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：

filp
: 文件对象的地址

buf
: 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

count
: 要读取的字符个数

ppos
: 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filp文件对象的f_pos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
struct list_head	ki_run_list	以后要重新操作的I/O链表指针
long	ki_flags	kiocb描述符的标志
int	ki_users	kiocb描述符的引用计数器
unsigned int	ki_key	异步I/O操作标识符，同步I/O操作标识符为KIOCB_SYNC_KEY (0xffffffff)
struct file *	ki_filp	与正在进行的I/O操作相关的文件对象指针
struct kioctx *	ki_ctx	异步I/O环境描述符指针（参见本章后面的“异步I/O”一节）
int (*) (struct kiocb *, struct io_event *)	ki_cancel	当取消异步I/O操作时所调用的方法
ssize_t (*) (struct kiocb *)	ki_retry	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字段的值除于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. 如果函数跳至这里，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。

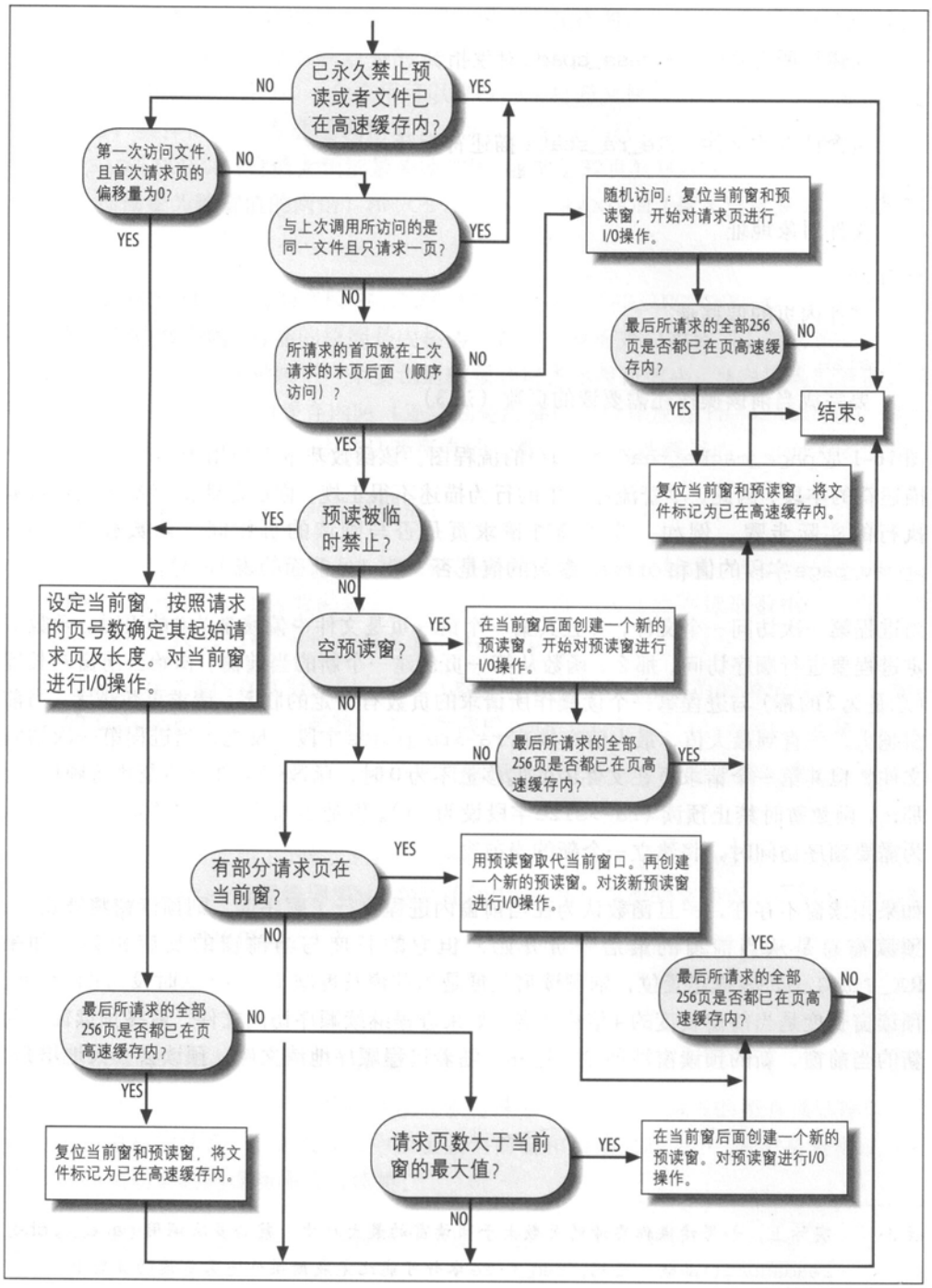

图 16-1：函数 page_cache_readahead()的流程图

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它用于多段情形。

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，然后把这些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_kiocb初始化kiocb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kiocb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writepages方法（如果有定义）或mpage_writepages()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writev()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writepages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writepages方法：

```
int ext2_writepages(struct address_space *mapping, struct
writeback_control *wbc)
{
    return mpage_writepages(mapping, wbc, ext2_get_block);
}
```

你可以看到，该函数是通用mpage_writepages()的一个简单的封装函数。事实上，若文件系统没有定义writepages方法，内核则直接调用mpage_writepages()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_writepages()函数执行下列步骤：

1. 如果请求队列写拥塞，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writepages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writepage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_writepage()实现writepage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writepage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writepage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会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。

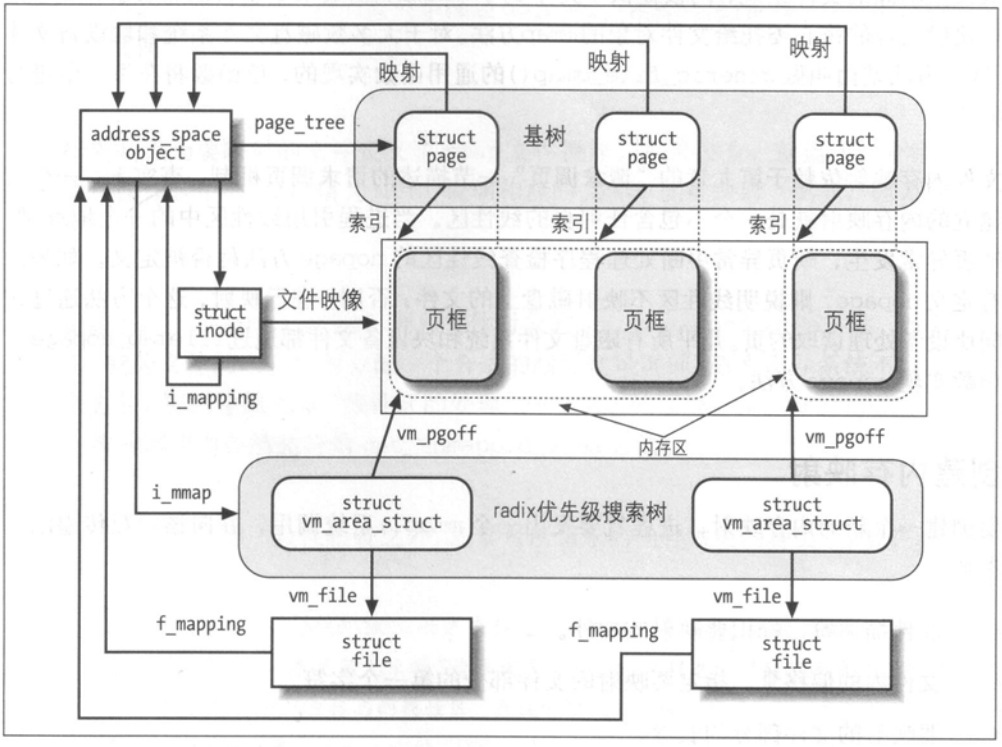

图 16-2：文件内存映射的数据结构

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-pg off()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，当初始化新线性区描述符的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来避免进一步的“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，我们不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都已经被刷新到磁盘。

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writepages方法。如果有，则调用这个函数后返回。如果没有，就执行mpage_writepages()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

iove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->nrpages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iov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
aio_read()	从文件异步读数据
aio_write()	向文件异步写数据
aio_fsync()	请求刷新所有正在运行的异步I/O操作（不阻塞）
aio_error()	获得正在运行的异步I/O操作的错误代码
aio_return()	获得一个已完成异步I/O操作的返回码
aio_cancel()	取消正在运行的异步I/O操作
aio_suspend()	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
io_setup()	为当前进程初始化一个异步环境
io_submit()	提交一个或多个异步I/O操作
io_getevents()	获得正在运行的异步I/O操作的完成状态
io_cancel()	取消一个正在运行的异步I/O操作
io_destroy()	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 kioctx 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 kioctx 描述符存放在一个单向链表中，该链表位于内存描述符的 ioctx_list 字段（参见第九章中的表 9-2）。

我们不再详细讨论kioctx对象。但是我们应当注意一个 kioctx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。kioctx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 kioctx 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kioctx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。