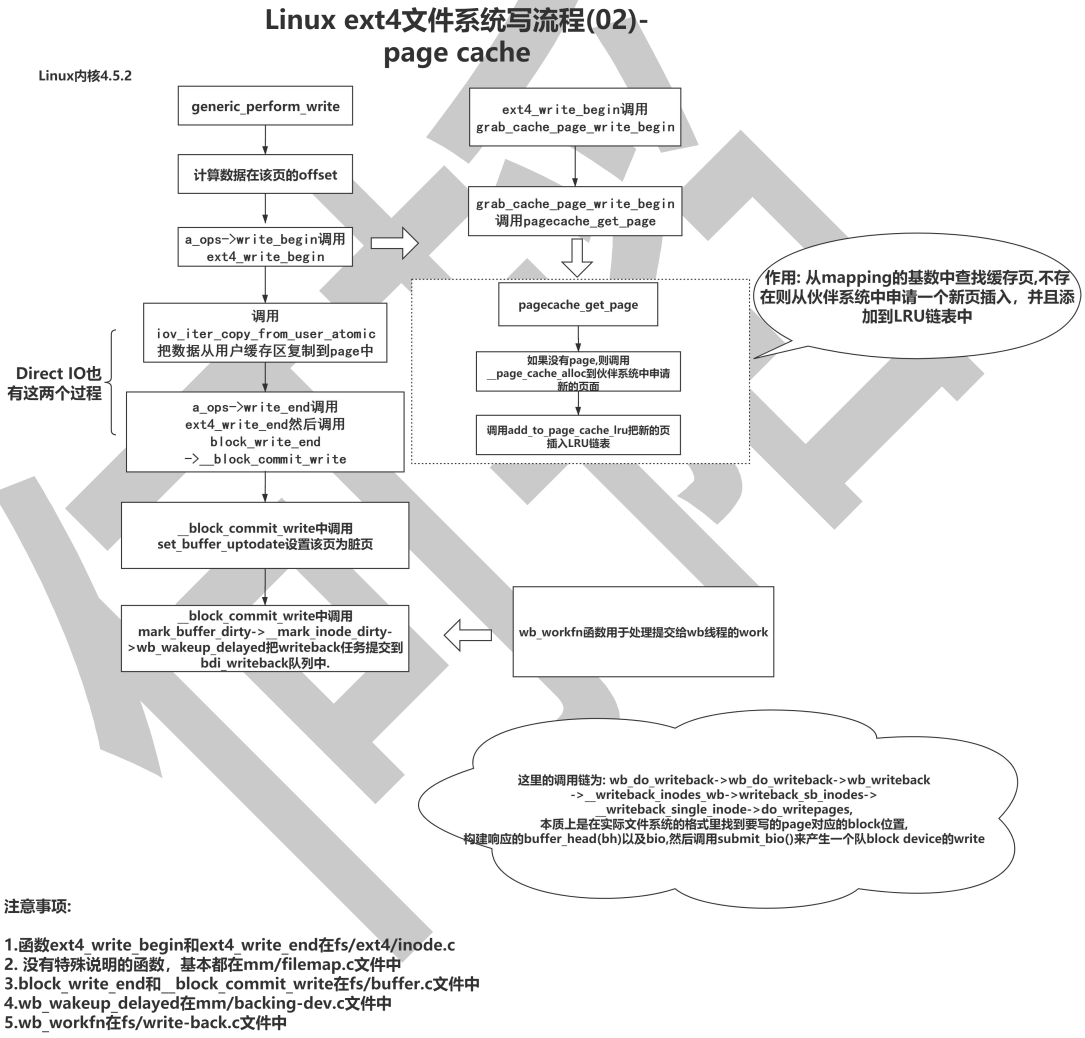
2.1.3. page cahe 的作用

回到前面提的内容，对于一个读写请求来说，经过前面的 VFS 和文件系统的调用，那么下面就会进入到内核调用中，而如果要使用内核缓存，这里就有了 page cache 的存在目的了。下面先来简单看一张代码调用逻辑图，理解一下其中的思路。



对于一个读写请求，如果这里发现在缓存中没有找到对应的数据，那么就需要申请 page 缓存页了，而这里会涉及到 linux 操作系统的内存管理模块伙伴系统和 mapp 相关的内容。而所谓的伙伴系统，如果熟悉 java jvm 的话，其实这里的思想是类似的。伙伴系

统是一个结合了 2 的方幂个分配器和空闲缓冲区合并计技术的内存分配方案，其基本思想很简单．内存被分成含有很多页面的大块，每一块都是 2 个页面大小的方幂．如果找不到想要的块，一个大块会被分成两部分，这两部分彼此就成为伙伴．其中一半被用来分配，而另一半则空闲．这些块在以后分配的过程中会继续被二分直至产生一个所需大小的块．当一个块被最终释放时，其伙伴将被检测出来，如果伙伴也空闲则合并两者。

那么不管内存是如何管理的，申请出来的页面，最后会被存放到一个叫做 LRU 链表进行管理。在 Linux 中，操作系统对 LRU 的实现主要是基于一对双向链表：active 链表和 inactive 链表，这两个链表是 Linux 操作系统进行页面回收所依赖的关键数据结构，每个内存区域都存在一对这样的链表。顾名思义，那些经常被访问的处于活跃状态的页面会被放在 active 链表上，而那些虽然可能关联到一个或者多个进程，但是并不经常使用的页面则会被放到 inactive 链表上。页面会在这两个双向链表中移动，操作系统会根据页面的活跃程度来判断应该把页面放到哪个链表上。页面可能会从 active 链表上被转移到 inactive 链表上，也可能从 inactive 链表上被转移到 active 链表上，但是，这种转移并不是每次页面访问都会发生，页面的这种转移发生的间隔有可能比较长。那些最近最少使用的页面会被逐个放到 inactive 链表的尾部。进行页面回收的时候，Linux 操作系统会从 inactive 链表的尾部开始进行回收。

而内核的内存管理部分，会有三个概念，分别是 node,zone 和 page,简单理解，page 就是一个数据页，zone 是一个区域分组，CPU 被划分为多个节点(node)，内存则被分簇，每个 CPU 对应一个本地物理内存，即一个 CPU-node 对应一个内存簇 bank，即每个内存簇被认为是一个节点。

有了这些缓存的数据页之后，那么数据就会从用户缓存区复制到 page 当中，这里就会

涉及到内核调用，同时为了进一步优化，这里会有零拷贝的技术出现了。当然因为这里并不打算深入具体去了解每一部分的内容，因此如果感兴趣的话，可以自行查阅相关资料。

而当数据存放到内核缓存之后，那么写入的请求会把请求封装成一个 bio 对象，提交到 bdi_writeback 队列中，而这里会有一个 writeback 机制，也就是回写机制，下面就是 block 层的任务了。

2.1.4. Fuse block

为了更好地简单理解读写请求后面要做的事情，可以先看看下面这张图。

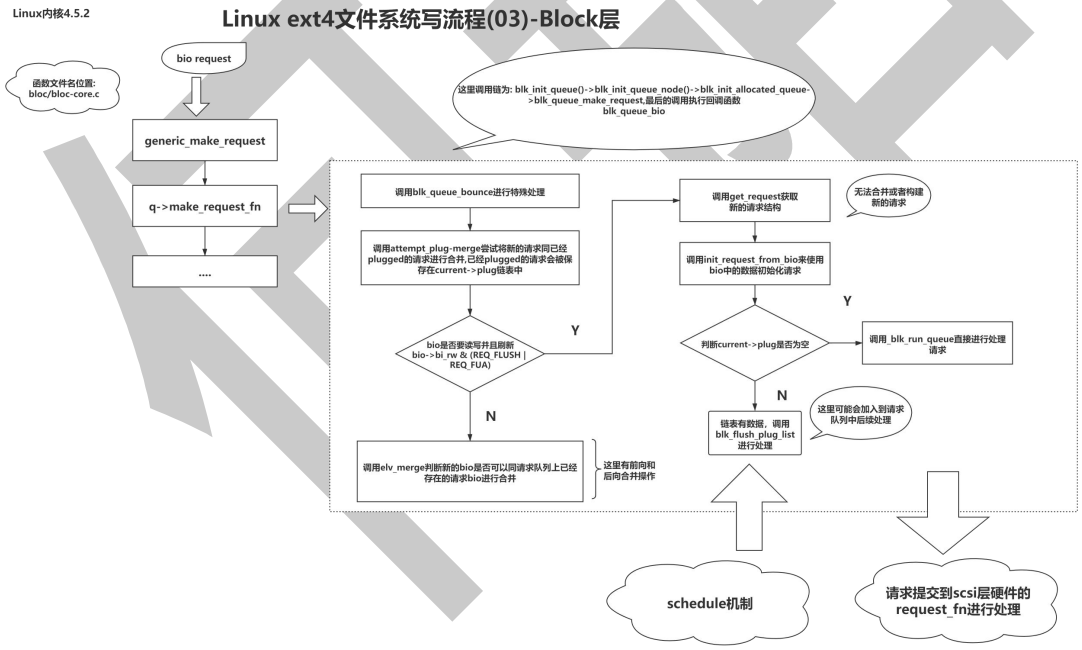


图 2.1.3-1 fuse block 层原理

对于一个 IO 请求，除了前面提到的把数据写入到缓存以外，其实还有一个叫直接 IO 的内容，那么关于 IO 我们常常还有听到同步与异步 IO 的内容，而异步 IO 则只能使用直接 IO 来实现的。另外对于一个 bio 请求来说，这里还会进行前向与后向的合并，之后会放入

到调度队列里面，等待调度算法来进行调度处理。

最后，在经历了一系列的原理的理解之后，我们使用 strace 命令来了解一下，在 linux 系统中创建一个文件时使用的函数调用吧。

```
# strace touch 1.txt
execve("/usr/bin/touch", ["touch", "1.txt"], 0x7ffe4906edc8 /* 24 vars */) = 0
brk(NULL)                               = 0x13ef000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff5b1044000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=25560, ...}) = 0
mmap(NULL, 25560, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5b103d000
close(3)                                = 0
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`&\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2156352, ...}) = 0
mmap(NULL, 3985920, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ff5b0a56000
mprotect(0x7ff5b0c1a000, 2093056, PROT_NONE) = 0
mmap(0x7ff5b0e19000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c3000)
 = 0x7ff5b0e19000
mmap(0x7ff5b0e1f000, 16896, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x
7ff5b0e1f000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff5b103c000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff5b103a000
arch_prctl(ARCH_SET_FS, 0x7ff5b103a740) = 0
mprotect(0x7ff5b0e19000, 16384, PROT_READ) = 0
mprotect(0x60d000, 4096, PROT_READ)     = 0
mprotect(0x7ff5b1045000, 4096, PROT_READ) = 0
munmap(0x7ff5b103d000, 25560)           = 0
brk(NULL)                               = 0x13ef000
brk(0x1410000)                          = 0x1410000
brk(NULL)                               = 0x1410000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=106176928, ...}) = 0
mmap(NULL, 106176928, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5aa513000
close(3)                                = 0
open("1.txt", O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK, 0666) = 3
dup2(3, 0)                              = 0
close(3)                                = 0
```

```
utimensat(0, NULL, NULL, 0)             = 0
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

这里有常见的 mmap 和 brk 是和内存管理分配有关的函数，open 是文件打开的函数等，其中 open 函数中的参数 O_CREAT 是创建并打开一个新文件,关于这些不同的函数的作用与参数意义，可以根据需要时去查阅接口文档。

章节语:

这一章我们主要是理解了一些 linux 文件系统中的概念，一个正常的读写请求所经过的层次结构，而这里面涉及到内存管理，进程调度等模块，本章也只是非常粗糙简略地讲解了 VFS 和缓存的一些内容，而这个章节的内容，也是为了方便理解后续 glusterfs fuse 中的一些参数做准备的，因为 glusterfs fuse 的底层也是很多我们所熟悉的系统调用，而且 glusterfs fuse 是一个用户文件系统。