页的存放方式是隔页存放，并有下列特征：

注 10： 有些情况下，交换高速缓存还能够提高系统性能：如服务器后台程序通过创建子进程提供服务请求时。系统负载很高时，页可能从父进程换出，而且再也不会成为父进程的页。如果不使用交换高速缓存，每个被创建的子进程就需要从交换区选择调入的页。

- 页描述符的 mapping 字段为 NULL。
- 页描述符的 PG_swapcache 标志置位。
- private 字段存放与该页有关的换出页标识符。

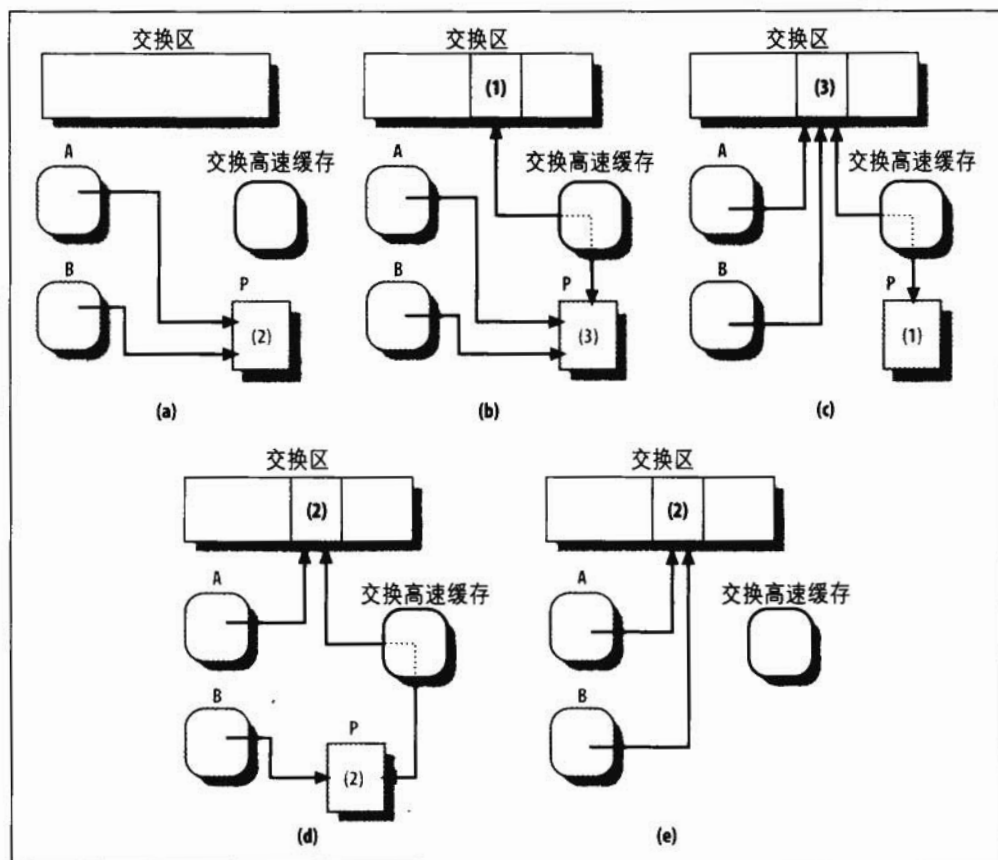


图 17-8: 交换高速缓存的作用

此外, 当页被放入交换高速缓存时, 则页描述符的 count 字段和页槽引用计数器的值都增加, 因为交换高速缓存既要使用页框, 也要使用页槽。

最后, 交换高速缓存中的所有页只使用一个 swapper_space 地址空间, 因此只有一个基树 (由 swapper_space.page_tree 指向) 对交换高速缓存中的页进行寻址。swapper_space 地址空间的 nrpages 字段存放交换高速缓存中的页数。

交换高速缓存的辅助函数

内核使用几个函数来处理交换高速缓存，它们主要是基于第十五章的“页高速缓存”一节中所讨论的那些函数。稍后我们将说明这些相对低层的函数是如何被高层函数调用来按需换入和换出页的。

处理交换高速缓存的函数主要有：

`lookup_swap_cache()`

通过传递来的参数（换出页标识符）在交换高速缓存中查找页并返回页描述符的地址。如果该页不在交换高速缓存中，就返回0。该函数调用`radix_tree_lookup()`函数，把指向`swapper_space.page_tree`的指针（用于交换高速缓存中页的基树）和换出页标识符作为参数传递，以查找所需要的页。

`add_to_swap_cache()`

把页插入交换高速缓存中。它本质上调用`swap_duplicate()`检查作为参数传递来的页槽是否有效，并增加页槽引用计数器；然后调用`radix_tree_insert()`把页插入高速缓存；最后递增页引用计数器并将`PG_swapcache`和`PG_locked`标志置位。

`__add_to_swap_cache()`

与`add_to_swap_cache()`类似，但是，在把页框插入交换高速缓存前，这个函数不调用`swap_duplicate()`。

`delete_from_swap_cache()`

调用`radix_tree_delete()`从交换高速缓存中删除页，递减`swap_map`中相应的使用计数器，递减页引用计数器。

`free_page_and_swap_cache()`

如果除了当前进程外，没有其它用户态进程正在引用相应的页槽，则从交换高速缓存中删除该页，并递减页使用计数器。

`free_pages_and_swap_cache()`

与`free_page_and_swap_cache()`相似，但它是对一组页操作。

`free_swap_and_cache()`

释放一个交换表项，并检查该表项引用的页是否在交换高速缓存。如果没有用户态进程（除了当前进程之外）引用该页，或者超过50%的交换表项在用，则从交换高速缓存中释放该页。

换出页

我们从本章前面“内存紧缺回收”一节可看到，PFRA是如何确定一个给定的匿名页是否该被换出。在这一节，我们描述内核如何执行换出操作。

向交换高速缓存插入页框

换出操作的第一步就是准备交换高速缓存。如果 `shrink_list()` 函数确认某页为匿名页 (`PageAnon()` 函数返回1) 而且交换高速缓存中没有相应的页框 (页描述符的 `PG_swapcache` 标志清0), 内核就调用 `add_to_swap()` 函数。

`add_to_swap()` 函数在交换区中分配一个新页槽, 并把一个页框 (其页描述符地址作为参数传递) 插入交换高速缓存。函数执行如下主要步骤:

1. 调用 `get_swap_page()` 函数分配一个新页槽 (参见本章前面“分配和释放页槽”一节)。如果失败 (例如没有发现空闲页槽), 则返回0。
2. 调用 `__add_to_page_cache()`, 传给它页槽索引、页描述符地址和一些分配标志。
3. 将页描述符中的 `PG_uptodate` 和 `PG_dirty` 标志置位, 从而强制 `shrink_list()` 函数把页写入磁盘 (见下一节)。
4. 返回1 (成功)。

更新页表项

一旦 `add_to_swap()` 结束, `shrink_list()` 就调用 `try_to_unmap()`, 它确定引用匿名页的每个用户态页表项地址, 然后将换出页标识符写入其中 (参见本章前面“匿名页的反向映射”一节)。

将页写入交换区

为完成换出操作需执行的下一个步骤是将页的数据写入交换区。这一 I/O 传输是由 `shrink_list()` 函数激活的, 它检查页框的 `PG_dirty` 标志是否置位, 然后执行 `pageout()` 函数 (参见本章前面的图 17-5)。

在本章前面的“内存紧缺回收”一节我们描述过, `pageout()` 函数建立一个 `writeback_control` 描述符, 且调用页 `address_space` 对象的 `writepage` 方法。而 `swapper_state` 对象的 `writepage` 方法是由 `swap_writepage()` 函数实现的。

`swap_writepage()` 函数所执行的主要步骤如下:

1. 检查是否至少有一个用户态进程引用该页。如果没有, 则从交换高速缓存删除该页, 并返回0。这一检查之所以必须做, 是因为一个进程可能会与 PFRA 发生竞争并在 `shrink_list()` 检查完后释放一页。
2. 调用 `get_swap_bio()` 分配并初始化一个 `bio` 描述符 (参见第十四章“bio 结构”一节)。函数从换出页标识符算出交换区描述符地址。然后它搜索交换子区链表, 以

找到页槽的初始磁盘扇区。bio描述符将包含一个单页数据请求（页槽），其完成方法设为 `end_swap_bio_write()` 函数。

3. 置位页描述符的 `PG_writeback` 标志和交换高速缓存基树的 `writeback` 标记（参见第十五章的“基树的标记”一节）。此外函数还清 `OPG_locked` 标志。
4. 调用 `submit_bio()`，传给它 `WRITE` 命令和 bio 描述符地址。
5. 返回 0。

一旦 I/O 数据传输结束，就执行 `end_swap_bio_write()` 函数。实际上，这个函数唤醒正等待页 `PG_writeback` 标志清 0 的所有进程，清除 `PG_writeback` 标志和基树中的相关标记，并释放用于 I/O 传输的 bio 描述符。

从交换高速缓存中删除页框

换出操作的最后一步还是由 `shrink_list()` 执行。如果它验证在 I/O 数据传输时没有进程试图访问该页框，它实际就调用 `delete_from_swap_cache()` 从交换高速缓存中删除该页框。因为交换高速缓存是该页的唯一拥有者，该页框被释放到伙伴系统。

换入页

当进程试图对一个已被换出到磁盘的页进行寻址时，必然会发生页的换入。在以下条件发生时，缺页异常处理程序就会触发一个换入操作（参见第九章中的“处理地址空间内的错误地址”一节）：

- 引起异常的地址所在的页是一个有效的页，也就是说，它属于当前进程的一个线性区。
- 页不在内存中，也就是说，页表项中的 `Present` 标志被清除。
- 与页有关的页表项不为空，但是 `Dirty` 位清 0，这意味着页表项包含一个换出页标识符（参见第九章的“请求调页”一节）。

如果上面的所有条件满足，则 `handle_pte_fault()` 调用相对简易的 `do_swap_page()` 函数换入所需页。

do_swap_page()函数

`do_swap_page()` 函数作用于如下参数：

`mm`

引起缺页异常的进程的内存描述符地址

vma	address 所在的线性区的线性区描述符地址
address	引起异常的线性地址
page_table	映射 address 的页表项的地址
Pmd	映射 address 的页中间目录的地址
orig_pte	映射 address 的页表项的内容
write_access	一个标志, 表示试图执行的访问是读操作还是写操作

与其他函数相反, `do_swap_page()` 从不返回 0。如果页已经在交换高速缓存中就返回 1 (次错误), 如果页已经从交换区读入就返回 2 (主错误), 如果在进行换入时发生错误就返回 -1。该函数本质上执行下列步骤:

1. 从 `orig_pte` 获得换出页标识符。
2. 调用 `pte_unmap()` 释放任何页表的临时内核映射, 该页表由 `handle_mm_fault()` 函数建立 (参见第九章的“处理地址空间内的错误地址”一节)。正如第八章“高端内存页框的内核映射”一节所述, 访问高端内存页表需要进行内核映射。
3. 释放内存描述符的 `page_table_lock` 自旋锁 (它是由调用者函数 `handle_pte_fault()` 获取的)。
4. 调用 `lookup_swap_cache()` 检查交换高速缓存是否已经含有换出页标识符对应的页; 如果页已经在交换高速缓存中, 就跳到第 6 步。
5. 调用 `swpin_readahead()` 函数从交换区读取至多有 $2n$ 个页的一组页, 其中包括所请求的页。值 n 存放在 `page_cluster` 变量中, 通常等于 3 (注 11)。其中的每个页是通过调用 `read_swap_cache_async()` 函数读入的 (参见下面)。
6. 再一次调用 `read_swap_cache_async()` 换入由引起缺页异常的进程所访问的那一页。这一步可能看起来有点多余, 但其实不然。`swpin_readahead()` 函数可能在读取请求的页时失败——例如, 因为 `page_cluster` 被置为 0, 或者该函数试图读取一组含

注 11: 系统管理员通过写 `/proc/sys/vm/page-cluster` 文件可以调整这个值。通过把 `page-cluster` 置为 0 可以禁止提前换入页。

有空闲或有缺陷页槽 (SWAP_MAP_BAD) 的页。另一方面, 如果 `swpin_readahead()` 成功, 这次对 `read_swap_cache_async()` 的调用就很快结束, 因为它在交换高速缓存找到了页。

7. 尽管如此, 如果请求的页还是没有被加到交换高速缓存, 那么, 另一个内核控制路径可能已经代表这个进程的一个子进程换入了所请求的页。这种情况的检查可以通过临时获取 `page_table_lock` 自旋锁, 并把 `page_table` 所指向的表项与 `orig_pte` 进行比较来实现。如果二者有差异, 则说明这一页已经被某个其他的内核控制路径换入, 因此, 函数返回 1 (次错误); 否则, 返回 -1 (失败)。
8. 函数执行到此, 我们知道页已经在高速缓存中。如果页已被换入 (主错误), 函数就调用 `grab_swap_token()` 试图获得一个交换标记 (参见本章前面“交换标记”一节)。
9. 调用 `mark_page_accessed()` [参见前面“最近最少使用 (LRU) 链表”一节] 并对页加锁。
10. 获取 `page_table_lock` 自旋锁。
11. 检查另一个内核控制路径是否代表这个进程的一个子进程换入了所请求的页。如果是, 就释放 `page_table_lock` 自旋锁, 打开页上的锁, 并返回 1 (次错误)。
12. 调用 `swap_free()` 减少 `entry` 对应的页槽的引用计数器。
13. 检查交换高速缓存是否至少占满 50% (`nr_swap_pages` 小于 `total_swap_pages` 的一半)。如果是, 则检查页是否仅被引起异常的进程 (或其一个子进程) 拥有; 如果是这样, 则从交换高速缓存中删去这一页。
14. 增加进程的内存描述符的 `rss` 字段。
15. 更新页表项以便进程能找到这一页。这一操作的实现是通过把所请求页的物理地址和在线性区的 `vm_page_prot` 字段所找到的保护位写入 `page_table` 所指向的页表项中来达到的。此外, 如果引起缺页的访问是一个写访问, 且造成缺页的进程是页的唯一拥有者, 那么, 函数还要设置 `Dirty` 和 `Read/Write` 标志以防止无用的写时复制错误。
16. 打开页上的锁。
17. 调用 `page_add_anon_rmap()` 把匿名页插入面向对象的反向映射数据结构 (参见本章前面“匿名页的反向映射”一节)。
18. 如果 `write_access` 参数等于 1, 则函数调用 `do_wp_page()` 复制一份页框 (参见第九章的“写时复制”一节)。
19. 释放 `mm->page_table_lock` 自旋锁, 并返回 1 (次错误) 或 2 (主错误)。

read_swap_cache_async() 函数

只要内核必须换入一个页，就调用 `read_swap_cache_async()` 函数，它接收的参数为：

`entry`

换出页标识符

`vma`

指向该页所在线性区的指针

`addr`

页的线性地址

我们知道，在访问交换分区之前，该函数必须检查交换高速缓存是否已经包含了所要的页框。因此，该函数本质上执行下列操作：

1. 调用 `radix_tree_lookup()`，搜索 `swapper_space` 对象的基树，寻找由换出页标识符 `entry` 给出位置的页框。如果找到该页，递增它的引用计数器，返回它的描述符地址。
2. 页不在交换高速缓存。调用 `alloc_page()` 分配一个新的页框。如果没有空闲的页框可用，则返回 0（表示系统没有足够的内存）。
3. 调用 `add_to_swap_cache()` 把新页框的页描述符插入交换高速缓存。正如前面“交换高速缓存的辅助函数”一节提到的那样，这个函数也对页加锁。
4. 如果 `add_to_swap_cache()` 在交换高速缓存找到页的一个副本，则前一步可能失败。例如，进程可能在第 2 步阻塞，因此允许另一个进程在同一个页槽上开始换入操作。在这种情况下，该函数释放在第 2 步分配的页框，并从第 1 步重新开始。
5. 调用 `lru_cache_add_active()` 把页插入 LRU 的活动链表[参见本章前面“最近最少使用（LRU）链表”一节]。
6. 新页框的页描述符现已在交换高速缓存。调用 `swap_readpage()` 从交换区读入该页数据。这个函数与前面“换出页”一节所描述的 `swap_writepage()` 函数很相似，它将页描述符的 `PG_uptodate` 标志清 0，调用 `get_swap_bio()` 为 I/O 传输分配与初始化一个 `bio` 描述符，再调用 `submit_bio()` 向块设备子系统层发出 I/O 请求。
7. 返回页描述符的地址。



Ext2 和 Ext3 文件系统

在本章，我们通过当与一个特定的文件系统交互时内核所关注的细节来结束对 I/O 和文件系统进行的广泛讨论。因为第二扩展文件系统（Ext2）是 Linux 所固有的，事实上已在每个 Linux 系统中得以使用，因此我们自然要对 Ext2 进行讨论。此外，Ext2 在对现代文件系统的高性能支持方面也显示出很多良好的实践性。固然，Linux 支持的其他文件系统有很多有趣的特点，但因篇幅限制，我们并不对此一一讨论。

在“Ext2 的一般特征”一节中引入 Ext2 后，我们会像在其他章节中一样，接着描述所需的数据结构。因为我们着眼于磁盘上存放数据的特定方式，因此必须考虑同一数据结构的两种形式：“Ext2 磁盘数据结构”一节说明把 Ext2 存放在磁盘上的数据结构，而“Ext2 的内存数据结构”一节说明如何把磁盘上的数据结构复制到内存中。

然后，我们讨论 Ext2 文件系统上所执行的操作。在“创建 Ext2 文件系统”一节，我们讨论如何在磁盘分区创建 Ext2。接下来的一节描述使用磁盘时内核所执行的操作。其中的大部分操作是为索引节点和数据块分配磁盘空间，这些操作相对比较低级。

在最后一节，我们将对 Ext3 文件系统给予简短描述，Ext3 是 Ext2 文件系统的发展。

Ext2 的一般特征

类 Unix 操作系统使用多种文件系统。尽管所有这些文件系统都有少数 POSIX API（如 `stat()`）所需的共同的属性子集，但每种文件系统的实现方式是不同的。

Linux 的第一个版本是基于 MINIX 文件系统的。当 Linux 成熟时，引入了扩展文件系统 (*Extended Filesystem, Ext FS*)，它包含了几个重要的扩展但提供的性能不令人满意。在 1994 年引入了第二扩展文件系统 (Ext2)，它除了包含几个新的特点外，还相当高效和稳定，Ext2 及它的下代文件系统 Ext3 已成为广泛使用的 Linux 文件系统。

下列特点有助于 Ext2 的效率：

- 当创建 Ext2 文件系统时，系统管理员可以根据预期的文件平均长度来选择最佳块大小（从 1024B ~ 4096B）。例如，当文件的平均长度小于几千字节时，块的大小为 1024B 是最佳的，因为这会产生较少的内部碎片——也就是文件长度与存放它的磁盘分区有较少的不匹配（参见第八章中的“内存区管理”一节，在那里讨论了动态内存的内部碎片）。另一方面，大的块对于大于几千字节的文件通常比较合适，因为这样的磁盘传送较少，因而减轻了系统的开销。
- 当创建 Ext2 文件系统时，系统管理员可以根据在给定大小的分区上预计存放的文件数来选择给该分区分配多少个索引节点。这可以有效地利用磁盘的空间。
- 文件系统把磁盘块分为组。每组包含存放在相邻磁道上的数据块和索引节点。正是这种结构，使得可以用较少的磁盘平均寻道时间对存放在一个单独块组中的文件进行访问。
- 在磁盘数据块被实际使用之前，文件系统就把这些块预分配给普通文件。因此，当文件的大小增加时，因为物理上相邻的几个块已被保留，这就减少了文件的碎片。
- 支持快速符号链接（参见第一章中的“硬链接和软链接”一节）。如果符号链接表示一个短路径名（小于或等于 60 个字符），就把它存放在索引节点中而不用通过读一个数据块进行转换。

此外，Ext2 还包含了一些使它既健壮又灵活的特点：

- 文件更新策略的谨慎实现将系统崩溃的影响减到最少。例如，当给文件创建一个新的硬链接时，首先增加磁盘索引节点中的硬链接计数器，然后把这个名字加到合适的目录中。在这种方式下，如果在更新索引节点后而改变这个目录之前出现一个硬件故障，这样即使索引节点的计数器产生错误，但目录是一致的。因此，尽管删除文件时无法自动收回文件的数据块，但并不导致灾难性的后果。如果这种操作的顺序相反（更新索引节点前改变目录），同样的硬件故障将会导致危险的不一致：删除原始的硬链接就会从磁盘删除它的数据块，但新的目录项将指向一个不存在的索引节点。如果那个索引节点号以后又被另外的文件所使用，那么向这个旧目录项的写操作将毁坏这个新的文件。
- 在启动时支持对文件系统的状态进行自动的一致性检查。这种检查是由外部程序

e2fsck 完成的, 这个外部程序不仅可以在系统崩溃之后被激活, 也可以在一个预定义的文件系统安装数 (每次安装操作之后对计数器加 1) 之后被激活, 或者在自从最近检查以来所花的预定义时间之后被激活。

- 支持不可变 (*immutable*) 的文件 (不能修改、删除和更名) 和仅追加 (*append-only*) 的文件 (只能把数据追加在文件尾)。
- 既与 Unix System V Release 4 (SVR4) 相兼容, 也与新文件的用户组 ID 的 BSD 语义相兼容。在 SVR4 中, 新文件采用创建它的进程的用户组 ID; 而在 BSD 中, 新文件继承包含它的目录的用户组 ID。Ext2 包含一个安装选项, 由你指定采用哪种语义。

即使 Ext2 文件系统是如此成熟、稳定的程序, 也还要考虑引入另外几个特性。一些特性已被实现并以外部补丁的形式来使用。另外一些还仅仅处于计划阶段, 但在一些情况下, 已经在 Ext2 的索引节点中为这些特性引入新的字段。最重要的一些特点如下:

块片 (*block fragmentation*)

系统管理员对磁盘的访问通常选择较大的块, 因为计算机应用程序常常处理大文件。因此, 在大块上存放小文件就会浪费很多磁盘空间。这个问题可以通过把几个文件存放在同一块的不同片上来解决。

透明地处理压缩和加密文件

这些新的选项 (创建一个文件时必须指定) 将允许用户透明地在磁盘上存放压缩和 (或) 加密的文件版本。

逻辑删除

一个 *undelete* 选项将允许用户在必要时很容易恢复以前已删除的文件内容。

日志

日志避免文件系统在突然卸载 (例如, 作为系统崩溃的后果) 时对其自动进行的耗时检查。

实际上, 这些特点没有一个正式地包含在 Ext2 文件系统中。有人可能说 Ext2 是这种成功的牺牲品; 直到几年前, 它仍然是大多数 Linux 发布公司采用的首选文件系统, 每天有成千上万的用户在使用它, 这些用户会对用其他文件系统来代替 Ext2 的任何企图产生质疑。

Ext2 中缺少的最突出的功能就是日志, 日志是高可用服务器必需的功能。为了平顺过渡, 日志没有引入到 Ext2 文件系统; 但是, 我们在后面 “Ext3 文件系统” 一节会讨论, 完全与 Ext2 兼容的一种新文件系统已经创建, 这种文件系统提供了日志。不真正需要日

志的用户可以继续使用良好而老式的Ext2文件系统,而其他用户可能采用这种新的文件系统。现在发行的大部分系统采用 Ext3 作为标准的文件系统。

Ext2 磁盘数据结构

任何 Ext2 分区中的第一个块从不受 Ext2 文件系统的管理,因为这一块是为分区的引导扇区所保留的(参见附录一)。Ext2 分区的其余部分分成块组 (*block group*), 每个块组的分布图如图 18-1 所示。正如你从图中所看到的, 一些数据结构正好可以放在一块中, 而另一些可能需要更多的块。在 Ext2 文件系统的所有块组大小相同并被顺序存放, 因此, 内核可以从块组的整数索引很容易地得到磁盘中一个块组的位置。

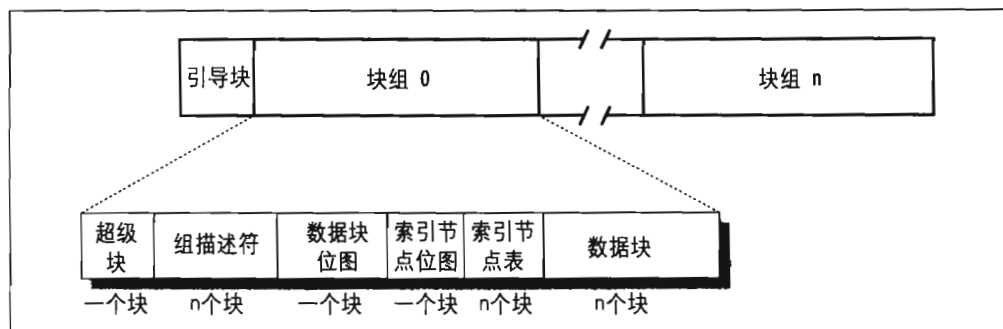


图 18-1: Ext2 分区和 Ext2 块组的分布图

由于内核尽可能地把属于一个文件的数据块存放在同一块组中, 所以块组减少了文件的碎片。块组中的每个块包含下列信息之一:

- 文件系统的超级块的一个拷贝
- 一组块组描述符的拷贝
- 一个数据块位图
- 一个索引节点位图
- 一个索引节点表
- 属于文件的一大块数据, 即数据块

如果一个块中不包含任何有意义的信息, 就说这个块是空闲的。

从图 18-1 中可以看出, 超级块与组描述符被复制到每个块组中。只有块组 0 中所包含的超级块和组描述符才由内核使用, 而其余的超级块和组描述符保持不变; 事实上, 内核甚至不考虑它们。当 *e2fsck* 程序对 Ext2 文件系统的状态执行一致性检查时, 就引用存放

在块组 0 中的超级块和组描述符，然后把它们拷贝到其他所有的块组中。如果出现数据损坏，并且块组 0 中的主超级块和主描述符变为无效，那么，系统管理员就可以命令 *e2fsck* 引用存放在某个块组（除了第一个块组）中的超级块和组描述符的旧拷贝。通常情况下，这些多余的拷贝所存放的信息足以让 *e2fsck* 把 Ext2 分区带回到一个一致的状态。

有多少块组呢？这取决于分区的大小和块的大小。其主要限制在于块位图，因为块位图必须存放在一个单独的块中。块位图用来标识一个组中块的占用和空闲状况。所以，每组中至多可以有 $8 \times b$ 个块， b 是以字节为单位的块大小。因此，块组的总数大约是 $s / (8 \times b)$ ，这里 s 是分区所包含的总块数。

举例说明，让我们考虑一下 32GB 的 Ext2 分区，块的大小为 4KB。在这种情况下，每个 4KB 的块位图描述 32K 个数据块，即 128MB。因此，最多需要 256 个块组。显然，块的大小越小，块组数越大。

超级块

Ext2 在磁盘上的超级块存放在一个 `ext2_super_block` 结构中，它的字段在表 18-1 中列出（注 1）。`__u8`、`__u16` 及 `__u32` 数据类型分别表示长度为 8、16 及 32 位的无符号数，而 `__s8`、`__s16` 及 `__s32` 数据类型表示长度为 8、16 及 32 位的有符号数。为清晰地表示磁盘上字或双字中字节的存放顺序，内核又使用了 `__le16`、`__le32`、`__be16` 和 `__be32` 数据类型，前两种类型分别表示字或双字的“小尾（*little-endian*）”排序方式（低阶字节在高位地址），而后两种类型分别表示字或双字的“大尾（*big-endian*）”排序方式（高阶字节在高位地址）。

表 18-1：Ext2 超级块的字段

类型	字段	说明
<code>__le32</code>	<code>s_inodes_count</code>	索引节点的总数
<code>__le32</code>	<code>s_blocks_count</code>	以块为单位的文件系统的大小
<code>__le32</code>	<code>s_r_blocks_count</code>	保留的块数
<code>__le32</code>	<code>s_free_blocks_count</code>	空闲块计数器
<code>__le32</code>	<code>s_free_inodes_count</code>	空闲索引节点计数器
<code>__le32</code>	<code>s_first_data_block</code>	第一个使用的块号（总为 1）
<code>__le32</code>	<code>s_log_block_size</code>	块的大小

注 1：为了确保 Ext2 和 Ext3 文件系统之间的兼容性，`ext2_super_block` 数据结构包含了一些 Ext3 特有的字段，它们并未在表 18-1 中列出。

表 18-1: Ext2 超级块的字段 (续)

类型	字段	说明
__le32	s_log_frag_size	片的大小
__le32	s_blocks_per_group	每组中的块数
__le32	s_frags_per_group	每组中的片数
__le32	s_inodes_per_group	每组中的索引节点数
__le32	s_mtime	最后一次安装操作的时间
__le32	s_wtime	最后一次写操作的时间
__le16	s_mnt_count	安装操作计数器
__le16	s_max_mnt_count	检查之前安装操作的次数
__le16	s_magic	魔术签名
__le16	s_state	状态标志
__le16	s_errors	当检测到错误时的行为
__le16	s_minor_rev_level	次版本号
__le32	s_lastcheck	最后一次检查的时间
__le32	s_checkinterval	两次检查之间的时间间隔
__le32	s_creator_os	创建文件系统的操作系统
__le32	s_rev_level	版本号
__le16	s_def_resuid	保留块的缺省 UID
__le16	s_def_resgid	保留块的缺省用户组 ID
__le32	s_first_ino	第一个非保留的索引节点号
__le16	s_inode_size	磁盘上索引节点结构的大小
__le16	s_block_group_nr	这个超级块的块组号
__le32	s_feature_compat	具有兼容特点的位图
__le32	s_feature_incompat	具有非兼容特点的位图
__le32	s_feature_ro_compat	只读兼容特点的位图
__u8 [16]	s_uuid	128 位文件系统标识符
char [16]	s_volume_name	卷名
char [64]	s_last_mounted	最后一个安装点的路径名
__le32	s_algorithm_usage_bitmap	用于压缩
__u8	s_prealloc_blocks	预分配的块数
__u8	s_prealloc_dir_blocks	为目录预分配的块数
__u16	s_padding1	按字对齐
__u32 [204]	s_reserved	用 null 填充 1024 字节

s_inodes_count 字段存放索引节点的个数，而 s_blocks_count 字段存放 Ext2 文件系统的块的个数。

s_log_block_size 字段以 2 的幂次方表示块的大小，用 1024 字节作为单位。因此，0 表示 1024 字节的块，1 表示 2048 字节的块，如此等等。目前 s_log_frag_size 字段与 s_log_block_size 字段相等，因为块片还没有实现。

s_blocks_per_group、s_frags_per_group 与 s_inodes_per_group 字段分别存放每个块组中的块数、片数及索引节点数。

一些磁盘块保留给超级用户（或由 s_def_resuid 和 s_def_resgid 字段挑选给某一其他用户或用户组）。即使当普通用户没有空闲块可用时，系统管理员也可以用这些块继续使用 Ext2 文件系统。

s_mnt_count、s_max_mnt_count、s_lastcheck 及 s_checkinterval 字段使系统启动时自动地检查 Ext2 文件系统。在预定义的安装操作数完成之后，或自最后一次一致性检查以来预定义的时间已经用完，这些字段就导致 *e2fsck* 执行（两种检查可以一起进行）。如果 Ext2 文件系统还没有被全部卸载（例如系统崩溃以后），或内核在其中发现一些错误，则一致性检查在启动时要强制进行。如果 Ext2 文件系统被安装或未被全部卸载，则 s_state 字段存放的值为 0；如果被正常卸载，则这个字段的值为 1；如果包含错误，则值为 2。

组描述符和位图

每个块组都有自己的组描述符，它是一个 ext2_group_desc 结构，它的字段在表 18-2 中列出。

表 18-2: Ext2 组描述符的字段

类型	字段	说明
__le32	bg_block_bitmap	块位图的块号
__le32	bg_inode_bitmap	索引节点位图的块号
__le32	bg_inode_table	第一个索引节点表块的块号
__le16	bg_free_blocks_count	组中空闲块的个数
__le16	bg_free_inodes_count	组中空闲索引节点的个数
__le16	bg_used_dirs_count	组中目录的个数
__le16	bg_pad	按字对齐
__le32 [3]	bg_reserved	用 null 填充 24 个字节

当分配新索引节点和数据块时,会用到**bg_free_blocks_count**、**bg_free_inodes_count**和**bg_used_dirs_count**字段。这些字段确定在最合适的块中给每个数据结构进行分配。位图是位的序列,其中值0表示对应的索引节点块或数据块是空闲的,1表示占用。因为每个位图必须存放在一个单独的块中,又因为块的大小可以是1024、2048或4096字节,因此,一个单独的位图描述8192、16384或32768个块的状态。

索引节点表

索引节点表由一连串连续的块组成,其中每一块包含索引节点的一个预定义号。索引节点表第一个块的块号存放在组描述符的**bg_inode_table**字段中。

所有索引节点的大小相同,即128字节。一个1024字节的块可以包含8个索引节点,一个4096字节的块可以包含32个索引节点。为了计算出索引节点表占用了多少块,用一个组中的索引节点总数(存放在超级块的**s_inodes_per_group**字段中)除以每块中的索引节点数。

每个Ext2索引节点为**ext2_innode**结构,其字段如表18-3所示。

表 18-3: Ext2 磁盘索引节点的字段

类型	字段	说明
__le16	i_mode	文件类型和访问权限
__le16	i_uid	拥有者标识符
__le32	i_size	以字节为单位的文件长度
__le32	i_atime	最后一次访问文件的时间
__le32	i_ctime	索引节点最后改变的时间
__le32	i_mtime	文件内容最后改变的时间
__le32	i_dtime	文件删除的时间
__le16	i_gid	用户组标识符
__le16	i_links_count	硬链接计数器
__le32	i_blocks	文件的数据块数
__le32	i_flags	文件标志
union	osdl	特定的操作系统信息
__le32 [EXT2_N_BLOCKS]	i_block	指向数据块的指针
__le32	i_generation	文件版本(当网络文件系统访问文件时使用)
__le32	i_file_acl	文件访问控制列表

表 18-3: Ext2 磁盘索引节点的字段 (续)

类型	字段	说明
__le32	i_dir_acl	目录访问控制列表
__le32	i_faddr	片的地址
union	osd2	特定的操作系统信息

与 POSIX 规范相关的很多字段类似于 VFS 索引节点对象的相应字段，这已在第十二章的“索引节点对象”一节中讨论过。其余的字段与 Ext2 的特殊实现相关，主要处理块的分配。

特别地，i_size 字段存放以字节为单位的文件的有效长度，而 i_blocks 字段存放已分配给文件的数据块数（以 512 字节为单位）。

i_size 和 i_blocks 的值没有必然的联系。因为一个文件总是存放在整数块中，一个非空文件至少接受一个数据块（因为还没实现片）且 i_size 可能小于 $512 \times i\_blocks$ 。另一方面，我们将在本章后面的“文件的洞”一节中看到，一个文件可能包含有洞。在那种情况下，i_size 可能大于 $512 \times i\_blocks$ 。

i_blocks 字段是具有 EXT2_N_BLOCKS（通常是 15）个指针元素的一个数组，每个元素指向分配给文件的数据块（参见本章后面的“数据块寻址”一节）。

留给 i_size 字段的 32 位把文件的大小限制到 4GB。事实上，i_size 字段的最高位没有使用，因此，文件的最大长度限制为 2GB。然而，Ext2 文件系统包含一种“脏技巧”，允许像 AMD 的 Opteron 和 IBM 的 PowerPC G5 这样的 64 位体系结构使用大型文件。从本质上说，索引节点的 i_dir_acl 字段（普通文件没有使用）表示 i_size 字段的 32 位扩展。因此，文件的大小作为 64 位整数存放在索引节点中。Ext2 文件系统的 64 位版本与 32 位版本在某种程度上兼容，因为在 64 位体系结构上创建的 Ext2 文件系统可以安装在 32 位体系结构上，反之亦然。但是，在 32 位体系结构上不能访问大型文件，除非以 O_LARGEFILE 标志打开文件（参见第十二章“open() 系统调用”一节）。

回忆一下，VFS 模型要求每个文件有不同的索引节点号。在 Ext2 中，没有必要在磁盘上存放文件的索引节点号与相应块号之间的转换，因为后者的值可以从块组号和它在索引节点表中的相对位置而得出。例如，假设每个块组包含 4096 个索引节点，我们想知道索引节点 13021 在磁盘上的地址。在这种情况下，这个索引节点属于第三个块组，它的磁盘地址存放在相应索引节点表的第 733 个表项中。正如你看到的，索引节点号是 Ext2 例程用来快速搜索磁盘上合适的索引节点描述符的一个关键字。

索引节点的增强属性

Ext2索引节点的格式对于文件系统设计者就好像一件紧身衣，索引节点的长度必须是2的幂，以免造成存放索引节点表的块内碎片。实际上，一个Ext2索引节点的128个字符空间中充满了信息，只有少许空间可以增加新的字段。另一方面，将索引节点的长度增加至256不仅相当浪费，而且使用不同索引节点长度的Ext2文件系统之间还会造成兼容问题。

引入增强属性（*extended attribute*）就是要克服上面的问题。这些属性存放在索引节点之外的磁盘块中。索引节点的 `i_file_acl` 字段指向一个存放增强属性的块。具有同样增强属性的不同索引节点可以共享同一个块。

每个增强属性有一个名称和值。两者都编码为变长字符数组，并由 `ext2_xattr_entry` 描述符来确定。图 18-2 表示 Ext2 中增强属性块的结构。每个属性分成两部分：在块首部的是 `ext2_xattr_entry` 描述符与属性名，而属性值则在块尾部。块前面的表项按照属性名称排序，而值的位置是固定的，因为它们是由属性的分配次序决定的。

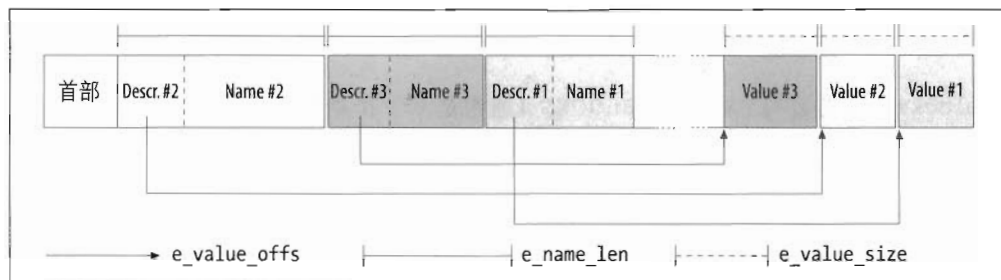


图 18-2：含增强属性的块的结构

有很多系统调用用来设置、取得、列表和删除一个文件的增强属性。系统调用 `setxattr()`、`lsetxattr()` 和 `fsetxattr()` 设置文件的增强属性，它们在符号链接的处理与文件限定的方式（或者传递路径名或者是文件描述符）上根本不同。类似地，系统调用 `getxattr()`、`lgetxattr()` 和 `fgetxattr()` 返回增强属性的值。系统调用 `listxattr()`、`llistxattr()` 和 `flistxattr()` 则列出一个文件的所有增强属性。最后，系统调用 `removexattr()`、`lremovexattr()` 和 `fremovexattr()` 从文件删除一个增强属性。

访问控制列表

很早以前访问控制列表就被建议用来改善 Unix 文件系统的保护机制。不是将文件的用户分成三类：拥有者、组和其他，访问控制列表（*access control list, ACL*）可以与每个

文件关联。有了这种列表，用户可以为他的文件限定可以访问的用户（或用户组）名称以及相应的权限。

Linux 2.6 通过索引节点的增强属性完整实现 ACL。实际上，增强属性主要就是为了支持 ACL 才引入的。因此，能让你处理文件 ACL 的库函数 `chacl()`、`setfacl()` 和 `getfacl()` 就是通过上一节中介绍的 `setxattr()` 和 `getxattr()` 系统调用实现的。

不幸的是，在 POSIX 1003.1 系列标准内，定义安全增强属性的工作组所完成的成果从没有正式成为新的 POSIX 标准。因此现在，不同的类 Unix 文件系统都支持 ACL，但不同的实现之间有一些微小的差别。

各种文件类型如何使用磁盘块

Ext2 所认可的文件类型（普通文件、管道文件等）以不同的方式使用数据块。有些文件不存放数据，因此根本就不需要数据块。本节讨论每种文件类型的存储要求，如表 18-4 所示。

表 18-4：Ext2 文件类型

文件类型	说明
0	未知
1	普通文件
2	目录
3	字符设备
4	块设备
5	命名管道
6	套接字
7	符号链接

普通文件

普通文件是最常见的情况，本章主要关注它。但普通文件只有在开始有数据时才需要数据块。普通文件在刚创建时是空的，并不需要数据块，也可以用 `truncate()` 或 `open()` 系统调用清空它。这两种情况是相同的，例如，当你发出一个包含字符串 `>filename` 的 shell 命令时，shell 创建一个空文件或截断一个现有文件。

目录

Ext2以一种特殊的文件实现了目录，这种文件的数据块把文件名和相应的索引节点号存放在一起。特别说明的是，这样的数据块包含了类型为 `ext2_dir_entry_2` 的结构。表 18-5 列出了这个结构的字段。因为该结构最后一个 `name` 字段是最大为 `EXT2_NAME_LEN`（通常是 255）个字符的变长数组，因此这个结构的长度是可变的。此外，因为效率的原因，目录项的长度总是 4 的倍数，并在必要时用 `null` 字符（`\0`）填充文件名的末尾。`name_len` 字段存放实际的文件名长度（参见图 18-3）。

表 18-5: Ext2 目录项中的字段

类型	字段	说明
<code>__le32</code>	<code>inode</code>	索引节点号
<code>__le16</code>	<code>rec_len</code>	目录项长度
<code>__u8</code>	<code>name_len</code>	文件名长度
<code>__u8</code>	<code>file_type</code>	文件类型
<code>char [EXT2_NAME_LEN]</code>	<code>name</code>	文件名

`file_type` 字段存放指定文件类型的值（见表 18-4）。`rec_len` 字段可以被解释为指向下一个有效目录项的指针：它是偏移量，与目录项的起始地址相加就得到下一个有效目录项的起始地址。为了删除一个目录项，把它的 `inode` 字段置为 0 并适当地增加前一个有效目录项 `rec_len` 字段的值就足够了。仔细看一下图 18-3 的 `rec_len` 字段，你会发现 `oldfile` 项已被删除，因为 `usr` 的 `rec_len` 字段被置为 `12+16`（`usr` 和 `oldfile` 目录项的长度）。

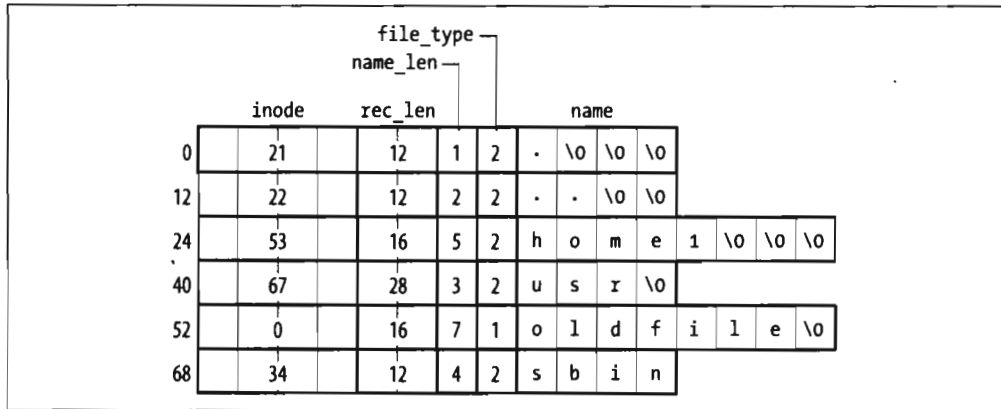


图 18-3: Ext2 目录的一个例子

符号链接

如前所述，如果符号链接的路径名小于等于 60 个字符，就把它存放在索引节点的 `i_blocks` 字段，该字段是由 15 个 4 字节整数组成的数组，因此无需数据块。但是，如果路径名大于 60 个字符，就需要一个单独的数据块。

设备文件、管道和套接字

这些类型的文件不需要数据块。所有必要的信息都存放在索引节点中。

Ext2 的内存数据结构

为了提高效率，当安装 Ext2 文件系统时，存放在 Ext2 分区的磁盘数据结构中的大部分信息被拷贝到 RAM 中，从而使内核避免了后来的很多读操作。那么一些数据结构如何经常更新呢？让我们考虑一些基本的操作：

- 当一个新文件被创建时，必须减少 Ext2 超级块中 `s_free_inodes_count` 字段的值和相应的组描述符中 `bg_free_inodes_count` 字段的值。
- 如果内核给一个现有的文件追加一些数据，以使分配给它的数据块数因此也增加，那么就必须修改 Ext2 超级块中 `s_free_blocks_count` 字段的值和组描述符中 `bg_free_blocks_count` 字段的值。
- 即使仅仅重写一个现有文件的部分内容，也要对 Ext2 超级块的 `s_wtime` 字段进行更新。

因为所有的 Ext2 磁盘数据结构都存放在 Ext2 分区的块中，因此，内核利用页高速缓存来保持它们最新（参见第十五章中的“把脏页写入磁盘”一节）。

对于与 Ext2 文件系统以及文件相关的每种数据类型，表 18-6 详细说明了在磁盘上用来表示数据的数据结构、在内存中内核所使用的数据结构以及决定使用多大容量高速缓存的经验方法。频繁更新的数据总是存放在高速缓存，也就是说，这些数据一直存放在内存并包含在页高速缓存中，直到相应的 Ext2 分区被卸载。内核通过让缓冲区的引用计数器一直大于 0 来达到此目的。

表 18-6：Ext2 数据结构的 VFS 映像

类型	磁盘数据结构	内存数据结构	缓存模式
超级块	<code>ext2_super_block</code>	<code>ext2_sb_info</code>	总是缓存
组描述符	<code>ext2_group_desc</code>	<code>ext2_group_desc</code>	总是缓存

表 18-6: Ext2 数据结构的 VFS 映像 (续)

类型	磁盘数据结构	内存数据结构	缓存模式
块位图	块中的位数组	缓冲区中的位数组	动态
索引节点位图	块中的位数组	缓冲区中的位数组	动态
索引节点	ext2_inode	ext2_inode_info	动态
数据块	字节数组	VFS 缓冲区	动态
空闲索引节点	ext2_inode	无	从不缓存
空闲块	字节数组	无	从不缓存

在任何高速缓存中不保存“从不缓存”的数据，因为这种数据表示无意义的信息。相反，“总是缓存”的数据也总在 RAM 中，这样就不必从磁盘读数据了（但是，数据必须周期性地写回磁盘）。除了这两种极端模式外，还有一种动态模式。在动态模式下，只要相应的对象（索引节点、数据块或位图）还在使用，它就保存在高速缓存中；而当文件关闭或数据块被删除后，页框回收算法会从高速缓存中删除有关数据。

有意思的是，索引节点与块位图并不永久保存在内存里，而是需要时从磁盘读。有了页高速缓存，最近使用的磁盘块保存在内存里，这样可以避免很多磁盘读（参见第十五章“把块存放在页高速缓存中”一节）（注 2）。

Ext2 的超级块对象

在第十二章“超级块对象”一节我们介绍过，VFS 超级块的 `s_fs_info` 字段指向一个包含文件系统信息的数据结构。对于 Ext2，该字段指向 `ext2_sb_info` 类型的结构，它包含如下信息：

- 磁盘超级块中的大部分字段
- `s_sbh` 指针，指向包含磁盘超级块的缓冲区的缓冲区首部
- `s_es` 指针，指向磁盘超级块所在的缓冲区
- 组描述符的个数 `s_desc_per_block`，可以放在一个块中
- `s_group_desc` 指针，指向一个缓冲区（包含组描述符的缓冲区）首部数组（通常一项就够）

注 2：在 Linux 2.4 和早期的版本中，最近使用的索引节点和块位图被保存在特殊高速缓存的有限空间里。

- 其他与安装状态、安装选项等有关的数据

图 18-4 表示的是与 Ext2 超级块和组描述符有关的缓冲区与缓冲区首部和 `ext2_sb_info` 数据结构之间的关系。

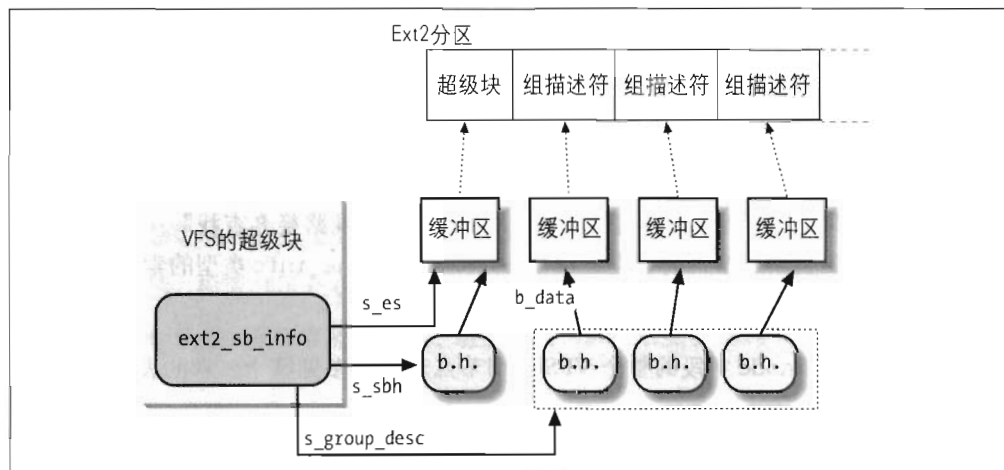


图 18-4: `ext2_sb_info` 数据结构

当内核安装 Ext2 文件系统时，它调用 `ext2_fill_super()` 函数来为数据结构分配空间，并写入从磁盘读取的数据（参见第十二章“安装普通文件系统”一节）。这里是对该函数的一个简要说明，只强调缓冲区与描述符的内存分配。

1. 分配一个 `ext2_sb_info` 描述符，将其地址当作参数传递并存放在超级块的 `s_fs_info` 字段。
2. 调用 `__bread()` 在缓冲区页中分配一个缓冲区和缓冲区首部。然后从磁盘读入超级块存放在缓冲区中。在第十五章的“在页高速缓存中搜索块”一节我们讨论过，如果一个块已在页高速缓存的缓冲区页而且是最初的，那么无需再分配。将缓冲区首部地址存放在 Ext2 超级块对象的 `s_sbh` 字段。
3. 分配一个字节数组，每组一个字节，把它的地址存放在 `ext2_sb_info` 描述符的 `s_debts` 字段（参见本章后面的“创建索引节点”一节）。
4. 分配一个数组用于存放缓冲区首部指针，每个组描述符一个，把该数组地址存放在 `ext2_sb_info` 的 `s_group_desc` 字段。
5. 重复调用 `__bread()` 分配缓冲区，从磁盘读入包含 Ext2 组描述符的块。把缓冲区首部地址存放在上一步得到的 `s_group_desc` 数组中。

6. 为根目录分配一个索引节点和目录项对象,为超级块建立相应的字段,从而能够从磁盘读入根索引节点对象。

很显然, `ext2_fill_super()` 函数返回后,分配的所有数据结构都保存在内存里,只有当 Ext2 文件系统卸载时才会被释放。当内核必须修改 Ext2 超级块的字段时,它只要把新值写入相应缓冲区内的相应位置然后将该缓冲区标记为脏即可。

Ext2 的索引节点对象

在打开文件时,要执行路径名查找。对于不在目录项高速缓存内的路径名元素,会创建一个新的目录项对象和索引节点对象(参见第十二章“标准路径名查找”一节)。当 VFS 访问一个 Ext2 磁盘索引节点时,它会创建一个 `ext2_inode_info` 类型的索引节点描述符。该描述符包含下列信息:

- 存放在 `vfs_inode` 字段的整个 VFS 索引节点对象(参见第十二章的表 12-3)
- 磁盘索引节点对象结构中的大部分字段(不保存在 VFS 索引节点中)
- 索引节点对应的 `i_block_group` 块组索引(参见本章前面“Ext2 磁盘数据结构”一节)
- `i_next_alloc_block` 和 `i_next_alloc_goal` 字段,分别存放着最近为文件分配的磁盘块的逻辑块号与物理块号
- `i_prealloc_block` 和 `i_prealloc_count` 字段,用于数据块预分配(参见本章后面“分配数据块”一节)
- `xattr_sem` 字段,一个读写信号量,允许增强属性与文件数据同时读入
- `i_acl` 和 `i_default_acl` 字段,指向文件的 ACL

当处理 Ext2 文件时, `alloc_inode` 超级块方法是由 `ext2_alloc_inode()` 函数实现的。它首先从 `ext2_inode_cache` slab 分配器高速缓存得到一个 `ext2_inode_info` 描述符,然后返回在这个 `ext2_inode_info` 描述符中的索引节点对象的地址。

创建 Ext2 文件系统

在磁盘上创建一个文件系统通常有两个阶段。第二步格式化磁盘,以使磁盘驱动程序可以读和写磁盘上的块。现在的硬磁盘已经由厂家预先格式化,因此不需要重新格式化;在 Linux 上可以使用 *superformat* 或 *fdformat* 等实用程序对软盘进行格式化。第二步才涉及创建文件系统,这意味着建立本章前面详细描述的结构。

Ext2 文件系统是由实用程序 *mke2fs* 创建的。*mke2fs* 采用下列缺省选项，用户可以用命令行的标志修改这些选项：

- 块大小：1024 字节（小文件系统的缺省值）
- 片大小：块的大小（块的分片还没有实现）
- 所分配的索引节点个数：每 8192 字节的组分配一个索引节点
- 保留块的百分比：5%

mke2fs 程序执行下列操作：

1. 初始化超级块和组描述符。
2. 作为选择，检查分区是否包含有缺陷的块，如果有，就创建一个有缺陷块的链表。
3. 对于每个块组，保留存放超级块、组描述符、索引节点表及两个位图所需要的所有磁盘块。
4. 把索引节点位图和每个块组的数据映射位图都初始化为 0。
5. 初始化每个块组的索引节点表。
6. 创建 */root* 目录。
7. 创建 *lost+found* 目录，由 *e2fsck* 使用这个目录把丢失和找到的缺陷块连接起来。
8. 在前两个已经创建的目录所在的块组中，更新块组中的索引节点位图和数据块位图。
9. 把有缺陷的块（如果存在）组织起来放在 *lost+found* 目录中。

让我们看一下 *mke2fs* 是如何以缺省选项初始化 Ext2 的 1.44 MB 软盘的。

软盘一旦被安装，VFS 就把它看作由 1412 个块组成的一个卷，每块大小为 1024 字节。为了查看磁盘的内容，我们可以执行如下 Unix 命令：

```
$ dd if=/dev/fd0 bs=1k count=1440 | od -tx1 -Ax > /tmp/dump_hex
```

从而获得了 */tmp* 目录下的一个文件，这个文件包含十六进制的软盘内容的转储（注 3）。

通过查看 *dump_hex* 文件我们可以看到，由于软盘有限的容量，一个单独的块组描述符就足够了。我们还注意到保留的块数为 72（1440 块的 5%），并且根据缺省选项，索引节点表必须为每 8192 个字节设置一个索引节点，也就是有 184 个索引节点存放在 23 个块中。

注 3： 使用 *dumpe2fs* 和 *debugfs* 实用程序也可以获得有关 Ext2 文件系统的一些信息。

表 18-7 总结了按缺省选项如何在软盘上建立 Ext2 文件系统。

表 18-7: 软盘的 Ext2 块分配

块	内容
0	引导块
1	超级块
2	包含一个单独的块组描述符的块
3	数据块位图
4	索引节点位图
5 ~ 27	索引节点表: 5 ~ 10 —— 保留 (2 是 root); 11 —— lost+found; 12 ~ 184 —— 空闲
28	根目录 (包括 “.”、 “..” 及 “lost+found”)
29	lost+found 目录 (包括 “.” 和 “..”)
30 ~ 40	给 lost+found 目录预分配保留的块
41 ~ 1439	空闲块

Ext2 的方法

在第十二章所描述的关于 VFS 的很多方法在 Ext2 都有相应的实现。因为对所有的方法都进行描述将需要整整一本书, 因此我们仅仅简单地回顾一下在 Ext2 中所实现的方法。一旦你真正搞明白了磁盘和内存数据结构, 你就应当能理解实现这些方法的 Ext2 函数的代码。

Ext2 超级块的操作

很多 VFS 超级块操作在 Ext2 中都有具体的实现, 这些方法为 `alloc_inode`、`destroy_inode`、`read_inode`、`write_inode`、`delete_inode`、`put_super`、`write_super`、`statfs`、`remount_fs` 和 `clear_inode`。超级块方法的地址存放在 `ext2_sops` 指针数组中。

Ext2 索引节点的操作

一些 VFS 索引节点的操作在 Ext2 中都有具体的实现, 这取决于索引节点所指的文件类型。

Ext2 的普通文件和目录文件的索引节点操作见表 18-8。每个方法的目的在第十二章的“索

引节点对象”一节有介绍。表中没有列出普通文件和目录中未定义的方法（NULL 指针）。回忆一下，如果方法未定义，VFS 要么调用通用函数，要么什么也不做。普通文件与目录的 Ext2 方法地址分别存放在 `ext2_file_inode_operations` 和 `ext2_dir_inode_operations` 表中。

表 18-8：Ext2 普通文件与目录的索引节点操作

VFS 索引节点操作	普通文件	目录
<code>create</code>	NULL	<code>ext2_create()</code>
<code>lookup</code>	NULL	<code>ext2_lookup()</code>
<code>link</code>	NULL	<code>ext2_link()</code>
<code>unlink</code>	NULL	<code>ext2_unlink()</code>
<code>symlink</code>	NULL	<code>ext2_symlink()</code>
<code>mkdir</code>	NULL	<code>ext2_mkdir()</code>
<code>rmdir</code>	NULL	<code>ext2_rmdir()</code>
<code>mknod</code>	NULL	<code>ext2_mknod()</code>
<code>rename</code>	NULL	<code>ext2_rename()</code>
<code>truncate</code>	<code>ext2_truncate()</code>	NULL
<code>permission</code>	<code>ext2_permission()</code>	<code>ext2_permission()</code>
<code>setattr</code>	<code>ext2_setattr()</code>	<code>ext2_setattr()</code>
<code>setxattr</code>	<code>generic_setxattr()</code>	<code>generic_setxattr()</code>
<code>getxattr</code>	<code>generic_getxattr()</code>	<code>generic_getxattr()</code>
<code>listxattr</code>	<code>ext2_listxattr()</code>	<code>ext2_listxattr()</code>
<code>removexattr</code>	<code>generic_removexattr()</code>	<code>generic_removexattr()</code>

Ext2 的符号链接的索引节点操作见表 18-9（省略未定义的方法）。实际上有两种符号链接：快速符号链接（路径名全部存放在索引节点内）与普通符号链接（较长的路径名）。因此，有两套索引节点操作，分别存放在 `ext2_fast_symlink_inode_operations` 和 `ext2_symlink_inode_operations` 表中。

表 18-9：Ext2 的快速及普通符号链接的索引节点操作

VFS 索引节点操作	快速符号链接	普通符号链接
<code>readlink</code>	<code>generic_readlink()</code>	<code>generic_readlink()</code>
<code>follow_link</code>	<code>ext2_follow_link()</code>	<code>page_follow_link_light()</code>
<code>put_link</code>	NULL	<code>page_put_link()</code>

表 18-9: Ext2 的快速及普通符号链接的索引节点操作 (续)

VFS 索引节点操作	快速符号链接	普通符号链接
setxattr	generic_setxattr()	generic_setxattr()
getxattr	generic_getxattr()	generic_getxattr()
listxattr	ext2_listxattr()	ext2_listxattr()
removexattr	generic_removexattr()	generic_removexattr()

如果索引节点指的是一个字符设备文件、块设备文件或命名管道 (参见第十九章中的“FIFO”一节), 那么这种索引节点的操作不依赖于文件系统, 其操作分别位于 `chrdev_inode_operations`、`blkdev_inode_operations` 和 `fifo_inode_operations` 表中。

Ext2 的文件操作

表 18-10 列出了 Ext2 文件系统特定的文件操作。正如你看到的, 一些 VFS 方法是由很多文件系统共用的通用函数实现的。这些方法的地址存放在 `ext2_file_operations` 表中。

表 18-10: Ext2 文件操作

VFS 文件操作	Ext2 方法
llseek	generic_file_llseek()
read	generic_file_read()
write	generic_file_write()
aio_read	generic_file_aio_read()
aio_write	generic_file_aio_write()
ioctl	ext2_ioctl()
mmap	generic_file_mmap()
open	generic_file_open()
release	ext2_release_file()
fsync	ext2_sync_file()
readv	generic_file_readv()
writv	generic_file_writev()
sendfile	generic_file_sendfile()

注意，Ext2 的 read 和 write 方法是分别通过 generic_file_read() 和 generic_file_write() 函数实现的。这两个函数在第十五章的“从文件中读取数据”和“写入文件”两节进行了描述。

管理 Ext2 磁盘空间

文件在磁盘的存储不同于程序员所看到的文件，这表现在两个方面：块可以分散在磁盘上（尽管文件系统尽力保持块连续存放以提高访问速度），以及程序员看到的文件似乎比实际的文件大，这是因为程序可以把洞引入文件（通过 lseek() 系统调用）。

在本节，我们将介绍 Ext2 文件系统如何管理磁盘空间，也就是说，如何分配和释放索引节点和数据块。有两个主要的问题必须考虑：

- 空间管理必须尽力避免文件碎片，也就是说，避免文件在物理上存放于几个小的、不相邻的盘块上。文件碎片增加了对文件的连续读操作的平均时间，因为在读操作期间，磁头必须频繁地重新定位（注 4）。这个问题类似于在第八章的“伙伴系统算法”一节中所讨论的 RAM 的外部碎片问题。
- 空间管理必须考虑效率，也就是说，内核应该能从文件的偏移量快速地导出 Ext2 分区上相应的逻辑块号。为了达到此目的，内核应该尽可能地限制对磁盘上寻址表的访问次数，因为对该表的访问会极大地增加文件的平均访问时间。

创建索引节点

ext2_new_inode() 函数创建 Ext2 磁盘的索引节点，返回相应的索引节点对象的地址（或失败时为 NULL）。该函数谨慎地选择存放该新索引节点的块组；它将无联系的目录散放在不同的组，而且同时把文件存放在父目录的同一组。为了平衡普通文件数与块组中的目录数，Ext2 为每一个块组引入“债（debt）”参数。

该函数作用于两个参数：dir，一个目录对应的索引节点对象的地址，新创建的索引节点必须插入到这个目录中；mode，要创建的索引节点的类型。后一个参数还包含一个 MS_SYNCHRONOUS 标志，该标志请求当前进程一直挂起，直到索引节点被分配。该函数执行如下操作：

1. 调用 new_inode() 分配一个新的 VFS 索引节点对象，并把它的 i_sb 字段初始化为存放在 dir->i_sb 中的超级块地址。然后把它追加到正在用的索引节点链表与超级块链表中（参见第十二章“索引节点对象”一节）。

注 4： 请注意，把一个文件跨过块组进行分片是一件坏事，而为了在一个块中存放多个文件把块进行分片（还没实现）是一件好事，这二者之间是不同的。

2. 如果新的索引节点是一个目录,函数就调用`find_group_orlov()`为目录找到一个合适的块组(注5)。该函数执行如下试探法:
 - a. 以文件系统根`root`为父目录的目录应该分散在各个组。这样,函数在这些块组去查找一个组,它的空闲索引节点数和空闲块数比平均值高。如果没有这样的组则跳到第2c步。
 - b. 如果满足下列条件,嵌套目录(父目录不是文件系统根`root`)就应被存放到父目录组:
 - 该组没有包含太多的目录
 - 该组有足够多的空闲索引节点
 - 该组有一点小“债”。(块组的债存放在一个`ext2_sb_info`描述符的`s_debts`字段所指向的计数器数组中。每当一个新目录加入,债加一;每当其他类型的文件加入,债减一)如果父目录组不满足这些条件,那么选择第一个满足条件的组。如果没有满足条件的组,则跳到第2c步。
 - c. 这是一个“退一步”原则,当找不到合适的组时使用。函数从包含父目录的块组开始选择第一个满足条件的块组,这个条件是:它的空闲索引节点数比每块组空闲索引节点数的平均值大。
3. 如果新索引节点不是个目录,则调用`find_group_other()`,在有空闲索引节点的块组中给它分配一个。该函数从包含父目录的组开始往下找。具体如下:
 - a. 从包含父目录`dir`的块组开始,执行快速的对数查找。这种算法要查找 $\log(n)$ 个块组,这里 n 是块组总数。该算法一直向前查找直到找到一个可用的块组,具体如下:如果我们把开始的块组称为 i ,那么,该算法要查找的块组为 $i \bmod (n)$, $i+1 \bmod (n)$, $i+1+2 \bmod (n)$, $i+1+2+4 \bmod (n)$, 等等。
 - b. 如果该算法没有找到含有空闲索引节点的块组,就从包含父目录`dir`的块组开始执行彻底的线性查找。
4. 调用`read_inode_bitmap()`得到所选块组的索引节点位图,并从中寻找第一个空位,这样就得到了第一个空闲磁盘索引节点号。
5. 分配磁盘索引节点:把索引节点位图中的相应位置位,并把含有这个位图的缓冲区标记为脏。此外,如果文件系统安装时指定了`MS_SYNCHRONOUS`标志(参见第十二章中的“安装普通文件系统”一节),则调用`sync_dirty_buffer()`开始I/O写操作并等待,直到写操作终止。

注5: 安装Ext2文件系统还可以带有一个选项标志,它强制内核使用一个更简单、更老式的分配策略,该策略是由`find_group_dir()`函数实现的。

6. 减小组描述符的 `bg_free_inodes_count` 字段。如果新的索引节点是一个目录，则增加 `bg_used_dirs_count` 字段，并把含有这个组描述符的缓冲区标记为脏。
7. 依据索引节点指向的是普通文件或目录，相应增减超级块内 `s_debts` 数组中的组计数器。
8. 减小 `ext2_sb_info` 数据结构中的 `s_freeinodes_counter` 字段；而且如果新索引节点是目录，则增大 `ext2_sb_info` 数据结构的 `s_dirs_counter` 字段。
9. 将超级块的 `s_dirt` 标志置 1，并把包含它的缓冲区标记为脏。
10. 把 VFS 超级块对象的 `s_dirt` 字段置 1。
11. 初始化这个索引节点对象的字段。特别是，设置索引节点号 `i_no`，并把 `xtime.tv_sec` 的值拷贝到 `i_atime`、`i_mtime` 及 `i_ctime`。把这个块组的索引赋给 `ext2_inode_info` 结构的 `i_block_group` 字段。关于这些字段的含义请参考表 18-3。
12. 初始化这个索引节点对象的访问控制列表（ACL）。
13. 将新索引节点对象插入散列表 `inode_hashtable`，调用 `mark_inode_dirty()` 把该索引节点对象移进超级块脏索引节点链表（参见第十二章“索引节点对象”一节）。
14. 调用 `ext2_preread_inode()` 从磁盘读入包含该索引节点的块，将它存入页高速缓存。进行这种预读是因为最近创建的索引节点可能会被很快写入。
15. 返回新索引节点对象的地址。

删除索引节点

用 `ext2_free_inode()` 函数删除一个磁盘索引节点，把磁盘索引节点表示为索引节点对象，其地址作为参数来传递。内核在进行一系列的清除操作（包括清除内部数据结构和文件中的数据）之后调用这个函数。具体来说，它在下列操作完成之后才执行：索引节点对象已经从散列表中删除，指向这个索引节点的最后一个硬链接已经从适当的目录中删除，文件的长度截为 0 以回收它的所有数据块（参见本章后面“释放数据块”一节）。函数执行下列操作：

1. 调用 `clear_inode()`，它依次执行如下步骤：
 - a. 删除与索引节点关联的“间接”脏缓冲区（参见后面“数据块寻址”一节）。它们都存放在一个链表中，该链表的首部在 `address_space` 对象 `inode->i_data` 的 `private_list` 字段（参见第十五章“`address_space` 对象”一节）。
 - b. 如果索引节点的 `I_LOCK` 标志置位，则说明索引节点中的某些缓冲区正处于 I/O 数据传送中；于是，函数挂起当前进程，直到这些 I/O 数据传送结束。

- c. 调用超级块对象的 `clear_inode` 方法（如果已定义），但 Ext2 文件系统没有定义这个方法。
 - d. 如果索引节点指向一个设备文件，则从设备的索引节点链表中删除索引节点对象，这个链表要么在 `cdev` 字符设备描述符的 `cdev` 字段（参见第十三章“字符设备驱动程序”一节），要么在 `block_device` 块设备描述符的 `bd_inodes` 字段（参见第十四章“块设备”一节）。
 - e. 把索引节点的状态置为 `I_CLEAR`（索引节点对象的内容不再有意义）。
2. 从每个块组的索引节点号和索引节点数计算包含这个磁盘索引节点的块组的索引。
 3. 调用 `read_inode_bitmap()` 得到索引节点位图。
 4. 增加组描述符的 `bg_free_inodes_count` 字段。如果删除的索引节点是一个目录，那么也要减小 `bg_used_dirs_count` 字段。把这个组描述符所在的缓冲区标记为脏。
 5. 如果删除的索引节点是一个目录，就减小 `ext2_sb_info` 结构的 `s_dirs_counter` 字段，把超级块的 `s_dirt` 标志置 1，并把它所在的缓冲区标记为脏。
 6. 清除索引节点位图中这个磁盘索引节点对应的位，并把包含这个位图的缓冲区标记为脏。此外，如果文件系统以 `MS_SYNCHRONIZE` 标志安装，则调用 `sync_dirty_buffer()` 并等待，直到在位图缓冲区上的写操作终止。

数据块寻址

每个非空的普通文件都由一组数据块组成。这些块或者由文件内的相对位置（它们的文件块号）来标识，或者由磁盘分区内的位置（它们的逻辑块号）来标识（参见第十四章的“块设备的处理”一节）。

从文件内的偏移量 f 导出相应数据块的逻辑块号需要两个步骤：

1. 从偏移量 f 导出文件的块号，即在偏移量 f 处的字符所在的块索引。
2. 把文件的块号转化为相应的逻辑块号。

因为 Unix 文件不包含任何控制字符，因此，导出文件的第 f 个字符所在的文件块号是相当容易的，只是用 f 除以文件系统块的大小，并取整即可。

例如，让我们假定块的大小为 4KB。如果 f 小于 4096，那么这个字符就在文件的第一个数据块中，其文件的块号为 0。如果 f 等于或大于 4096 而小于 8192，则这个字符就在文件块号为 1 的数据块中，以此类推。

只用关注文件的块号确实不错。但是，由于 Ext2 文件的数据块在磁盘上不必是相邻的，因此把文件的块号转化为相应的逻辑块号可不是这么直截了当的。

因此，Ext2 文件系统必须提供一种方法，用这种方法可以在磁盘上建立每个文件块号与相应逻辑块号之间的关系。在索引节点内部部分实现了这种映射（回到了 AT&T Unix 的早期版本）。这种映射也涉及一些包含额外指针的专用块，这些块用来处理大型文件的索引节点的扩展。

磁盘索引节点的 `i_block` 字段是一个有 `EXT2_N_BLOCKS` 个元素且包含逻辑块号的数组。在下面的讨论中，我们假定 `EXT2_N_BLOCKS` 的默认值为 15。如图 18-5 所示，这个数组表示一个大型数据结构体的初始化部分。正如从图中所看到的，数组的 15 个元素有 4 种不同的类型：

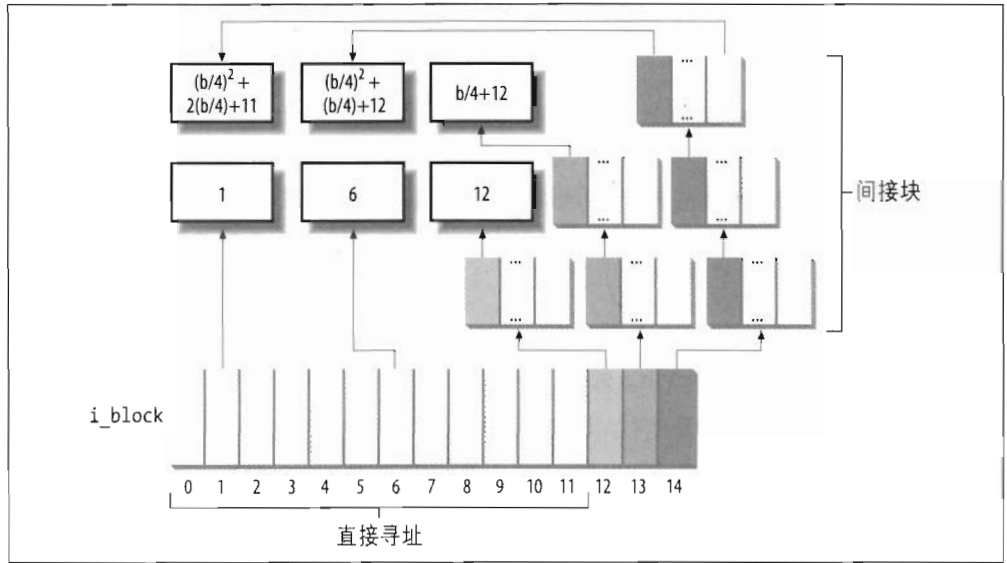


图 18-5：对文件的数据块进行寻址的数据结构

- 最初的 12 个元素产生的逻辑块号与文件最初的 12 个块对应，即对应的文件块号从 0~11。
- 下标 12 中的元素包含一个块的逻辑块号（叫做间接块），这个块表示逻辑块号的一个二级数组。这个数组的元素对应的文件块号从 12~ $b/4+11$ ，这里 b 是文件系统的块大小（每个逻辑块号占 4 个字节，因此我们在式子中用 4 作除数）。因此，内核为了查找指向一个块的指针必须先访问这个元素，然后，在这个块中找到另一个指向最终块（包含文件内容）的指针。

- 下标 13 中的元素包含一个间接块的逻辑块号，而这个块包含逻辑块号的一个二级数组，这个二级数组的数组项依次指向三级数组，这个三级数组存放的才是文件块号对应的逻辑块号，范围从 $b/4+12 \sim (b/4)^2+(b/4)+11$ 。
- 最后，下标 14 中的元素使用三级间接索引，第四级数组中存放的才是文件块号对应的逻辑块号，范围从 $(b/4)^2+(b/4)+12 \sim (b/4)^3+(b/4)^2+(b/4)+11$ 。

在图 18-5 中，块内的数字表示相应的文件块号。箭头（表示存放在数组元素中的逻辑块号）指示了内核如何通过间接块找到包含文件实际内容的块。

注意这种机制是如何支持小文件的。如果文件需要的数据块小于 12，那么两次磁盘访问就可以检索到任何数据：一次是读磁盘索引节点 `i_block` 数组的一个元素，另一次是读所需要的数据块。对于大文件来说，可能需要三四次的磁盘访问才能找到需要的块。实际上，这是一种最坏的估计，因为目录项、索引节点、页高速缓存都有助于极大地减少实际访问磁盘的次数。

还要注意文件系统的块大小是如何影响寻址机制的，因为大的块允许 Ext2 把更多的逻辑块号存放在一个单独的块中。表 18-11 显示了对每种块大小和每种寻址方式所存放文件大小的上限。例如，如果块的大小是 1024 字节，并且文件包含的数据最多为 268KB，那么，通过直接映射可以访问文件最初的 12KB 数据，通过简单的间接映射可以访问剩余的 13~268KB 的数据。大于 2GB 的大型文件通过指定 `O_LARGEFILE` 打开标志必须在 32 位体系结构上进行打开。

表 18-11：数据块寻址的文件大小上界

块大小	直接	一次间接	二次间接	三次间接
1024	12 KB	268 KB	64.26 MB	16.06 GB
2048	24 KB	1.02 MB	513.02 MB	256.5 GB
4096	48 KB	4.04 MB	4 GB	~ 4 TB

文件的洞

文件的洞(*file hole*)是普通文件的一部分，它是一些空字符但没有存放在磁盘的任何数据块中。洞是 Unix 文件一直存在的一个特点。例如，下列的 Unix 命令创建了第一个字节是洞的文件。

```
$ echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

现在，`/tmp/hole` 有 6145 个字符（6144 个空字符加一个 X 字符），然而，这个文件在磁盘上只占一个数据块。

引入文件的洞是为了避免磁盘空间的浪费。它们被广泛地用在数据库应用中，更一般地说，用于在文件上进行散列的所有应用。

文件洞在 Ext2 中的实现是基于动态数据块的分配的：只有当进程需要向一个块写数据时，才真正把这个块分配给文件。每个索引节点的 `i_size` 字段定义程序所看到的文件大小，包括洞，而 `i_blocks` 字段存放分配给文件有效的数据块数（以 512 字节为单位）。

在前面 `dd` 命令的例子中，假定 `/tmp/hole` 文件创建在块大小为 4096 的 Ext2 分区上。其相应磁盘索引节点的 `i_size` 字段存放的数为 6145，而 `i_blocks` 字段存放的数为 8（因为每 4096 字节的块包含 8 个 512 字节的块）。`i_block` 数组的第二个元素（对应块的文件块号为 1）存放已分配块的逻辑块号，而数组中的其他元素都为空（参看图 18-6）。

分配数据块

当内核要分配一个数据块来保存 Ext2 普通文件的数据时，就调用 `ext2_get_block()` 函数。如果块不存在，该函数就自动为文件分配块。请记住，每当内核在 Ext2 普通文件上执行读或写操作时就调用这个函数（参见第十六章“从文件中读取数据”和“写入文件”两节）；显然，这个函数只在页高速缓存内没有相应的块时才被调用。

`ext2_get_block()` 函数处理在“数据块寻址”一节描述的数据结构，并在必要时调用 `ext2_alloc_block()` 函数在 Ext2 分区真正搜索一个空闲块。如果需要，该函数还为间接寻址分配相应的块（参见图 18-5）。

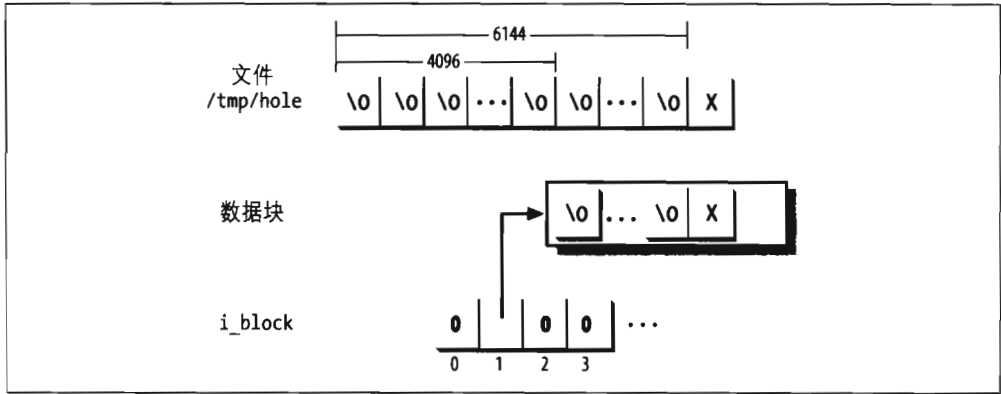


图 18-6：起始部分有洞的文件

为了减少文件的碎片，Ext2 文件系统尽力在已分配给文件的最后一个块附近找一个新块

分配给该文件。如果失败, Ext2 文件系统又在包含这个文件索引节点的块组中搜寻一个新的块。作为最后一个办法, 可以从其他一个块组中获得空闲块。

Ext2 文件系统使用数据块的预分配策略。文件并不仅仅获得所需要的块, 而是获得一组多达 8 个邻接的块。ext2_inode_info 结构的 i_prealloc_count 字段存放预分配给某一文件但还没有使用的数据块数, 而 i_prealloc_block 字段存放下一次要使用的预分配块的逻辑块号。当下列情况发生时, 释放预分配而一直没有使用的块: 当文件被关闭时, 当文件被缩短时, 或者当一个写操作相对于引发块预分配的写操作不是顺序的时。

ext2_alloc_block() 函数接收的参数为指向索引节点对象的指针、目标 (goal) 和存放错误码的变量地址。目标是一个逻辑块号, 表示新块的首选位置。ext2_getblk() 函数根据下列的试探法设置目标参数:

1. 如果正被分配的块与前面已分配的块有连续的文件块号, 则目标就是前一块的逻辑块号加 1。这很有意义, 因为程序所看到的连续的块在磁盘上将会是相邻的。
2. 如果第一条规则不适用, 并且至少给文件已分配了一个块, 那么目标就是这些块的逻辑块号中的一个。更确切地说, 目标是已分配块的逻辑块号, 位于文件中待分配块之前。
3. 如果前面的规则都不适用, 那么目标就是文件索引节点所在的块组中第一个块的逻辑块号 (不必空闲)。

ext2_alloc_block() 函数检查目标是否指向文件的预分配块中的一块。如果是, 就分配相应的块并返回它的逻辑块号; 否则, 丢弃所有剩余的预分配块并调用 ext2_new_block()。

ext2_new_block() 函数用下列策略在 Ext2 分区内搜寻一个空闲块:

1. 如果传递给 ext2_alloc_block() 的首选块 (目标块) 是空闲的, 就分配它。
2. 如果目标为忙, 就检查首选块后的其余块之中是否有空闲的块。
3. 如果在首选块附近没有找到空闲块, 就从包含目标的块组开始, 查找所有的块组。对每个块组:
 - a. 寻找至少有 8 个相邻空闲块的一个组块。
 - b. 如果没有找到这样的一组块, 就寻找一个单独的空闲块。

只要找到一个空闲块, 搜索就结束。在结束前, ext2_new_block() 函数还尽力在找到的空闲块附近的块中找 8 个空闲块进行预分配, 并把磁盘索引节点的 i_prealloc_block 和 i_prealloc_count 字段置为适当的块位置及块数。

释放数据块

当进程删除一个文件或把它的长度截为 0 时，其所有数据块必须回收。这是通过调用 `ext2_truncate()` 函数（其参数是这个文件的索引节点对象的地址）来完成的。实际上，这个函数扫描磁盘索引节点的 `i_block` 数组，以确定所有数据块的位置和间接寻址用的块的位置。然后反复调用 `ext2_free_blocks()` 函数释放这些块。

`ext2_free_blocks()` 函数释放一组含有一个或多个相邻块的数据块。除 `ext2_truncate()` 调用它外，当丢弃文件的预分配块时也主要调用它（参见前面的“分配数据块”一节）。函数参数如下：

`inode`

文件的索引节点对象的地址。

`block`

要释放的第一个块的逻辑块号。

`count`

要释放的相邻块数。

这个函数对每个要释放的块执行下列操作：

1. 获得要释放块所在块组的块位图。
2. 把块位图中要释放的块的对应位清 0，并把位图所在的缓冲区标记为脏。
3. 增加块组描述符的 `bg_free_blocks_count` 字段，并把相应的缓冲区标记为脏。
4. 增加磁盘超级块的 `s_free_blocks_count` 字段，并把相应的缓冲区标记为脏，把超级块对象的 `s_dirt` 标记置位。
5. 如果 Ext2 文件系统安装时设置了 `MS_SYNCHRONOUS` 标志，则调用 `sync_dirty_buffer()` 并等待，直到对这个位图缓冲区的写操作终止。

Ext3 文件系统

在本节我们将简单描述从 Ext2 发展而来的增强型文件系统，即 Ext3。这个新的文件系统在设计时曾秉持两个简单的概念：

- 成为一个日志文件系统（参见下一节）
- 尽可能与原来的 Ext2 文件系统兼容

Ext3 完全达到了这两个目标。尤其是，它很大程度上是基于 Ext2 的，因此，它在磁盘上的数据结构从本质上与 Ext2 文件系统的结构是相同的。事实上，如果 Ext3 文件系统已经被彻底卸载，那么就可以把它作为 Ext2 文件系统来重新安装；反之，创建 Ext2 文件系统的日志并把它作为 Ext3 文件系统来重新安装，也是一种简单、快速的操作。

由于 Ext3 与 Ext2 之间的兼容性，本章前面几节的很多描述也适用于 Ext3。因此，本节我们集中于 Ext3 所提供的新特点——“日志”。

日志文件系统

随着磁盘变得越来越大，传统 Unix 文件系统（像 Ext2）的一种设计选择证明是不相称的。从第十四章我们已经知道，对文件系统块的更新可能在内存保留相当长的时间后才刷新到磁盘。因此，像断电故障或系统崩溃这样不可预测的事件可能导致文件系统处于不一致状态。为了克服这个问题，每个传统的 Unix 文件系统在安装之前都要进行检查，如果它没有被正常卸载，那么，就有一个特定的程序执行彻底、耗时的检查，并修正磁盘上文件系统的所有数据结构。

例如，Ext2 文件系统的状态存放在磁盘上超级块的 `s_mount_state` 字段中。由启动脚本调用 `e2fsck` 实用程序检查存放在这个字段中的值；如果它不等于 `EXT2_VALID_FS`，说明文件系统没有正常卸载，因此，`e2fsck` 开始检查文件系统的所有磁盘数据结构。

显然，检查文件系统一致性所花费的时间主要取决于要检查的文件数和目录数；因此，它也取决于磁盘的大小。如今，随着文件系统达到几百个 GB，一次一致性检查就可能花费数个小时。造成的停机时间对任何生产环境和高可用服务器都是无法接受的。

日志文件系统的目标就是避免对整个文件系统进行耗时的一致性检查，这是通过查看一个特殊的磁盘区达到的，因为这种磁盘区包含所谓日志（*journal*）的最新磁盘写操作。系统出现故障后，安装日志文件系统只不过是几秒钟的事。

Ext3 日志文件系统

Ext3 日志所隐含的思想就是对文件系统进行的任何高级修改都分两步进行。首先，把待写块的一个副本存放在日志中；其次，当发往日志的 I/O 数据传送完成时（简而言之，把数据提交到日志），块就被写入文件系统。当发往文件系统的 I/O 数据传送终止时（把数据提交给文件系统），日志中的块副本就被丢弃。

当从系统故障中恢复时，`e2fsck` 程序区分下列两种情况：

提交到日志之前系统故障发生。与高级修改相关的块副本或者从日志中丢失，或者是不完整的；在这两种情况下，*e2fsck* 都忽略它们。

提交到日志之后系统故障发生。块的副本是有效的，且 *e2fsck* 把它们写入文件系统。

在第一种情况下，对文件系统的高级修改被丢失，但文件系统的状态还是一致的。在第二种情况下，*e2fsck* 应用于整个高级修改，因此，修正由于把未完成的 I/O 数据传送到文件系统而造成的任何不一致。

不要对日志文件系统有太多的期望。它只能确保系统调用级的一致性。例如，当你正在发出几个 `write()` 系统调用拷贝一个大型文件时发生了系统故障，这将会使拷贝操作中断，因此，复制的文件就会比原来的文件短。

因此，日志文件系统通常不把所有的块都拷贝到日志中。事实上，每个文件系统都由两种块组成：包含所谓元数据 (*metadata*) 的块和包含普通数据的块。在 Ext2 和 Ext3 的情形中，有六种元数据：超级块、块组描述符、索引节点、用于间接寻址的块（间接块）、数据位图块和索引节点位图块。其他的文件系统可能使用不同的元数据。

很多日志文件系统（如 SGI 的 XFS 以及 IBM 的 JFS）都限定自己把影响元数据的操作记入日志。事实上，元数据的日志记录足以恢复磁盘文件系统数据结构的一致性。然而，因为文件的数据块不记入日志，因此就无法防止系统故障造成的文件内容的损坏。

不过，可以把 Ext3 文件系统配置为把影响文件系统元数据的操作和影响文件数据块的操作都记入日志。因为把每种写操作都记入日志会导致极大的性能损失，因此，Ext3 让系统管理员决定应当把什么记入日志；具体来说，它提供三种不同的日志模式：

日志 (*Journal*)

文件系统所有数据和元数据的改变都被记入日志。这种模式减少了丢失每个文件修改的机会，但是它需要很多额外的磁盘访问。例如，当一个新文件被创建时，它的所有数据块都必须复制一份作为日志记录。这是最安全和最慢的 Ext3 日志模式。

预定 (*Ordered*)

只有对文件系统元数据的改变才被记入日志。然而，Ext3 文件系统把元数据和相关的数据块进行分组，以便在元数据之前把数据块写入磁盘。这样，就可以减少文件内数据损坏的机会；例如，确保增大文件的任何写访问都完全受日志的保护。这是缺省的 Ext3 日志模式。

写回 (*Writeback*)

只有对文件系统元数据的改变才被记入日志，这是在其他日志文件系统中发现的方法，也是最快的模式。

Ext3 文件系统的日志模式由 *mount* 系统命令的一个选项来指定。例如，为了在 */jdisk* 安装点对存放在 */dev/sda2* 分区上的 Ext3 文件系统以“写回”模式进行安装，系统管理员可以键入如下命令：

```
# mount -t ext3 -o data=writeback /dev/sda2 /jdisk
```

日志块设备层

Ext3 日志通常存放在名为 *.journal* 的隐藏文件中，该文件位于文件系统的根目录。

Ext3 文件系统本身不处理日志，而是利用所谓日志块设备 (*Journaling Block Device*, *JBD*) 的通用内核层。现在，只有 Ext3 使用 JBD 层，而其他文件系统可能在将来才使用它。

JBD 层是相当复杂的软件部分。Ext3 文件系统调用 JBD 例程，以确保在系统万一出现故障时它的后续操作不会损坏磁盘数据结构。然而，JBD 典型地使用同一磁盘来把 Ext3 文件系统所做的改变记入日志，因此，它与 Ext3 一样易受系统故障的影响。换言之，JBD 也必须保护自己免受任何系统故障引起的日志损坏。

因此，Ext3 与 JBD 之间的交互本质上基于三个基本单元：

日志记录

描述日志文件系统一个磁盘块的一次更新。

原子操作处理

包括文件系统的一次高级修改对应的日志记录；一般来说，修改文件系统的每个系统调用都引起一次单独的原子操作处理。

事务

包括几个原子操作处理，同时，原子操作处理的日志记录对 *e2fsck* 标记为有效。

日志记录

日志记录 (*log record*) 本质上是文件系统将要发出的一个低级操作的描述。在某些日志文件系统中，日志记录只包括操作所修改的字节范围及字节在文件系统起始位置。然而，JBD 层使用的日志记录由低级操作所修改的整个缓冲区组成。这种方式可能浪费很多日志空间（例如，当低级操作仅仅改变位图的一个位时），但是，它还是相当快的，因为 JBD 层直接对缓冲区和缓冲区首部进行操作。

因此，日志记录在日志内部表示为普通的数据块（或元数据）。但是，每个这样的块都是与类型为 *journal_block_tag_t* 的小标签相关联的，这种小标签存放块在文件系统中的逻辑块号和几个状态标志。

随后, 只要一个缓冲区得到 JBD 的关注, 或者因为它属于日志记录, 或者因为它是一个数据块, 该数据块应当在相应的元数据之前刷新到磁盘 (处于 “预定” 模式), 那么, 内核把 `journal_head` 数据结构加入到缓冲区首部。在这种情况下, 缓冲区首部的 `b_private` 字段存放 `journal_head` 数据结构的地址, 并把 `BH_JBD` 标志置位 (参见第十五章 “块缓冲区和缓冲区首部” 一节)。

原子操作处理

修改文件系统的任一系统调用通常都被划分为操纵磁盘数据结构的一系列低级操作。

例如, 假定 Ext3 必须满足用户把一个数据块追加到普通文件的请求。文件系统层必须确定文件的最后一个块, 定位文件系统中的空闲块, 更新适当块组内的数据块位图, 存放新块的逻辑块号在文件的索引节点或间接寻址块中, 写新块的内容, 并在最后更新索引节点的几个字段。你可以看到, 追加操作转换为对文件系统数据块和元数据块很多低级的操作。

现在, 仅仅想象一下, 如果在追加操作的中间一些低级操作已经执行, 另一些还没有执行, 而系统出现了故障会发生什么事情。当然, 对于影响两个或多个文件的高级操作 (例如, 把文件从一个目录移到另一个目录), 情况会更糟。

为了防止数据损坏, Ext3 文件系统必须确保每个系统调用以原子的方式进行处理。原子操作处理 (*atomic operation handle*) 是对磁盘数据结构的一组低级操作, 这组低级操作对应一个单独的高级操作。当从系统故障中恢复时, 文件系统确保要么整个高级操作起作用, 要么没有一个低级操作起作用。

任何原子操作处理都用类型为 `handle_t` 的描述符来表示。为了开始一个原子操作, Ext3 文件系统调用 `journal_start()` JBD 函数, 该函数在必要时分配一个新的原子操作处理并把它插入到当前的事务中 (见下一节)。因为对磁盘的任何低级操作都可能挂起进程, 因此, 活动原子操作处理的地址存放在进程描述符的 `journal_info` 字段中。为了通知原子操作已经完成, Ext3 文件系统调用 `journal_stop()` 函数。

事务

出于效率的原因, JBD 层对日志的处理采用分组的方法, 即把属于几个原子操作处理的日志记录分组放在一个单独的事务 (*transaction*) 中。此外, 与一个处理相关的所有日志记录都必须包含在同一个事务中。

一个事务的所有日志记录存放在日志的连续块中。JBD 层把每个事务作为整体来处理。

例如,只有当包含在一个事务的日志记录中的所有数据都提交给文件系统时才回收该事务所使用的块。

事务一旦被创建,它就能接受新处理的日志记录。当下列情况之一发生时,事务就停止接受新处理:

- 固定的时间已经过去,典型情况下为 5s。
- 日志中没有空闲块留给新处理

事务是由类型为 `transaction_t` 的描述符来表示的。其最重要的字段为 `t_state`,该字段描述事务的当前状态。

从本质上说,事务可以是:

完成的

包含在事务中的所有日志记录都已经从物理上写入日志。当从系统故障中恢复时, `e2fsck` 考虑日志中每个完成的事务,并把相应的块写入文件系统。在这种情况下, `t_state` 字段存放值 `T_FINISHED`。

未完成的

包含在事务中的日志记录至少还有一个没有从物理上写入日志,或者新的日志记录还正在追加到事务中。在系统故障的情况下,存放在日志中的事务映像很可能不是最新的。因此,当从系统故障中恢复时, `e2fsck` 不信任日志中未完成的事务,并跳过它们。在这种情况下, `i_state` 存放下列值之一:

`T_RUNNING`

还在接受新的原子操作处理。

`T_LOCKED`

不接受新的原子操作处理,但其中的一些还没有完成。

`T_FLUSH`

所有的原子操作处理都已完成,但一些日志记录还正在写入日志。

`T_COMMIT`

原子操作处理的所有日志记录都已经写入磁盘,但在日志中,事务仍然被标记为完成。

在任何时刻,日志可能包含多个事务,但其中只有一个处于 `T_RUNNING` 状态,即它是活动事务 (*active transaction*)。所谓活动事务就是正在接受由 Ext3 文件系统发出的新原子操作处理的请求。

日志中的几个事务可能是未完成的，因为包含相关日志记录的缓冲区还没有写入日志。

如果事务完成，说明所有日志记录已被写入日志，但是一部分相应的缓冲区还没有写入文件系统。只有当 JDB 层确认日志记录描述的所有缓冲区都已成功写入 Ext3 文件系统时，一个完成的事务才能从日志中删除。

日志如何工作

让我们用一个例子来试图解释日志如何工作：Ext3 文件系统层接受向普通文件写一些数据块的请求。

你可能很容易猜到，我们打算详细描述 Ext3 文件系统层和 JDB 层的每个单独操作。那将会涉及太多问题！但是，我们描述本质的操作：

1. `write()` 系统调用服务例程触发与 Ext3 普通文件相关的文件对象的 `write` 方法。对于 Ext3 来说，这个方法是由 `generic_file_write()` 函数实现的，这已在第十六章“写入文件”一节进行了描述。
2. `generic_file_write()` 函数几次调用 `address_space` 对象的 `prepare_write` 方法，写方法涉及的每个数据页都调用一次。对 Ext3 来说，这个方法是由 `ext3_prepare_write()` 函数实现的。
3. `ext3_prepare_write()` 函数调用 `journal_start()` JBD 函数开始一个新的原子操作。这个原子操作处理被加到活动事务中。实际上，原子操作处理是在第一次调用 `journal_start()` 函数时创建的。后续的调用确认进程描述符的 `journal_info` 字段已经被置位，并使用这个处理。
4. `ext3_prepare_write()` 函数调用第十六章已描述过的 `block_prepare_write()` 函数，传递给它的参数为 `ext3_get_block()` 函数的地址。回想一下，`block_prepare_write()` 负责准备文件页的缓冲区和缓冲区首部。
5. 当内核必须确定 Ext3 文件系统的逻辑块号时，就执行 `ext3_get_block()` 函数。这个函数实际上类似于 `ext2_get_block()`，后者在前面“分配数据块”一节已经描述。但是，有一个主要的差异在于 Ext3 文件系统调用 JDB 层的函数来确保低级操作记入日志：
 - 在对 Ext3 文件系统的元数据块发出低级写操作之前，该函数调用 `journal_get_write_access()`。后一个函数主要把元数据缓冲区加入到活动事务的链表中。但是，它也必须检查元数据是否包含在日志的一个较老的未完成的事务中；在这种情况下，它把缓冲区复制一份以确保老的事务以老的内容提交。

- 在更新元数据块所在的缓冲区之后,Ext3文件系统调用`journal_dirty_metadata()`把元数据缓冲区移到活动事务的适当脏链表中,并在日志中记录这一操作。

注意,由JDB层处理的元数据缓冲区通常并不包含在索引节点的缓冲区的脏链表中,因此,这些缓冲区并不由第十五章描述的正常磁盘高速缓存的刷新机制写入磁盘。

6. 如果Ext3文件系统已经以“日志”模式安装,则`ext3_prepare_write()`函数在写操作触及的每个缓冲区上也调用`journal_get_write_access()`。
7. 控制权回到`generic_file_write()`函数,该函数用存放在用户态地址空间的数据更新页,并调用`address_space`对象的`commit_write`方法。对于Ext3,函数如何实现这个方法取决于Ext3文件系统的安装方式:
 - 如果Ext3文件系统已经以“日志”模式安装,那么`commit_write`方法是由`ext3_journalled_commit_write()`函数实现的,它对页中的每个数据(不是元数据)缓冲区调用`journal_dirty_metadata()`。这样,缓冲区就包含在活动事务的适当脏链表中,但不包含在拥有者索引节点的脏链表中;此外,相应的日志记录写入日志。最后,`ext3_journalled_commit_write()`调用`journal_stop`通知JBD层原子操作处理已关闭。
 - 如果Ext3文件系统已经以“预定”模式安装,那么`commit_write`方法是由`ext3_ordered_commit_write()`函数实现的,它对页中的每个数据缓冲区调用`journal_dirty_data()`函数以把缓冲区插入到活动事务的适当链表中。JBD层确保在事务中的元数据缓冲区写入之前这个链表中的所有缓冲区写入磁盘。没有日志记录写入日志。然后,`ext3_ordered_commit_write()`函数执行第十五章描述的常规`generic_commit_write()`函数,该函数把数据缓冲区插入拥有者索引节点的脏缓冲区链表中。最后,`ext3_ordered_commit_write()`调用`journal_stop()`通知JBD层原子操作处理已关闭。
 - 如果Ext3文件系统已经以“写回”模式安装,那么`commit_write`方法是由`ext3_writeback_commit_write()`函数实现的,它执行第十五章描述的常规`generic_commit_write()`函数,该函数把数据缓冲区插入拥有者索引节点的脏缓冲区链表中。然后,`ext3_writeback_commit_write()`调用`journal_stop()`通知JBD层原子操作处理已关闭。
8. `write()`系统调用的服务例程到此结束。但是,JBD层还没有完成它的工作。终于,当事务的所有日志记录都物理地写入日志时,我们的事务才完成。然后,执行`journal_commit_transaction()`。
9. 如果Ext3文件系统已经以“预定”模式安装,则`journal_commit_transaction()`函数为事务链表包含的所有数据缓冲区激活I/O数据传送,并等待直到数据传送终止。

10. `journal_commit_transaction()` 函数为包含在事务中的所有元数据缓冲区激活 I/O 数据传送（如果 Ext3 以“日志”模式安装，则也为所有的数据缓冲区激活 I/O 数据传送）。
11. 内核周期性地为日志中每个完成的事务激活检查点活动。检查点主要验证由 `journal_commit_transaction()` 触发的 I/O 数据传送是否已经成功结束。如果是，则从日志中删除事务。

当然，除非发生系统故障，否则日志中的日志记录根本没有什么积极作用。事实上，只有在系统发生故障时，`e2fsck` 实用程序才扫描存放在文件系统中的日志，并重新安排完成的事务中的日志记录所描述的所有写操作。

第十九章

进程通信



本章介绍用户态的进程之间如何进行同步和交换数据。在第五章我们已经介绍了很多同步的主题，但是参与的对象是内核控制路径，而不是用户态程序。我们在详细讨论了I/O管理和文件系统之后，就可以继续讨论用户态进程的同步。这些进程都要依靠内核来实现彼此之间的同步以及通信。

正如我们在第十二章“Linux 文件加锁”一节已经看到的那样，通过创建一个文件（可能是空文件）并使用适当的VFS系统调用对该文件加锁和解锁就可以在用户态进程之间实现某种同步。通过把数据存放在使用锁保护的临时文件中就可以在进程之间实现类似的数据共享，然而这种方法的代价很高，因为它需要访问磁盘文件系统。出于这个原因，所有的Unix内核都包含一组系统调用，这些系统调用不用与文件系统打交道就可以支持进程通信；而且，已经开发了几个封装函数并将其加入到适当的库来加速进程对内核发出同步请求。

通常，应用程序员有使用不同通信机制的各种需求。这里列出Unix系统提供的进程间通信的基本机制：

管道和FIFO（命名管道）

最适合在进程之间实现生产者/消费者的交互。有些进程向管道中写入数据，而另外一些进程则从管道中读出数据。这将在“管道”与“FIFO”一节讨论。

信号量

顾名思义，这是在第五章中的“信号量”一节讨论过的内核信号量的用户态版本。这将在“System V IPC”一节讨论。

消息

允许进程在预定义的消息队列中读和写消息来交换消息（小块数据）。Linux 内核提供两种不同的消息版本：System V IPC 消息（参见“System V IPC”一节）和 POSIX 消息一节（参见“POSIX 消息队列”一节）。

共享内存区

允许进程通过共享内存块来交换信息。在必须共享大量数据的应用中，这可能是最高效的进程通信形式。这将在“System V IPC”一节讨论。

套接字

允许不同计算机上的进程通过网络交换数据。套接字还可以用作相同主机上的进程之间的通信工具；例如，X Window 系统图形接口就是使用套接字来允许客户端和 X 服务器交换数据的。

管道

管道 (*pipe*) 是所有 Unix 都愿意提供的一种进程间通信机制。管道是进程之间的一个单向数据流：一个进程写入管道的所有数据都由内核定向到另一个进程，另一个进程由此就可以从管道中读取数据。

在 Unix 的命令 shell 中，可以使用“|”操作符来创建管道。例如，下面的语句通知 shell 创建两个进程，并使用一个管道把这两个进程连接在一起：

```
$ ls | more
```

① 第一个进程（执行 *ls* 程序）的标准输出被重定向到管道中；② 第二个进程（执行 *more* 程序）从这个管道中读取输入。

注意，执行下面这两条命令也可以得到相同的结果：

```
$ ls > temp  
$ more < temp
```

第一个命令把 *ls* 的输出重定向到一个普通文件中；接下来，第二个命令强制 *more* 从这个普通文件中读取输入。当然，通常使用管道比使用临时文件更方便，这是因为：

- shell 语句比较短，也比较简单。
- 没有必要创建将来还必须删除的临时普通文件。

使用管道

管道被看作是打开的文件，但在已安装的文件系统中没有相应的映像。可以使用 `pipe()` 系统调用来创建一个新管道，这个系统调用返回一对文件描述符，然后进程通过 `fork()` 把这两个描述符传递给它的子进程，由此与子进程共享管道。进程可以在 `read()` 系统调用中使用第一个文件描述符从管道中读取数据，同样也可以在 `write()` 系统调用中使用第二个文件描述符向管道中写入数据。

POSIX 只定义了半双工的管道，因此即使 `pipe()` 系统调用返回了两个描述符，每个进程在使用一个文件描述符之前仍得把另外一个文件描述符关闭。如果所需要的是双向数据流，那么进程必须通过两次调用 `pipe()` 来使用两个不同的管道。

有些 Unix 系统，例如 System V Release 4，实现了全双工的管道。在全双工管道中，允许两个文件描述符既可以被写入也可以被读取，这就有两个双向信息通道。Linux 采用了另外一种解决方法：每个管道的文件描述符仍然都是单向的，但是在使用一个描述符之前不必把另外一个描述符关闭。

让我们回顾一下前面的那个例子。当 shell 命令对 `ls | more` 语句进行解释时，实际上要执行以下操作：

1. 调用 `pipe()` 系统调用；让我们假设 `pipe()` 返回文件描述符 3（管道的读通道）和 4（管道的写通道）。
2. 两次调用 `fork()` 系统调用。
3. 两次调用 `close()` 系统调用来释放文件描述符 3 和 4。

第一个子进程必须执行 `ls` 程序，它执行以下操作：

1. 调用 `dup2(4, 1)` 把文件描述符 4 拷贝到文件描述符 1。从现在开始，文件描述符 1 就代表该管道的写通道。
2. 两次调用 `close()` 系统调用来释放文件描述符 3 和 4。
3. 调用 `execve()` 系统调用来执行 `ls` 程序（参见第二十章“`exec` 函数”一节）。缺省情况下，这个程序要把自己的输出写到文件描述符为 1 的那个文件（标准输出）中，也就是说，写入管道中。

第二个子进程必须执行 `more` 程序；因此，该进程执行以下操作：

1. 调用 `dup2(3, 0)` 把文件描述符 3 拷贝到文件描述符 0。从现在开始，文件描述符 0 就代表管道的读通道。

2. 两次调用 `close()` 系统调用来释放文件描述符 3 和 4。
3. 调用 `execve()` 系统调用来执行 *more* 程序。缺省情况下, 这个程序要从文件描述符为 0 的那个文件 (标准输入) 中读取输入, 也就是说, 从管道中读取输入。

在这个简单的例子中, 管道完全被两个进程使用。但是, 由于管道的这种实现方式, 一个管道可以供任意个进程使用 (注 1)。显然, 如果两个或者更多个进程对同一个管道进行读写, 那么这些进程必须使用文件加锁机制 (参见第十二章中的“Linux 文件加锁”一节) 或者 IPC 信号量机制 (参见本章后面的“IPC 信号量”一节) 对自己的访问进行显式的同步。

除了 `pipe()` 系统调用之外, 很多 Unix 系统都提供了两个名为 `popen()` 和 `pclose()` 的封装函数来处理在使用管道的过程中产生的所有脏工作。只要使用 `popen()` 函数创建一个管道, 就可以使用包含在 C 函数库中的高级 I/O 函数 (`fprintf()`, `fscanf()` 等等) 对这个管道进行操作。

在 Linux 中, `popen()` 和 `pclose()` 都包含在 C 函数库中。`popen()` 函数接收两个参数: 可执行文件的路径名 `filename` 和定义数据传输方向的字符串 `type`。该函数返回一个指向 `FILE` 数据结构的指针。`popen()` 函数实际上执行以下操作:

1. 使用 `pipe()` 系统调用创建一个新管道。
2. 创建一个新进程, 该进程又执行以下操作:
 - a. 如果 `type` 是 `r`, 就把与管道的写通道相关的文件描述符拷贝到文件描述符 1 (标准输出); 否则, 如果 `type` 是 `w`, 就把与管道的读通道相关的文件描述符拷贝到文件描述符 0 (标准输入)。
 - b. 关闭 `pipe()` 返回的文件描述符。
 - c. 调用 `execve()` 系统调用执行 `filename` 所指定的程序。
3. 如果 `type` 是 `r`, 就关闭与管道的写通道相关的文件描述符; 否则, 如果 `type` 是 `w`, 就关闭与管道的读通道相关的文件描述符。
4. 返回 `FILE` 文件指针所指向的地址, 这个指针指向仍然打开的管道所涉及的任一文件描述符。

注 1: 由于大部分 shell 都提供只连接两个进程的管道, 所以应用程序要通过管道连接多于两个的进程就必须使用诸如 C 之类的编程语言自行编写。

在 `popen()` 函数被调用之后，父进程和子进程就可以通过管道交换信息：父进程可以使用该函数所返回的 `FILE` 指针来读（如果 `type` 是 `r`）写（如果 `type` 是 `w`）数据。子进程所执行的程序分别把数据写入标准输出或从标准输入中读取数据。

`pclose()` 函数接收 `popen()` 所返回的文件指针作为参数，它会简单地调用 `wait4()` 系统调用并等待 `popen()` 所创建的进程结束。

管道数据结构

我们现在又一次在系统调用的层次考虑问题。只要管道一被创建，进程就可以使用 `read()` 和 `write()` 这两个 VFS 系统调用来访问管道。因此，对于每个管道来说，内核都要创建一个索引节点对象和两个文件对象，一个文件对象用于读，另外一个对象用于写。当进程希望从管道中读取数据或向管道中写入数据时，必须使用适当的文件描述符。

当索引节点指的是管道时，其 `i_pipe` 字段指向一个如表 19-1 所示的 `pipe_inode_info` 结构。

表 19-1: `pipe_inode_info` 结构

类型	字段	说明
<code>struct wait_queue *</code>	<code>wait</code>	管道 /FIFO 等待队列
<code>unsigned int</code>	<code>nrbufs</code>	包含待读数据的缓冲区数
<code>unsigned int</code>	<code>curbuf</code>	包含待读数据的第一个缓冲区的索引
<code>struct pipe_buffer [16]</code>	<code>bufs</code>	管道缓冲区描述符数组
<code>struct page *</code>	<code>tmp_page</code>	高速缓存页框指针
<code>unsigned int</code>	<code>start</code>	当前管道缓冲区读的位置
<code>unsigned int</code>	<code>readers</code>	读进程的标志（或编号）
<code>unsigned int</code>	<code>writers</code>	写进程的标志（或编号）
<code>unsigned int</code>	<code>waiting_writers</code>	在等待队列中睡眠的写进程的个数
<code>unsigned int</code>	<code>r_counter</code>	与 <code>readers</code> 类似，但当等待读取 FIFO 的进程时使用
<code>unsigned int</code>	<code>w_counter</code>	与 <code>writers</code> 类似，但当等待写入 FIFO 的进程时使用
<code>struct fasync_struct *</code>	<code>fasync_readers</code>	用于通过信号进行的异步 I/O 通知
<code>struct fasync_struct *</code>	<code>fasync_writers</code>	用于通过信号进行的异步 I/O 通知

除了一个索引节点对象和两个文件对象之外，每个管道都还有自己的管道缓冲区 (*pipe buffer*)。实际上，它是一个单独的页，其中包含了已经写入管道等待读出的数据。在 Linux 2.6.10 以前，每个管道一个管道缓冲区。而 2.6.11 内核中，管道（与 FIFO）的数据缓冲区已有很大改变，每个管道可以使用 16 个管道缓冲区。这个改变大大增强了向管道写大量数据的用户态应用的性能。

`pipe_inode_info` 数据结构的 `bufs` 字段存放一个具有 16 个 `pipe_buffer` 对象的数组，每个对象代表一个管道缓冲区。该对象的字段如表 19-2 所示。

表 19-2: `pipe_buffer` 对象的字段

类型	字段	说明
<code>struct page *</code>	<code>page</code>	管道缓冲区页框的描述符地址
<code>unsigned int</code>	<code>offset</code>	页框内有效数据的当前位置
<code>unsigned int</code>	<code>len</code>	页框内有效数据的长度
<code>struct pipe_buf_operations*</code>	<code>ops</code>	管道缓冲区方法表的地址(管道缓冲区空时为 NULL)

`ops` 字段指向管道缓冲区方法表 `anon_pipe_buf_ops`，它是一个类型为 `pipe_buf_operations` 的数据结构。实际上，它有三个方法：

`map`

在访问缓冲区数据之前调用。它只在管道缓冲区在高端内存时对管道缓冲区页框调用 `kmap()` (参见第八章“高端内存页框的内核映射”一节)。

`unmap`

不再访问缓冲区数据时调用。它对管道缓冲区页框调用 `kunmap()`。

`release`

当释放管道缓冲区时调用。该方法实现了一个单页内存高速缓存：释放的不是存放缓冲区的那个页框，而是由 `pipe_inode_info` 数据结构（如果不是 NULL）的 `tmp_page` 字段指向的高速缓存页框。存放缓冲区的页框变成新的高速缓存页框。

16 个缓冲区可以被看作一个整体环形缓冲区：写进程不断向这个大缓冲区追加数据，而读进程则不断移出数据。所有管道缓冲区中当前写入而等待读出的字节数就是所谓的管道大小。为提高效率，仍然要读的数据可以分散在几个未填满的管道缓冲区内：事实上，在上一个管道缓冲区没有足够空间存放新数据时，每个写操作都可能会把数据拷贝到一个新的空管道缓冲区。因此，内核必须记录：

- 下一个待读字节所在的管道缓冲区、页框中的对应偏移量。该管道缓冲区的索引存放在 `pipe_inode_info` 数据结构的 `curbuf` 字段，而偏移量在相应 `pipe_buffer` 对象的 `offset` 字段。
- 第一个空管道缓冲区。它可以通过增加当前管道缓冲区的索引得到（模为16），并存放在 `pipe_inode_info` 数据结构的 `curbuf` 字段，而存放有效数据的管道缓冲区号存放在 `nrbufs` 字段。

为了避免对管道数据结构的竞争条件，内核使用包含在索引节点对象中的 `i_sem` 信号量。

pipefs 特殊文件系统

管道是作为一组 VFS 对象来实现的，因此没有对应的磁盘映象。在 Linux 2.6 中，把这些 VFS 对象组织为 *pipefs* 特殊文件系统以加速它们的处理（参见第十二章“特殊文件系统”一节）。因为这种文件系统在系统目录树中没有安装点，因此用户根本看不到它。但是，有了 *pipefs*，管道完全被整合到 VFS 层，内核就可以以命名管道或 FIFO 的方式处理它们，FIFO 是以终端用户认可的文件而存在的（参见后面“FIFO”一节）。

`init_pipe_fs()` 函数（一般是在内核初始化期间执行）注册 *pipefs* 文件系统并安装它（参见第十二章“安装普通文件系统”一节）：

```
struct file_system_type pipe_fs_type;
pipe_fs_type.name = "pipefs";
pipe_fs_type.get_sb = pipefs_get_sb;
pipe_fs_type.kill_sb = kill_anon_super;
register_filesystem(&pipe_fs_type);
pipe_mnt = do_kern_mount("pipefs", 0, "pipefs", NULL);
```

表示 *pipefs* 根目录的已安装文件系统对象存放在 `pipe_mnt` 变量中。

创建和撤消管道

`pipe()` 系统调用由 `sys_pipe()` 函数处理，后者又会调用 `do_pipe()` 函数。为了创建一个新的管道，`do_pipe()` 函数执行以下操作：

1. 调用 `get_pipe_inode()` 函数，该函数为 *pipefs* 文件系统管道分配一个索引节点对象并对其进行初始化。具体来说，该函数执行下列操作：
 - a. 在 *pipefs* 文件系统中分配一个新的索引节点。
 - b. 分配 `pipe_inode_info` 数据结构，并把它的地址存放在索引节点的 `i_pipe` 字段。
 - c. 设置 `pipe_inode_info` 的 `curbuf` 和 `nrbufs` 字段为 0，并将 `bufs` 数组中的管道缓冲区对象的所有字段都清 0。

- d. 把 `pipe_inode_info` 结构的 `r_counter` 和 `w_counter` 字段初始化为 1。
- e. 把 `pipe_inode_info` 结构的 `readers` 和 `writers` 字段初始化为 1。
2. 为管道的读通道分配一个文件对象和一个文件描述符，并把这个文件对象的 `f_flag` 字段设置成 `O_RDONLY`，把 `f_op` 字段初始化成 `read_pipe_fops` 表的地址。
3. 为管道的写通道分配一个文件对象和一个文件描述符，并把这个文件对象的 `flag` 字段设置成 `O_WRONLY`，把 `f_op` 字段初始化成 `write_pipe_fops` 表的地址。
4. 分配一个目录项对象，并使用它把两个文件对象和索引节点对象连接在一起（参见第十二章的“通用文件模型”一节）；然后，把新的索引节点插入 *pipefs* 特殊文件系统中。
5. 把两个文件描述符返回给用户态进程。

发出一个 `pipe()` 系统调用的进程是最初唯一一个可以读写访问新管道的进程。为了表示该管道实际上既有一个读进程，又有一个写进程，就要把 `pipe_inode_info` 数据结构的 `readers` 和 `writers` 字段都初始化为 1。通常，只要相应管道的文件对象仍然由某个进程打开，这两个字段中的每个字段就应该都被设置成 1；如果相应的文件对象已经被释放，那么这个字段就被设置成 0，因为不会再有任何进程访问这个管道。

创建一个新进程并不增加 `readers` 和 `writers` 字段的值，因此这两个值从不超过 1（注 2）。但是，父进程仍然使用的所有文件对象的引用计数器的值都会增加（参见第三章“`clone()`、`fork()` 及 `vfork()` 系统调用”一节）。因此，即使父进程死亡时这个对象都不会被释放，管道仍会一直打开供子进程使用。

只要进程对与管道相关的一个文件描述符调用 `close()` 系统调用，内核就对相应的文件对象执行 `fput()` 函数，这会减少它的引用计数器的值。如果这个计数器变成 0，那么该函数就调用这个文件操作的 `release` 方法（参见第十二章的“`close()` 系统调用”和“与进程相关的文件”两节）。

根据文件是与读通道还是与写通道关联，`release` 方法或者由 `pipe_read_release()` 或者由 `pipe_write_release()` 函数来实现。这两个函数都调用 `pipe_release()`，后者把 `pipe_inode_info` 结构的 `readers` 字段或 `writers` 字段设置成 0。`pipe_release()` 还要检查 `readers` 和 `writers` 是否都等于 0。如果是，就调用所有管道缓冲区的 `release` 方法，向伙伴系统（buddy system）释放所有管道缓冲区页框；此外，函数还释放由 `tmp_page` 字段指向的高速缓存页框。否则，`readers` 或者 `writers` 字段不为 0，函数唤醒在管道的等待队列上睡眠的任一进程，以使它们可以识别管道状态的变化。

注 2：正如我们将看到的，当与 FIFO 相关时，`readers` 和 `writers` 字段用作计数器而不是标志。

从管道中读取数据

希望从管道中读取数据的进程发出一个 `read()` 系统调用，为管道的读端指定一个文件描述符。正如在第十二章的“`read()`和`write()`系统调用”一节中描述的那样，内核最终调用与这个文件描述符相关的文件操作表中所找到的`read`方法。在管道的情况下，`read`方法在 `read_pipe_fops` 表中的表项指向 `pipe_read()` 函数。

`pipe_read()` 相当复杂，因为 POSIX 标准定义了管道的读操作的一些要求。表 19-3 概述了所期望的 `read()` 系统调用的行为，该系统调用从一个管道大小（管道缓冲区中待读的字节数）为 p 的管道中读取 n 个字节。

表 19-3：从一个管道中读取 n 个字节

管道大小 p	至少有一个写进程		没有写进程	
	阻塞读			非阻塞读
	睡眠的写者进程	无睡眠的写者进程		
	$p=0$	等待某一数据、拷贝它并返回它的 大小		
$0 < p < n$	拷贝 n 个字节并返回 n ， 当管道缓冲区为空时 等待数据	拷贝 p 个字节并返回 p ；在管道的缓冲区中 还剩 0 个字节	返回 0	
$p \geq n$	拷贝 n 个字节并返回 n ；在管道的缓冲区中 还剩 $p-n$ 个字节			

这个系统调用可能以两种方式阻塞当前进程：

- 当系统调用开始时管道缓冲区为空。
- 管道缓冲区没有包含所有请求的字节，写进程在等待缓冲区的空间时曾被置为睡眠。

注意，读操作可以是非阻塞的。在这种情况下，只要所有可用的字节（即使是 0 个）一被拷贝到用户地址空间中，读操作就完成（注 3）。

还要注意，只有在管道为空而且当前没有进程正在使用与管道的写通道相关的文件对象时，`read()` 系统调用才会返回 0。

注 3：非阻塞操作通常都是通过通过在 `open()` 系统调用中指定 `O_NONBLOCK` 标志进行请求。这个方法并不适合管道，因为管道不能被打开；但是，进程可以通过对相应的文件描述符发出一个 `fcntl()` 系统调用来请求对管道执行非阻塞操作。

pipe_read()函数执行以下操作：

1. 获取索引节点的 `i_sem` 信号量。
2. 确定存放在 `pipe_inode_info` 结构 `nrbufs` 字段中的管道大小是否为 0。如果是，说明所有管道缓冲区为空。这时还要确定函数必须返回还是进程在等待时必须被阻塞，直到其他进程向管道中写入一些数据（参见表 19-3）。I/O 操作的类型（阻塞或非阻塞）是通过文件对象的 `f_flags` 字段中的 `O_NONBLOCK` 标志来表示的。如果当前进程必须被阻塞，则函数执行下列操作：
 - a. 调用 `prepare_to_wait()` 把 `current` 加到管道的等待队列 (`pipe_inode_info` 结构的 `wait` 字段)。
 - b. 释放索引节点的信号量。
 - c. 调用 `schedule()`。
 - d. 一旦 `current` 被唤醒，就调用 `finish_wait()` 把它从等待队列中删除，再次获取 `i_sem` 索引节点信号量，然后跳回第 2 步。
3. 从 `pipe_inode_info` 数据结构的 `curbuf` 字段得到当前管道缓冲区索引。
4. 执行管道缓冲区的 `map` 方法。
5. 从管道缓冲区拷贝请求的字节数（如果较小，就是管道缓冲区可用字节数）到用户地址空间。
6. 执行管道缓冲区的 `unmap` 方法。
7. 更新相应 `pipe_buffer` 对象的 `offset` 和 `len` 字段。
8. 如果管道缓冲区已空（`pipe_buffer` 对象的 `len` 字段现在等于 0），则调用管道缓冲区的 `release` 方法释放对应的页框，把 `pipe_buffer` 对象的 `ops` 字段置为 `NULL`，增加在 `pipe_inode_info` 数据结构的 `curbuf` 字段中存放的当前管道缓冲区索引，并减小 `nrbufs` 字段中非空管道缓冲区计数器的值。
9. 如果所有请求字节拷贝完毕，则跳至第 12 步。
10. 目前，还没有把所有请求字节拷贝到用户态地址空间。如果管道大小大于 0（`pipe_inode_info` 的 `nrbufs` 字段不是 `NULL`），则跳到第 3 步。
11. 管道缓冲区内已没有剩余字节。如果至少有一个写进程正在睡眠（即 `pipe_inode_info` 数据结构的 `waiting_writers` 字段大于 0），且读操作是阻塞的，那么调用 `wake_up_interruptible_sync()` 唤醒在管道等待队列中所有睡眠的进程，然后跳至第 2 步。
12. 释放索引节点的 `i_sem` 信号量。

- 13. 调用 `wake_up_interruptible_sync()` 函数唤醒在管道的等待队列中所有睡眠的写者进程。
- 14. 返回拷贝到用户地址空间的字节数。

向管道中写入数据

希望向管道中写入数据的进程发出一个 `write()` 系统调用，为管道的写端指定一个文件描述符。内核通过调用适当文件对象的 `write` 方法来满足这个请求；`write_pipe_fops` 表中相应的项指向 `pipe_write()` 函数。

表 19-4 概述了由 POSIX 标准所定义的 `write()` 系统调用的行为，该系统调用请求把 n 个字节写入一个管道中，而该管道在它的缓冲区中有 u 个未用的字节。具体地说，该标准要求涉及少量字节数的写操作必须原子地执行。更确切地说，如果两个或者多个进程并发地在写入一个管道，那么任何少于 4096 个字节（管道缓冲区的大小）的写操作都必须单独完成，而不能与唯一进程对同一个管道的写操作交叉进行。但是，超过 4096 个字节的写操作是可分割的，也可以强制调用进程睡眠。

表 19-4：把 n 个字节写入管道

可用缓冲区的空间 u	至少有一个读进程		没有读进程
	阻塞写	非阻塞写	
$u < n \leq 4096$	等待，直到有 $n - u$ 个字节被释放为止，拷贝 n 个字节，并返回 n	返回 <code>-EAGAIN</code>	发送 <code>SIGPIPE</code> 信号并返回 <code>-EPIPE</code>
$n > 4096$	拷贝 n 个字节（必要时等待）并返回 n	如果 $u > 0$ ，就拷贝 u 个字节并返回 u ，否则就返回 <code>-EAGAIN</code>	
$u \geq n$	拷贝 n 个字节并返回 n		

还有，如果管道没有读进程（也就是说，如果管道的索引节点对象的 `readers` 字段的值是 0），那么任何对管道执行的写操作都会失败。在这种情况下，内核会向写进程发送一个 `SIGPIPE` 信号，并停止 `write()` 系统调用，使其返回一个 `-EPIPE` 错误码，这个错误码就表示我们熟悉的“Broken pipe（损坏的管道）”消息。

`pipe_write()` 函数执行以下操作：

1. 获取索引节点的 `i_sem` 信号量。
2. 检查管道是否至少有一个读进程。如果不是,就向当前进程发送一个 `SIGPIPE` 信号,释放索引节点信号量并返回 `-EPIPE` 值。
3. 将 `pipe_inode_info` 数据结构 `curbuf` 和 `nrbufs` 字段相加并减一得到最后写入的管道缓冲区索引。如果该管道缓冲区有足够空间存放待写字节,就拷入这些数据:
 - a. 执行管道缓冲区的 `map` 方法。
 - b. 把所有字节拷贝到管道缓冲区。
 - c. 执行管道缓冲区的 `unmap` 方法。
 - d. 更新相应 `pipe_buffer` 对象的 `len` 字段。
 - e. 跳至第 11 步。
4. 如果 `pipe_inode_info` 数据结构的 `nrbufs` 字段等于 16,就表明没有空闲管道缓冲区来存放待写字节。这种情况下:
 - a. 如果写操作是非阻塞的,跳至第 11 步,结束并返回错误码 `-EAGAIN`。
 - b. 如果写操作是阻塞的,将 `pipe_inode_info` 结构的 `waiting_writers` 字段加 1,调用 `prepare_to_wait()` 将当前操作加入管道等待队列 (`pipe_inode_info` 结构的 `wait` 字段),释放索引节点信号量,调用 `schedule()`。一旦唤醒,则调用 `finish_wait()` 从等待队列中移出当前操作,重新获得索引节点信号量,递减 `waiting_writers` 字段,然后跳回第 4 步。
5. 现在至少有一个空缓冲区,将 `pipe_inode_info` 数据结构的 `curbuf` 和 `nrbufs` 字段相加得到第一个空管道缓冲区索引。
6. 除非 `pipe_inode_info` 数据结构的 `tmp_page` 字段不是 `NULL`,否则从伙伴系统中分配一个新页框。
7. 从用户态地址空间拷贝多达 4096 个字节到页框(如果必要,在内核态线性地址空间作临时映射)。
8. 更新与管道缓冲区关联的 `pipe_buffer` 对象的字段:将 `page` 字段设为页框描述符的地址, `ops` 字段设为 `anon_pipe_buf_ops` 表的地址, `offset` 字段设为 0, `len` 字段设为写入的字节数。
9. 增加非空管道缓冲区计数器的值,该缓冲区计数器存放在 `pipe_inode_inf` 结构的 `nrbufs` 字段。
10. 如果所有请求的字节还没有写完,则跳至第 4 步。

11. 释放索引节点信号量。
12. 唤醒在管道等待队列上睡眠的所有读进程。
13. 返回写入管道缓冲区的字节数（如果无法写入，则返回错误码）。

FIFO

虽然管道是一种十分简单、灵活、有效的通信机制，但它们有一个主要的缺点，也就是无法打开已经存在的管道。这就使得任意的两个进程不可能共享同一个管道，除非管道由一个共同的祖先进程创建。

这个缺点在很多应用程序中都存在。例如，考虑一个数据库引擎服务器，该服务器连续地轮流询问发出查询请求的客户端进程，并把数据库查询的结果返回客户端进程。服务器和给定客户端之间的每次交互都可以使用一个管道进行处理。但是，当用户显式查询数据库时，通常由shell命令根据需要创建客户端进程；因此，服务器进程和客户端进程就不能方便地共享管道。

为了突破这种限制，Unix系统引入了一种称为命名管道(*named pipe*)或者FIFO[FIFO代表“先进先出 (first in, first out)”：最先写入文件的字节总是被最先读出]的特殊文件类型。FIFO在这几个方面都非常类似于管道：在文件系统中不拥有磁盘块，打开的FIFO总是与一个内核缓冲区相关联，这一缓冲区中临时存放两个或多个进程之间交换的数据。

然而，有了磁盘索引节点，使得任何进程都可以访问FIFO，因为FIFO文件名包含在系统的目录树中。因此，在前面那个数据库的例子中，服务器和客户端之间的通信可以很容易地使用FIFO而不是管道。服务器在启动时创建一个FIFO，由客户端程序用来发出自己的请求。每个客户端程序在建立连接之前都另外创建一个FIFO，并在自己对服务器发出的最初请求中包含这个FIFO的名字，服务器程序就可以把查询结果写入这个FIFO。

在Linux 2.6中，FIFO和管道几乎是相同的，并使用相同的pipe_inode_info结构。事实上，FIFO的read和write操作就是由前面“从管道中读取数据”和“向管道中写入数据”这两节描述的pipe_read()和pipe_write()函数实现的。事实上，只有两点主要的差别：

- FIFO索引节点出现在系统目录树上而不是pipefs特殊文件系统中。
- FIFO是一种双向通信管道；也就是说，可能以读/写模式打开一个FIFO。

因此，为了完成我们的描述，我们仅说明如何创建和打开FIFO。

创建并打开 FIFO

进程通过执行 `mknod()` (注 4) 系统调用创建一个 FIFO (参见第十三章的“设备文件”一节), 传递的参数是新 FIFO 的路径名以及 `S_IFIFO` (`0x10000`) 与这个新文件的权限位掩码进行逻辑或的结果。POSIX 引入了一个名为 `mkfifo()` 的系统调用专门用来创建 FIFO。这个系统调用在 Linux 以及 System V Release 4 中是作为调用 `mknod()` 的 C 库函数实现的。

FIFO 一旦被创建, 就可以使用普通的 `open()`、`read()`、`write()` 和 `close()` 系统调用访问 FIFO, 但是 VFS 对 FIFO 的处理方法比较特殊, 因为 FIFO 的索引节点及文件操作都是专用的, 并且不依赖于 FIFO 所在的文件系统。

POSIX 标准定义了 `open()` 系统调用对 FIFO 的操作; 这种操作本质上与所请求的访问类型、I/O 操作的种类 (阻塞或非阻塞) 以及其他正在访问 FIFO 的进程的存在状况有关。

进程可以为读操作、写操作或者读写操作打开一个 FIFO。根据这三种情况, 把与相应的文件对象相关的文件操作设置成特定的方法。

当进程打开一个 FIFO 时, VFS 就执行一些与设备文件所执行的操作相同的操作 (参见第十三章的“设备文件的 VFS 处理”一节)。与打开的 FIFO 相关的索引节点对象是由依赖于文件系统的 `read_inode` 超级块方法进行初始化的。这个方法总要检查磁盘上的索引节点是否表示一个特殊文件, 并在必要时调用 `init_special_inode()` 函数。这个函数又把索引节点对象的 `i_fop` 字段设置为 `def_fifo_fops` 表的地址。随后, 内核把文件对象的文件操作表设置为 `def_fifo_fops`, 并执行它的 `open` 方法, 这个方法由 `fifo_open()` 实现。

`fifo_open()` 函数初始化专用于 FIFO 的数据结构; 具体来说, 它执行下列操作:

1. 获取 `i_sem` 索引节点信号量。
2. 检查索引节点对象的 `i_pipe` 字段; 如果为 `NULL`, 则分配并初始化一个新的 `pipe_inode_info` 结构, 这与本章前面“创建和撤销管道”一节的第 1b~1e 步相同。
3. 根据 `open()` 系统调用的参数中指定的访问模式, 用合适的文件操作表的地址初始化文件对象的 `f_op` 字段 (如表 I9-5 所示)。

注 4: 实际上, 用 `mknod()` 几乎可以创建任何种类的文件, 如块设备文件、字符设备文件、FIFO 甚至是普通文件 (但是该函数不能创建目录和套接字)。

表 19-5: FIFO 的文件操作

访问类型	文件操作	读方法	写方法
只读	read_fifo_fops	pipe_read()	bad_pipe_w()
只写	write_fifo_fops	bad_pipe_r()	pipe_write()
读 / 写	rdwr_fifo_fops	pipe_read()	pipe_write()

4. 如果访问模式或者为只读或者为读 / 写, 则把 1 加到 pipe_inode_info 结构的 readers 字段和 r_counter 字段。此外, 如果访问模式是只读的, 且没有其他的读进程, 则唤醒等待队列上的任何写进程。
5. 如果访问模式或者为只写或者为读 / 写, 则把 1 加到 pipe_inode_info 结构的 writers 字段和 w_counter 字段。此外, 如果访问模式是只写的, 且没有其他的写进程, 则唤醒等待队列上的任何读进程。
6. 如果没有读进程或没有写进程, 则确定函数是应当阻塞还是返回一个错误码而终止 (如表 19-6 所示)。

表 19-6: fifo_open() 函数的行为

访问类型	阻塞	非阻塞
只读, 有写者	成功返回	成功返回
只读, 无写者	等待一个写者	成功返回
只写, 有读者	成功返回	成功返回
只写, 无读者	等待一个读者	返回 -ENXIO
读 / 写	成功返回	成功返回

7. 释放索引节点信号量, 并终止, 返回 0 (成功)。

FIFO 的三个专用文件操作表的主要区别是 read 和 write 方法的实现不同。如果访问类型允许读操作, 那么 read 方法是使用 pipe_read() 函数实现的; 否则, read 方法就是使用 bad_pipe_r() 函数实现的, 该函数只是返回一个错误码。类似地, 如果访问类型允许写操作, 那么 write 方法就是使用 pipe_write() 函数实现的; 否则, write 方法就是使用 bad_pipe_w() 函数实现的, 该函数也只是返回一个错误代码。

System V IPC

IPC 是进程间通信 (Interprocess Communication) 的缩写, 通常指允许用户态进程执行下列操作的一组机制:

- 通过信号量与其他进程进行同步
- 向其他进程发送消息或者从其他进程接收消息
- 和其他进程共享一段内存区

System V IPC 最初是在一个名为“Columbus Unix”的开发版 Unix 变体中引入的，之后在 AT&T 的 System III 中采用。现在在大部分 Unix 系统（包括 Linux）中都可以找到。

IPC 数据结构是在进程请求 IPC 资源（信号量、消息队列或者共享内存区）时动态创建的。每个 IPC 资源都是持久的：除非被进程显式地释放，否则永远驻留在内存中（直到系统关闭）。IPC 资源可以由任一进程使用，包括那些不共享祖先进程所创建的资源的进程。

由于一个进程可能需要同类型的多个 IPC 资源，因此每个新资源都是使用一个 32 位的 IPC 关键字来标识的，这和系统的目录树中的文件路径名类似。每个 IPC 资源都有一个 32 位的 IPC 标识符，这与和打开文件相关的文件描述符有些类似。IPC 标识符由内核分配给 IPC 资源，在系统内部是唯一的，而 IPC 关键字可以由程序员自由地选择。

当两个或者更多的进程要通过一个 IPC 资源进行通信时，这些进程都要引用该资源的 IPC 标识符。

使用 IPC 资源

根据新资源是信号量、消息队列还是共享内存区，分别调用 `semget()`、`msgget()` 或者 `shmget()` 函数创建 IPC 资源。

这三个函数的主要目的都是从 IPC 关键字（作为第一个参数传递）中导出相应的 IPC 标识符，进程以后就可以使用这个标识符对资源进行访问。如果还没有 IPC 资源和 IPC 关键字相关联，就创建一个新的资源。如果一切都顺利，那么函数就返回一个正的 IPC 标识符；否则，就返回一个如表 19-7 所示的错误码。

表 19-7：当请求 IPC 标识符时返回的错误码

错误码	说明
EACCESS	进程没有适当的访问权限
EEXIST	进程试图创建一个和已有的关键字相同的 IPC 资源
EINVAL	在 <code>semget()</code> 、 <code>msgget()</code> 或 <code>shmget()</code> 函数中有非法参数
ENOENT	不存在具有所请求的关键字的 IPC 资源，而且进程没有请求创建这个资源
ENOMEM	没有更多的存储空间供 IPC 另外的资源使用
ENOSPC	已经超过了 IPC 资源最大数目的限制

假设两个独立的进程想共享一个公共的 IPC 资源。这可以使用两种方法来达到：

- 这两个进程统一使用固定的、预定义的 IPC 关键字。这是最简单的情况，对于由很多进程实现的任一复杂的应用程序也工作得很好。然而，另外一个无关的程序也可能使用了相同的 IPC 关键字。在这种情况下，IPC 函数可能被成功地调用，但返回错误资源的 IPC 标识符（注 5）。
- 一个进程通过指定 `IPC_PRIVATE` 作为自己的 IPC 关键字来调用 `semget()`、`msgget()` 或 `shmget()` 函数。一个新的 IPC 资源因此而被分配，这个进程或者可以与应用程序中的另一个进程共享自己的 IPC 标识符（注 6），或者自己创建另一个进程。这种方法确保 IPC 资源不会偶然被其他应用程序使用。

`semget()`、`msgget()` 和 `shmget()` 函数的最后一个参数可以包括三个标志。`IPC_CREAT` 说明如果 IPC 资源不存在，就必须创建它；`IPC_EXCL` 说明如果资源已经存在而且设置了 `IPC_CREAT` 标志，那么函数就必定失败；`IPC_NOWAIT` 说明访问 IPC 资源时进程从不阻塞（典型的情况如取得消息或获取信号量）。

即使进程使用了 `IPC_CREAT` 和 `IPC_EXCL` 标志，也没有办法保证对一个 IPC 资源进行排它访问，因为其他进程也可能用自己的 IPC 标识符引用了这个资源。

为了把不正确地引用错误资源的风险降到最小，内核不会在 IPC 标识符一空闲时就再利用它。相反，分配给资源的 IPC 标识符总是大于给同类型的前一个资源所分配的标识符（唯一的例外发生在 32 位的 IPC 标识符溢出时）。每个 IPC 标识符都是通过结合使用与资源类型相关的位置使用序号（*slot usage sequence number*）、已分配资源的任意位置索引（*slot index*）以及内核中为可分配资源所选定的最大值而计算出来的。如果我们使用 s 来代表位置使用序号， M 来代表可分配资源的最大数目， i 来代表位置索引，此处 $0 \leq i < M$ ，则每个 IPC 资源的 ID 都可以按如下公式来计算：

$$\text{IPC 标识符} = s \times M + i$$

在 Linux 2.6 中， M 的值设为 32768（`IPCMNI` 宏）。位置使用序号 s 被初始化成 0，每次分配资源时增加 1。当 s 达到预定的阈值时（这取决于 IPC 资源类型），它从 0 重新开始。

注 5：`ftok()` 函数试图从作为参数传递的文件路径名和一个 8 位对象标识符中获得一个新关键字。但是这并不能担保是一个唯一关键字，因为也有可能使用不同路径名和对象标识符的两个不同应用程序会返回同一个 IPC 关键字，不过这种机会很小。

注 6：当然，这就意味着进程之间另一个通信通道的存在并不基于 IPC。

IPC 资源的每种类型（信号量、消息队列和共享内存区）都拥有 `ipc_ids` 数据结构，该结构包括的字段如表 19-8 所示。

表 19-8: `ipc_ids` 数据结构的字段

类型	字段	说明
int	<code>in_use</code>	已分配 IPC 资源数
int	<code>max_id</code>	在使用的最大位置索引
unsigned short	<code>seq</code>	下一个分配的位置使用序号
unsigned short	<code>seq_max</code>	最大位置使用序号
struct semaphore	<code>sem</code>	保护 <code>ipc_ids</code> 数据结构的信号量
struct ipc_id_ary	<code>nullentry</code>	如果 IPC 资源无法初始化，则 <code>entries</code> 字段指向伪数据结构（一般不使用）
struct ipc_id_ary *	<code>entries</code>	指向资源的 <code>ipc_id_ary</code> 数据结构的指针

`ipc_id_ary` 数据结构有两个字段：`p` 和 `size`。`P` 字段是一个指向 `kern_ipc_perm` 数据结构的指针数组，每个结构对应一个可分配资源。`size` 字段是这个数组的大小。最初，数组为共享内存区、消息队列与信号量分别存放 1、16 或 128 个指针。当太小时，内核动态地增大数组。但是每种资源都有个上限。系统管理员可以修改 `/proc/sys/kernel/sem`、`/proc/sys/kernel/msgmni` 和 `/proc/sys/kernel/shmmni` 这三个文件以改变这些上限。

每个 `kern_ipc_perm` 数据结构与一个 IPC 资源相关联，并且包含如表 19-9 所示的字段。`uid`、`gid`、`cuid` 和 `cgid` 分别存放资源的创建者的用户标识符和组标识符以及当前资源属主的用户标识符和组标识符。`mode` 位掩码包括六个标志，分别存放资源的属主、组以及其他用户的读、写访问权限。IPC 访问许可权和第一章的“访问权限和文件模式”一节中介绍的文件访问许可权类似，唯一不同的是这里没有执行许可权标志。

表 19-9: `kern_ipc_perm` 结构中的字段

类型	字段	说明
spinlock_t	<code>lock</code>	保护 IPC 资源描述符的自旋锁
int	<code>deleted</code>	如果资源已被释放，则设置该标志
int	<code>key</code>	IPC 关键字
unsigned int	<code>uid</code>	属主用户 ID
unsigned int	<code>gid</code>	属主组 ID
unsigned int	<code>cuid</code>	创建者用户 ID
unsigned int	<code>cgid</code>	创建者组 ID

表 19-9: kern_ipc_perm 结构中的字段 (续)

类型	字段	说明
unsigned short	mode	许可权位掩码
unsigned long	seq	位置使用序号
void*	security	安全结构指针 (用于 SELinux)

kern_ipc_perm 数据结构也包括一个 key 字段和一个 seq 字段, 前者指的是相应资源的 IPC 关键字, 后者存放的是用来计算该资源的 IPC 标识符所使用的位置使用序号。

semctl()、msgctl() 和 shmctl() 函数都可以用来处理 IPC 资源。IPC_SET 子命令允许进程改变属主的用户标识符和组标识符以及 ipc_perm 数据结构中的许可权位掩码。IPC_STAT 和 IPC_INFO 子命令取得和资源有关的信息。最后, IPC_RMID 子命令释放 IPC 资源。根据 IPC 资源的种类不同, 还可以使用其他专用的子命令 (注 7)。

一旦一个 IPC 资源被创建, 进程就可以通过一些专用函数对这个资源进行操作。进程可以执行 semop() 函数获得或释放一个 IPC 信号量。当进程希望发送或接收一个 IPC 消息时, 就分别使用 msgsnd() 和 msgrcv() 函数。最后, 进程可以分别使用 shmat() 和 shmdt() 函数把一个共享内存区附加到自己的地址空间中或者取消这种附加关系。

ipc() 系统调用

所有的 IPC 函数都必须通过适当的 Linux 系统调用实现。实际上, 在 80 × 86 体系结构中, 只有一个名为 ipc() 的 IPC 系统调用。当进程调用一个 IPC 函数时, 比如说 msgget(), 该函数实际上调用 C 库中的一个封装函数, 该函数又通过传递 msgget() 的所有参数加上一个适当的子命令代码 (在本例中是 MSGGET) 来调用 ipc() 系统调用。sys_ipc() 服务例程检查子命令代码, 并调用内核函数实现所请求的服务。

ipc() “多路复用” 系统调用是从早期的 Linux 版本中继承而来的, 早期 Linux 版本把 IPC 代码包含在动态模块中 (参见附录二)。在 system_call 表中为可能未实现的内核部件保留几个系统调用入口并没有什么意义, 因此内核设计者就采用了这种多路复用的方法。

现在, System V IPC 不再作为动态模块被编译, 因此也就没有理由使用单个 IPC 系统调用。事实上, Linux 在 HP 的 Alpha 体系结构和 Intel 的 IA-64 上为每个 IPC 函数都提供了一个系统调用。

注 7: IPC 在设计上的一个缺陷是用户态进程不能原子地创建和初始化一个 IPC 信号量, 因为这两个操作是由两个不同的 IPC 函数执行的。

IPC 信号量

IPC 信号量和在第五章中介绍的内核信号量非常类似：二者都是计数器，用来为多个进程共享的数据结构提供受控访问。

如果受保护的资源是可用的，那么信号量的值就是正数；如果受保护的资源现不可用，那么信号量的值就是 0。要访问资源的进程试图把信号量的值减 1，但是，内核阻塞这个进程，直到在这个信号量上的操作产生一个正值。当进程释放受保护的资源时，就把信号量的值增加 1；在这样处理的过程中，其他所有正在等待这个信号量的进程就都被唤醒。

实际上，IPC 信号量比内核信号量的处理更复杂是由于两个主要的原因：

- 每个 IPC 信号量都是一个或者多个信号量值的集合，而不像内核信号量一样只有一个值。这意味着同一个 IPC 资源可以保护多个独立、共享的数据结构。在资源正在被分配的过程中，必须把每个 IPC 信号量中的信号量的个数指定为 `semget()` 函数的一个参数。从现在开始，我们就把信号量内部的计数器作为原始信号量 (*primitive semaphore*) 来引用。IPC 信号量资源的个数和单个 IPC 资源内原始信号量的个数都有界限，其缺省值前者为 128，后者为 250；不过，系统管理员可以通过 `/proc/sys/kernel/sem` 文件很容易地修改这两个界限。
- System V IPC 信号量提供了一种失效安全机制，这是用于进程不能取消以前对信号量执行的操作就死亡的情况的。当进程选择使用这种机制时，由此引起的操作就是所谓的可取消的(*undoable*)信号量操作。当进程死亡时，所有 IPC 信号量都可以恢复成原来的值，就好像从来都没有开始它的操作。这有助于防止出现这种情况：由于正在结束的进程不能手工取消它的信号量操作，其他使用相同信号量的进程无限地停留在阻塞状态。

首先我们简要描绘一下，当进程想访问 IPC 信号量所保护的一个或者多个资源时所执行的典型步骤：

1. 调用 `semget()` 封装函数来获得 IPC 信号量标识符，作为参数指定对共享资源进行保护的 IPC 信号量的 IPC 关键字。如果进程希望创建一个新的 IPC 信号量，则还要指定 `IPC_CREATE` 或者 `IPC_PRIVATE` 标志以及所需要的原始信号量（参见本章前面的“使用 IPC 资源”一节）。
2. 调用 `semop()` 封装函数来测试并递减所有原始信号量所涉及的值。如果所有的测试全部成功，就执行递减操作，结束函数并允许这个进程访问受保护的资源。如果有些信号量正在使用，那么进程通常都会被挂起，直到某个其他进程释放这个资源为

止。函数接收的参数为IPC信号量标识符、用来指定对原始信号量所进行的原子操作的一组整数以及这种操作的个数。作为选项，进程也可以指定SEM_UNDO标志，这个标志通知内核：如果进程没有释放原始信号量就退出，那么撤消那些操作。

3. 当放弃受保护的资源时，就再次调用semop()函数来原子地增加所有有关的原始信号量。
4. 作为选择，调用semctl()封装函数，在参数中指定IPC_RMID命令把这个IPC信号量从系统中删除。

现在我们可以讨论内核是如何实现IPC信号量的。有关的数据结构如图19-1所示。sem_ids变量存放IPC信号量资源类型的ipc_ids数据结构，对应的ipc_id_ary数据结构包含一个指针数组，它指向sem_array数据结构，每个元素对应一个IPC信号量资源。

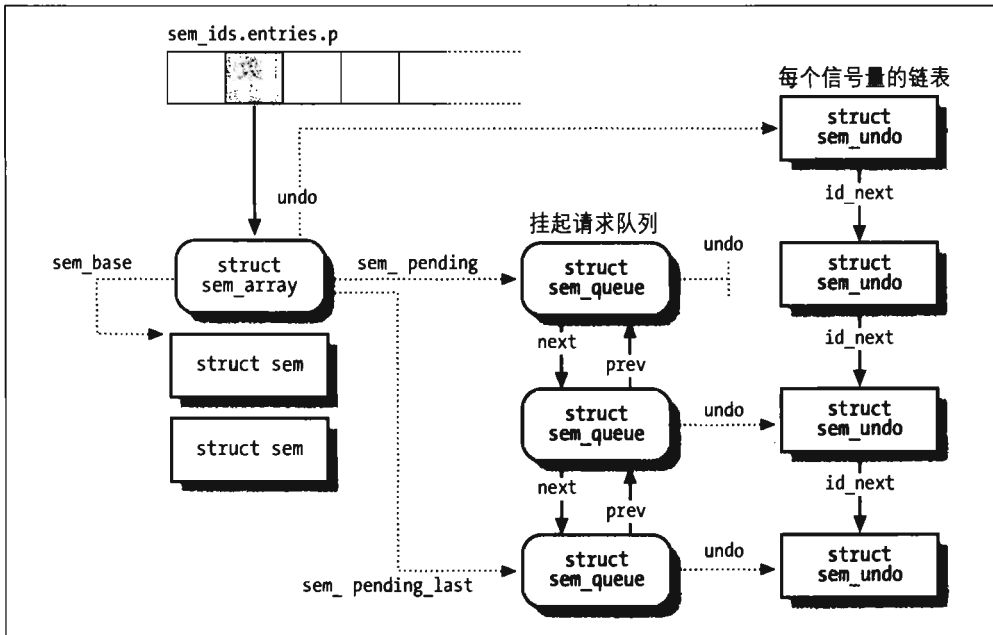


图 19-1：IPC 信号量数据结构

从形式上说，这个数组存放指向kern_ipc_perm数据结构的指针，但是每个结构只不过是sem_array数据结构的一个字段。sem_array数据结构的所有字段如表19-10所示。

表 19-10: sem_array 数据结构中的字段

类型	字段	说明
struct kern_ipc_perm	sem_perm	kern_ipc_perm 数据结构
long	sem_otime	最后一次调用 semop() 的时间戳
long	sem_ctime	最后一次修改的时间戳
struct sem *	sem_base	指向第一个 sem 结构的指针
struct sem_queue *	sem_pending	挂起操作
struct sem_queue **	sem_pending_last	最后一次挂起操作
struct sem_undo *	undo	取消请求
unsigned short	sem_nsems	数组中信号量的个数

sem_base 字段指向 sem 数据结构的数组，每个元素对应一个 IPC 原始信号量。sem 数据结构只包括两个字段：

semval
 信号量的计数器的值。

sempid
 最后一个访问信号量的进程的 PID。进程可以使用 semctl() 封装函数查询该值。

可取消的信号量操作

如果一个进程突然放弃执行，那么它就不能取消已经开始执行的操作（例如，释放自己保留的信号量），因此通过把这些操作定义成可取消的，进程就可以让内核把信号量返回到一致状态并允许其他进程继续执行。进程可以在 semop() 函数中指定 SEM_UNDO 标志来请求可取消的操作。

为了有助于内核撤消给定进程对给定的 IPC 信号量资源所执行的可撤销操作，有关的信息存放在 sem_undo 数据结构中。这个结构实际上包含信号量的 IPC 标识符及一个整数数组，这个数组表示由进程执行的所有可取消操作对原始信号量值引起的修改。

有一个简单的例子可以说明如何使用这种 sem_undo 元素。考虑一个进程使用具有 4 个原始信号量的一个 IPC 信号量资源，并假设该进程调用 semop() 函数把第一个计数器的值增加 1 并把第二个计数器的值减 2。如果该函数指定了 SEM_UNDO 标志，sem_undo 数据结构中的第一个数组元素中的整数值就被减少 1，而第二个元素就被增加 2，其他两个整数都保持不变。同一进程对这个 IPC 信号量执行的更多的可取消操作将相应地改变存放在 sem_undo 结构中的整数值。当进程退出时，该数组中的任何非零值就表示对相应

原始信号量的一个或者多个错乱操作,内核只简单地给相应的原始信号量计数器增加这个非零值来取消这些操作。换言之,把异常中断的进程所做的修改退回,而其他进程所做的修改仍然能反映信号量的状态。

对于每个进程来说,内核都要记录以可取消操作处理的所有信号量资源,这样如果进程意外退出,就可以回滚这些操作。还有,内核还必须对每个信号量都记录它所有的 `sem_undo` 结构,这样只要进程使用 `semctl()` 来强行给一个原始信号量的计数器赋一个明确的值或者撤消一个 IPC 信号量资源时,内核就可以快速访问这些结构。

正是由于两个链表(我们称之为每个进程的链表和每个信号量的链表),使得内核可以有效地处理这些任务。第一个链表记录给定进程以可取消操作处理的所有信号量。第二个链表记录对以可取消操对给定信号量进行操作的所有进程。更确切地说:

- 每个进程链表包含所有的 `sem_undo` 数据结构,该结构对应于进程执行了可取消操作的 IPC 信号量。进程描述符的 `sysvsem.undo_list` 字段指向一个 `sem_undo_list` 类型的数据结构,而该结构又包含了指针指向该链表的第一个元素。每个 `sem_undo` 数据结构的 `proc_next` 字段指向该链表的下一个元素(在第三章“`clone()`、`fork()` 及 `vfork()` 系统调用”一节我们讲过,因为都共享一个 `sem_undo_list` 描述符,将 `CLONE_SYSVSEM` 标志传给 `clone()` 系统调用而克隆的进程都共享同一个可取消信号量操作链表)。
- 每个信号量链表包含的所有 `sem_undo` 数据结构对应于在该信号量上执行可取消操作的进程。`sem_array` 数据结构的 `undo` 字段指向链表的第一个元素,而每个 `sem_undo` 数据结构的 `id_next` 字段指向链表的下一个元素。

当进程结束时,每个进程的链表才被使用。`exit_sem()` 函数由 `do_exit()` 调用,后者会遍历这个链表,并为进程所涉及的每个 IPC 信号量平息错乱操作产生的影响。与此相对照,当进程调用 `semctl()` 函数强行给一个原始信号量赋一个明确的值时,每个信号量的链表才被使用。内核把指向 IPC 信号量资源的所有 `sem_undo` 数据结构中的数组的相应元素都设置成 0,因为撤消原始信号量的一个可取消操作不再有任何意义。此外,在 IPC 信号量被清除时,每个信号量链表也被使用。通过把 `semid` 字段设置成 -1 而使所有有关的 `sem_undo` 数据结构都变为无效(注 8)。

注 8: 注意仅仅是使这些数据结构无效而已,并没有释放它们,因为从所有进程的每个进程链表中删除这些数据结构的代价太高了。

挂起请求的队列

内核给每个 IPC 信号量都分配了一个挂起请求队列，用来标识正在等待数组中的一个（或多个）信号量的进程。这个队列是一个 `sem_queue` 数据结构的双向链表，其字段如表 19-11 所示。队列中的第一个和最后一个挂起请求分别由 `sem_array` 结构中的 `sem_pending` 和 `sem_pending_last` 字段所指向。这最后一个字段允许把链表作为一个 FIFO 进行简单的处理。新的挂起请求都被追加到链表的末尾，这样就可以稍后得到服务。挂起请求最重要的字段是 `nsops` 和 `sops`，前者存放挂起操作所涉及的原始信号量的个数，后者指向描述每个信号量操作的整型数组。`sleeper` 字段存放发出请求操作的睡眠进程的描述符地址。

表 19-11: `sem_queue` 数据结构中的字段

类型	字段	说明
<code>struct sem_queue *</code>	<code>next</code>	指向下一个队列元素的指针
<code>struct sem_queue **</code>	<code>prev</code>	指向上一个队列元素的指针
<code>struct wait_queue *</code>	<code>sleeper</code>	指向请求信号量操作的睡眠进程
<code>struct sem_undo *</code>	<code>undo</code>	指向 <code>sem_undo</code> 结构的指针
<code>int</code>	<code>pid</code>	进程标识符
<code>int</code>	<code>status</code>	操作的完成状态
<code>struct sem_array *</code>	<code>sma</code>	IPC 信号量的描述符指针
<code>int</code>	<code>id</code>	IPC 信号量的位置索引
<code>struct sembuf *</code>	<code>sops</code>	指向挂起操作的数组的指针
<code>int</code>	<code>nsops</code>	挂起操作的个数
<code>int</code>	<code>alter</code>	标志，表示操作是否修改了信号量数组

图 19-1 显示有三个挂起请求的一个 IPC 信号量。第二个和第三个请求涉及可取消操作，因此 `sem_queue` 数据结构的 `undo` 字段指向相应的 `sem_undo` 结构；第一个挂起请求的 `undo` 字段为 `NULL`，因为相应的操作是不可取消的。

IPC 消息

进程彼此之间可以通过 IPC 消息进行通信。进程产生的每条消息都被发送到一个 IPC 消息队列中，这个消息一直存放在队列中直到另一个进程将其读走为止。

消息是由固定大小的首部和可变长度的正文组成的，可以使用一个整数值（消息类型）标识消息，这就允许进程有选择地从消息队列中获取消息（注9）。只要进程从IPC消息队列中读出一条消息，内核就把这个消息删除；因此，只能有一个进程接收一条给定的消息。

为了发送一条消息，进程要调用 `msgsnd()` 函数，传递给它以下参数：

- 目标消息队列的 IPC 标识符
- 消息正文的大小
- 用户态缓冲区的地址，缓冲区中包含消息类型，之后紧跟消息正文

进程要获得一条消息就要调用 `msgrcv()` 函数，传递给它如下参数：

- IPC 消息队列资源的 IPC 标识符
- 指向用户态缓冲区的指针，消息类型和消息正文应该到被拷贝这个缓冲区
- 缓冲区的大小
- 一个值 t ，指定应该获得什么消息

如果 t 的值为 0，就返回队列中的第一条消息。如果 t 为正数，就返回队列中类型等于 t 的第一条消息。最后，如果 t 为负数，就返回消息类型小于等于 t 绝对值的最小的第一条消息。

为了避免资源耗尽，IPC 消息队列资源在这几个方面是有限制的：IPC 消息队列数（缺省为 16），每个消息的大小（缺省为 8192 字节）及队列中全部信息的大小（缺省为 16384 字节）。不过和前面类似，系统管理员可以分别修改 `/proc/sys/kernel/msgmni`、`/proc/sys/kernel/msgmnb` 和 `/proc/sys/kernel/msgmax` 文件调整这些值。

与 IPC 消息队列有关的数据结构如图 19-2 所示。`msg_ids` 变量存放 IPC 消息队列资源类型的 `ipc_ids` 数据结构，相应的 `ipc_id_ary` 数据结构包含一个指向 `shmid_kernel` 数据结构的指针数组——每个 IPC 消息资源对应一个元素。从形式上看，数组中存放指向 `kern_ipc_perm` 数据结构的指针，但是，每个这样的结构只不过是 `msg_queue` 数据结构的第一个字段。`msg_queue` 数据结构的所有字段如表 19-12 所示。

注9： 我们将看到，消息队列是使用一个链表实现的。因为消息可以按照非“先进先出”的次序获得，因此“消息队列”这个名字并不恰当。不过，新消息通常都放在链表的末尾。

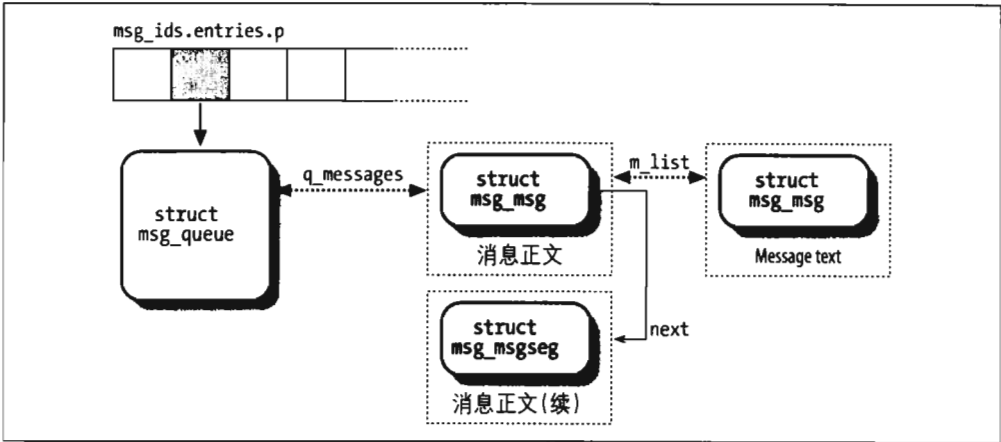


图 19-2：IPC 消息队列数据结构

表 19-12：msg_queue 数据结构

类型	字段	说明
struct kern_ipc_perm	q_perm	kern_ipc_perm 数据结构
long	q_stime	最后调用 msgsnd() 的时间
long	q_rtime	最后调用 msgrcv() 的时间
long	q_ctime	最后修改的时间
unsigned long	q_qcbytes	队列中的字节数
unsigned long	q_qnum	队列中的消息数
unsigned long	q_qbytes	队列中的最大字节数
int	q_lspid	最后调用 msgsnd() 的 PID
int	q_lrpid	最后调用 msgrcv() 的 PID
struct list_head	q_messages	队列中的消息链表
struct list_head	q_receivers	接收消息的进程链表
struct list_head	q_senders	发送消息的进程链表

最重要的字段是 q_messages，它表示包含队列中当前所有消息的双向循环链表的首部（也就是第一个哑元素）。

每条消息分开存放在一个或多个动态分配的页中。第一页的起始部分存放消息头，消息头是一个 msg_msg 类型的数据结构，它的字段如表 19-13 所示。m_list 字段指向队列中前一条和后一条消息的指针。消息的正文正好从 msg_msg 描述符之后开始，如果消息（页

的大小减去 msg_msg 描述符的大小) 大于 4072 字节, 就继续放在另一页, 它的地址存放在 msg_msg 描述符的 next 字段中。第二个页框以 msg_msgseg 类型的描述符开始, 这个描述符只包含一个 next 指针, 该指针存放可选的第三个页, 以此类推。

表 19-13: msg_msg 数据结构

类型	字段	说明
struct list_head	m_list	用于消息链表的指针
long	m_type	消息类型
int	m_ts	消息正文的大小
struct msg_msgseg *	next	消息的下一部分
void *	security	安全数据结构指针 (用于 SELinux)

当消息队列满时 (或者达到了最大消息数, 或者达到了队列最大字节数), 则试图让新消息入队的进程可能被阻塞。msg_queue 数据结构的 q_senders 字段是所有阻塞的发送进程的描述符形成的链表的头。

当消息队列为空时 (或者当进程指定的一条消息类型不在队列中时), 则接收进程也会被阻塞。msg_queue 数据结构的 q_receivers 字段是 msg_receiver 数据结构链表的头, 每个阻塞的接收进程对应其中一个元素。其中的每个结构本质上都包含一个指向进程描述的指针、一个指向消息的 msg_msg 结构的指针和所请求的消息类型。

IPC 共享内存

最有用的 IPC 机制是共享内存, 这种机制允许两个或多个进程通过把公共数据结构放入一个共享内存区 (*IPC shared memory region*) 来访问它们。如果进程要访问这种存放在共享内存区的数据结构, 就必须在自己的地址空间中增加一个新内存区 (参见第九章的“线性区”一节), 它将映射与这个共享内存区相关的页框。这样的页框可以很容易地由内核通过请求调页进行处理 (参见第九章的“请求调页”一节)。

与信号量以及消息队列一样, 调用 shmget() 函数来获得一个共享内存区的 IPC 标识符, 如果这个共享内存区不存在, 就创建它。

调用 shmat() 函数把一个共享内存区“附加 (attach)”到一个进程上。该函数使用 IPC 共享内存资源的标识符作为参数, 并试图把一个共享内存区加入到调用进程的地址空间中。调用进程可以获得这个内存区域的起始线性地址, 但是这个地址通常并不重要, 访问这个共享内存区域的每个进程都可以使用自己地址空间中的不同地址。shmat() 函数

不修改进程的页表。我们稍后会介绍在进程试图访问属于新内存区域的页时内核究竟怎样进行处理。

调用 `shmdt()` 函数来“分离”由 IPC 标识符所指定的共享内存区域，也就是说把相应的共享内存区域从进程的地址空间中删除。回想一下，IPC 共享内存资源是持久的；即使现在没有进程在使用它，相应的页也不能被丢弃，但是可以被换出。

与 IPC 资源的其他类型一样，为了避免用户态进程过分使用内存，也有一些限制施加于所允许的 IPC 共享内存区域数（缺省为 4096）、每个共享段的大小（缺省为 32MB）以及所有共享段的最大字节数（缺省为 8GB）。不过，系统管理员照样可以调整这些值，这是通过分别修改 `/proc/sys/kernel/shmmni`、`/proc/sys/kernel/shmmax` 和 `/proc/sys/kernel/shmall` 文件完成的。

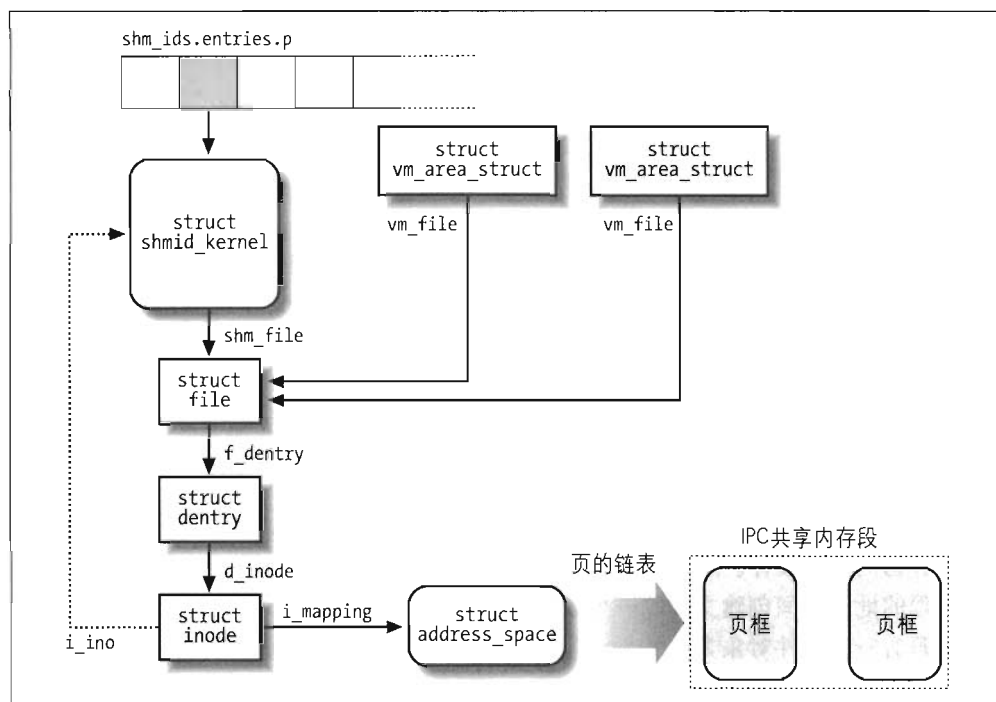


图 19-3: IPC 共享内存数据结构

图 19-3 显示与 IPC 共享内存区相关的数据结构。`shm_ids` 变量存放 IPC 共享内存资源类型 `ipc_ids` 的数据结构，相应的 `ipc_id_ary` 数据结构包含一个指向 `shmid_kernel` 数据结构的指针数组，每个 IPC 共享内存资源对应一个数组元素。从形式上看，这个数组

存放指向 `kern_ipc_perm` 数据结构指针，但是每个这样的结构只不过是 `msg_queue` 数据结构的第一个字段。`shmid_kernel` 数据结构的所有字段如表 19-14 所示。

表 19-14：在 `shmid_kernel` 数据结构中的字段

类型	字段	说明
<code>struct kern_ipc_perm</code>	<code>shm_perm</code>	<code>kern_ipc_perm</code> 数据结构
<code>struct file *</code>	<code>shm_file</code>	共享段的特殊文件
<code>int</code>	<code>id</code>	共享段的位置索引
<code>unsigned long</code>	<code>shm_nattch</code>	当前附加的内存区数
<code>unsigned long</code>	<code>shm_segsz</code>	内存区字节数
<code>long</code>	<code>shm_atime</code>	最后访问时间
<code>long</code>	<code>shm_dtime</code>	最后分离时间
<code>long</code>	<code>shm_ctime</code>	最后修改时间
<code>int</code>	<code>shm_cprid</code>	创建者的 PID
<code>int</code>	<code>shm_lprid</code>	最后访问进程的 PID
<code>struct user_struct *</code>	<code>mlock_user</code>	锁定在共享内存 RAM 中的用户的 <code>user_struct</code> 描述符的指针 (参见第三章“ <code>clone()</code> 、 <code>fork()</code> 及 <code>vfork()</code> 系统调用”一节)

最重要的字段是 `shm_file`，该字段存放文件对象的地址。这反映 Linux 2.6 中 IPC 共享内存与 VFS 的紧密结合。具体来说，每个 IPC 共享内存区与属于 *shm* 特殊文件系统的一个普通文件相关联（参见第十二章“特殊文件系统”一节）。

因为 *shm* 文件系统在系统目录树中没有安装点，因此，用户不能通过普通的 VFS 系统调用打开并访问它的文件。但是，只要进程“附加”一个内存段，内核就调用 `do_mmap()`，并在进程的地址空间创建文件的一个新的共享内存映射。因此，属于 *shm* 特殊文件系统的文件只有一个文件对象方法 `mmap`，该方法是由 `shm_mmap()` 函数实现的。

如图 19-3 所示，与 IPC 共享内存区对应的内存区是用 `vm_area_struct` 对象描述的（参见第十六章“内存映射”一节）；它的 `vm_file` 字段指回特殊文件的文件对象，而特殊文件又依次引用目录项对象和索引节点对象。存放在索引节点 `i_ino` 字段的索引节点号实际上是 IPC 共享内存区的位置索引，因此，索引节点对象间接引用 `shmid_kernel` 描述符。

同样，对于任何共享内存映射，通过 `address_space` 对象把页框包含在页高速缓存中，而 `address_space` 对象包含在索引节点中而且被索引节点的 `i_mapping` 字段引用（你也

可以参看图 16-2)。万一页框属于 IPC 共享内存区, `address_space` 对象的方法就存放在全局变量 `shmem_aops` 中。

换出 IPC 共享内存区的页

内核在把包含在共享内存区的页换出时一定要谨慎,并且交换高速缓存的作用是至关重要的(这个主题已经在第十七章“交换高速缓存”一节讨论过)。

因为 IPC 共享内存区映射的是在磁盘上没有映像的特殊索引节点,因此其页是可交换的(而不是可同步的,参见第十七章的表 17-1)。因此为了回收 IPC 共享内存区的页,内核必须把它写入交换区。因为 IPC 共享内存区是持久的——也就是说即使内存段不附加到进程,也必须保留这些页。因此即使这些页没有进程在使用,内核也不能简单地删除它们。

让我们看看 PFRA 是如何回收 IPC 共享内存区页框的。一直到 `shrink_list()` 函数处理页之前,都与第十七章“内存紧缺回收”一节所描述的一样。因为这个函数并不为 IPC 共享内存区域作任何检查,因此它会调用 `try_to_unmap()` 函数从用户态地址空间删除对页框的每个引用。正如第十七章“反向映射”一节描述的一样,相应的页表项就被删除。

然后, `shrink_list()` 函数检查页的 `PG_dirty` 标志,调用 `pageout()` (当 IPC 共享内存区域的页框在分配时总是被标记为脏,因此 `pageout()` 总是被调用)。而 `pageout()` 函数又调用所映射文件的 `address_space` 对象的 `writepage` 方法。

`shmem_writepage()` 函数实现了 IPC 共享内存区页的 `writepage` 方法。它实际上给交换区域分配一个新页槽 (page slot),然后将它从页高速缓存移到交换高速缓存(实际上就是改变页所有者的 `address_space` 对象)。该函数还在 `shmem_inode_info` 结构中存放换出页标识符,这个结构包含了 IPC 共享内存区的索引节点对象,它再次设置页的 `PG_dirty` 标志。如第十七章的表 17-5 所示, `shrink_list()` 函数检查 `PG_dirty` 标志,并通过把页留在非活动链表而中断回收过程。

迟早, PFRA 还会处理该页框。 `shrink_list()` 又一次调用 `pageout()` 尝试将页刷新到磁盘。但这一次,页已在交换高速缓存内,因而它的所有者是交换子系统的 `address_space` 对象,即 `swapper_space`。相应的 `writepage` 方法 `swap_writepage()` 开始有效地向交换区进行写入操作(参见第十七章“换出页”一节)。一旦 `pageout()` 结束, `shrink_list()` 确认该页已干净,于是从交换高速缓存删除页并释放给伙伴系统。

IPC 共享内存区的请求调页

通过 `shmat()` 加入进程的页都是哑元页 (dummy page),该函数把一个新内存区加入一

个进程的地址空间中，但是它不修改该进程的页表。此外，我们已经看到，IPC 共享内存区的页可以被换出。因此，可以通过请求调页机制来处理这些页。

我们知道，当进程试图访问IPC 共享内存区的一个单元，而其基本的页框还没有分配时则发生缺页异常。相应的异常处理程序确定引起缺页的地址是在进程的地址空间内，且相应的页表项为空；因此，它就调用 `do_no_page()` 函数（参见第九章中的“请求调页”一节）。这个函数又检查是否为此内存区定义了 `nopage` 方法。然后调用这个方法，并把页表项设置成所返回的地址（参见第十六章“内存映射的请求调页”一节）。

IPC 共享内存所使用的内存区通常都定义了 `nopage` 方法。这是通过 `shmem_nopage()` 函数实现的，该函数执行以下操作：

1. 遍历 VFS 对象的指针链表，并导出 IPC 共享内存资源的索引节点对象的地址（参见图 19-3）。
2. 从内存区域描述符的 `vm_start` 字段和请求的地址计算共享段内的逻辑页号。
3. 检查页是否已经在交换高速缓存中，如果是，则结束并返回该描述符的地址。
4. 检查页是否在交换高速缓存内且是否最新，如果是，则结束并返回该描述符的地址。
5. 检查内嵌在索引节点对象的 `shmem_inode_info` 是否存放着逻辑页号对应的换出页标识符。如果是，就调用 `read_swap_cache_async()` 执行换入操作（参见第十七章“换入页”一节），并一直等到数据传送完成，然后结束并返回页描述符的地址。
6. 否则，页不在交换区中，因此就从伙伴系统分配一个新页框，把它插入页高速缓存，并返回它的地址。

`do_no_page()` 函数对引起缺页的地址在进程的页表中所对应的表项进行设置，以使该函数指向 `nopage` 方法所返回的页框。

POSIX 消息队列

POSIX 标准（IEEE Std 1003.1-2001）基于消息队列定义了一个 IPC 机制，就是大家知道的 POSIX 消息队列。它很像本章前面“IPC 消息”一节介绍的 System V IPC 消息队列。但是 POSIX 消息队列比老的队列具有许多优点：

- 更简单的基于文件的应用接口
- 完全支持消息优先级（优先级最终决定队列中消息的位置）

- 完全支持消息到达的异步通知，这通过信号或是线程创建实现
- 用于阻塞发送与接收操作的超时机制

POSIX 消息队列通过一套库函数来实现，参见表 19-15。

表 19-15: POSIX 消息队列的库函数

函数名	说明
<code>mq_open()</code>	打开（或创建）POSIX 消息队列
<code>mq_close()</code>	关闭 POSIX 消息队列（并不删除）
<code>mq_unlink()</code>	删除 POSIX 消息队列
<code>mq_send()</code>	给 POSIX 消息队列发送一个消息
<code>mq_timedsend()</code>	在操作时限内给 POSIX 消息队列发送一个消息
<code>mq_receive()</code>	从 POSIX 消息队列接收一个消息
<code>mq_timedreceive()</code>	在操作时限内从 POSIX 消息队列接收一个消息
<code>mq_notify()</code>	在空 POSIX 消息队列中，为消息到达建立异步通知机制
<code>mq_getattr()</code>	获得 POSIX 消息队列的属性（实际上就是发送和接收操作应当是阻塞还是非阻塞）
<code>mq_setattr()</code>	设置 POSIX 消息队列的属性（实际上就是发送和接收操作应当是阻塞还是非阻塞）

我们来看看应用如何典型地使用这些函数。首先，应用调用 `mq_open()` 库函数打开一个 POSIX 消息队列。函数的第一个参数是一个指定队列名字的字符串，这与文件名类似，而且必须以 “/” 开始。该库函数接收一个 `open()` 系统调用的标志子集：`O_RDONLY`、`O_WRONLY`、`O_RDWR`、`O_CREAT`、`O_EXCL` 和 `O_NONBLOCK`（用于非阻塞发送与接收）。注意应用可以通过指定一个 `O_CREAT` 标志来创建一个新的 POSIX 消息队列。`mq_open()` 函数返回一个队列描述符，与 `open()` 系统调用返回的文件描述符非常类似。

一旦 POSIX 消息队列打开，应用可以通过库函数 `mq_send()` 和 `mq_receive()` 来发送与接收消息，并传给它们 `mq_open()` 返回的队列描述符作为参数。应用也可以通过 `mq_timedsend()` 和 `mq_timedreceive()` 指定应用程序等待发送与接收操作完成所需的最长时间。

应用除了在 `mq_receive()` 上阻塞，或者如果 `O_NONBLOCK` 标志置位则继续在消息队列上轮询外，还可以通过执行 `mq_notify()` 库函数建立异步通知机制。实际当一个消息插入空队列时，应用可以要求：要么给指定进程发出信号，要么创建一个新线程。

最后，当应用使用完消息队列，它调用 `mq_close()` 库函数，传给它队列描述符。注意这个函数并不删除队列，这与 `close()` 系统调用不会删除文件一样。要删除队列，应用需要调用 `mq_unlink()` 函数。

在 Linux 2.6 中，POSIX 消息队列的实现是简单的。已经引入了一个叫做 *mqueue* 的特殊文件系统（参见第十二章“特殊文件系统”一节），每个现存队列在其中都有一个相应的索引节点。内核提供了几个系统调用：`mq_open()`、`mq_unlink()`、`mq_timedsend()`、`mq_timedreceive()`、`mq_notify()` 和 `mq_getsetattr()`。这些系统调用大略对应前面表 19-15 中的库函数。这些系统调用透明地对 *mqueue* 文件系统的文件进行操作，而大部分工作交由 VFS 层处理。例如，注意到内核不提供 `mq_close()` 函数，而事实上返回给应用的队列描述符实际上是一个文件描述符，因此 `mq_close()` 的工作由 `close()` 系统调用来做就可以了。

mqueue 特殊文件系统不能安装在系统目录树中。但是如果安装了，用户可以通过使用文件系统根目录中的文件来创建 POSIX 消息队列，也可以读入相应文件来得到队列的有关信息。最后，应用可以使用 `select()` 和 `poll()` 获得队列状态变化的通知。

每个队列有一个 `mqueue_inode_info` 描述符，它包含有 `inode` 对象，该对象与 *mqueue* 特殊文件系统的一个文件相对应。当 POSIX 消息队列系统调用接收一个队列描述符作为参数时，它就调用 VFS 的 `fget()` 函数计算出对应文件对象的地址。然后，系统调用得到 *mqueue* 文件系统中文件的索引节点对象。最后，就可以得到该索引节点对象所对应的 `mqueue_inode_info` 描述符地址。

队列中挂起的消息被收集到 `mqueue_inode_info` 描述符中的一个单向链表。每个消息由一个 `msg_msg` 类型的描述符来表示，这与 System V IPC 中使用的消息描述符是完全一样的（参见本章前面“IPC 消息”一节）。



程序的执行

第三章所描述的“进程”概念在 Unix 中是用来表示正在运行的一组程序竞争系统资源的行为。本章作为最后的一章，将集中讨论程序和进程之间的关系。我们会专门描述内核如何通过程序文件的内容建立进程的执行上下文。尽管把一组指令装入内存并让 CPU 执行看起来并不是什么大问题，但内核还必须灵活处理以下几方面的问题：

不同的可执行文件格式

Linux 的出色表现之一就是能执行为其他操作系统编译的二进制代码。具体地说，Linux 可以在 64 位版本的机器上执行 32 位可执行代码。例如，在 Pentium 上创建的可执行代码可以在 64 位的 AMD Opteron CPU 平台上运行。

共享库

很多可执行文件并不包含执行程序所需的所有代码，而是期望内核在运行时从共享库中加载函数。

执行上下文的其他信息

这包括程序员熟悉的命令行参数与环境变量。

程序是以可执行文件 (*executable file*) 的形式存放在磁盘上的，可执行文件既包括被执行函数的目标代码，也包括这些函数所使用的数据。程序中的很多函数是所有程序员都可使用的服务例程，它们的目标代码包含在所谓“库”的特殊文件中。实际上，一个库函数的代码或被静态地拷贝到可执行文件中（静态库），或在运行时被连接到进程（共享库，因为它们的代码由很多独立的进程所共享）。

当装入并运行一个程序时，用户可以提供影响程序执行方式的两种信息：命令行参数和

环境变量。用户在 shell 提示符下紧跟文件名输入的就是命令行参数。环境变量（例如 HOME 和 PATH）是从 shell 继承来的，但用户在装入并运行程序前可以修改任何环境变量。

我们在“可执行文件”一节将解释一个程序的执行上下文到底是什么。在“可执行格式”一节我们会提及一些 Linux 所支持的可执行格式，并说明 Linux 如何改变它的“个性”以执行其他操作系统所编译的程序。最后，在“exec 函数”一节会描述执行一个新程序的进程所需的系统调用。

可执行文件

在第一章中我们把进程定义为“执行上下文”。这就意味着进行特定的计算需要收集必要的信息，包括所访问的页，打开的文件，硬件寄存器的内容等等。可执行文件是一个普通文件，它描述了如何初始化一个新的执行上下文，也就是如何开始一个新的计算。

假定一位用户想在当前目录下显示文件，他知道在 shell 提示符下只要简单地敲入外部命令 `/bin/ls`（注 1）就可得到这个结果。命令 shell 创建一个新进程，新进程又调用系统调用 `execve()`（参看本章后面的“exec 函数”一节），其中传递的一个参数就是 `ls` 可执行文件的全路径名，在本例中即 `/bin/ls`。`sys_execve()` 服务例程找到相应的文件，检查可执行格式，并根据存放在其中的信息修改当前进程的执行上下文。因此，当这个系统调用终止时，新进程开始执行存放在可执行文件中的代码，也就是执行目录显示。

当进程开始执行一个新程序时，它的执行上下文发生很大的变化，这是因为在进程的前一个计算执行期间所获得的大部分资源会被抛弃。在前面的例子中，当进程开始执行 `/bin/ls` 时，它用 `execve()` 系统调用传递来的新参数代替 shell 的参数，并获得一个新的 shell 环境（参见后面的“命令行参数和 shell 环境”一节），从父进程继承的所有页（并通过写时复制机制实现共享）被释放，以便在一个新的用户态地址空间开始执行新的计算；甚至进程的特权都可能改变（参看后面的“进程的信任状和权能”一节）。然而，进程的 PID 不改变，并且新的计算从前一个计算继承所有打开的文件描述符，当然这些文件描述符是在执行 `execve()` 系统调用时还没有自动关闭的描述符（注 2）。

注 1： 在 Linux 中，可执行文件的路径不是固定的，这取决于所使用的发布版本。对于所有的 Unix 系统，已经提议了几个标准的命名模式，如 *FHS* (*F: lesystem Hierarchy standard*)。

注 2： 默认情况下，在发出 `execve()` 系统调用后由进程已经打开的文件仍然是打开的。但是，如果进程把 `files_struct` 结构的 `close_on_exec` 字段的相应位置置位，则文件自动关闭（参见第十二章的表 12-7）；这是通过 `fcntl()` 系统完成的。

进程的信任状和权能

从传统上看, Unix 系统与每个进程的一些信任状 (*credential*) 相关, 信任状把进程与一个特定的用户或用户组捆绑在一起。信任状在多用户系统上尤为重要, 因为信任状可以决定每个进程能做什么, 不能做什么, 这样既保证了每个用户的个人数据的完整性, 也保证了系统整体上的稳定性。

信任状的使用既需要在进程的数据结构方面给予支持, 也需要在被保护的资源方面给予支持。文件就是一种显而易见的资源。因此, 在 Ext2 文件系统中, 每个文件都属于一个特定的用户, 并被捆绑于某个用户组。文件的拥有者可以决定对某个文件允许哪些操作, 以在文件的拥有者、文件的用户组及其他所有用户之间做出区别。当某个进程试图访问一个文件时, VFS 总是根据文件的拥有者和进程的信任状所建立的许可权检查访问的合法性。

进程的信任状存放在进程描述符的几个字段中, 如表 20-1 所示。这些字段包括系统中用户和用户组的标识符, 与之可以相比较的通常是存放在所访问文件索引节点中的标识符。

表 20-1: 传统的进程信任状

名字	说明
uid, gid	用户和组的实际标识符
euid, egid	用户和组的有效标识符
fsuid, fsgid	文件访问的用户和组的有效标识符
groups	补充的组标识符
suid, sgid	用户和组保存的标识符

值为 0 的 UID 指定给 root 超级用户, 而值为 0 的用户 GID 指定给 root 超级组。只要有关进程的信任状存放了一个零值, 则内核将放弃权限检查, 始终允许这个进程做任何事情, 如涉及系统管理或硬件处理的那些操作, 而这些操作对于非特权进程是不允许的。

当一个进程被创建时, 总是继承父进程的信任状。不过, 这些信任状以后可以被修改, 这发生在当进程开始执行一个新程序时, 或者当进程发出合适的系统调用时。通常情况下, 进程的 uid、euid、fsuid 及 suid 字段具有相同的值。然而, 当进程执行 setuid 程序时, 即可执行文件的 setuid 标志被设置时, euid 和 fsuid 字段被置为这个文件拥有者的标识符。几乎所有的检查都涉及这两个字段中的一个: fsuid 用于与文件相关的操作, 而 euid 用于其他所有的操作。这也同样适用于组标识符的 gid、egid、fsgid 及 sgid 字段。

我们用一个例子来说明如何使用 `fsuid` 字段，考虑一下当用户想改变她的口令时的典型情况。所有的口令都存放在一个公共文件中，但用户不能直接编辑这样的文件，因为它是受保护的。因此，用户调用一个名为 `/usr/bin/passwd` 的系统程序，它可以设置 `setuid` 标志，而且它的拥有者是超级用户。当 shell 创建的进程执行这样一个程序时，进程的 `eid` 和 `fsuid` 字段被置为 0，即超级用户的 PID。现在，这个进程可以访问这个文件，因为当内核执行访问控制表时在 `fsuid` 字段发现了值 0。当然，`usr/bin/passwd` 程序除了让用户改变自己的口令外，并不允许做其他任何事情。

从 Unix 的历史发展可以得出一个教训，即 `setuid` 程序是相当危险的：恶意用户可以以这样的方式触发代码中的一些编程错误 (bug)，从而强迫 `setuid` 程序执行程序的最初设计者从未安排的操作。这可能常常危及整个系统的安全。为了减少这样的风险，Linux 与所有现代 Unix 操作系统一样，让进程只有在必要时才获得 `setuid` 特权，并在不需要时取消它们。可以证明，当以几个保护级别实现用户应用程序时，这种特点是很有用的。进程描述符包含一个 `suid` 字段，在 `setuid` 程序执行以后在该字段中正好存放有效标识符 (`eid` 和 `fsuid`) 的值。进程可以通过 `setuid()`、`setresuid()`、`setfsuid()` 和 `setreuid()` 系统调用改变有效标识符 (注 3)。

表 20-2 显示了这些系统调用是怎样影响进程的信任状的。请注意，如果调用进程还没有超级用户特权，即它的 `eid` 字段不为 0，那么，只能用这些系统调用来设置在这个进程的信任状字段已有的值。例如，一个普通用户进程可以通过调用系统调用 `setfsuid()` 强迫它的 `fsuid` 值为 500，但这只有在其他信任状字段中有一个字段已经有相同的值 500 时才可行。

表 20-2：设置进程信任状的系统调用语义

字段	setuid (e)		setresuid (u,e,s)	setreuid (u,e)	setfsuid (f)
	eid=0	eid ≠ 0			
uid	设置为 e	不改变	设置为 u	设置为 u	不改变
eid	设置为 e	设置为 e	设置为 e	设置为 e	不改变
fsuid	设置为 e	设置为 e	设置为 e	设置为 e	设置为 f
suid	设置为 e	不改变	设置为 s	设置为 e	不改变

为了理解四个用户 ID 字段之间的复杂关系，让我们考虑一下 `setuid()` 系统调用的效果。

注 3：通过发出相应的 `setgid()`、`setresgid()`、`setfsgid()` 和 `setregid()` 系统调用，可以改变组的有效信任状。

这些操作是不同的，这取决于调用者进程的 `euid` 字段是否被置为 0（即进程有超级用户特权）或被置为一个正常的 `UID`。

如果 `euid` 字段为 0，这个系统调用就把调用进程的所有信任状字段(`uid`、`euid`、`fsuid` 及 `suid`)置为参数 `e` 的值。超级用户进程因此就可以删除自己的特权而变为由普通用户拥有的一个进程。例如，在用户登录时，系统以超级用户特权创建一个新进程，但这个进程通过调用 `setuid()` 系统调用删除自己的特权，然后开始执行用户的 `login shell` 程序。

如果 `euid` 字段不为 0，那么这个系统调用只修改存放在 `euid` 和 `fsuid` 中的值，让其他两个字段保持不变。当运行 `setuid` 程序来提高和降低进程有效权限时（这些权限存放在 `euid` 和 `fsuid` 字段），该系统调用的这种功能是非常有用的。

进程的权能

POSIX.1e 草案（现已撤销）用“权能（capability）”一词引入进程信任状的另一种模型。Linux 内核支持 POSIX 权能，但是大部分 Linux 的发行版本不用它。

一种权能仅仅是一个标志，它表明是否允许进程执行一个特定的操作或一组特定的操作。这个模型不同于传统的“超级用户 VS 普通用户”模型，在后一种模型中，一个进程要么能做任何事情，要么什么也不能做，这取决于它的有效 `UID`。如表 20-3 所示，在 Linux 内核中已包含了很多权能。

表 20-3: Linux 的权能

名字	说明
<code>CAP_AUDIT_WRITE</code>	通过在 <code>netlink</code> 套接字进行写入而产生审计消息
<code>CAP_AUDIT_CONTROL</code>	通过 <code>netlink</code> 套接字控制内核审计操作
<code>CAP_CHOWN</code>	忽略对文件和组的拥有者进行改变的限制
<code>CAP_DAC_OVERRIDE</code>	忽略文件的访问许可权
<code>CAP_DAC_READ_SEARCH</code>	忽略文件 / 目录读和搜索的许可权
<code>CAP_FOWNER</code>	一般是忽略对文件拥有者的权限检查
<code>CAP_FSETID</code>	忽略对文件 <code>setid</code> 和 <code>setgid</code> 标志设置的限制
<code>CAP_KILL</code>	产生信号时绕过权限检查
<code>CAP_LINUX_IMMUTABLE</code>	允许修改仅追加和不可变的 <code>Ext2/Ext3</code> 文件
<code>CAP_IPC_LOCK</code>	允许页加锁和共享内存段加锁
<code>CAP_IPC_OWNER</code>	跳过 <code>IPC</code> 拥有者检查
<code>CAP_LEASE</code>	允许对文件进行租借（参看第十二章“Linux 文件加锁”一节）

表 20-3: Linux 的权能 (续)

名字	说明
CAP_MKNOD	允许有特权的 <code>mknod()</code> 操作
CAP_NET_ADMIN	允许一般的联网管理
CAP_NET_BIND_SERVICE	允许绑定到低于 1024 的 TCP/UDP 套接字
CAP_NET_BROADCAST	允许广播与组播
CAP_NET_RAW	允许使用 RAW 和 PACKET 套接字
CAP_SETGID	忽略对组进程信任状操作的限制
CAP_SETPCAP	允许对其他进程进行权能操作
CAP_SETUID	忽略对用户进程信任状操作的限制
CAP_SYS_ADMIN	允许一般的系统管理
CAP_SYS_BOOT	允许使用 <code>reboot()</code>
CAP_SYS_CHROOT	允许使用 <code>chroot()</code>
CAP_SYS_MODULE	允许内核模块的插入和删除
CAP_SYS_NICE	跳过 <code>nice()</code> 和 <code>setpriority()</code> 系统调用的权限检查, 并允许创建实时进程
CAP_SYS_PACCT	允许配置进程的记账
CAP_SYS_PTRACE	允许对任何进程使用 <code>ptrace()</code>
CAP_SYS_RAWIO	允许通过 <code>ioperm()</code> 和 <code>iopl()</code> 访问 I/O 端口
CAP_SYS_RESOURCE	允许增加资源限制
CAP_SYS_TIME	允许系统时钟和实时时钟的操作
CAP_SYS_TTY_CONFIG	允许配置终端并执行 <code>vhangup()</code> 系统调用

权能的主要优点是, 任何时候每个进程只需要有限种权能。因此, 即使有恶意的用户发现一种利用有潜在错误的程序的方法, 他也只能非法地执行有限个操作类型。

例如, 假定一个有潜在错误的程序只有 CAP_SYS_TIME 权能。在这种情况下, 利用其错误的恶意用户只能在非法地改变实时时钟和系统时钟方面获得成功。她并不能执行任何其他特权的操作。

不管是 VFS 还是 Ext2 文件系统目前都不支持权能模型, 所以, 当进程执行一个可执行文件时, 无法把这个文件与本该强加的一组权能联系起来。然而, 进程可以分别用 `capget()` 和 `capset()` 系统调用显式地获得和降低它的权能。例如, 完全可以通过修改 `login` 程序只保留其权能的一个子集而删除其他权能。

事实上, Linux 内核已经考虑权能。例如, 让我们考虑一下 `nice()` 系统调用, 它允许用户改变进程的静态优先级。在传统的模型中, 只有超级用户才能提升一个优先级, 内核因此应该检查调用进程描述符的 `euid` 字段是否为 0。然而, Linux 内核定义了一个名为 `CAP_SYS_NICE` 的权能, 正好对应着这种操作。内核通过调用 `capable()` 函数并把 `CAP_SYS_NICE` 值传给这个函数来检查这个标志的值。

正是由于一些“兼容性小巧程序”已被加入到内核代码中, 这种方法才起作用。每当一个进程把 `euid` 和 `fsuid` 字段设置为 0 时 (或者通过调用表 20-2 中的一个系统调用, 或者通过执行超级用户所拥有的 `setuid` 程序), 内核就设置进程的所有权能, 以便使所有的检查成功。类似地, 当进程把 `euid` 和 `fsuid` 字段重新置为进程拥有者的实际 UID 时, 内核检查进程描述符中的 `keep_capabilities` 标志, 并在该标志设置时删除进程的所有权能。进程可以调用 Linux 专有的 `prctl()` 系统调用来设置和重新设置 `keep_capabilities` 标志。

Linux 安全模块框架

在 Linux 2.6 中, 权能是与 Linux 安全模块 (LSM) 框架紧密结合在一起的。简单地说, LSM 框架允许开发人员定义几种可以选择的内核安全模型。

每个安全模型是由一组安全钩 (*security hook*) 实现的。安全钩是由内核调用的一个函数, 用于执行与安全有关的重要操作。钩函数决定一个操作是否可以执行。

钩函数存放在 `security_operations` 类型的表中。当前使用的安全模型钩表地址存放在 `security_ops` 变量中。内核默认使用 `dummy_security_ops` 表实现最小安全模型。表中的每个钩函数实际上去检查相应的权能(如果有)是否允许, 否则无条件返回 0 (允许操作)。

例如, `stime()` 和 `settimeofday()` 函数的服务例程在改变系统日期时间之前调用 `settime` 安全钩。`dummy_security_ops` 表指向相应的函数, 而该函数约束自己去检查当前进程是否有 `CAP_SYS_TIME` 的权能, 并相应地返回 0 或者 `-EPERM`。

Linux 内核更复杂的安全模型已经开发出来。一个广为人知的例子是由美国国家安全局开发的 `securtty_Enhanced Linux (SELinux)`。

命令行参数和 shell 环境

当用户键入一个命令时, 为满足这个请求而装入的程序可以从 shell 接收一些命令行参数 (*command-line argument*)。例如, 当用户键入命令:

```
$ ls -l /usr/bin
```

以获得在 `/usr/bin` 目录下的全部文件列表时，shell 进程创建一个新进程执行这个命令。这个新进程装入 `/bin/ls` 可执行文件。在这样做的过程中，从 shell 继承的大多数执行上下文被丢弃，但三个单独的参数 `ls`、`-l` 和 `/usr/bin` 依然保持。一般情况下，新进程可以接收任意多个参数。

传递命令行参数的约定依赖于所用的高级语言。在 C 语言中，程序的 `main()` 函数把传递给程序的参数个数和指向字符串指针数组的地址作为参数。下列原型形式化地表示了这种标准格式：

```
int main(int argc, char *argv[])
```

再回到前面的例子，当 `/bin/ls` 程序被调用时，`argc` 的值为 3，`argv[0]` 指向 `ls` 字符串，`argv[1]` 指向 `-l` 字符串，而 `argv[2]` 指向 `/usr/bin` 字符串。`argv` 数组的末尾处总以空指针来标记，因此，`argv[3]` 为 `NUL`。

在 C 语言中，传递给 `main()` 函数的第三个可选参数是包含环境变量的参数。环境变量用来定制进程的执行上下文，由此为用户或其他进程提供通用的信息，或者允许进程在执行 `execve()` 系统调用的过程中保持一些信息。

为了使用环境变量，`main()` 可以声明如下：

```
int main(int argc, char *argv[], char *envp[])
```

`envp` 参数指向环境串的指针数组，形式如下：

```
VAR_NAME=something
```

这里，`VAR_NAME` 表示一个环境变量的名字，而“=”后面的子串表示赋给变量的实际值。`envp` 数组的结尾用一个空指针标记，就像 `argv` 数组。`envp` 数组的地址存放在 C 库的 `environ` 全局变量中。

命令行参数和环境串都存放在用户态堆栈中，正好位于返回地址之前（参见第十章的“参数传递”一节）。图 20-1 显示了用户态堆栈的底部单元。注意，环境变量位于栈底附近正好在一个 0 长整数之后。

库

每个高级语言的源码文件都是经过几个步骤才转化为目标文件的，目标文件中包含的是汇编语言指令的机器代码，它们和相应的高级语言指令对应。目标文件并不能被执行，因为它不包含源代码文件所用的全局外部符号名的线性地址（例如库函数或同一程序中的其他源代码文件）。这些地址的分配或解析是由链接程序完成的，链接程序把程序所

有的目标文件收集起来并构造可执行文件。链接程序还分析程序所用的库函数，并以本章后面所描述的方式把它们粘合成可执行文件。

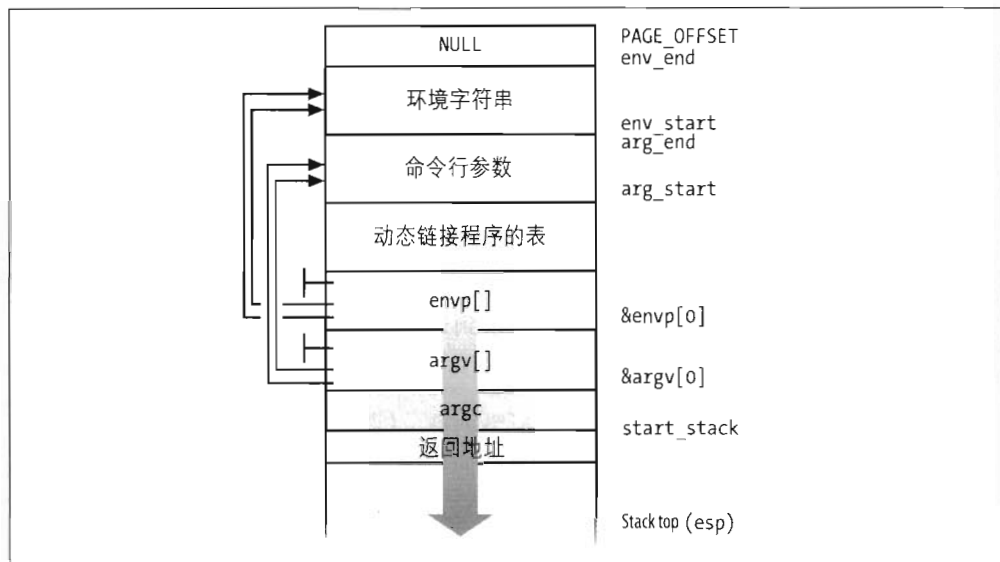


图 20-1：用户态堆栈的底部单元

大多数程序，甚至是最小的程序都会利用 C 库。例如，请看下面只有一行的 C 程序：

```
void main(void) { }
```

尽管这个程序没有做任何事情，但还是需要做很多工作来建立执行环境（参见本章后面的“exec函数”一节），并在程序终止时杀死这个进程（参见第三章的“撤消进程”一节）。尤其是当main()函数终止时，C 编译程序把 `exit_group()` 函数插入到目标代码中。

从第十章我们知道，程序通常通过 C 库中的封装例程调用系统调用。C 编译器亦如此。任何可执行文件除了包括对程序的语句进行编译所直接产生的代码外，还包括一些“粘合”代码来处理用户态进程与内核之间的交互。这样的粘合代码有一部分存放在 C 库中。

除了 C 库，Unix 系统中还包含很多其他的函数库。一般的 Linux 系统通常就有几百个不同的库。这里仅仅列举其中的两个：数学库 *libm* 包含浮点操作的基本函数，而 X11 库 *libX11* 收集了所有 X11 窗口系统图形接口的基本底层函数。

传统 Unix 系统中的所有可执行文件都是基于静态库（*static library*）的。这就意味着链接程序所产生的可执行文件不仅包括原程序的代码，还包括程序所引用的库函数的代码。

静态库的一大缺点是：它们占用大量的磁盘空间。的确，每个静态链接的可执行文件都复制库代码的某些部分。

现代 Unix 系统利用共享库 (*shared library*)。可执行文件不用再包含库的目标代码，而仅仅指向库名。当程序被装入内存执行时，一个名为动态链接器 (*dynamic linker*, 也叫 `ld.so`) 的程序就专注于分析可执行文件中的库名，确定所需库在系统目录树中的位置，并使执行进程可以使用所请求的代码。进程也可以使用 `dlopen()` 库函数在运行时装入额外的共享库。

共享库对提供文件内存映射的系统尤为方便，因为它们减少了执行一个程序所需的主内存量。当动态链接程序必须把某一共享库链接到进程时，并不拷贝目标代码，而是仅仅执行一个内存映射，把库文件的相关部分映射到进程的地址空间中。这就允许共享库机器代码所在的页框被使用同一代码的所有进程共享。显然，如果程序是静态链接的，那么共享是不可能的。

共享库也有一些缺点。动态链接的程序启动时间通常比静态链接的程序长。此外，动态链接的程序的移植性也不如静态链接的好，因为当系统中所包含的库版本发生变化时，动态链接的程序运行时就可能出现问题的。

用户可以始终请求一个程序被静态地链接。例如，GCC 编译器提供 `-static` 选项，即告诉链接程序使用静态库而不是共享库。

程序段和进程的线性区

从逻辑上说，Unix 程序的线性地址空间传统上被划分为几个叫做段 (*segment*) (注 4) 的区间：

正文段

包含程序的可执行代码。

已初始化数据段

包含已初始化的数据，也就是初值存放在可执行文件中的所有静态变量和全局变量（因为程序在启动时必须知道它们的值）。

未初始化数据段 (*bss* 段)

包含未初始化的数据，也就是初值没有存放在可执行文件中的所有全局变量（因为程序在引用它们之前才赋值）；历史上把这个段叫做 *bss* 段。

注 4：“段”这个术语有其历史根源，因为第一个 Unix 系统用不同的段寄存器实现每一个线性地址区间。不过，Linux 并不利用 80x86 微处理器的段机制实现程序分段。

堆栈段

包含程序的堆栈，堆栈中有返回地址、参数和被执行函数的局部变量。

每个 `mm_struct` 内存描述符（参见第九章中的“内存描述符”一节）都包含一些字段来标识相应进程特定线性区的作用：

`start_code, end_code`

程序的源代码所在线性区的起始和终止线性地址，即可执行文件中的代码。

`start_data, end_data`

程序的初始化数据所在线性区的起始和终止线性地址，正如在可执行文件中所指定的那样。这两个字段指定的线性区大体上与数据段对应。

`start_brk, brk`

存放线性区的起始和终止线性地址，该线性区包含动态分配给进程的内存区（参看第九章的“堆的管理”一节）。有时把这部分线性区叫做堆（*heap*）。

`start_stack`

正好在 `main()` 的返回地址之上的地址。如图 20-1 所示，更高的地址被保留（回想一下，栈是向低地址增长）。

`arg_start, arg_end`

命令行参数所在的堆栈部分的起始地址和终止地址。

`env_start, env_end`

环境串所在的堆栈部分的起始地址和终止地址。

注意，共享库和文件的内存映射使得基于程序段的进程地址空间分类有点过时，因为每个共享库被映射到与前面所讨论的线性区不同的线性区。

灵活线性区布局

灵活线性区布局（*flexible memory region layout*）在内核版本 2.6.9 中引入：实际上，每个进程是按照用户态堆栈预期的增长量来进行内存布局的。但是仍然可以使用老的经典布局（主要用于：当内核无法限制进程用户态堆栈的大小时）。表 20-4 以 80x86 结构的默认用户态地址空间为例描述了这两种布局，地址空间最大可以到 3GB。

正如你所看到的，布局之间只在文件内存映射与匿名映射时线性区的位置上有区别。在经典布局下，这些区域从整个用户态地址空间的 1/3 开始，通常在地址 0x40000000。新的区域往更高线性地址追加，因此，这些区域往用户态堆栈方向扩展。

表 20-4：80x86结构的线性区布局

线性区种类	经典布局	灵活布局
正文段 (ELF)	起自：0x08048000	
数据与 bss 段	起自：紧接正文段之后	
堆	起自：紧接数据与 bss 段之后	
文件内存映射与匿名线性区	起自：0x40000000 (该地址对应于整个用户态地址空间的 1/3)，库连续往高地址追加	起自：紧接用户态堆栈尾（最小地址），库连续往低地址追加
用户态堆栈	起自：0xc0000000 并向低地址增长	

而相反的是,在灵活布局中,文件内存映射与匿名映射的线性区是紧接用户态堆栈尾的。新的区域往更低线性地址追加,因此,这些区域往堆的方向扩展。记住,堆栈也是连续往低地址追加的。

当内核能通过 RLIMIT_STACK 资源限制来限定用户态堆栈的大小时,通常使用灵活布局 (参见第三章“进程资源限制”一节)。这个限制确定了为堆栈保留的线性地址空间大小。但是这个空间大小不能小于 128MB 或大于 2.5GB。

另外,如果 RLIMIT_STACK 资源限制设为无限 (infinity),或者系统管理员将 sysctl_legacy_va_layout 变量设为 1 (通过修改 /proc/sys/vm/legacy_va_layout 文件或调用相应的 sysctl() 系统调用实现),内核无法确定用户态堆栈的上限,就仍然使用经典线性区布局。

为什么引入灵活布局? 其主要优点是可以允许进程更好地使用用户态线性地址空间。在经典布局中,堆的限制是小于 1GB,而其他线性区可以使用到约 2GB (要减去堆栈大小)。在灵活布局中,这些限制没有了,堆和其他线性区可以自由扩展,可以使用除了用户态堆栈和程序用固定大小的段以外的所有线性地址空间。

现在,一个实用的小试验很有启发意义。让我们录入和编译下面的 C 程序:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    char cmd[32];
    brk((void *)0x8051000);
    sprintf(cmd, "cat /proc/self/maps");
    system(cmd);
}
```

```
    return 0;
}
```

实际上，程序将它的进程堆变大（参见第九章“堆的管理”一节），然后在 */proc* 特殊文件系统下读入 *maps* 文件，该文件产生进程自身的线性区清单。

让我们对堆栈大小不加任何限制并运行程序：

```
# ulimit -s unlimited; /tmp/memorylayout
08048000-08049000 r-xp 00000000 03:03 5042408 /tmp/memorylayout
08049000-0804a000 rwxp 00000000 03:03 5042408 /tmp/memorylayout
0804a000-08051000 rwxp 0804a000 00:00 0
40000000-40014000 r-xp 00000000 03:03 620801 /lib/ld-2.3.2.so
40014000-40015000 rwxp 00013000 03:03 620801 /lib/ld-2.3.2.so
40015000-40016000 rwxp 40015000 00:00 0
4002f000-40157000 r-xp 00000000 03:03 620804 /lib/libc-2.3.2.so
40157000-4015b000 rwxp 00128000 03:03 620804 /lib/libc-2.3.2.so
4015b000-4015e000 rwxp 4015b000 00:00 0
bffe000-c0000000 rwxp bffe0000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

（由于C编译器的版本不同与程序链接方式不同，见到的结果可能略有不同。）前两个十六进制数表示线性区的范围，后面是权限标志。最后面是线性区映射的文件的有关信息，如果有信息就是：文件内的开始偏移量、块设备号、索引节点号和文件名。

请注意，列出的所有区域是由私有内存映射实现的（权限列的p字母）。这并不奇怪，因为这些线性区是只为进程提供数据而存在的。当执行指令时，进程可以修改这些线性区的内容，但是与它们相关的磁盘文件会保持不变。私有内存映射就具有如此作用。

从0x08048000开始的线性区是与 */tmp/memorylayout* 文件的0~4095字节部分对应的内存映射。而相应的权限表示是可执行的（它包含了目标代码）、只读的（因为指令在执行期间是不改变的，因此不可写）和私有的。这很正确，这是程序正文段的映射区域。

从0x08049000开始的线性区也是与 */tmp/memorylayout* 文件的0~4095字节部分对应的另一个内存映射。这个程序太小，以至于程序的正文、数据和bss段都在同一个文件页里。因此，包含数据段和bss段的线性区与上一个线性区在线性地址空间是重叠的。

第三个线性区包含进程的堆。注意，它在线性地址0x08051000处终止，传递给 *brk()* 系统调用的就是该地址。

接下来从0x40000000和0x40014000开始的两个线性区，分别对应这个系统ELF共享库（*/lib/ld-2.3.2.so*）动态链接程序的正文段和数据、bss段。动态链接程序决不单独执行，它总是以内存映射的方式映射到执行其他程序的进程地址空间内。从0x40015000开始的匿名线性区已由动态链接程序分配。

在这个系统上，C库正好存放在文件 `/lib/libc-2.3.2.so` 中。C库的正文段和数据、bss段被映射到从 `0x4002f000` 地址开始的两个线性区。还记得私有区域所在的页框，只要没被修改，就可以通过写时复制机制在几个进程间共享。因此，因为正文段是只读的，所示包含C库执行代码的页框几乎在所有当前运行进程间共享（除了静态链接程序）。从 `0x4015b000` 开始的匿名线性区已由C库分配。

从 `0xbffeb000` 到 `0xc0000000` 的匿名内存区对应于用户态堆栈。我们在第九章“缺页异常处理程序”一节已讨论过堆栈是如何在必要时自动地向低地址方向扩展的。

最后，从 `0xffffe000` 开始的单页匿名线性区包含进程的 `vsyscall` 页，当发出系统调用和从信号处理程序返回时会访问该区域（参见第十章“通过 `sysenter` 指令发出系统调用”一节和第十一章“捕获信号”一节）。

现在我们对用户态堆栈大小施加限制后再运行该程序：

```
# ulimit -s 100; /tmp/memorylayout
08048000-08049000 r-xp 00000000 03:03 5042408      /tmp/memorylayout
08049000-0804a000 rwxp 00000000 03:03 5042408      /tmp/memorylayout
0804a000-08051000 rwxp 0804a000 00:00 0
b7ea3000-b7fcb000 r-xp 00000000 03:03 620804      /lib/libc-2.3.2.so
b7fcb000-b7fcf000 rwxp 00128000 03:03 620804      /lib/libc-2.3.2.so
b7fcf000-b7fd2000 rwxp b7fcf000 00:00 0
b7feb000-b7fec000 rwxp b7feb000 00:00 0
b7fec000-b8000000 r-xp 00000000 03:03 620801      /lib/ld-2.3.2.so
b8000000-b8001000 rwxp 00013000 03:03 620801      /lib/ld-2.3.2.so
bffeb000-c0000000 rwxp bffeb000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

我们注意到布局发生了变化，即在最高堆栈地址之上为动态链接程序映射了一个约128MB的区域。而且，因为C库的线性区在稍后创建，所以就得到一个较低的线性地址。

执行跟踪

执行跟踪 (*execution tracing*) 是一个程序监视另一个程序执行的一种技术。被跟踪的程序一步一步地执行，直到接收到一个信号或调用一个系统调用。执行跟踪由调试程序 (debugger) 广泛使用，当然还使用其他技术（包括在被调试程序中插入断点及运行时访问它的变量）。与往常一样，我们将集中讨论内核怎样支持执行跟踪，而不讨论调试程序怎样工作。

在Linux中，通过 `ptrace()` 系统调用进行执行跟踪，这个系统调用能处理如表20-5所示的命令。设置了 `CAP_SYS_PTRACE` 权能的进程可以跟踪系统中的任何进程（除了 `init`）。相反，没有 `CAP_SYS_PTRACE` 权能的进程 *P* 只能跟踪与 *P* 有相同属主的进程。此外，两个进程不能同时跟踪一个进程。

表 20-5: 80x86结构的 ptrace 命令

命令	说明
PTRACE_ATTACH	对另一个进程开始执行跟踪
PTRACE_CONT	重新恢复执行
PTRACE_DETACH	终止执行跟踪
PTRACE_GET_THREAD_AREA	代表被跟踪进程获得线程局部存储区 (TLS)
PTRACE_GETEVENTMSG	从被跟踪进程获得附加数据 (如, 新创建进程的 PID)
PTRACE_GETFPREGS	读浮点寄存器
PTRACE_GETFPXREGS	读 MMX 和 XMM 寄存器
PTRACE_GETREGS	读有特权的 CPU 的寄存器
PTRACE_GETSIGINFO	获得传给被跟踪进程最后一条信号的信息
PTRACE_KILL	删除被跟踪的进程
PTRACE_OLDSETOPTIONS	依赖于结构的命令, 等价于 PTRACE_SETOPTIONS
PTRACE_PEEKDATA	从数据段读一个 32 位值
PTRACE_PEEKTEXT	从文本段读一个 32 位值
PTRACE_PEEKUSR	读 CPU 的普通和调试寄存器
PTRACE_POKEDATA	把一个 32 位值写入数据段
PTRACE_POKETEXT	把一个 32 位值写入正文段
PTRACE_POKEUSR	写 CPU 的普通和调试寄存器
PTRACE_SET_THREAD_AREA	代表被跟踪进程设置线程局部存储区 (TLS)
PTRACE_SETFPREGS	写浮点寄存器
PTRACE_SETFPXREGS	写 MMX 和 XMM 寄存器
PTRACE_SETOPTIONS	修改 ptrace() 的行为
PTRACE_SETREGS	写有特权的 CPU 寄存器
PTRACE_SETSIGINFO	建立传给被跟踪进程最后一条信号的信息
PTRACE_SINGLESTEP	恢复单条汇编指令的执行
PTRACE_SYSCALL	恢复执行直到下一个系统调用的边界
PTRACE_TRACEME	对当前进程开始执行跟踪

ptrace() 系统调用修改被跟踪进程描述符的 parent 字段以使它指向跟踪进程, 因此, 跟踪进程变为被跟踪进程的有效父进程。当执行跟踪终止时, 也就是当以 PTRACE_DETACH 命令调用 ptrace() 时, 这个系统调用把 p_pptr 设置为 real_parent 的值, 恢复被跟踪进程原来的父进程 (参见第三章的“进程之间的关系”一节)。

与被跟踪程序相关的几个监控事件为：

- 一条单独汇编指令执行的结束
- 进入系统调用
- 退出系统调用
- 接收到一个信号

当一个监控的事件发生时，被跟踪的程序停止，并且将 SIGCHLD 信号发送给它的父进程。当父进程希望恢复子进程的执行时，就使用 PTRACE_CONT、PTRACE_SINGLESTEP 和 PTRACE_SYSCALL 命令中的一条命令，这取决于父进程要监控哪种事件。

PTRACE_CONT 命令只继续执行，子进程将一直执行到收到另一个信号。这种跟踪是通过进程描述符的 ptrace 字段中的 PF_PTRACED 标志实现的，而这个标志的检查是由 do_signal() 函数进行的（参看第十一章中的“传递信号”一节）。

PTRACE_SINGLESTEP 命令强迫子进程执行下一条汇编语言指令，然后又停止它。这种跟踪是基于 80x86 机器的 eflags 寄存器的 TF 陷阱标志而实现的。当这个标志为 1 时，在任一条汇编语言指令之后正好产生一个“Debug”异常。相应的异常处理程序只是清除掉这个标志，强迫当前进程停止，并发送 SIGCHLD 信号给父进程。注意，设置 TF 标志并不是特权操作，因此用户态进程即使在没有 ptrace() 系统调用的情况下，也能强迫单步执行。内核检查进程描述符中的 PT_DTRACE 标志，以跟踪子进程是否通过 ptrace() 进行单步执行。

PTRACE_SYSCALL 命令使被跟踪的进程重新恢复执行，直到一个系统调用被调用。进程停止两次，第一次是在系统调用开始时，第二次是在系统调用终止时。这种跟踪是利用进程描述符中的 TIF_SYSCALL_TRACE 标志实现的。这个标志是在进程 thread_info 结构的 flags 字段中，并在 system_call() 汇编语言的函数中被检查（参见第十章“通过 int \$0 × 80 指令发出系统调用”一节）。

也可以利用 Intel Pentium 处理器的一些调试特点来跟踪进程。例如，父进程使用 PTRACE_POKEUSR 命令为子进程设置 dr0, ..., dr7 调试寄存器的值。当由某调试寄存器监控的事情发生时，CPU 产生“Debug”异常，异常处理程序然后挂起被调试的进程并给父进程发送 SIGCHLD 信号。

可执行格式

Linux 标准的可执行格式是 ELF(*Executable and Linking Format*)，它由 Unix 系统实验

室开发并在 Unix 世界相当流行。几个著名的 Unix 操作系统（如 System V Release 4 和 Sun 的 Solaris 2）都把 ELF 作为它们的主要可执行格式。

Linux 的旧版支持另一种名叫 Assembler OUTput Format (a.out) 的格式，实际上，在 Unix 世界有好几种版本使用这种格式。因为现在 ELF 非常实用，因此已经很少用 a.out 格式。

Linux 支持很多其他不同格式的可执行文件。在这种方式下，Linux 能运行为其他操作系统所编译的程序，如 MS-DOS 的 EXE 程序，或 BSD Unix 的 COFF 可执行格式。有几种可执行格式，如 Java 或 bash 脚本，是与平台无关的。

由类型为 linux_binfmt 的对象所描述的可执行格式实质上提供以下三种方法：

load_binary

通过读存放在可执行文件中的信息为当前进程建立一个新的执行环境。

load_shlib

用于动态地把一个共享库捆绑到一个已经在运行的进程，这是由 uselib() 系统调用激活的。

core_dump

在名为 core 的文件中存放当前进程的执行上下文。这个文件通常是在进程接收到一个缺省操作为“dump”的信号时被创建的，其格式取决于被执行程序的可执行类型（参见第十一章的“传递信号之前所执行的操作”一节）。

所有的 linux_binfmt 对象都处于一个单向链表中，第一个元素的地址存放在 formats 变量中。可以通过调用 register_binfmt() 和 unregister_binfmt() 函数在链表中插入和删除元素。在系统启动期间，为每个编译进内核的可执行格式都执行 register_binfmt() 函数。当实现了一个新的可执行格式的模块正被装载时，也执行这个函数，当模块被卸载时，执行 unregister_binfmt() 函数。

在 formats 链表中的最后一个元素总是对解释脚本 (*interpreted script*) 的可执行格式进行描述的一个对象。这种格式只定义了 load_binary 方法。其相应的 load_script() 函数检查这种可执行文件是否以两个 #! 字符开始。如果是，这个函数就把第一行的其余部分解释为另一个可执行文件的路径名，并把脚本文件名作为参数传递以执行它（注 5）。

注 5： 只要以用户 shell 能识别的语言把文件写入脚本文件，即使不以 #! 字符开始，也可能执行这个脚本文件。但是，在这种情况下，用 shell（用户在 shell）或缺省的 Bourne shell sh 来对这种脚本进行解释，因此并不直接涉及内核。

Linux 允许用户注册自己定义的可执行格式。对这种格式的识别或者通过存放在文件前 128 字节的魔数，或者通过表示文件类型的扩展名。例如，MS-DOS 的扩展名由“.”把三个字符从文件名中分离出来：*.exe* 扩展名标识可执行文件，而 *.bat* 扩展名标识 shell 脚本。

当内核确定可执行文件是自定义格式时，它就启动相应的解释程序(*interpreter program*)。解释程序运行在用户态，读入可执行文件的路径名作为参数，并执行计算。例如，包含 Java 程序的可执行文件就由 Java 虚拟机（如 */usr/lib/java/bin/java*）来解释。

这种机制与脚本格式类似，但功能更加强大，这是因为它对自定义格式不加任何限制。要注册一个新格式，就必须在 *binfmt_misc* 特殊文件系统（通常在 */proc/sys/fs/binfmt_misc*）的注册文件内写入一个字符串，其格式如下：

```
:name:type:offset:string:mask:interpreter:flags
```

这里，每个字段的含义如下：

name

新格式的标识符。

type

识别类型（M 表示魔数，E 表示扩展）。

offset

魔数在文件中的起始偏移量。

string

以魔数或者以扩展名匹配的字节序列。

mask

用来屏蔽掉 string 中的一些位的字符串。

interpreter

解释程序的完整路径名。

flags

可选标志，控制必须怎样调用解释程序。

例如，超级用户执行的下列命令将使内核识别出 Microsoft Windows 的可执行格式：

```
$ echo ':DOSWin:M:0:MZ:0xff:/usr/bin/wine:'  
> /proc/sys/fs/binfmt_misc/register
```

Windows 可执行文件的前两个字节是魔数 MZ，由解释程序 */usr/bin/wine* 执行这个可执行文件。

执行域

在第一章已提到，Linux 的一个巧妙的特点就是能执行其他操作系统所编译的程序。当然，只有内核运行的平台与可执行文件包含的机器代码对应的平台相同时这才是可能的。对这些“外来”程序提供两种支持：

- 模拟执行 (emulated execution)：程序中包含的系统调用与 POSIX 不兼容时才有必要执行这种程序。
- 原样执行 (native execution)：只有程序中所包含的系统调用完全与 POSIX 兼容时才有效。

Microsoft MS-DOS和Windows程序是被模拟执行的，因为它们包含的API不能被Linux所认识，因此不能原样执行。像 DOSEmu 或 Wine 这样的模拟程序（出现在上一节末尾的例子中）被调用来把每个 API 调用转换为一个模拟的封装函数调用，而封装函数调用又使用现有的 Linux 系统调用。因为模拟程序主要是作为用户态的应用程序来执行，因此我们在此不做进一步的讨论。

另一方面，不用太费力就可以执行为其他操作系统编译的与POSIX兼容的程序，因为与POSIX兼容的操作系统都提供了类似的API（尽管实际上并不总是这种情况，但API应该相同）。内核必须消除的细微差别通常涉及如何调用系统调用或如何给各种信号编号。这种信息存放在类型为exec_domain的执行域描述符(execution domain descriptor)中。

进程可以指定它的执行域，这是通过设置进程描述符的personality字段，以及把相应exec_domain数据结构的地址存放到thread_info结构的exec_domain字段来实现的。进程可以通过发布一个叫做personality()的系统调用来改变它的个性(personality)；表20-6列出了这个系统调用的参数所接收的典型值。程序员通常不希望直接改变其程序的个性；相反，应该通过建立进程的执行上下文的“粘合”代码来发出Personality()系统调用（参见下一节）。

表 20-6：Linux 内核所支持的主要个性

个性	操作系统
PER_LINUX	标准执行域
PER_LINUX_32BIT	Linux，64 位结构中 32 位物理地址
PER_LINUX_FDPIC	Linux 程序，格式为 ELF FDPIC
PER_SVR4	System V Release 4
PER_SVR3	System V Release 3
PER_SCOSVR3	SCO Unix Version 3.2

表 20-6: Linux 内核所支持的主要个性 (续)

个性	操作系统
PER_OSR5	SCO OpenServer Release 5
PER_WYSEV386	Unix System V/386 Release 3.2.1
PER_ISCR4	交互式 Unix
PER_BSD	BSD Unix
PER_SUNOS	SunOS
PER_XENIX	Xenix
PER_LINUX32	64 位结构中 32 位 Linux 程序模拟 (使用 4GB 用户态地址空间)
PER_LINUX32_3GB	64 位结构中 32 位 Linux 程序模拟 (使用 3GB 用户态地址空间)
PER_IRIX32	SGI Irix-5 32 位
PER_IRIXN32	SGI Irix-6 32 位
PER_IRIX64	SGI Irix-6 64 位
PER_RISCOS	RISC OS
PER_SOLARIS	Sun 的 Solaris
PER_UW7	SCO(正式为 Caldera)的 UnixWare 7
PER_OSF4	Digital UNIX (Compaq Tru64 UNIX)
PER_HPUX	HP 的 HP-UX

exec 函数

Unix 系统提供了一系列函数, 这些函数能用可执行文件所描述的新上下文代替进程的上下文。这样的函数名以前缀 `exec` 开始, 后跟一个或两个字母, 因此, 家族中的一个普通函数被当作 `exec` 函数来引用。

表 20-7 中列出了 `exec` 函数, 它们之间的差别在于如何解释参数。

表 20-7: `exec` 函数

函数名	路径搜索	命令行参数	环境数组
<code>execl()</code>	否	列表	否
<code>execlp()</code>	是	列表	否
<code>execle()</code>	否	列表	是
<code>execv()</code>	否	数组	否

表 20-7: exec 函数 (续)

函数名	路径搜索	命令行参数	环境数组
execvp()	是	数组	否
execve()	否	数组	是

每个函数的第一个参数表示被执行文件的路径名。路径名可以是绝对路径或是当前进程目录的相对路径。此外，如果路径名中不包含 “/” 字符，execlp()和 execvp()函数就在 PATH 环境变量指定的所有目录中搜索这个可执行文件。

除了第一个参数，execl()、execlp()和 execl()函数包含的其他参数个数都是可变的。每个参数指向一个字符串，这个字符串是对新程序命令行参数的描述，正如函数名中 “l” 字符所隐含的一样，这些参数组织成一个列表（最后一个值为 NULL）。通常情况下，第一个命令行参数复制可执行文件名。相反，execv()、execvp()和 execve()函数指定单个参数的命令行参数，正如函数名中的 “v” 字符所隐含的一样，这单个参数是指向命令行参数串的指针向量地址。数组的最后一个元素必须存放 NULL 值。

execl()和 execve()函数的最后一个参数是指向环境串的指针数组的地址；数组的最后一个元素照样必须为 NULL。其他函数对新程序环境参数的访问是通过 C 库定义的外部全局变量 environ 进行的。

所有的 exec 函数（除 execve()外）都是 C 库定义的封装例程，并利用了 execve()系统调用，这是 Linux 所提供的处理程序执行的唯一系统调用。

sys_execve()服务例程接收下列参数：

- 可执行文件路径名的地址（在用户态地址空间）。
- 以 NULL 结束的字符串指针数组的地址（在用户态地址空间）。每个字符串表示一个命令行参数。
- 以 NULL 结束的字符串指针数组的地址（也在用户态地址空间）。每个字符串以 NAME=value 形式表示一个环境变量。

sys_execve()把可执行文件路径名拷贝到一个新分配的页框。然后调用 do_execve()函数，传递给它的参数为指向这个页框的指针、指针数组的指针及把用户态寄存器内容保存到内核态堆栈的位置。do_execve()依次执行下列操作：

1. 动态地分配一个linux_binprm数据结构,并用新的可执行文件的数据填充这个结构。

2. 调用 `path_lookup()`、`dentry_open()` 和 `path_release()`，以获得与可执行文件相关的目录项对象、文件对象和索引节点对象。如果失败，则返回相应的错误码。
3. 检查是否可以由当前进程执行该文件，再检查索引节点的 `i_writecount` 字段，以确定可执行文件没被写入；把 -1 存放在这个字段以禁止进一步的写访问。
4. 在多处理器系统中，调用 `sched_exec()` 函数来确定最小负载 CPU 以执行新程序，并把当前进程转移过去（参见第七章）。
5. 调用 `init_new_context()` 检查当前进程是否使用自定义局部描述符表，参见第二章“Linux LDT”一节）。如果是，函数为新程序分配和准备一个新的 LDT。
6. 调用 `prepare_binprm()` 函数填充 `linux_binprm` 数据结构，这个函数又依次执行下列操作：
 - a. 再一次检查文件是否可执行（至少设置一个执行访问权限）。如果不可执行，则返回错误码（因为带有 `CAP_DAC_OVERRIDE` 权能的进程总能通过检查，所以第 3 步中的检查还不够。参见本章前面“进程的信任状和权能”一节）。
 - b. 初始化 `linux_binprm` 结构的 `e_uid` 和 `e_gid` 字段，考虑可执行文件的 `setuid` 和 `setgid` 标志的值。这些字段分别表示有效的用户 ID 和组 ID。也要检查进程的权能（在本章前面的“进程的信任状和权能”一节中介绍了兼容性技巧）。
 - c. 用可执行文件的前 128 字节填充 `linux_binprm` 结构的 `buf` 字段。这些字节包含的是适合于识别可执行文件格式的一个魔数和其他信息。
7. 把文件路径名、命令行参数及环境串拷贝到一个或多个新分配的页框中（最终，它们会被分配给用户态地址空间）。
8. 调用 `search_binary_handler()` 函数对 `formats` 链表进行扫描，并尽力应用每个元素的 `load_binary` 方法，把 `linux_binprm` 数据结构传递给这个函数。只要 `load_binary` 方法成功应答了文件的可执行格式，对 `formats` 的扫描就终止。
9. 如果可执行文件格式不在 `formats` 链表中，就释放所分配的所有页框并返回错误码 `-ENOEXEC`，表示 Linux 不认识这个可执行文件格式。
10. 否则，函数释放 `linux_binprm` 数据结构，返回从这个文件可执行格式的 `load_binary` 方法中所获得的代码。

可执行文件格式对应的 `load_binary` 方法执行下列操作（我们假定这个可执行文件所在的文件系统允许文件进行内存映射并需要一个或多个共享库）：

1. 检查存放在文件前 128 字节中的一些魔数以确认可执行格式。如果魔数不匹配，则返回错误码 `-ENOEXEC`。

2. 读可执行文件的首部。这个首部描述程序的段和所需的共享库。
3. 从可执行文件获得动态链接程序的路径名,并用它来确定共享库的位置并把它们映射到内存。
4. 获得动态链接程序的目录项对象(也就获得了索引节点对象和文件对象)。
5. 检查动态链接程序的执行许可权。
6. 把动态链接程序的前 128 字节拷贝到缓冲区。
7. 对动态链接程序类型执行一些一致性检查。
8. 调用 `flush_old_exec()` 函数释放前一个计算所占用的几乎所有资源。这个函数又依次执行下列操作:
 - a. 如果信号处理程序的表为其他进程所共享,那么就分配一个新表并把旧表的引用计数器减 1,而且它将进程从旧的线程组脱离(参见第三章“标识一个进程”一节)。这是通过调用 `de_thread()` 函数完成的。
 - b. 如果与其他进程共享,就调用 `unshare_files()` 函数拷贝一份包含进程已打开文件的 `files_struct` 结构。
 - c. 调用 `exec_mmap()` 函数释放分配给进程的内存描述符、所有线性区及所有页框,并清除进程的页表。
 - d. 将可执行文件路径名赋给进程描述符的 `comm` 字段。
 - e. 调用 `flush_thread()` 函数清除浮点寄存器的值和 `TSS` 段保存的调试寄存器的值。
 - f. 调用 `flush_signal_handlers()` 函数,用于将每个信号恢复为默认操作,从而更新信号处理程序的表。
 - g. 调用 `flush_old_files()` 函数关闭所有打开的文件,这些打开的文件在进程描述符的 `files->close_on_exec` 字段设置了相应的标志(参见第十二章中的“与进程相关的文件”一节)(注 6)。

现在,我们已经不能返回了:如果真出了差错,这个函数再不能恢复前一个计算。

9. 清除进程描述符的 `PF_FORKNOEXEC` 标志。这个标志用于在进程创建时设置进程记账,在执行一个新程序时清除进程记账。
10. 设立进程新的个性,即设置进程描述符的 `personality` 字段。

注 6: 可以通过 `fcntl()` 系统调用来读取和修改这些标志。

11. 调用 `arch_pick_mmap_layout()`，以选择进程线性区的布局（参见本章前面“程序段与进程的线性”一节）。
12. 调用 `setup_arg_pages()` 函数为进程的用户态堆栈分配一个新的线性区描述符，并把那个线性区插入到进程的地址空间。`setup_arg_pages()` 还把命令行参数和环境变量串所在的页框分配给新的线性区。
13. 调用 `do_mmap()` 函数创建一个新线性区来对可执行文件正文段（即代码）进行映射。这个线性区的起始线性地址依赖于可执行文件的格式，因为程序的可执行代码通常是不可重定位的。因此，这个函数假定从某一特定逻辑地址的偏移量开始（因此就从某一特定的线性地址开始）装入正文段。ELF 程序被装入的起始线性地址为 `0x08048000`。
14. 调用 `do_mmap()` 函数创建一个新线性区来对可执行文件的数据段进行映射。这个线性区的起始线性地址也依赖于可执行文件的格式，因为可执行代码希望在特定的偏移量（即特定的线性地址）处找到它自己的变量。在 ELF 程序中，数据段正好被装在正文段之后。
15. 为可执行文件的其他专用段分配另外的线性区，通常是无。
16. 调用一个装入动态链接程序的函数。如果动态链接程序是 ELF 可执行的，这个函数就叫做 `load_elf_interp()`。一般情况下，这个函数执行第 12~14 步的操作，不过要用动态链接程序代替被执行的文件。动态链接程序的正文段和数据段在线性区的起始线性地址是由动态链接程序本身指定的，但它们处于高地址区（通常高于 `0x40000000`），这是为了避免与被执行文件的正文段和数据段所映射的线性区发生冲突（参见前面的“程序段和进程的线性区”一节）。
17. 把可执行格式的 `linux_binfmt` 对象的地址存放在进程描述符的 `binfmt` 字段中。
18. 确定进程的新权限。
19. 创建特定的动态链接程序表并把它们存放在用户态堆栈，如图 20-1 所示，这些表处于命令行参数和指向环境串的指针数组之间。
20. 设置进程的内存描述符的 `start_code`、`end_code`、`start_data`、`end_data`、`start_brk`、`brk` 及 `start_stack` 字段。
21. 调用 `do_brk()` 函数创建一个新的匿名线性区来映射程序的 `bss` 段（当进程写入一个变量时，就触发请求调页，进而分配一个页框）。这个线性区的大小是在可执行程序被链接时就计算出来的。因为程序的可执行代码通常是不可重新定位的，因此，必须指定这个线性区的起始线性地址。在 ELF 程序中，`bss` 段正好装在数据段之后。

22. 调用`start_thread()`宏修改保存在内核态堆栈但属于用户态寄存器的`eip`和`esp`的值,以使它们分别指向动态链接程序的入口点和新的用户态堆栈的栈顶。
23. 如果进程正被跟踪,就通知调试程序`execve()`系统调用已完成。
24. 返回0值(成功)。

当`execve()`系统调用终止且调用进程重新恢复它在用户态的执行时,执行上下文被大幅度改变,调用系统调用的代码不复存在。从这个意义上看,我们可以说`execve()`从未成功返回。取而代之的是,要执行的新程序已被映射到进程的地址空间。

但是,新程序还不能执行,因为动态链接程序还必须考虑共享库的装载(注7)。

尽管动态链接程序运行在用户态,但我们还要在这里简要概述一下它是如何运作的。它的第一个工作就是从内核保存在用户态堆栈的信息(处于环境串指针数组和`arg_start`之间)开始,为自己建立一个基本的执行上下文。然后,动态链接程序必须检查被执行的程序,以识别哪个共享库必须装入及在每个共享库中哪个函数被有效地请求。接下来,解释器发出几个`mmap()`系统调用来创建线性区,以对将存放程序实际使用的库函数(正文和数据)的页进行映射。然后,解释器根据库的线性区的线性地址更新对共享库符号的所有引用。最后,动态链接程序通过跳转到被执行程序的主入口点而终止它的执行。从现在开始,进程将执行可执行文件的代码和共享库的代码。

你可能已注意到,执行程序是一个相当复杂的活动,它涉及内核设计的很多方面,如进程抽象、内存管理、系统调用及文件系统。这会使你认识到:Linux真是一个杰作!

注7: 如果可执行文件是静态链接的,即如果不需要共享库,事情就简单多了。`Load_binary`方法只需将程序的正文段、数据段、`bss`段和堆栈段映射到进程线性区,然后把用户态`eip`寄存器的内容设置为新程序的入口点即可。

系统启动

本附录介绍当用户打开计算机电源之后所发生的事情，也就是说，Linux 内核映像是如何被拷贝到内存的，又是如何被执行的。简而言之，我们讨论内核，继而是整个系统，是如何启动的。

“启动 (*bootstrap*)”这个术语的原意是一个人要穿上靴子站起来。在操作系统中，这个术语专门表示把一部分操作系统装载到主存中并让处理器执行它，也表示内核数据结构的初始化、一些用户进程的创建以及把控制权转移到其中某个进程。

计算机启动是一个冗长乏味的任务，因为最开始时几乎每个硬件设备（包括RAM）都处于一种随机的、不可预知的状态。此外，启动过程在很大程度上都依赖于计算机的体系结构；和以前一样，我们在本附录中特指 80x86 体系结构。

史前时代：BIOS

计算机在加电的那一刻几乎是毫无用处的，因为RAM 芯片中包含的是随机数据，此时还没有操作系统在运行。在开始启动时，有一个特殊的硬件电路在CPU的一个引脚上产生一个RESET逻辑值。在RESET产生以后，就把处理器的一些寄存器（包括cs和eip）设置成固定的值，并执行在物理地址 0xffffffff0 处找到的代码。硬件把这个地址映射到某个只读、持久的存储芯片中，该芯片通常称为ROM (Read-Only Memory, 只读内存)。ROM中所存放的程序集在80x86体系中通常叫作基本输入/输出系统 (*Basic Input/Output System*, BIOS)，因为它包括几个中断驱动的低级过程。所有操作系统在启动时，都要通过这些过程对计算机硬件设备初始化。一些操作系统，如微软的MS-DOS，依赖于BIOS实现大部分系统调用。

Linux 一旦进入保护模式（参见第二章“硬件中的分段”一节），就不再使用 BIOS，而是为计算机上的每个硬件设备提供各自的设备驱动程序。实际上，因为 BIOS 过程必须在实模式下运行，所以即使有益，两者之间也不能共享函数。

BIOS 使用实模式的地址，因为在计算机加电启动时只有这些可以使用。一个实模式的地址由一个 *seg* 段和一个 *off* 偏移量组成。相应的物理地址可以这样计算： $seg * 16 + off$ 。所以 CPU 寻址电路根本就不需要全局描述符表、局部描述符表或者页表把逻辑地址转换成物理地址。显然，对 GDT、LDT 和页表进行初始化的代码必须在实模式下运行。

Linux 在启动阶段必须使用 BIOS，此时 Linux 必须要从磁盘或者其他外部设备中获取内核映像。BIOS 启动过程实际上执行以下 4 个操作：

1. 对计算机硬件执行一系列的测试，用来检测现在都有什么设备以及这些设备是否正常工作。这个阶段通常称为 *POST* (*Power-On Self-Test*，上电自检)。在这个阶段中，会显示一些信息，例如 BIOS 版本号。

如今的 80x86、AMD64 和 Itanium 计算机使用高级配置与开机界面 (*Advanced Configuration and Power Interface*, *ACPI*) 标准。在 ACPI 兼容的 BIOS 中，启动代码会建立几个表来描述当前系统中的硬件设备。这些表的格式独立于设备生产商，而且可由操作系统读取以获得如何调用这些设备的信息。

2. 初始化硬件设备。这个阶段在现代基于 PCI 的体系结构中相当重要，因为它可以保证所有的硬件设备操作不会引起 IRQ 线与 I/O 端口的冲突。在本阶段的最后，会显示系统中所安装的所有 PCI 设备的一个列表。
3. 搜索一个操作系统来启动。实际上，根据 BIOS 的设置，这个过程可能要试图访问（按照用户预定义的次序）系统中软盘、硬盘和 CD-ROM 的第一个扇区（引导扇区）。
4. 只要找到一个有效的设备，就把第一个扇区的内容拷贝到 RAM 中从物理地址 0x00007c00 开始的位置，然后跳转到这个地址处，开始执行刚才装载进来的代码。

本附录其余的部分会带你体验从最原始的开始状态到运行 Linux 系统的整个历程。

远古时代：引导装入程序

引导装入程序 (*boot loader*) 是由 BIOS 用来把操作系统的内核映像装载到 RAM 中所调用的一个程序。让我们简要地描绘一下引导装入程序在 IBM 的 PC 体系结构中是如何工作的。

为了从软盘上启动，必须把第一个扇区中所存放的指令装载到 RAM 中并执行；这些指令再把包含内核映像的其他所有扇区都拷贝到 RAM 中。

从硬盘启动的实现有点不同。硬盘的第一个扇区称为主引导记录 (*Master Boot Record*, *MBR*), 该扇区中包括分区表 (注 1) 和一个小程序, 这个小程序用来装载被启动的操作系统所在分区的第一个扇区。诸如 Microsoft Windows 98 之类的操作系统使用分区表中所包含的一个活动 (*active*) 标志来标识这个分区 (注 2)。按照这种方法, 只有那些内核映像存放在活动分区中的操作系统才可以被启动。正如我们将在后面看到的一样, Linux 的处理方式更加灵活, 因为 Linux 使用一个巧妙的引导装入程序取代这个 MBR 中不完善的程序, 它允许用户来选择要启动的操作系统。

Linux 早期版本 (一直到 2.4 系列) 的内核映像, 在第一个 512 字节有一个最小的引导装入程序, 因此在第一扇区拷贝一个内核映像就可以使软盘可启动。但是在 Linux 2.6 中就不再有这样的引导装入程序, 所以要从软盘启动, 就必须在第一个磁盘扇区存放一个合适的引导装入程序。而现在从软盘启动与从硬盘或 CD-ROM 启动是十分相似的。

从磁盘启动 Linux

从磁盘启动 Linux 内核需要一个两步的引导装入程序。在 80x86 体系中, 众所周知的 Linux 引导装入程序叫作 *LI*nx *LO*ader (*LILO*)。确实还有一些 80x86 体系的引导装入程序, 如广泛使用的 *GR*and *U*nified *B*ootloader (*GRUB*)。GRUB 比 LILO 更为先进, 因为它可识别多个基于磁盘的文件系统, 而且可以从文件中读入部分引导程序。当然, 对于 Linux 支持的所有体系结构都有各自专门的引导装入程序。

LILO 或许被装在 MBR 上 (代替那个装载活动引导扇区的小程序), 或许被装在每个磁盘分区的引导扇区上。在这两种情况下, 最终的结果是相同的: 装入程序在启动过程中被执行时, 用户都可以选择装入哪个操作系统。

实际上, LILO 引导装入程序被分为两部分, 因为不划分的话, 它就太大而无法装进单个扇区。MBR 或者分区引导扇区包括一个小的引导装入程序, 由 BIOS 把这个小程序装入从地址 0x00007c00 开始的 RAM 中。这个小程序又把自己移到地址 0x00096a00, 建立实模式栈 (0x00098000~0x000969ff), 并把 LILO 的第二部分装入到从地址 0x00096c00 开始的 RAM 中。

第二部分又依次从磁盘读取可用操作系统的映射表, 并提供给用户一个提示符, 因此用户就可以从中选择一个操作系统。最后, 用户选择了被装入的内核后 (或经过一个延迟

注 1: 每个分区表项通常包含分区的起止扇区和处理它的操作系统类型。

注 2: 活动标志可由程序 *fdisk* 设置。

时间以使 LILO 选择一个缺省值)，引导装入程序就可以把相应分区的引导扇区拷贝到 RAM 中并执行它，或直接把内核映像拷贝到 RAM 中。

假定 Linux 内核映像必须被导入，LILO 引导装入程序依赖于 BIOS 例程，主要执行如下步骤：

1. 调用一个 BIOS 过程显示“Loading”信息。
2. 调用一个 BIOS 过程从磁盘装入内核映像的初始部分，即将内核映像的第一个 512 字节从地址 0x00090000 开始存入 RAM 中，而将 setup() 函数的代码（参见下面）从地址 0x00090200 开始存入 RAM 中。
3. 调用一个 BIOS 过程从磁盘中装载其余的内核映像，并把内核映像放入从低地址 0x00010000（适用于使用 make zImage 编译的小内核映像）或者从高地址 0x00100000（适用于使用 make bzImage 编译的大内核映像）开始的 RAM 中。在以下的讨论中，我们将分别称内核映像是“低装载”到 RAM 中或者“高装载”到 RAM 中。大内核映像的支持虽然本质上与其他启动模式相同，但是它却把数据放在不同的物理内存地址，以避免在第二章“物理内存布局”一节所介绍的 ISA 黑洞问题。
4. 跳转到 setup() 代码。

中世纪：setup() 函数

setup() 汇编语言函数的代码由链接程序放在内核映像文件的偏移量 0x200 处。引导装入程序因此就可以很容易地确定 setup() 代码的位置，并把它拷贝到从物理地址 0x00090200 开始的 RAM 中。

setup() 函数必须初始化计算机中的硬件设备，并为内核程序的执行建立环境。虽然 BIOS 已经初始化了大部分硬件设备，但是 Linux 并不依赖于 BIOS，而是以自己的方式重新初始化设备以增强可移植性和健壮性。setup() 本质上执行以下操作：

1. 在 ACPI 兼容的系统中，它调用一个 BIOS 例程，以在 RAM 中建立系统物理内存布局表（通过检索“IOS-e820”标签，就可在引导内核信息中看到该表）。在早期系统中，它调用 BIOS 例程，返回系统可用内存。
2. 设置键盘重复延时和速率（当用户一直按下一个键超过一定的时间，键盘设备就反复地向 CPU 发送相应的键盘码）。
3. 初始化视频卡。
4. 重新初始化磁盘控制器并检测硬盘参数。

5. 检查 IBM 微通道总线 (MCA)。
6. 检查 PS/2 指针设备 (总线鼠标)。
7. 检查对高级电源管理 (APM) BIOS 的支持。
8. 如果 BIOS 支持增强磁盘驱动服务 (*Enhanced Disk Drive Service, EDD*)，它就调用相应的 BIOS 过程在 RAM 中建立系统可用硬盘表 (表中的信息可以通过 *sysfs* 特殊文件系统的 *firmware/edd* 目录查看)。
9. 如果内核映像被低装载到 RAM 中 (在物理地址 0x00010000 处)，就把它移动到物理地址 0x00001000 处。反之，如果内核映像被高装载到 RAM 中，就不用移动。这个步骤是必需的，因为为了能在软盘上存储内核映像并节省启动的时间，存放在磁盘上的内核映像都是压缩的，解压程序需要一些空闲空间作为临时缓冲区 (在 RAM 中紧挨内核映像的地方)。
10. 置位 8042 键盘控制器的 A20 引脚。A20 引脚是在 80286 系统中引入的，为的是与古老的 8088 微处理器物理地址兼容。不幸的是，在切换到保护模式之前必须将 A20 引脚正确置位，否则，每个物理地址的第 21 位都会被 CPU 看作 0。置位 A20 引脚是件讨厌的事情。
11. 建立一个临时中断描述符表 (IDT) 和一个临时全局描述符表 (GDT)。
12. 如果需要，重置浮点单元 (FPU)。
13. 重新编写可编程中断控制器 (Programmable Interrupt Controller, PIC)，以屏蔽所有中断，但保留 IRQ2，它是两个 PIC 之间的级联中断。
14. 通过设置 `cr0` 状态寄存器中的 PE 位，把 CPU 从实模式切换到保护模式。`cr0` 状态寄存器中的 PG 位被清 0，因此分页还没有启用。
15. 跳转到 `startup_32()` 汇编语言函数。

文艺复兴时期：startup_32()函数

有两个不同的 `startup_32()` 函数，我们此处所指的是在 *arch/i386/boot/compressed/head.S* 文件中实现的那一个。在 `setup()` 结束之后，`startup_32()` 就已经被移动到物理地址 0x00100000 处或者 0x00001000 处，这取决于内核映像是被高装载到 RAM 中还是低装载到 RAM 中。

该函数执行以下操作：

1. 初始化段寄存器和一个临时堆栈。

2. 清零 `eflags` 寄存器的所有位。
3. 用 0 填充由 `_edata` 和 `_end` 符号标识的内核未初始化数据区（参见第二章的“物理内存布局”一节）。
4. 调用 `decompress_kernel()` 函数来解压内核映像。首先显示“Uncompressing Linux ...”信息。完成内核映像的解压之后，显示“OK, booting the kernel.”信息。如果内核映像是低装载的，那么解压后的内核就被放在物理地址 `0x00100000` 处。否则，如果内核映像是高装载的，那么解压后的内核就被放在位于这个压缩映像之后的临时缓冲区中。然后，解压后的映像就被移动到从物理地址 `0x00100000` 开始的最终位置。
5. 跳转到物理地址 `0x00100000` 处。

解压的内核映像以包含在 `arch/i386/kernel/head.S` 中的另一个 `startup_32()` 函数开始。这两个函数使用相同的名字不会产生任何问题（除了使读者容易混淆外），因为这两个函数会跳转到自己的起始物理地址去执行。

第二个 `startup_32()` 函数为第一个 Linux 进程（进程 0）建立执行环境。该函数执行以下操作：

1. 把段寄存器初始化为最终值。
2. 把内核的 `bss` 段填充为 0（参见第二十章“程序段和进程的内存区域”一节）。
3. 初始化包含在 `swapper_pg_dir` 的临时内核页表，并初始化 `pg0`，以使线性地址一致地映射同一物理地址，这在第二章“内核页表”一节已经作了说明。
4. 把页全局目录的地址存放在 `cr3` 寄存器中，并通过设置 `cr0` 寄存器的 `PG` 位启用分页。
5. 为进程 0 建立内核态堆栈（参见第三章的“内核线程”一节）。
6. 该函数再一次清零 `eflags` 寄存器的所有位。
7. 调用 `setup_idt()` 用空的中断处理程序填充 IDT（参见第四章的“IDT 的初步初始化”一节）。
8. 把从 BIOS 中获得的系统参数和传递给操作系统的参数放入第一个页框中（参见第二章的“物理内存布局”一节）。
9. 识别处理器的型号。
10. 用 GDT 和 IDT 表的地址来填充 `gdtr` 和 `idtr` 寄存器。
11. 跳转到 `start_kernel()` 函数。

现代：start_kernel()函数

start_kernel()函数完成Linux内核的初始化工作。几乎每天内核部件都是由这个函数进行初始化的，我们只提及其中的少部分：

- 调用 sched_init()函数来初始化调度程序（参见第七章）。
- 调用 build_all_zonelists()函数来初始化内存管理区（参见第八章“内存管理区”一节）。
- 调用 page_alloc_init()函数来初始化伙伴系统分配程序（参见第八章“伙伴系统算法”一节）。
- 调用 trap_init()函数（参见第四章“异常处理”一节）和 init_IRQ()函数（参见第四章“IRQ数据结构”一节）以完成 IDT 初始化。
- 调用 softirq_init()函数初始化 TASKLET_SOFTIRQ 和 HI_SOFTIRQ（参见第四章“软中断”一节）。
- 调用 time_init()函数来初始化系统日期和时间（参见第六章“Linux 计时体系结构”一节）。
- 调用 kmem_cache_init()函数来初始化 slab 分配器（参见第八章的“普通和专用高速缓存”一节）。
- 调用 calibrate_delay()函数以确定 CPU 时钟的速度（参见第六章“延迟函数”一节）。
- 调用 kernel_thread()函数为进程 1 创建内核线程。正如我们在第三章的“内核线程”一节中已经描述的一样，这个内核线程又会创建其他的内核线程并执行 /sbin/init 程序。

在 start_kernel()开始执行之后会显示“Linux version 2.6.11...”信息，除此之外，在 init 程序和内核线程执行的最后阶段还会显示很多其他信息。最后，就会在控制台上出现熟悉的登录提示符（如果在启动时所启动的是 X Window 系统，那么登录提示符就会出现在一个图形窗口中），通知用户 Linux 内核已经启动，现在正在运行。

附录二

模块

正如我们在第一章中所介绍的那样，模块（*module*）是 Linux 用来高效地利用微内核的理论优点而不会降低系统性能的一种方法。

是否使用模块？

当系统程序员希望给 Linux 内核增加新功能时，就面临一个进退两难的问题：他们应该编写新代码从而将其作为一个模块进行编译，还是应该将这些代码静态地链接到内核中？

通常，系统程序员都倾向于把新代码作为一个模块来实现。因为模块可以根据需要进行链接，这样内核就不会因为装载那些数以百计的很少使用的程序而变得非常庞大，这一点我们后面就会看到。几乎 Linux 内核的每个高层组件——文件系统、设备驱动程序、可执行格式、网络层等等——都可以作为模块进行编译。Linux 的发布版，充分使用模块方式全面地支持多种硬件设备。例如，发布版中会将几十种声卡驱动程序模块放在某个目录下，但是在某个计算机上只会有效加载其中一个声卡驱动程序。

然而，有些 Linux 代码必须被静态链接，也就是说相应组件或者被包含在内核中，或者根本不被编译。典型情况下，这发生在组件需要对内核中静态链接的某个数据结构或函数进行修改时。

例如，假设某个组件必须在进程描述符中引入新字段。链接一个模块并不能修改诸如 `task_struct` 之类已经定义的数据结构，因为即使这个模块使用其数据结构的修改版，所有静态链接的代码看到的仍是原来的版本，这样就很容易发生数据崩溃。对此问题的一种局部解决方法就是“静态地”把新字段加到进程描述符，从而让这个内核组件可以

使用这些字段，而不用考虑组件究竟是如何被链接的。然而，如果该内核组件从未被使用，那么，在每个进程描述符中都复制这些额外的字段就是对内存的浪费。如果新内核组件对进程描述符的大小有很大的增加，那么，只有新内核组件被静态地链接到内核，才可能通过在这个数据结构中增加需要的字段获得较好的系统性能。

再例如，考虑一个内核组件，它要替换静态链接的代码。显然，这样的组件不能作为一个模块来编译，因为在链接模块时内核不能修改已经在RAM中的机器码。例如，系统不可能链接一个改变页框分配方法的模块，因为伙伴系统函数总是被静态地链接到内核（注1）。

内核有两个主要的任务来进行模块的管理。第一个任务是确保内核的其他部分可以访问该模块的全局符号，例如指向模块主函数的入口。模块还必须知道这些符号在内核及其他模块中的地址。因此，在链接模块时，一定要解决模块间的引用关系。第二个任务是记录模块的使用情况，以便在其他模块或者内核的其他部分正在使用这个模块时，不能卸载这个模块。系统使用了一个简单的引用计数器来记录每个模块的引用次数。

模块许可证

Linux 内核许可证（GPL，版本2）不限制用户与企业使用其源代码，但是它严格禁止在非GPL许可证下发行相关的源代码，而这些代码起源于或大部分起源于Linux代码。也就是说，内核开发者要确保他们的代码及其衍生代码可由所有用户自由使用。

但是，模块对这一机制造成了威胁。可能有人只发行一个用于Linux内核的二进制格式模块；例如，厂商可能只以二进制格式模块发行一个硬件驱动程序。现在，这种情况较为少见。理论上说，Linux内核的特性和功能可被只有二进制格式的模块极大地改变，从而把基于Linux的内核转变成商业产品。

因此，Linux内核开发者社团不太接受只有二进制格式的模块。Linux模块的实现就反映出这一点。一般地，使用MODULE_LICENSE宏，每个模块开发者应当在模块源代码中标出许可证类型。如果许可证是非GPL兼容（或根本没有标出），模块就不能使用内核的许多核心函数和数据结构。而且，使用非GPL许可证的模块会“玷污”内核，也就是说内核开发者不再考虑内核中可能的缺陷。

注1： 你可能疑惑为什么不把你所钟爱的内核组件模块化。实际上，总的原因是软件许可证的问题，而不是技术原因。内核开发者想确保核心组件永远不会被仅发布二进制“黑盒”模块的私有代码所代替。

模块的实现

模块是作为 ELF 对象文件存放在文件系统上的，并通过执行 `insmod` 程序链接到内核中（参见后面的“模块的链接和取消”一节）。对于每个模块，系统都分配一个包含以下数据的内存区：

- 一个 `module` 对象
- 表示模块名的一个以 `null` 结束的字符串（所有的模块都必须有唯一的名字）
- 实现模块功能的代码

`module` 对象描述一个模块，其字段如表 B-1 所示。一个双向循环列表存放所有 `module` 对象。链表头部存放在 `modules` 变量中，而指向相邻单元的指针存放在每个 `module` 对象的 `list` 字段中。

表 B-1: `module` 对象

类型	字段名	说明
<code>enum module_state</code>	<code>state</code>	模块的内部状态
<code>struct list_head</code>	<code>list</code>	模块链表的指针
<code>char [60]</code>	<code>name</code>	模块名
<code>struct module_kobject</code>	<code>mkobj</code>	包含一个 <code>kobject</code> 数据结构和指向这个模块对象的指针
<code>struct module_param_attrs *</code>	<code>param_attrs</code>	指向模块参数描述符数组的指针
<code>const struct kernel_symbol *</code>	<code>syms</code>	指向导出符号数组的指针
<code>unsigned int</code>	<code>num_syms</code>	导出符号数
<code>const unsigned long *</code>	<code>crcls</code>	指向导出符号 CRC 值数组的指针
<code>const struct kernel_symbol *</code>	<code>gpl_syms</code>	指向 GPL 格式导出符号数组的指针
<code>unsigned int</code>	<code>num_gpl_syms</code>	GPL 格式导出符号数
<code>const unsigned long *</code>	<code>gpl_crcls</code>	指向 GPL 格式导出符号 CRC 值数组的指针
<code>unsigned int</code>	<code>num_exentries</code>	模块异常表项数
<code>const struct exception_table_entry *</code>	<code>exstable</code>	指向模块异常表的指针

表 B-1: module 对象 (续)

类型	字段名	说明
int (*)(void)	init	模块初始化方法
void *	module_init	用于模块初始化的动态内存区指针
void *	module_core	用于模块核心函数与数据结构的动态内存区指针
unsigned long	init_size	用于模块初始化的动态内存区大小
unsigned long	core_size	用于模块核心函数与数据结构的动态内存区大小
unsigned long	init_text_size	模块初始化的可执行代码大小, 只当链接模块时使用
unsigned long	core_text_size	模块核心可执行代码大小, 只当链接模块时使用
struct mod_arch_specific	arch	依赖于体系结构的字段 (80 × 86 结构中 没有)
int	unsafe	如果模块不能安全卸载, 则将该标志置位
int	license_gpl	如果模块许可证是 GPL 兼容的, 则将该标志置位
struct module_ref [NR_CPUS]	ref	每 CPU 使用计数器
struct list_head	modules_which_use_me	依赖于该模块的模块链表
struct task_struct *	waiter	正卸载模块的进程
void (*)(void)	exit	模块退出方法
Elf_Sym *	symtab	/proc/kallsyms 文件中所列模块 ELF 符号数组指针
unsigned long	num_symtab	/proc/kallsyms 文件中所列模块 ELF 符号数
char *	strtab	/proc/kallsyms 文件中所列模块 ELF 符号的字符串表
struct module_sect_attrs *	sect_attrs	模块分节属性描述符数组指针 (在 sysfs 文件系统中显示)
void *	percpu	特定于 CPU 的内存区指针
char *	args	链接模块时使用的命令行参数

state 字段记录模块内部状态，它可以是：MODULE_STATE_LIVE（模块为活动的）、MODULE_STATE_COMING（模块正在初始化）和MODULE_STATE_GOING（模块正在卸载）。

正如我们在第十章的“动态地址检查：修正代码”一节中已经介绍的那样，每个模块都有自己的异常表。该表包括（如果有）模块的修正代码的地址。在链接模块时，该表被拷贝到 RAM 中，其开始地址保存在 module 对象的 extable 字段中。

模块使用计数器

每个模块都有一组使用计数器，每个 CPU 一个，存放在相应 module 对象的 ref 字段中。在模块功能所涉及的操作开始执行时递增这个计数器，在操作结束时递减这个计数器。只有所有使用计数器的和为 0 时，模块才可以被取消链接。

例如，假设 MS-DOS 文件系统层作为模块被编译，而且这个模块已经在运行时被链接。最开始时，该模块的引用计数器是 0。如果用户装载一张 MS-DOS 软盘，那么模块引用计数器其中的一个就被递增 1。反之，当用户卸载这张软盘时，计数器其中之一就被减 1（甚至不是刚才递增的那一个）。模块的总的引用计数器就是所有 CPU 计数器的总和。

导出符号

当链接一个模块时，必须用合适的地址替换在模块对象代码中引用的所有全局内核符号（变量和函数）。这个操作与在用户态编译程序时链接程序所执行的操作非常类似（参见第二十章的“库”一节），这是委托给 *insmod* 外部程序完成的（将在后面的“模块的链接和取消”一节进行介绍）。

内核使用一些专门的内核符号表 (*kernel symbol table*)，用于保存模块访问的符号和相应的地址。它们在内核代码段中分三个节：_kstrtab 节（保存符号名）、_ksymtab 节（所有模块可使用的符号地址）和 _ksymtab_gpl 节（GPL 兼容许可证下发布的模块可以使用的符号地址）。当用于静态链接内核代码内时，EXPORT_SYMBOL 与 EXPORT_SYMBOL_GPL 宏让 C 编译器分别往 _ksymtab 和 _ksymtab_gpl 部分相应地加入一个专用符号。

只有某一现有的模块实际使用的内核符号才会保存在这个表中。如果系统程序员在某些模块中需要访问一个尚未导出的内核符号，那么他只要在 Linux 源代码中增加相应的 EXPORT_SYMBOL_GPL 宏就可以了。当然，如果许可证不是 GPL 兼容的，他就不能为模块合法导出一个新符号。

已链接的模块也可以导出自己的符号，这样其他模块就可以访问这些符号。模块符号部分表 (*module symbol table*) 保存在模块代码段的 _ksymtab、_ksymtab_gpl 和 _kstrtab

部分中。要从模块中导出符号的一个子集，程序员可以使用上面描述的 `EXPORT_SYMBOL` 和 `EXPORT_SYMBOL_GPL` 宏。当模块链接时，模块的导出符号被拷贝到两个内存数组中，而其地址保存在 `module` 对象的 `syms` 和 `gpl_syms` 字段中。

模块依赖

一个模块 (B) 可以引用由另一个模块 (A) 所导出的符号；在这种情况下，我们就说 B 装载在 A 的上面，或者说 A 被 B 使用。为了链接模块 B，必须首先链接模块 A；否则，对于模块 A 所导出的那些符号的引用就不能适当地链接到 B 中。简而言之，在这两个模块之间存在着依赖 (*dependency*)。

A 模块对象的 `modules_which_use_me` 字段是一个依赖链表的头部，该链表保存 A 使用的所有模块。链表中的每个元素是一个小型 `module_use` 描述符，该描述符保存指向链表中相邻元素的指针及一个指向相应模块对象的指针。在本例中，指向 B 模块对象的 `module_use` 描述符将出现在 A 的 `modules_which_use_me` 链表中。只要有模块装载在 A 上，`modules_which_use_me` 链表就必须动态更新。如果 A 的依赖链表非空，模块 A 就不能卸载。

当然，除了 A 和 B 之外，还会有其他模块 (C) 装载到 B 上，依此类推。模块的堆叠是对内核源代码进行模块化的一种有效方法，目的是为了加速内核的开发。

模块的链接和取消

用户可以通过执行 `insmod` 外部程序把一个模块链接到正在运行的内核中。该程序执行以下操作：

1. 从命令行中读取要链接的模块名。
2. 确定模块对象代码所在的文件在系统目录树中的位置。对应的文件通常都是在 `/lib/modules` 的某个子目录中。
3. 从磁盘读入存有模块目标代码的文件。
4. 调用 `init_module()` 系统调用，传入参数：存有模块目标代码的用户态缓冲区地址、目标代码长度和存有 `insmod` 程序所需参数的用户态内存区。
5. 结束。

`sys_init_module()` 服务例程是实际执行者，主要操作步骤如下：

1. 检查是否允许用户链接模块（当前进程必须具有 `CAP_SYS_MODULE` 权能）。只要给内核增加功能，而它可以访问系统中的所有数据和进程，安全就是至关重要的。

2. 为模块目标代码分配一个临时内存区,然后拷入作为系统调用第一个参数的用户态缓冲区数据。
3. 验证内存区中的数据是否有效表示模块的 ELF 对象,如果不能,则返回错误码。
4. 为传给 *insmod* 程序的参数分配一个内存区,并存入用户态缓冲区的数据,该缓冲区地址是系统调用传入的第三个参数。
5. 查找 *modules* 链表,以验证模块未被链接。通过比较模块名 (*module* 对象的 *name* 字段) 进行这一检查。
6. 为模块核心可执行代码分配一个内存区,并存入模块相应节的内容。
7. 为模块初始化代码分配一个内存区,并存入模块相应节的内容。
8. 为新模块确定模块对象地址,对象映像保存在模块 ELF 文件的正文段 *gnu.linkonce.this_module* 一节,而模块对象保存在第 6 步中的内存区。
9. 将第 6 和 7 步中分配的内存区地址存入模块对象的 *module_code* 和 *module_init* 字段。
10. 初始化模块对象的 *modules_which_use_me* 链表。当前执行 CPU 的计数器设为 1,而其余所有的模块引用计数器设为 0。
11. 根据模块对象许可证类型设定模块对象的 *license_gpl* 标志。
12. 使用内核符号表与模块符号表,重置模块目标码。这意味着用相应的逻辑地址偏移量替换所有外部与全局符号的实例值。
13. 初始化模块对象的 *syms* 和 *gpl_syms* 字段,使其指向模块导出的内存中符号表。
14. 模块异常表 (参见第十章“异常表”一节) 保存在模块 ELF 文件的 *__ex_table* 一节,因此它在第 6 步中已拷入内存区,将其地址存入模块对象的 *extable* 字段。
15. 解析 *insmod* 程序的参数,并相应地设定模块变量的值。
16. 注册模块对象 *mkobj* 字段中的 *kobject* 对象,这样在 *sysfs* 特殊文件系统的 *module* 目录中就有一个新的子目录 (参见第十三章“*kobject*”一节)。
17. 释放第 2 步中分配的临时内存区。
18. 将模块对象追加到 *modules* 链表。
19. 将模块状态设为 *MODULE_STATE_COMING*。
20. 如果模块对象的 *init* 方法已定义,则执行它。
21. 将模块状态设为 *MODULE_STATE_LIVE*。
22. 结束并返回 0 (成功)。

为了取消模块的链接,用户需要调用 *rmmod* 外部程序,该程序执行以下操作:

1. 从命令行中读取要取消的模块的名字。
2. 打开 `/proc/modules` 文件，其中列出了所有链接到内核的模块，检查待取消模块是否有效链接。
3. 调用 `delete_module()` 系统调用，向其传递要卸载的模块名。
4. 结束。

相应的 `sys_delete_module()` 服务例程执行以下操作：

1. 检查是否允许用户取消模块链接（当前进程必须具有 `CAP_SYS_MODULE` 权能）。
2. 将模块名存入内核缓冲区。
3. 从 `modules` 链表查找模块的 `module` 对象。
4. 检查模块的 `modules_which_use_me` 依赖链表，如果非空就返回一个错误码。
5. 检查模块状态，如果不是 `MODULE_STATE_LIVE`，就返回错误码。
6. 如果模块有自定义 `init` 方法，函数就要检查是否有自定义 `exit` 方法。如果没有自定义 `exit` 方法，模块就不能卸载，那么返回一个退出码。
7. 为了避免竞争条件，除了运行 `sys_delete_module()` 服务例程的 CPU 外，暂停系统中所有 CPU 的运行。
8. 把模块状态设为 `MODULE_STATE_GOING`。
9. 如果所有模块引用计数器的累加值大于 0，就返回错误码。
10. 如果已定义模块的 `exit` 方法，则执行它。
11. 从 `modules` 链表删除模块对象，并且从 `sysfs` 特殊文件系统注销该模块。
12. 从刚才使用的模块依赖链表中删除模块对象。
13. 释放相应内存区，其中存有模块可执行代码、`module` 对象及有关符号和异常表。
14. 返回 0（成功）。

根据需要链接模块

模块可以在系统需要其所提供的功能时自动进行链接，之后也可以自动删除。

例如，假设 MS-DOS 文件系统既没有被静态链接，也没有被动态链接。如果用户试图装载 MS-DOS 文件系统，那么 `mount()` 系统调用通常就会失败，返回一个错误码，因为 MS-DOS 没有被包含在已注册文件系统的 `file_systems` 链表中。然而，如果内核已配

置为支持模块的动态链接,那么Linux就试图链接MS-DOS模块,然后再扫描已经注册过的文件系统的列表。如果该模块成功地被链接,那么`mount()`系统调用就可以继续执行,就好像MS-DOS文件系统从一开始就存在一样。

modprobe 程序

为了自动链接模块,内核要创建一个内核线程来执行`modprobe`外部程序(注2),该程序要考虑由于模块依赖所引起的所有可能因素。模块依赖在前面已介绍过:一个模块可能需要一个或者多个其他模块,这些模块又可能需要其他模块。例如,MS-DOS模块需要另外一个名为`fat`的模块,该模块包含基于文件分配表(File Allocation Table, FAT)的所有文件系统所通用的一些代码。因此,如果`fat`模块还不在于系统中,那么在系统请求MS-DOS模块时,`fat`模块也必须被动态链接到运行的内核中。对模块依赖进行解析以及对模块进行查找的操作最好都在用户态中实现,因为这需要查找和访问文件系统中的模块对象文件。

`modprobe`外部程序和`insmod`类似,因为它链接在命令行中指定的一个模块。然而,`modprobe`还可以递归地链接命令行中模块所使用的所有模块。例如,如果用户调用`modprobe`来链接MS-DOS模块,那么在需要的时候,`modprobe`就会在MS-DOS模块之后链接`fat`模块。实际上,`modprobe`只是检查模块依赖关系,每个模块的实际的链接工作是通过创建一个进程并执行`insmod`命令来实现的。

`modprobe`又是如何知道模块间的依赖关系的呢?另外一个称为`depmod`的外部命令在系统启动时被执行。该程序查找为正在运行的内核而编译的所有模块,这些模块通常存放在`/lib/modules`目录下。然后它就把所有的模块间依赖关系写入一个名为`modules.dep`的文件。这样,`modprobe`就可以对该文件中存放的信息和`/proc/modules`文件产生的链接模块链表进行比较。

request_module()函数

在某些情况下,内核可以调用`request_module()`函数来试图自动链接一个模块。

再次考虑用户试图装载MS-DOS文件系统的情况。如果`get_fs_type()`函数发现这个文件系统还没有注册,就调用`request_module()`函数,希望MS-DOS已经被编译为一个模块。

注2: 这是内核依赖于外部程序的少数例子之一。

如果 `request_module()` 函数成功地链接所请求的模块, `get_fs_type()` 就可以继续执行, 仿佛这个模块一直都存在一样。当然, 并非所有的情况都是如此; 在我们的例子中, MS-DOS 模块可能根本就没有被编译。在这种情况下, `get_fs_type()` 返回一个错误码。

`request_module()` 函数接收要链接的模块名作为参数。该函数调用 `kernel_thread()` 来创建一个新的内核线程并等待, 直到这个内核线程结束为止。

而此内核线程又接收待链接的模块名作为参数, 并调用 `execve()` 系统调用以执行 *modprobe* 外部程序 (注 3), 向其传递模块名。然后, *modprobe* 程序真正地链接所请求的模块以及这个模块所依赖的任何模块。

注 3: 由 `exec_modprobe()` 执行的程序名和路径名可以通过向 `/proc/sys/kernel/modprobe` 文件写入而自定义。