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） 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）	（不允许也无需回收）

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（倒如，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，FPRA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_mm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。

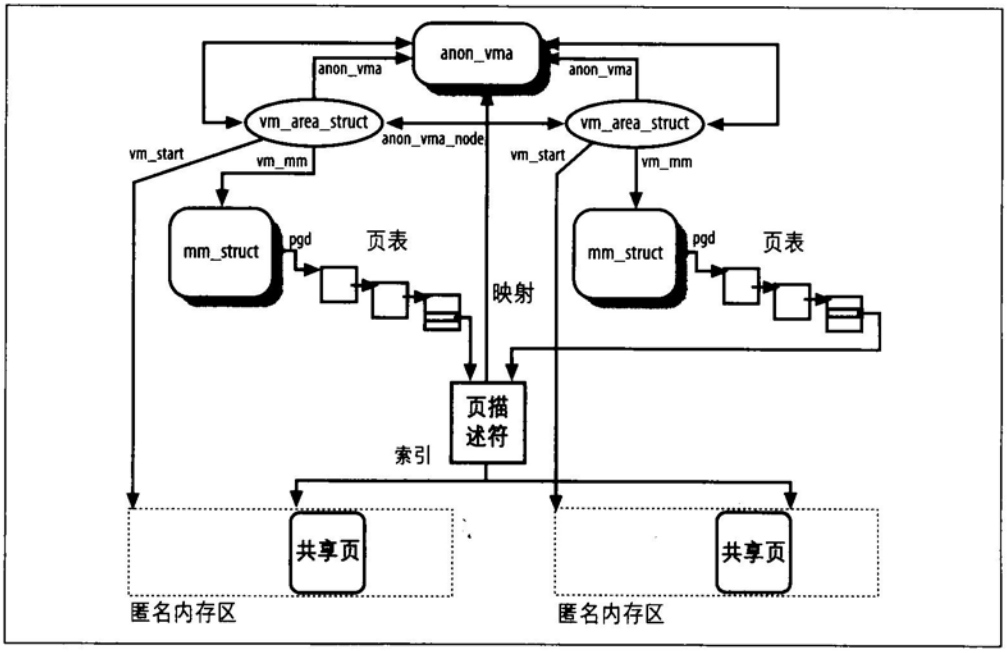

图 17-1：匿名页的面向对象反向映射

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。

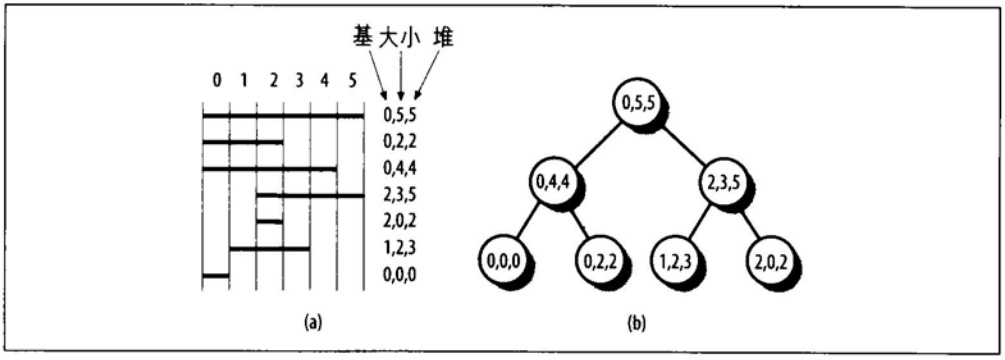

图17-2：简单的优先搜索树

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

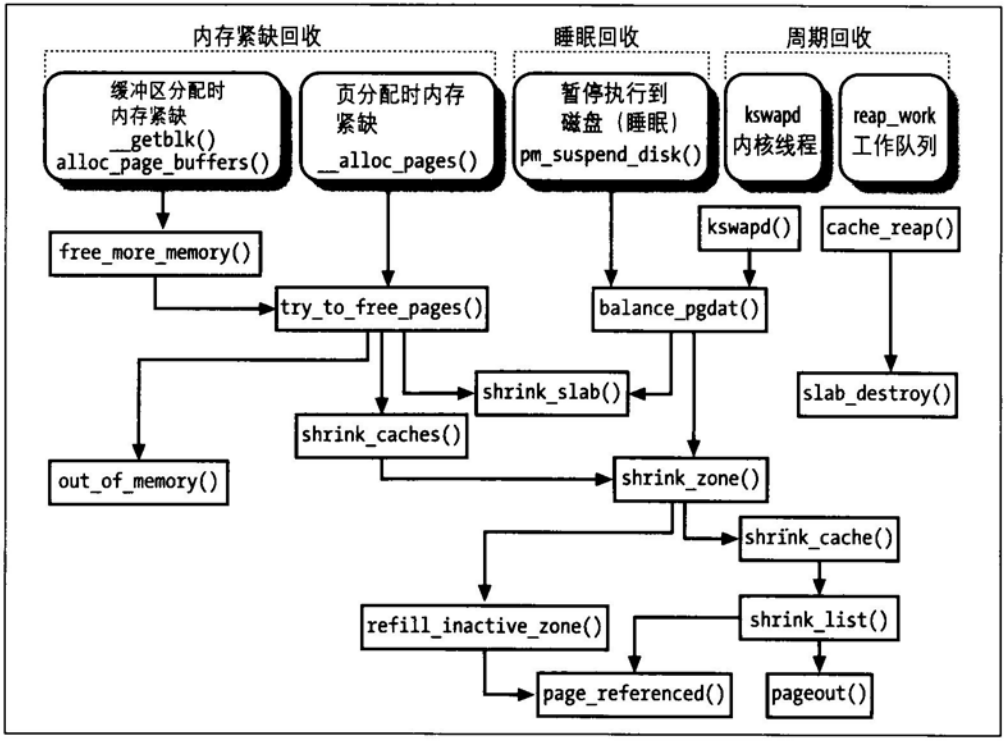

图 17-3：PFRA 的主要函数

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。

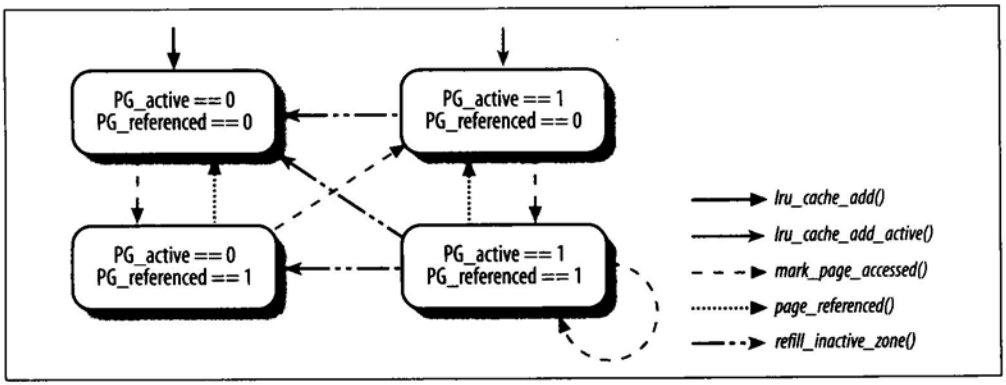

图 17-4：在 LRU 链表之间移动页

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：

交换倾向值 = 映射比率 / 2 + 负荷值 + 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_referened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_bdflush()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/2^{12}到zone->nr_active/2^{0}（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/2^{12}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，它的主要目的就是从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。

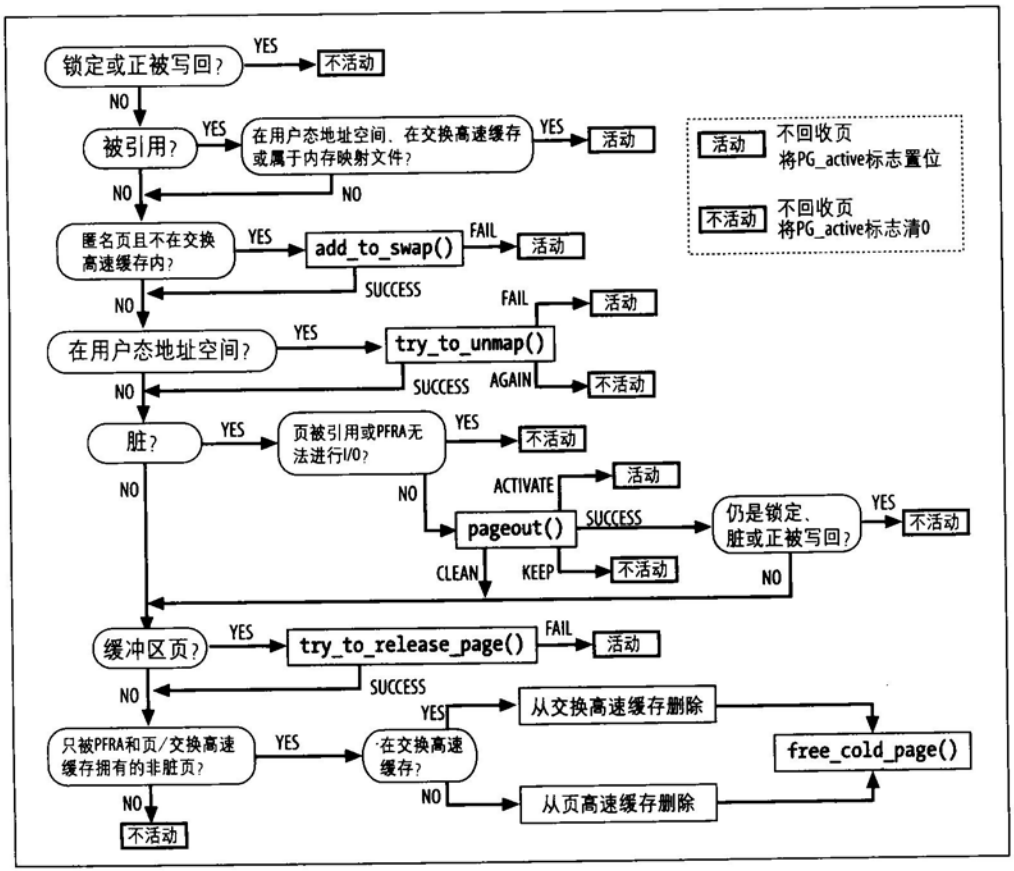

图 17-5：shrink_list()函数的页框回收流程

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_icache_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_icache()，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_icache()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()还不足够吗？

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。在极少的情况下，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子页框回收，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mmput()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*swapoff*命令关闭交换，但此时随着磁盘系统负载增加，很快就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一些坏块，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。

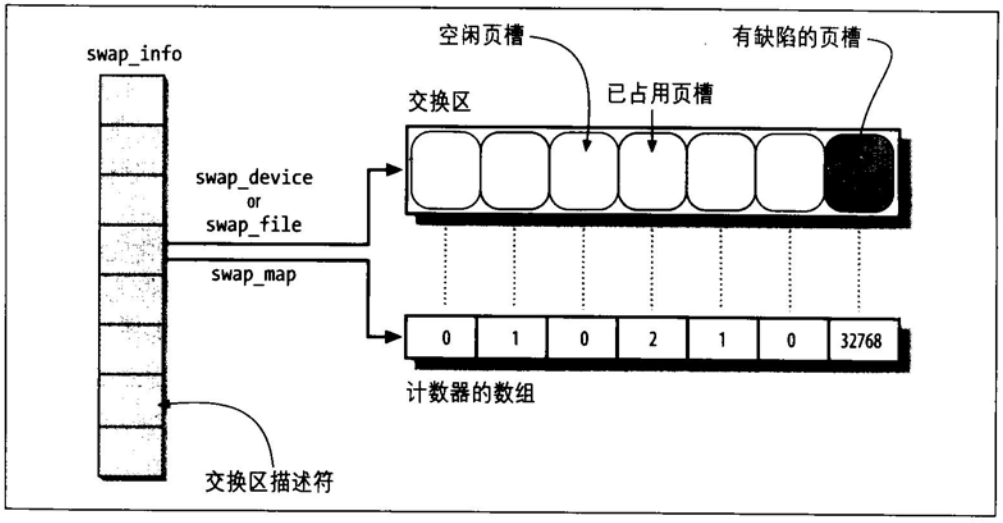

图17-6：交换区数据结构

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。

swap_list_t 类型的 swap_list 变量包括以下字段：

head
: 第一个链表元素在 swap_info 数组中的下标。

next
: 为换出页所选中的下一个交换区的描述符在swap_info数组中的下标。该字段用于在具有空闲页槽的最大优先级的交换区之间实现轮询算法。

swaplock 自旋锁防止在多处理器系统中对链表的并发访问。

交换区描述符的max字段存放以页为单位交换区的大小，而pages字段存放可用页槽的数目。这两个数字之所以不同是因为pages字段并没有考虑第一个页槽和有缺陷的页槽。

最后，nr_swap_pages变量包含所有活动交换区中可用的（空闲并且无缺陷）页槽数目，而 total_swap_pages 变量包含无缺陷页槽的总数。

换出页标识符

可以很简单地而又唯一地标识一个换出页，这是通过在swap_info数组中指定交换区的索引和在交换区内指定页槽的索引实现的。由于交换区的第一个页（索引为0）留给swap_header联合体，第一个可用页槽的索引就为1。换出页标识符的格式如图17-7所示。

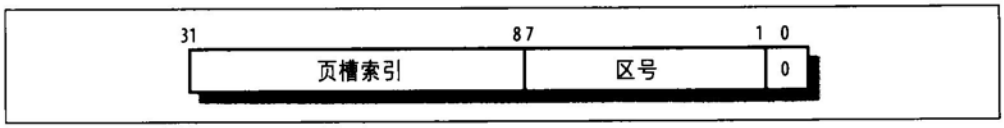

图 17-7：换出页标识符

swp_entry（type,offset）宏负责从交换区索引 type 和页槽索引 offset 中构造换出页标识符。swp_type和swp_offset宏正好相反，它们分别从换出页标识符中提取出交换区索引和页槽索引。

当页被换出时，其标识符就作为页的表项插入页表中，这样在需要时就可以再找到这个页。要注意这种标识符的最低位与 Present 标志对应，通常被清除来说明该页目前不在RAM中。但是，剩余31位中至少有一位被置位，因为没有页存放在交换区0的页槽0中。这样就可以从一个页表项中区分三种不同的情况：

空项
: 该页不属于进程的地址空间，或相应的页框还没有分配给进程（请求调页）。

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，并把它的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*swapoff*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。

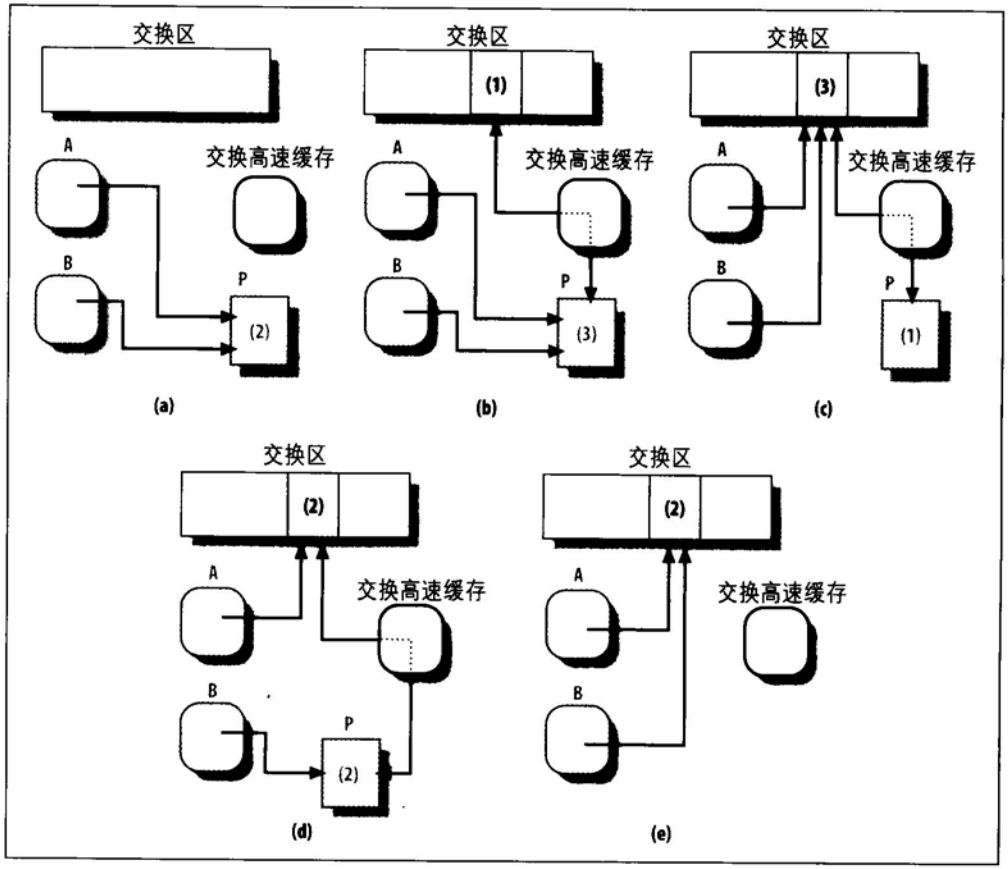

图17-8：交换高速缓存的作用

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0PG_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swapin_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。swapin_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swapin_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。