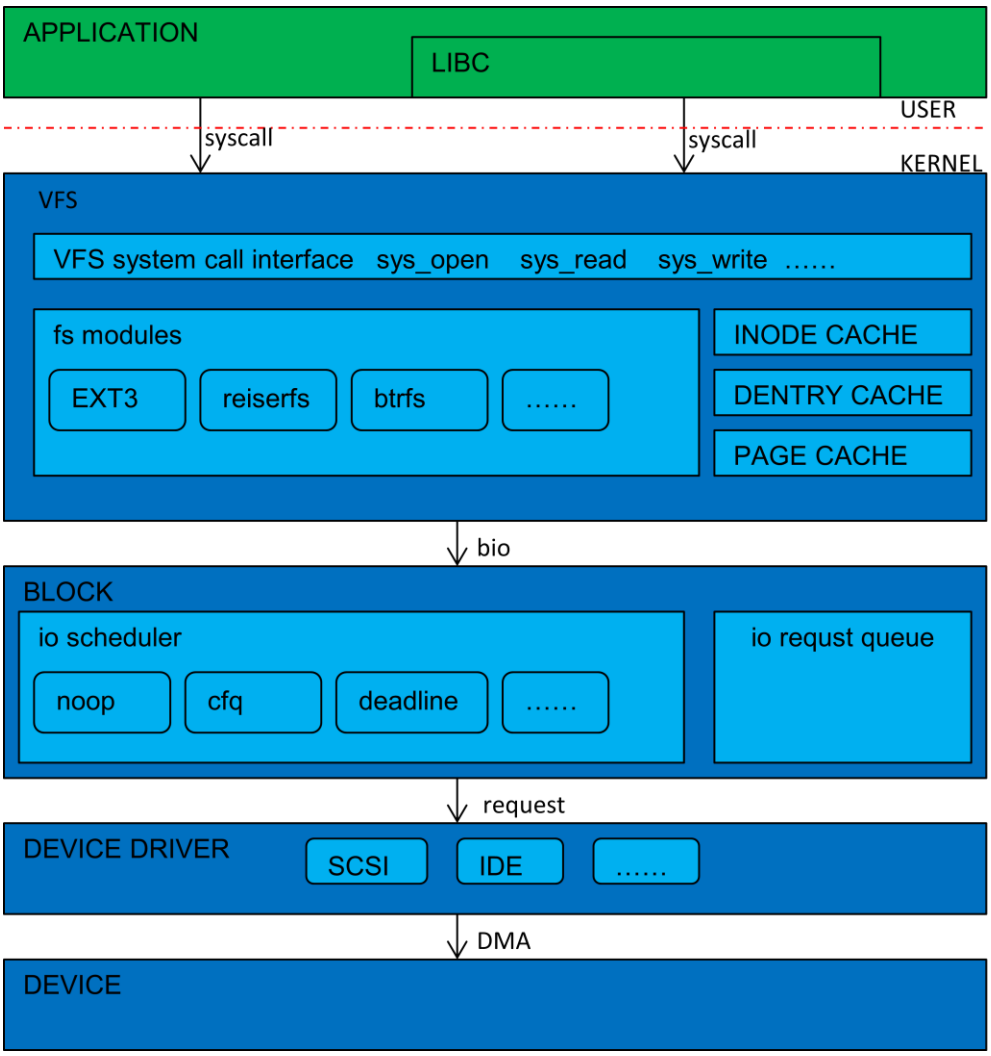
Linux IO子系统知识总结

2013.3 Version 1.01

yalung929@gmail.com

1 IO层次图



2 相关概念和定义

2.1 文件系统层

2.1.1 VFS

VFS建立了文件系统的统一模型，对文件系统的公共属性做了抽象和定义，封装了文件系统的统一界面。应用程序可以不加修改的、透明的访问操作不同文件系统上的文件。

同时VFS也提供了一些文件系统的基础能力比如cache。

我们可以这样理解：VFS是一个文件系统框架，具体的文件系统比如ext4是一个插件。

2.1.2 文件与目录

文件就是单纯的字节流，应用程序告诉内核读取偏移和长度，内核返回字节流，应用程序自己解读内容，在内核层不存在文本和二进制等文件类型的区别。

目录是文件的集合。在VFS模型中一个目录也是一个文件。（目录具备文件的所有属性，当然目录还有一些独特的东西）。

2.1.3 文件路径

在VFS模型中一个目录可以包含多个文件或子目录，子目录中可以继续包含其他文件和孙目录，这是一个树状结构。这个树上的任何一条自上往下[1]的分支或节点，就是一个文件路径。从根部开始的分支，称为绝对路径；否则称为相对路径。

VFS是单根的，所有的目录或文件其最顶层目录都是根“/”目录。

2.1.4 FILE

每个被打开的文件，内核为之分配一个文件对象，VFS使用struct file描述它。这个结构的关键成员就是**当前读写位置和操作文件的一系列函数指针**。

```
struct file {
    ……
```

[1] 根在上。

```
    struct path f_path;
    const struct file_operations *f_op;      //文件操作
    spinlock_t f_lock;
    int f_sb_list_cpu;
    atomic_long_t f_count;
    unsigned int f_flags;
    fmode_t f_mode;
    loff_t f_pos;                            //文件读写位置
    ……
}
crash> struct file_operations 0xffffffff8162e1c0   //一个位于ext4文件系统上的文件关联的file_operations
struct file_operations {
  owner = 0x0,
  llseek = 0xffffffff811dd120 <ext4_llseek>,
  read = 0xffffffff8114b140 <do_sync_read>,
  write = 0xffffffff8114b050 <do_sync_write>,
  aio_read = 0xffffffff810f25e0 <generic_file_aio_read>,
  aio_write = 0xffffffff811dd2b0 <ext4_file_write>,
  mmap = 0xffffffff811dd0d0 <ext4_file_mmap>,
  open = 0xffffffff811dcf10 <ext4_file_open>,
  release = 0xffffffff811dce50 <ext4_release_file>,
……
}
crash> struct file_operations 0xffffffff81658a40 //null 文件，内核为它定义了专门的操作函数。
struct file_operations {
  owner = 0x0,
  llseek = 0xffffffff8135aae0 <null_lseek>,
  read = 0xffffffff8135aaa0 <read_null>,
  write = 0xffffffff8135aab0 <write_null>,
}
```

每一个文件对象都包含打开、关闭、读、写等一系列文件操作的函数指针，不同文件系统注册不同的函数。

2.1.5 FD

每次打开（open）一个文件都会分配一个file对象，在此之外还要分配一个数字化的序号，称作文件描述符（fd），俗称句柄。句柄的作用在于方便打开之后的操作——你不需要再传送文件路径给内核，只需要一个序号：

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

文件描述符（fd）与文件对象（file）是属于进程的，每一个进程都有一个独立的已打开文

件列表。

我们可以通过proc查询到它们。

```
# ls -l /proc/17590/fd/
total 0
dr-x------ 2 root root    0 Jan    4 15:17 ./
dr-xr-xr-x 8 root root    0 Jan    4 15:16 ../
lrwx------ 1 root root 64 Jan    4 15:17 0 -> /dev/pts/1
lrwx------ 1 root root 64 Jan    4 15:17 1 -> /dev/pts/1
lrwx------ 1 root root 64 Jan    4 15:17 2 -> /dev/pts/1
lrwx------ 1 root root 64 Jan    4 15:17 3 -> /dev/null
lrwx------ 1 root root 64 Jan    4 15:17 4 -> /dev/mem
lr-x------ 1 root root 64 Jan    4 15:17 5 -> /usr/src/linux-3.1.10-1.9/vmlinux*
```

如上，进程17590打开了6个文件，分别对应fd 0~5，其中0~2都指向同一个文件/dev/pts/1。

我们使用crash的files命令可以查看更详细的信息：

```
crash> files 17590
PID: 17590    TASK: ffff880021e2a6c0    CPU: 0     COMMAND: "crash"
ROOT: /       CWD: /root
 FD          FILE                 DENTRY              INODE          TYPE PATH
  0 ffff880114827d80 ffff880023a663c0 ffff8801178b4da0 CHR    /dev/pts/1
  1 ffff880114827d80 ffff880023a663c0 ffff8801178b4da0 CHR    /dev/pts/1
  2 ffff880114827d80 ffff880023a663c0 ffff8801178b4da0 CHR    /dev/pts/1
  3 ffff880114869cc0 ffff88011b175a40 ffff88011895b830 CHR    /dev/null
  4 ffff88011484ed80 ffff88011b175bc0 ffff88011895bd40 CHR    /dev/mem
  5 ffff88011393b0c0 ffff880117547780 ffff88011754e7f0 REG    /usr/src/linux-3.1.10-1.9/vmlinux
  6 ffff88011393b5c0 ffff88011b2a4d40 ffff880112716930 FIFO
  7 ffff88011449e780 ffff88011b2a4d40 ffff880112716930 FIFO
  8 ffff880114092880 ffff8801174d6600 ffff880117793170 REG    /tmp/tmpf8XptMP
 10 ffff8801153a1d80 ffff8801174f7180 ffff8801127166f8 FIFO
```

可以看到同一个文件对象（file ffff880114827d80）可以分配给多个文件描述符fd，这是通过执行dup（man 2 dup）系统调用做到的。在这种情况下，由于共享同一个文件对象，我们通过fd 0进行操作从而导致文件读写位置的改变，通过fd 1和fd 2都可以感知到。

2.1.6 INODE

元数据是指描述数据本身的数据。文件内容之外的数据，比如文件时间戳、大小、属主、访问权限、文件内容的位置等等， 就是文件的元数据。我们使用stat命令可以看到部分：

```
# stat /etc/fstab
```

```
  File: `/etc/fstab'
  Size: 633             Blocks: 8          IO Block: 4096   regular file
Device: 802h/2050d      Inode: 2050        Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2012-12-31 10:13:54.172298406 +0800
Modify: 2012-06-04 09:58:16.039783121 +0800
Change: 2012-06-04 09:58:16.067783125 +0800
```

VFS使用struct inode描述一个文件的元数据。

实际上inode就是表示了文件系统上的文件实体，二者一一对应。

一个文件对应一个inode对象。 不同的进程打开相同的文件，分配不同的file，但内核中只存在一个与此文件对应的inode。每一个文件都有一个inode序号，在同一个文件系统内该序号唯一。

对文件元数据的操作函数指针保存在struct inode内：

```
crash>struct inode |grep operations
const struct inode_operations *i_op;
crash> struct inode_operations 0xffffffff8162e0c0
struct inode_operations {
  ......
  setattr = 0xffffffff811e5190 <ext4_setattr>,
  getattr = 0xffffffff811e39e0 <ext4_getattr>,
  ......
}
```

2.1.7 DENTRY

目录也是一个文件，目录这个文件的内容就是指向其他文件（或目录）的目录项。

内核使用struct dentry来描述一个目录项。

构成文件路径上的每一个节点都有一个dentry与之对应，包括最末端的文件节点。

dentry的最重要的一个字段就是名字（d_name）， 也就是该目录项对应的文件或目录的名字——**文件的名字并不保存在这个文件（的inode）里面，而是保存在它所在的目录里面。可以说文件名是目录这个特殊文件的主要内容。**

看一个实例：

/a/d.txt

“/” 对应一个dentry，其d_name是“/”，这个是万物之起源，我们可以认为是天然存在

的。

在“/” 目录内有指向很多目录和文件的目录项，其中一个的d_name是“a”。

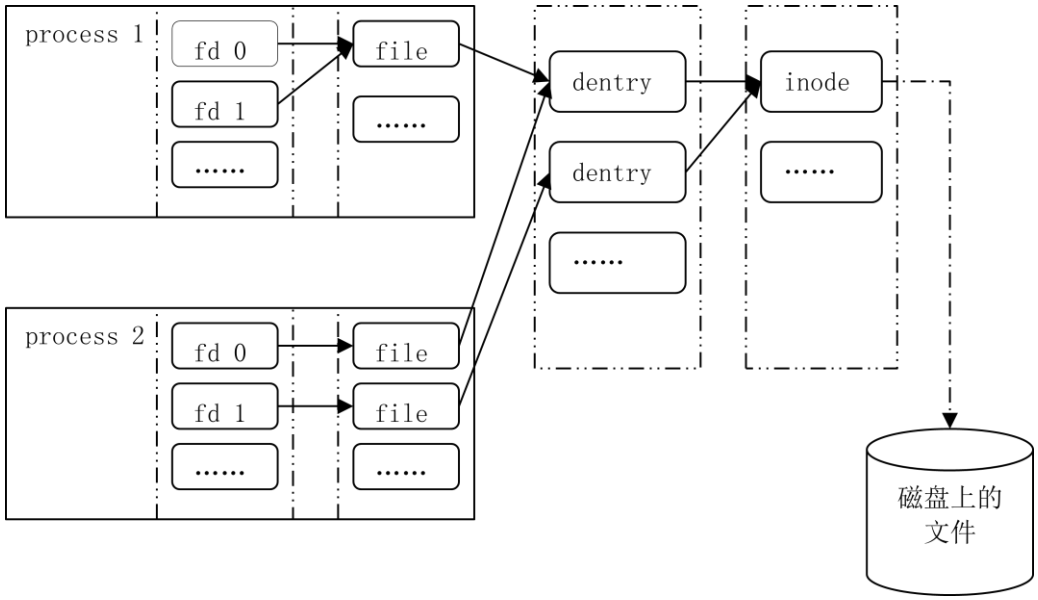
在“a” 目录内有指向很多目录和文件的目录项，其中一个的d_name是“d.txt”。

如果两个不同的目录项指向同一个文件，是什么情况？这种情况就是文件链接(hard link)，即我可以通过两个不同的路径访问同一个文件。这种情况下，该文件对应的inode仍然只有一个，但dentry会有多个。

同时联想到符号链接（soft link）的情况，在这种情况下，链接文件本身是一个独立的文件，有自己的inode，只不过这个文件被定义为特殊类型（symbolic link），这种类型文件的内容就是对其他文件的指向信息。对这些文件的读写等操作,VFS获取到它所指向的文件，然后把操作都映射过去。

```
# ll /lib64/libz.so.1     #inode不同的两个独立文件。
lrwxrwxrwx 1 root root 13 May 21   2010 /lib64/libz.so.1 -> libz.so.1.2.3
# stat libz.so.1
  File: `libz.so.1' -> `libz.so.1.2.5.2'
  Size: 15              Blocks: 0          IO Block: 4096   symbolic link
Device: 802h/2050d      Inode: 786928      Links: 1
# stat libz.so.1.2.5.2
  File: `libz.so.1.2.5.2'
  Size: 88376           Blocks: 176        IO Block: 4096   regular file
Device: 802h/2050d      Inode: 786544      Links: 1
```

2.1.8 FILE DENTRY INODE 之间的关系



如图所示：

一个磁盘上的文件，有两个hard link，在内核中：

有一个唯一的inode对象；

两个hard link都被访问了，对应两个dentry对象；

不同的进程打开了这个文件多次，有多个file对象指向这两个dentry对象；

在进程1中，对fd进行了dup操作，有两个fd指向同一个file对象。

file的f_path字段保存了其所属的dentry。

```
crash> struct file.f_path
struct file {
    [16] struct path f_path;
}
crash> struct path
struct path {
      struct vfsmount *mnt;
      struct dentry *dentry;
}
```

dentry的d_inode字段保存了其所属的inode。

```
crash> dentry.d_inode
struct dentry {
    [48] struct inode *d_inode;
```

}

整个文件系统层都围绕这三个对象干活儿，搞清楚它们的含义以及它们之间的关系至关重要。

2.1.9 文件系统

一个具体的文件系统比如ext4其核心工作就是完成文件在磁盘上的布局，即：

如何放置目录、如何放置目录中的项、如何放置元数据、如何放置文件内容……

通常一个完整的基于磁盘的文件系统包括以下三个内容：

1. 有一个工作在用户层的格式化程序（mkfs.ext4）对磁盘进行格式化，完成磁盘空间的初始分配和布局。
2. 有一个工作在用户层的挂载程序（mount.ext4[2]），读取磁盘上的文件系统信息，完成目录挂载。
3. 有一个工作在内核层的驱动程序(ext4.ko)， 注册一系列VFS层定义的操作函数，把VFS层对文件对象的操作转化为对磁盘扇区的操作。

2.1.10 挂载点

一个分区被格式化成一个文件系统之后，需要挂载到一个目录节点内，才可以访问，这个目录被称为挂载点，是被挂载文件系统的根节点。

挂载点可以是位于其他文件系统之上的任意目录。

VFS使用struct vfsmount来描述一个挂载点。

2.1.11 超级块

超级块存在于被格式化的磁盘分区之上，保存文件系统的管理控制信息。

超级块上通常保存着：

1. 该文件系统的格式化参数，比如块的大小、文件系统名称、版本等等。
2. 该文件系统的全局运行信息，比如inode（文件）的数量、空闲块数等等。
3. ……

[2] mount命令通常已经默认支持并理解常见的文件系统，mount.ext4这个子命令并不需要。

超级块具体怎么放、放置什么信息以及大小，全是文件系统自己的事，超级块的内容只有该文件系统自己理解。

VFS使用struct super_block来描述一个超级块，但这个结构其实并不保存磁盘上的超级块内容。

super_block最重要的成员就是文件系统驱动模块注册的针对文件系统而非文件的一些操作函数，比如创建inode（文件），这也是VFS最关心和需要的：

```
crash> super_block  |grep  super_operations
const struct super_operations *s_op;

crash> super_operations 0xffffffff8162ef80
struct super_operations {
   alloc_inode = 0xffffffff811f1720 <ext4_alloc_inode>,
   destroy_inode = 0xffffffff811fc5d0 <ext4_destroy_inode>,
   dirty_inode = 0xffffffff811e7360 <ext4_dirty_inode>,
   write_inode = 0xffffffff811e38d0 <ext4_write_inode>,
   drop_inode = 0xffffffff811f1660 <ext4_drop_inode>,
   evict_inode = 0xffffffff811e6f40 <ext4_evict_inode>,
   put_super = 0xffffffff811fdfa0 <ext4_put_super>,
```

系统中已有的挂载点以及对应的super_block对象，VFS使用一个链表维护。
可以通过/proc/mounts文件获得系统中的挂载点信息。

2.1.12 CACHE

与内存相比，显然磁盘太慢了，不能每次查看一下文件大小，都让磁头转吧。对inode、dentry以及文件内容都要缓存。对于文件系统而言缓存太重要了的，所以VFS层提供了一套公共的缓存机制：

1. 所有的inode对象，建立一个基于slab[3]管理的cache，减少磁盘访问。
2. 所有的dentry对象，建立一个基于slab管理的cache，减少磁盘访问。
3. 文件的内容全部映射至内存，这部分内容称为page cache[4]。

通过 /proc/sys/vm/drop_caches 这个内核控制参数我们可以控制这些cache的释放：

```
echo 1 > /proc/sys/vm/drop_caches #释放dentry和inode。
echo 2 > /proc/sys/vm/drop_caches #释放page cache。
```

[3] Linux内核的小片内存管理器，一个基于对象的内存池。

[4] 就是free命令显示的cache字段。

echo 3 > /proc/sys/vm/drop_caches #释放dentry、inode和page cache。

2.1.13 PROC 文件系统

PROC文件系统才是VFS最强大的利用。

PROC是Linux[5]系统下我最喜欢的设计之一，比起Windows使用注册表、函数调用的形式管理控制系统运行状态；Linux通过一个虚拟文件系统使用已有的简单文件读写命令就可以完成丰富的管理控制，这简直是太棒了！

PROC是一个文件系统，可以被挂载到一个目录，文件系统上有很多文件，使用普通的文件操作命令比如cat、echo就可以访问这些文件——达到和内核通讯修改内核参数或者查询内核运行状态的目的。

其工作原理与ext4这些磁盘文件系统没有什么区别，只不过注册给VFS一系列的函数接口的实现内部，不去访问磁盘，而是直接访问内核相关数据结构而已——

我们读一个proc上的文件，VFS会调用proc注册的proc_xxx_read，proc_xxx_read帮我们从磁盘上把文件内容读出来。但到底是否真的访问磁盘，VFS层并不关心和知道这一点，完全可以瞎捏一个文件内容出来！要是不瞎捏，搞点有意义的，比如我访问某个文件，你内核就把当前的内存使用信息返回给我。比如这样：

```
# cat /proc/meminfo
MemTotal:        3923860 kB
MemFree:          110712 kB
Buffers:          770684 kB
Cached:          2646260 kB
SwapCached:         1816 kB
Active:           747372 kB
Inactive:        2768848 kB
Active(anon):      73844 kB
Inactive(anon):    26000 kB
Active(file):     673528 kB
Inactive(file):  2742848 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       2104476 kB
SwapFree:        2099268 kB
```

有了/proc/meminfo之后，我们并不需要专门写一个通过调用系统API函数查询内存信息的程

[5] 当然这个设计是从UNIX系统沿袭下来的。敬拜Unix大神。

序。 我们只需要使用通用的文件操作命令就可以完成，如果再和grep awk等文本处理命令组合，形成一个脚本就可以了完成很多系统运行数据的统计和报告。

当然作为一个完整的操作系统， Linux上通常提供了一些专门的系统工具，比如free top vmstat等；这些工具无一例外都是通过读取proc文件系统上的相关文件实现的。

2.2 块层

2.2.1 磁盘

磁盘，没什么好说的。内核使用struct gendisk描述一个磁盘。

2.2.2 块设备

一个磁盘以及一个分区都是一个块设备，块设备以块为单位进行访问。

一个块设备对应一个设备文件，位于/dev目录。

内核使用struct block_device来描述它。

2.2.3 块

块是访问块设备的基本单位。所有下发给块设备的读写操作，都是以块为单位。

这意味着我写某个文件的某一个字节， 实际上可能会导致从磁盘上读取该字节所在的整个块（假设还没有在page cache中），然后再修改这一个字节，最后再把整个块写入磁盘。

过小的块可能会降低效率，过大的块可能会浪费空间。

块大小是格式化分区建立文件系统时的最重要参数。

通常Linux X86上文件系统的默认块大小是4096字节，这正好是一个内存页的大小。

这也就意味着1个字节的文件，实际占用4096字节的大小，如果有太多小文件，空间浪费就很严重；但各个文件系统对零碎小文件的存储可能有些优化，也许会多个文件共用一个块。

显然块不能小于一个扇区的大小——512字节。

2.2.4 BIO

struct bio封装对一个块设备的IO读写。包含了底层设备执行操作的信息：

1. bi_sector 起始扇区；
2. bi_size 读写的大小；
3. bi_io_vec 读写缓冲区；bi_io_vec是一个struct bio_vec数组，每一个bio_vec描述了一个缓冲区片段，每一个片段由所在的内存页、所在内存页中的偏移、长度构成。

```
struct bio_vec {
struct page *bv_page; //哪个页
unsigned int     bv_offset; //从该页中哪里开始
unsigned int     bv_len;    //长度
};
```

从BIO的设计我们可以看出：

1. 一次块设备访问中，对块设备的访问地址是连续的（扇区）。
2. 读写的内存缓冲区可以不连续，分散到不同的内存页面。

submit_bio 函数是块层的入口函数，接受两个参数：读写标记以及bio。

```
void submit_bio(int rw, struct bio *bio);
```

bio由文件系统层构造，**把对文件某字节的操作，转化为对磁盘某个扇区的操作**——某个文件的某字节躺在磁盘上哪个扇区，也只有文件系统知道这件事儿，这正是文件系统的活儿~。

2.2.5 IO 请求

进入块层之后，bio继续被封装为struct request下发给底层设备。

一个request可以包含一个或多个连续的bio（访问的扇区连续）。

2.2.6 IO 请求队列

每一个磁盘（NOT 分区）都有一个IO请求队列。内核使用struct request_queue描述它。

```
crash> dev -d
MAJOR GENDISK    NAME    REQUEST QUEUE      TOTAL ASYNC  SYNC    DRV
```

```
8    0xffff8801189a7400 sda          0xffff8801153860b8       0       0       0       0
```

顾名思义，request_queue上放的都是request。

2.2.7 IO 调度算法

进入request_queue中的request，会使用IO调度算法（又称电梯算法）合并、排序，以提高磁头移动的效率（尽量减少来回摆动）。

队列当前所使用的调度算法保存在elevator字段。

```
crash> struct request_queue.elevator
struct request_queue {
      [24] struct elevator_queue *elevator;
}
crash> struct elevator_queue 0xffff880114cdc240
struct elevator_queue {
   ops = 0xffffffff81a3feb0,

crash> struct elevator_ops 0xffffffff81a3feb0
struct elevator_ops {
   elevator_merge_fn = 0xffffffff8129d1e0 <cfq_merge>,
   elevator_merged_fn = 0xffffffff8129e3c0 <cfq_merged_request>,
   elevator_merge_req_fn = 0xffffffff8129d750 <cfq_merged_requests>,
   elevator_allow_merge_fn = 0xffffffff8129cc30 <cfq_allow_merge>,
   elevator_bio_merged_fn = 0xffffffff8129d1b0 <cfq_bio_merged>,
   elevator_dispatch_fn = 0xffffffff812a0240 <cfq_dispatch_requests>,
   elevator_add_req_fn = 0xffffffff812a06a0 <cfq_insert_request>,
   ……
}
```

IO调度算法通过sysfs[6]暴露出来，可以在线查看和调整：

```
# cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
```

这些算法的代码定义在：

```
# ls block/*iosched.c
block/as-iosched.c    block/cfq-iosched.c    block/deadline-iosched.c    block/noop-iosched.c
```

[6] 一个类似proc的虚拟文件系统。

3 关键流程分析

3.1 文件系统挂载流程

已经格式化并建立文件系统的分区，挂载之后才可以使用。

我们以mount -t ext4 /dev/sda2 /mnt为例说明文件系统挂载流程。

3.1.1 mount 命令

挂载流程始于mount命令，但文件系统对应的模块应先加载到内核中[7]，使内核具备理解所挂载文件系统的能力。

系统当前已经加载的文件系统类型保存在一个全局链表中。

```
static struct file_system_type *file_systems;
```

通过/proc/filesystems我们可以查看到。

mount 命令的核心工作就一个，调用mount系统调用（man 2 mount），让内核完成挂载：

```
mount("/dev/sda2", "/mnt/", "ext4", MS_MGC_VAL, NULL);

int mount(const char *source, const char *target,
                const char *filesystemtype, unsigned long mountflags,
                const void *data);
```

在调用mount系统调用之前，mount命令会做两个工作：

1. 在用户没有通过 -t 参数指明文件系统类型的情况下，尝试通过直接读取分区上的内容确定文件系统类型。常见的类型它内部已经支持，知道类型特征。
2. 根据文件系统类型，查找是否有对应的mount.fstype子命令，如果有就调用子命令完成。

```
stat("/sbin/mount.ext4", 0x7fff6c7ee800) = -1 ENOENT (No such file or directory)
stat("/sbin/fs.d/mount.ext4", 0x7fff6c7ee800) = -1 ENOENT (No such file or directory)
stat("/sbin/fs/mount.ext4", 0x7fff6c7ee800) = -1 ENOENT (No such file or directory)
```

3.1.2 内核处理流程

内核对mount系统调用的定义：

```
SYSCALL_DEFINE5(mount, char __user *, dev_name, char __user *, dir_name,
```

[7] 其实内核有个机制在挂载的时候可以自动加载对应的文件系统模块。参见3.1.2 内核处理流程。

```
                char __user *, type, unsigned long, flags, void __user *, data)
{
        int ret;
        char *kernel_type;
        char *kernel_dir;
        char *kernel_dev;
        unsigned long data_page;

        ret = copy_mount_string(type, &kernel_type);
        if (ret < 0)
                goto out_type;

        kernel_dir = getname(dir_name);
"fs/namespace.c" 2332 lines --89%--
```

早期的内核版本中，查找一个系统调用的代码定义很方便，一般是系统调用名字增加sys_前缀。现在都使用SYSCALL_DEFINEX宏封装了(展开后仍然是sys_mount)，SYSCALL_DEFINE5表示一个带有5个参数的系统调用定义。

内核还有很多其它地方通过宏展开定义一个变量或函数，这种情况下使用Source Insight、cscope等代码浏览工具查找或跳转到定义会玩不转，要是代码浏览工具支持对宏展开后的符号名建立索引就好了，这个并不难，却大有用处。或许有什么配置我还不知道。

mount的真正流程开始于长长的 SYSCALL_DEFINE5(mount …：

1. 拷贝和准备参数以及一些必要的检查。（sys_mount->do_mount）
2. 获取挂载点（/mnt）的dentry 和 vfsmount对象。

   这个vfsmount对应的是mnt目录当前所在文件系统的挂载点，这个获取过程与打开文件的流程相同。（参见3.2打开文件流程）
3. 获取文件系统类型的描述结构。（do_kern_mount-> get_fs_type）

   *struct file_system_type *type = get_fs_type(fstype);*

   get_fs_type 首先从已注册文件系统类型链表file_systems中查找，如果没找到会尝试加载以文件系统类型名字命名的模块。（get_fs_type->request_module）
4. 为挂载点/mnt准备一个新的vfsmount对象。（vfs_kern_mount-> alloc_vfsmnt）
5. 读取分区上的超级块获取文件系统信息，构造super_block对象并填充vfsmount。

这个过程通过调用file_system_type中的get_sb函数由具体的文件系统完成。

（vfs_kern_mount ： type->get_sb）

```
static struct file_system_type ext4_fs_type = {
        .owner          = THIS_MODULE,
        .name           = "ext4",
        .get_sb         = ext4_get_sb,
        .kill_sb        = kill_block_super,
        .fs_flags       = FS_REQUIRES_DEV,
};
```

get_sb 由文件系统模块在加载时注册。

每个文件系统上都有一个根节点“/”（root inode）， 无论是否被挂载，这个root inode在文件系统第一次创建时就有了，get_sb会把root inode对应的dentry赋给vfsmount的mnt_root字段以及super_block的s_root保存下来。

```
crash> vfsmount.mnt_root
struct vfsmount {
    [32] struct dentry *mnt_root;
}
crash> super_block.s_root
struct super_block {
    [96] struct dentry *s_root;
}
```

6. 把vfsmount对象和挂载点(/mnt)关联起来。（mnt_set_mountpoint）

   vfsmount的mnt_parent字段指向/mnt目录原来文件系统对应的vfsmout。实际上就是系统/目录所对应的vfsmount。

   vfsmount的mnt_mountpoint字段指向/mnt目录对应的dentry。

   并增加/mnt dentry的挂载计数，使用d_mounted字段保存。**这个字段大于0表示这个目录是一个挂载点。**

7. 把vfsmount对象插入全局的保存所有vfsmount对象的hash表mount_hashtable。

   （commit_tree）

   这是为了加速挂载点的查找访问。

8. 把vfsmount对象插入其父vfsmount对象子vfsmount对象链表。（commit_tree）

   这步操作使得整个系统的挂载点构成一个单根树状结构，从/挂载点开始，可以找到所有的挂载点。

到此挂载就处理完了。当我们再次访问/mnt目录的时候，内核发现/mnt的dentry的*d_mounted*字段非0，说明它一个挂载点，就会去mount_hashtable查找到它对应的vfsmount，vfsmount保存了文件系统的root inode和super_block，进而就可以转为对这个文件系统的访问了。**由此可见挂载的本质就在于对一个目录做了个特殊标记，在有这个标记的情况下，任何对此目录的访问都被转为对所挂载文件系统根目录的访问。**

成为挂载点之前，/mnt的inode位于/所在文件系统上，inode是***868353*** ：

```
# stat /mnt/
  File: `/mnt/'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 802h/2050d      Inode: 868353      Links: 3
```

成为挂载点之后，/mnt被替换为被挂载文件系统的root inode。原来的/mnt目录内容就访问不了。

```
# stat /mnt/
  File: `/mnt/'
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 802h/2050d      Inode: 2           Links: 27 #ext4文件系统的root inode号固定是2
```

3.2 打开文件流程

打开文件的过程就是查找并构建文件对应dentry、inode和file对象的过程。

应用程序调用C库的fopen，fopen调用系统调用open，系统调用open的内核定义：

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)
{
        long ret;
        if (force_o_largefile())
                flags |= O_LARGEFILE;
        ret = do_sys_open(AT_FDCWD, filename, flags, mode);
        /* avoid REGPARM breakage on x86: */
"fs/open.c" 1199 lines --85%--
```

内核处理打开文件的主要流程（*以open("/mnt/success", O_RDONLY);为例*）：

1. 分配一个文件描述符fd。（do_sys_open->get_unused_fd_flags）

每个进程有一个fd table，保存在task_struct[8]的files字段。

```
crash> task_struct.files
struct task_struct {
  [1368] struct files_struct *files;
}
```

先从fd table获取空闲的，如果不行就申请新的fd。（alloc_fd->expand_files）

这时候就会检查ulimit -n（控制进程句柄总数）和fs.nr_open（控制全局句柄总数）的设置。

2. 分配一个文件对象file。（path_lookup_open->get_empty_filp）

   这里会检查fs.file-max（控制全局文件对象总数）的设置。

3. 查找文件对应的inode和dentry。（do_path_lookup）

   这是一个漫长的过程，逐层查找路径中的每一个节点对应的dentry：

   以/mnt/success为例：

   a. 先找到 /的dentry，/的dentry常驻内核，对应根文件系统的root inode。（do_path_lookup ->path_init）

   b. 然后查找 / dentry下的hash表——已经打开的所有的位于 / 目录下的文件或目录对应的dentry都会放在这个hash表里面。（__link_path_walk-> do_lookup）

   ```
   crash> dentry.d_hash
   struct dentry {
       [8] struct hlist_bl_node d_hash;
   ```

   c. 如果运气不好没找到，那就只能读磁盘了。我们之前已经拿到了父目录/的dentry，同时也可以拿到它的inode（dentry.d_inode）。inode的i_op字段保存了打开inode时文件系统注册的lookup函数。

   ```
   crash> inode.i_op
   struct inode {
       [32] const struct inode_operations *i_op;
   }
   crash> inode_operations 0xffffffff8162ea40
   struct inode_operations {
     lookup = 0xffffffff811ea530 <ext4_lookup>,
     follow_link = 0,
   ```

[8] 描述一个任务（进程或线程）的结构体。

通过调用lookup函数，促使文件系统从磁盘上读取并构建mnt的inode和dentry。

inode_operations就是在这个时候填充到inode对象内的（do_lookup->real_lookup）

除了inode_operations, file对象所用的 file_operations也是在这个时候保存到inode的i_fop字段。

```
struct inode {
   [312] const struct file_operations *i_fop;
}
```

最后还有一个最最重要的事情——建立文件（磁盘）内容到内存的映射结构。(inode_init_always)

这个映射结构保存在inode的imapping字段。

```
struct inode {
    [48] struct address_space *i_mapping;
}
```

每一个文件分配一个地址空间用来映射文件内容。i_mapping所关联的物理页，就是这个文件的page cache，就是free命令cached字段统计的内容。

所谓文件缓存，VFS直接借助内核mm子系统的地址映射机制，把内存地址映射至文件内容就轻松的实现了！当然这本不是缓存的难点，缓存的难点在于生命周期管理。

d. 无论是直接在内存中找到，还是读磁盘，总之找到了mnt的inode和dentry。接着如法炮制，继续搞定下一个节点success的inode和dentry。不过在这之前有一个特殊处理，因为mnt是一个挂载点，/mnt被替换为被挂载文件系统的root inode, dentry也会替换。替换之后，再继续查找success就是在mnt所挂载的文件系统上了。如果mnt和/的文件系统不同，inode保存的inode_operations也不同，层层嵌套的文件系统就都完美的运转起来了！

4. 填充文件对象file。（do_filp_open ->nameidata_to_filp）

dentry 和 inode都搞定，我们需要把之前分配的file对象和他们关联起来，这样进程才可以访问。这里最重要的工作就是把inode保存的file_operations和address_space填充到file内。后面读写文件就能直接找到对应的操作函数了。

```
crash> struct file.f_op
struct file {
    [32] const struct file_operations *f_op;
```

```
}
crash> struct file.f_mapping
struct file {
   [184] struct address_space *f_mapping;
}
```

5. 把file和fd关联起来，并把fd返回给用户态程序。（do_sys_open -> fd_install）

   file建立好之后，其实文件打开工作就完成了，就可以访问了，把file对象的地址返回给用户态，作为read/write后续调用的参数就可以工作了。但这显然不是明智之举，基于封装原则，file对象不需要对应用程序暴露，而一个内核态地址暴露给用户态程序，显得很不安全。不如转化为一个数字吧。于是我们得到了数字。数字本身毫无意义，只不过唯一标识了一个file对象。除此之外，使用一个有序的序号代表一个打开的文件，这总比地址好理解和好管理多了！

3.3 写文件流程

还是从系统调用开始：

```
ssize_t write(int fd, const void *buf, size_t count);

SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
{
        struct file *file;
        ssize_t ret = -EBADF;
        int fput_needed;

        file = fget_light(fd, &fput_needed);
"fs/read_write.c" 912 lines --39%--
```

之前打开文件拿到了fd，把这个fd、缓冲区以及写多少告诉内核，内核就会帮你搞定。写文件过程比较长，可以分几个阶段来描述[9]。

3.3.1 第一阶段

1. 根据fd找到该文件对应的file对象。（sys_open->fget_light）
2. 从file对象获取当前的读写位置（*f_pos*）。（sys_open-> file_pos_read）

   ```
   struct file {
   ```

[9] 为了简便，没考虑文件系统对日志的处理过程。

```
    [64] loff_t  f_pos;
}
```

3. 调用file对象保存的对应文件系统注册的写函数。(vfs_write: file->f_op->write)

```
crash> file_operations 0xffffffff8162e1c0
struct file_operations {
owner = 0x0,
llseek = 0xffffffff811dd120 <ext4_llseek>,
read = 0xffffffff8114b140 <do_sync_read>,
write = 0xffffffff8114b050 <do_sync_write>,
aio_read = 0xffffffff810f25e0 <generic_file_aio_read>,
aio_write = 0xffffffff811dd2b0 <ext4_file_write>,
```

ext4并没有定义自己的write方法，而是直接注册了VFS提供的通用写函数do_sync_write，而do_sync_write继续调用文件系统注册的aio_write方法，一般的写文件也是统一到异步写的过程，但却并非就是真正的异步过程，内核对文件读写的异步和同步的处理与我们平常理解的并非一致，附录4.3有一节专门的分析，在此先按下不表。

在进入aio_write方法之前，对这次写文件请求构造一个kiocb对象，iocb是Io control block的意思，由于异步处理需要保存一个上下文环境，这样完成之后才能找到对应的对象发送通知，kiocb就做这个事情，这里可以先不管它。并不影响我们理解关键流程。

ext4注册的aio_write方法是ext4_file_write，也没干啥活儿，而是又回去调用了VFS的generic_file_aio_write，generic_file_aio_write判断O_DIRECT标记，对direct IO和buffered IO分开处理。direct IO后面再说，对于buffered IO，进入generic_file_buffered_write处理，开始第二阶段。

3.3.2 第二阶段

第一个阶段调来调去，绕来绕去，只不过是为了找到最终的干活儿的人，不过也很必要，如果需要一些特殊的处理，我可以很容易的插入自己的扩展。

我们打开文件大小的时候建立了文件对应的内存映射， 这一个阶段要做的主要工作就是把要写的数据写入文件对应的内存中（page cache）：

1. 根据f_pos算出要写的起始位置对应的page（内存页）序号，以及在page内的偏移。

（generic_file_buffered_write->generic_perform_write）

这些page不一定是连续的，但文件内容是按其序号依次分布的，即page 0对应文件的1~4096字节，page 1对应4097~8192字节。

2. 找到或新建这个page。（generic_perform_write: a_ops->write_begin）

这个过程通过文件系统注册的address_space_operations完成，这一组操作函数指针保存在文件对应的address_space内：

```
crash> address_space.a_ops
struct address_space {
   [112] const struct address_space_operations *a_ops;
}
crash> struct address_space_operations 0xffffffff8162e420
struct address_space_operations {
   writepage = 0xffffffff811e4640 <ext4_writepage>,
   readpage = 0xffffffff811e0c10 <ext4_readpage>,
   readpages = 0xffffffff811e0940 <ext4_readpages>,
   write_begin = 0xffffffff811e3de0 <ext4_da_write_begin>,
   write_end = 0xffffffff811e5dd0 <ext4_da_write_end>,
   direct_IO = 0xffffffff811e1bb0 <ext4_direct_IO>,
```

在打开文件的时候，已经初始化了文件的address_space对象，并保存至f_mapping字段。

一个文件的所有关联page被放在一个radix tree[10]上，树的根节点保存在address_space的page_tree字段。

```
crash> address_space.page_tree
struct address_space {
     [8] struct radix_tree_root page_tree;
}
```

page cache即所谓的文件缓存的实体结构就是所有文件对应的page_tree。

寻找page的时候，首先到page_tree上查找，如果没找到，就重新分配一个page，并加到page_tree上，（grab_cache_page_write_begin）。

只是这样还不够，我可能必须把page对应的磁盘块的内容，写到这个新建的page内。即写文件之前，可能会先从磁盘上读取这个文件的内容，转入读文件流程的

[10] 本质是一个hash表，但对hash的key值分了段，按层次构成树，节约了空间。 否则真要对page cache中的所有page建立hash，估计内存就别干别的事情了。

处理。（block_write_begin ->__block_prepare_write）

因为我们可能只操作page内的1个字节，但最后提交给磁盘的都是按块的，没法只提交这1个字节，只能预先读出来整个块，然后修改1个字节，再写回去。

3. 把用户传入buffer的内容拷贝至之前准备好的page上，即进入了page cache。

（generic_perform_write->iov_iter_copy_from_user_atomic）

4. 把个page标记为脏页，把这个inode标记为脏inode。（generic_perform_write: a_ops->write_end）

通过调用__set_page_dirty和__mark_inode_dirty实现。

3.3.3 第三阶段

上一个阶段把inode和page搞脏之后，write这个系统调用就拍屁股走人返回了[11]，应用进程结束阻塞状态，在应用进程看来写文件的动作就结束了，但是数据还没到磁盘上。

第三阶段的主要工作就是把这些脏数据转化为IO操作提交到块层。

每一个磁盘都分配了一个bdi_writeback对象，维护了若干个链表，其中b_dirty就是用来保存所有待会回写的脏inode的。在第二阶段write返回之前内核会把inode加入这个链表。

```
crash> struct bdi_writeback
struct bdi_writeback {
        struct backing_dev_info *bdi;
        unsigned int nr;
        long unsigned int last_old_flush;
        long unsigned int last_active;
        struct task_struct *task;
        struct timer_list wakeup_timer;
        struct list_head b_dirty;
        struct list_head b_io;
        struct list_head b_more_io;
        spinlock_t list_lock;
```

[11] 除非你加了SYNC标记强制同步；或者系统脏页太多进入流控状态强制等待。（balance_dirty_pages_ratelimited）。

bdi_writeback保存在request_queue的backing_dev_info上。

```
crash> struct request_queue.backing_dev_info
struct request_queue {
    [312] struct backing_dev_info backing_dev_info;
}
crash> struct backing_dev_info.wb
struct backing_dev_info {
   [296] struct bdi_writeback wb;
}
```

数据有了，那么谁来干活儿呢？ 自然不是应用线程——内核为此专门设立了一个写回线程。

这个线程就是我们通过ps命令看到的flush-major:minor，每一个磁盘对应一个。

```
1 S root       855       2    0    80    0 -        0 -      09:57 ?        00:00:00 [flush-8:0]

wb->task = kthread_run(bdi_start_fn, wb, "flush-%s", dev_name(bdi->dev));
"mm/backing-dev.c" 769 lines --53%--
```

这个线程的工作机制(bdi_start_fn)：

1. *bdi是一个任务框架，每个磁盘的backing_dev_info内保存了一个任务链表。*

   ```
   crash> struct backing_dev_info.work_list
   struct backing_dev_info {
      [480] struct list_head work_list;
   }
   ```

   flush线程遍历work_list，获得写回任务struct bdi_work，根据任务中提供的信息和参数干活儿。

2. *而写回任务的生成主要有两种：一种是主动的，一种是周期性的。*

   主动的比如执行了sync调用，sync调用实际上就是生成一个写回任务，并唤醒flush线程。

   ```
   SYSCALL_DEFINE0(sync)
   {
           wakeup_flusher_threads(0);
           sync_filesystems(0);
           sync_filesystems(1);
           if (unlikely(laptop_mode))
                   laptop_sync_completion();
           return 0;
   }
   ```

   周期性的任务比如每隔若干时间就主动检查是否有活儿干。这个时间通过以下内核参

数控制，默认是500，单位是centisecs（百分之一秒）。

```
# sysctl -a |grep "dirty_writeback_centisecs"

if (dirty_writeback_interval) {
        wait_jiffies = msecs_to_jiffies(dirty_writeback_interval * 10);
        schedule_timeout_interruptible(wait_jiffies);
"fs/fs-writeback.c" 1456 lines --69%--
```

3. *除此之外，在脏页面总数达到页面总数的一个比例之后，也会主动生成一个写回任务，这个比例通过以下内核参数控制。这个写回任务直到把脏页写到这个比例之下才会停止。*

```
# sysctl -a |grep back
vm.dirty_background_ratio = 10

static long wb_check_background_flush(struct bdi_writeback *wb)
{
        if (over_bground_thresh()) {
                struct wb_writeback_args args = {
                        .nr_pages       = LONG_MAX,
                        .sync_mode      = WB_SYNC_NONE,
                        .for_background = 1,
                        .range_cyclic   = 1,
                };
                return wb_writeback(wb, &args);
        }
"fs/fs-writeback.c" 1456 lines --65%--
```

以上就是bdi框架的基本工作原理。

flush线程调用writeback_inodes_wb从dirty_list上获取要回写的inode对象，最终又调用文件系统注册的address_space_operations.writepages函数进入下一个阶段。

```
int do_writepages(struct address_space *mapping, struct writeback_control *wbc)
{
        int ret;
        if (wbc->nr_to_write <= 0)
                return 0;
        if (mapping->a_ops->writepages)
                ret = mapping->a_ops->writepages(mapping, wbc);
```

```
        else
                ret = generic_writepages(mapping, wbc);
        return ret;
}
```

脏数据回写这部分，SUSE10的2.6.16内核上整个系统共享一个回写队列的（没有为每个设备维护一个bdi），这就可能导致一个问题。参见本人博文《SUSE10（2.6.16）上的内存回收和脏数据回写问题》。

3.3.4 第四阶段

这一个阶段的主要工作就是遍历文件的映射页面，把其中的脏页内容提交到块层。

我们需要知道一个页面对应磁盘上的第几个块（扇区）。这个信息只有文件系统知道。这个信息保存在struct buffer_head内。一个buffer_head对应1个块。一个page对应多个buffer_head。buffer_head并不保存块的内容，块的内容在内核中只有一份，就是在其对应的page里面；buffer_head仅仅描述内存页到磁盘块的映射关系。

一个page对应的buffer_head在第一次读取这个page对应块内容的时候就建立。（ext4_readpages-> ext4_get_block | ext4_da_write_begin->..-> ext4_get_block）。

一个page对应的所有buffer_head被穿在一个链表上，头节点保存在page.private上。这一阶段遍历所有page，对其中的dirty page执行address_space_operations.writepage，而writepage遍历page对应的所有buffer_head，最后调用submit_bh向块层提交这个IO请求。

但是前面提到bio才是块层处理的对象，而不是buffer_head。所以submit_bh就是把buffer_head转化为一个bio然后再调用submit_bio进入块层。

bio和buffer_head最大区别就是bio可以关联多个page。只要这些page表示的块是连续的。bio更好的代表了一个IO请求，而buffer_head仅仅是代表一个块。谁说一个IO请求只能包含一个块呢？块只是单位而已！

那么文件系统不可以直接转化为bio提交吗？当然可以了！buffer_head是旧事物，而bio是新事物，就是替代buffer_head的。当然演进过程中肯定有一个漫长的共存过程。

实际上ext3和ext4都提供了一个参数，禁用buffer_head。

nobh　　Do not attach buffer_heads to file pagecache. (Since 2.5.49.)

新的文件系统彻底不使用buffer_head也是完全可以的。

到此文件系统层的工作就彻底结束了。 一个完全新的篇章——块层就开始了。丫的层次真多，一光年了还没到硬盘呢。。。——不过要是没这些层次、封装和抽象，我们直接裸写磁盘扇区，估计早累死个球了！

层次、封装和抽象——这基本上是计算机这个学科的全部思想，哲学上的任何一个思想概念都比这深刻，懂这些思想再容易不过，但计算机是一个实践学科，要做好这三样就是数十年如一日的扎实积累了。

3.3.5 第五阶段

进入块层之后，文件的概念就彻底消失了，只有块，更准确的说是扇区。那么这一层的主要工作是什么？或者说为什么设置一个块层？直接写到磁盘不就好了？块层主要目的就一个——IO调度。

为什么调度？很简单，调度是为了提高效率。提高磁盘这个机械部件的运转效率，通过合并和排序减少磁头来回摆动。

那么既然要调度，就要队列。所以块层玩的主要对象就是request_queue以及request_queue上的request。

一切从submit_bio开始：

1. 入队。　　（submit_bio->…>__make_request）

   request_queue中管理的是request，而不是bio，所以要从队列中分配一个request封装这个bio。如果队列满了，说明底层设备处理不过来了，就需要等待。

   （__make_request　->get_request_wait）

   队列上的request总数通过/sys/block/sdX/queue/nr_requests控制:

   ```
   # cat /sys/block/sda/queue/nr_requests
   128
   ```

如果队列中有一个request要操作的扇区和要入队的bio相邻，其实并不用新建一个request，直接并入已有的request就可以了。所以入队之前会尝试和队中已有的request合并（__request->elv_merge）。

具体的合并规则由IO调度算法注册的elevator_ops.elevator_merge_fn方法搞定。

request如果是新建的，就插入到request_queue中，不同的IO调度算法可能会使用不同的结构保存这些request，比如cfq会使用一个红黑树，而noop只是一个链表。noop是不排序的，而cfq是插入时自动保持树的有序性，是排序的。

(cfq_add_rq_rb->elv_rb_add)

2. 出队。

出队由底层设备注册的request_queue.request_fn完成，比如对于基于SCSI协议连接的设备，通过scsi_request_fn函数进入SCSI层处理。

```
crash> request_queue.request_fn 0xffff880114e550b8
request_fn = 0xffffffff81389c70 <scsi_request_fn>
```

这个出队动作的触发会在入队动作结束后，在submit_bio的执行流上。
（__make_request ->__generic_unplug_device）
这就意味着调用submit_bio的文件系统层必须同步等待块层处理完毕，这可能并非必须，完全可以入队完就闪人，所以需要一个异步处理的过程。异步处理通过以下线程完成。

```
# ps -elf |grep kblockd
1 S root       138     2  0  80   0 -     0 worker Jan08 ?        00:00:00 [kblockd/0]
1 S root       139     2  0  80   0 -     0 worker Jan08 ?        00:00:00 [kblockd/1]
1 S root       140     2  0  80   0 -     0 worker Jan08 ?        00:00:00 [kblockd/2]
1 S root       141     2  0  80   0 -     0 worker Jan08 ?        00:00:00 [kblockd/3]
1 S root       142     2  0  80   0 -     0 worker Jan08 ?        00:00:00 [kblockd/4]
1 S root       143     2  0  80   0 -     0 worker Jan08 ?        00:00:00 [kblockd/5]
1 S root       144     2  0  80   0 -     0 worker Jan08 ?        00:00:00 [kblockd/6]
1 S root       145     2  0  80   0 -     0 worker Jan08 ?        00:00:00 [kblockd/7]
1 S root       146     2  0  80   0 -     0 worker Jan08 ?        00:00:00 [kblockd/8]
```

每一个CPU上都一个kblockd线程。这其实是利用了内核的工作队列机制
("kernel/workqueue.c")

简单来说，就是类似前面提到的bdi_work一样。每一个线程扫描一个链表(工作队列)，链表上保存了要干的活儿struct work_struct，而work_struct中保存了具体负责干活的函数。

```
crash>    work_struct
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
}
```

调用这个函数，就完成了任务。说白了就是一个异步任务执行框架，任务的本质就是一个函数。

IO调度算法会把自己的出队函数封装到为一个work_struct。

```
INIT_WORK(&cfqd->unplug_work, cfq_kick_queue);
"block/cfq-iosched.c" 2891 lines --91%--
```

在入队操作结束之后会调用kblockd_schedule_work触发这个任务。

```
 kblockd_schedule_work(q, &q->unplug_work);
"block/blk-core.c" 2517 lines --15%--
```

也即把unplug_work放入kblockd扫描的链表，并唤醒kblockd线程干活儿。

最终出队函数被调用，进入设备驱动层。

3.3.6 第六阶段

进入设备驱动层之后，最重要的一个操作就是DMA。这时候我们要写的内容还在page cache中的page里面。驱动通过request找到bio再找到其关联的page，把这些page DMA映射给设备。然后下发操作命令给设备。

IO数据在内核漫长的流程、层次内没有任何拷贝，只有用户态和内核态之间的一次拷贝。

由于IO设备在把内存中的数据写到设备上，不需要CPU参与，所以CPU这时可以干其他活儿。

IO WAIT高并非说明CPU忙，而恰恰说明CPU闲的发慌，DMA价值没体现！

在服务器上把计算密集型和IO密集型的应用部署到一起，可以最大化资源利用率，真正发挥DMA的作用。

设备驱动层的具体细节可参考《Linux SCSI子系统知识总结》。

到此写文件的流程就结束了。真是比电视剧还长！但是只是结束了，还不行，我们需要结局！需要知道写的结果。

3.3.7 THE END

把IO丢给设备处理之后，大家都HAPPY去了，不需要人看着并不停地问，喂，完成了没？——那是轮询，太土了。这里必然是靠异步的元祖——中断。

设备驱动注册了自己的中断处理函数，中断处理函数调用块层的blk_end_request。块层又调用文件系统层注册的保存在bio. bi_end_io处理函数：

```
crash> struct bio.bi_end_io
struct bio {
    [80] bio_end_io_t *bi_end_io;
}
crash> bio_end_io_t
typedef void (struct bio *, int) bio_end_io_t;
SIZE: 1
```

就这样样一层层地通知上去，总之就两字儿——**回调**。

3.4 读文件流程

写文件流程清楚了，读文件其实也就清楚了。在块层以及块层之下，其处理没有差别。

在文件系统层其实也没有本质差别，也是一个从fpos到page再到bio最后submit_bio的过程。

不过有几点特殊的地方：

1. 如果page cache已经存在要读的page并且是uptodate的就直接拷贝到用户buffer返回。
2. 如果page不存在或者不是uptodate就submit_bio读磁盘，但这是一个同步的过程，直到数据从磁盘拷贝到page cache，再拷贝到用户buffer，read过程（系统调用）才会返回。
3. 读的时候会有预读，即比实际要读的多读若干page。下次再读附近的page就不需要读磁盘了。(do_generic_file_read->page_cache_sync_readahead|page_cache_async_readahead)

4 附录

4.1 DIRECT IO

先来看一段有意思的评价：

"The thing that has always disturbed me about O_DIRECT is that the whole interface is just stupid, and was probably designed by a deranged monkey on some serious mind-controlling substances." —— Linus

Linus说O_DIRECT接口是吃了迷药的精神错乱的猴子设计的。这话也太狠了。。。

但存在即合理，它在某些场景下还是有一些用处的，或是某些历史原因。

在open的时候，指定O_DIRECT标记，就可以进行DIRECT IO，但要求用户层的IO buffer要对齐到512字节。如果不满足，会自动退化到普通IO（buffered IO）。

而且如果其他进程，不是使用O_DIRECT标记操作同一个文件，使用的是正常IO，这可能导致两个进程都慢下来。（主要是因为进行direct IO的时候会强制flush所有page cache）

DIRECT IO 由文件系统注册的address_space_operations.direct_IO方法实现。通常都会转至__blockdev_direct_IO方法，对应的代码在"fs/direct-io.c"内。

DIRECT IO的本质就是把用户buffer直接DMA映射给设备，这样就避免了用户内存（buffer）和内核内存（page cache）之间的拷贝，提高了效率。但实际上并非如此，因为这种方式之下，相当于没有了缓存：

1. 对于写，在内存充分的情况下，DIRECT IO远远慢于正常IO，因为后者只复制到内存，而后者需要写到磁盘。
2. 对于读，没有了缓存，每次都要读磁盘，即使之前读过相同数据。

于是使用DIRECT IO在性能上没有任何好处，如果需要性能，那就又需要在用户层做cache。 这就是比较蛋疼的，为什么不直接用内核提供的cache？你做的比内核做的更好？一般恐怕都不行。

但是像Oracle这样的数据库软件，对IO的可靠性和性能要求比较高，不仅可以配置为不用内核cache，而且甚至连整个文件系统层都不用，直接自己写磁盘（裸设备）。不过Oracle内部也少不了对数据的抽象（未必是文件的概念）以及缓存的组织和设计，只不过对自己的数据

模型做了强度适配，比使用内核通用的cache系统更好。

对于一般的应用，我觉得最好忘掉O_DIRECT这回事儿好了。

DIRECT IO可以理解为一种零拷贝[12]技术，但由于cache的必要性，零拷贝在存储IO方面其实没啥价值。 零拷贝更适用于网络IO， 因为网络IO是没有延续性的——不需要cache。

这里顺带啰嗦一下——buffer和cache的区别，很多人搞不清楚。buffer是用来暂存待处理数据的，一旦处理完毕就失去意义，再也不用了。cache是用来存储长时间有意义，并且会重复访问的数据的。

网络IO没有cache只有buffer，所以如果可以把我们的数据包的用户buffer直接DMA映射给网卡实现零拷贝，对网络性能提升会很明显。实际上内核也提供了一些接口，实现零拷贝，提高大数据量的网络传输性能，比如splice、sendfile之类的。如果需要更广泛的支持，则需要自己定制内核了。

4.2 mmap

mmap是一个系统调用，用来申请一段地址空间的（虚存）。实际上这就是C库malloc函数的底层函数。当我们访问这一段地址空间时，内核分配对应的内存页。

mmap可以把申请的地址空间映射至一个文件，实际上就是把一个文件page cache放到了用户态。通过访问mmap返回的线性地址，也就是直接访问文件的内容。而不需要像read/write等调用那样在内核态和用户态之间传递数据。

mmap也实现了零拷贝。相比DIRECT IO，mmap并没有越过文件系统的cache机制，只不过cache自动跑到用户态了。所以使用mmap的方式操作大文件进行大数据量读写对性能会有很好的提升。是一个值得推荐的方式。

但使用mmap一定要注意文件的保护，一旦一个文件被其他进程截断，本进程mmap的地址空间对应的物理内容不存在了，就会发生SIGBUS，导致进程coredump。这是X86平台上发生SIGBUS问题的一个主要原因。

[12]直接把用户内存DMA映射给设备，避免用户态和内核态之间的内存拷贝。

4.3 AIO

AIO就是异步IO。

io_submit[13]是内核提供的异步IO接口。按照对于异步的理解，当我们调用了io_submit提交了IO请求之后，应该立即返回，不需要等待结果。结果采用异步的方式通知给我们。io_submit通常应该是不阻塞的，瞬间完成的。

然而实际上并非如此。在现有的内核实现上，io_submit只不过是直接调用了文件系统注册的aio_read和aio_write函数，在这两个函数不返回之前io_submit是不会返回的，这和普通IO没有区别，无论读写都会被阻塞。io_submit毫无意义。

不过在DIRECT IO的下，却有一些差别，__blockdev_direct_IO方法会判断是否是AIO，如果是就不会等待磁盘读写完成，在调用submit_bio向块层提交了请求之后就会直接返回。然后块层通过回调它注册的dio_bio_end_aio方法唤醒调用了io_getevents等待结果通知处于阻塞状态的线程。

综上，目前内核的AIO机制仅针对DIRECT IO，对于普通IO内核也许认为都是基于cache的，访问内存是不会阻塞的，本身就是异步了的。不过内核的AIO架子是搭好了，只要文件系统注册的aio_read和aio_write是真正异步的，而不是像现在这样阻塞[14]的，整体上就是异步的了。这种情况下aio_read和aio_write就不能继续作为read和write的实现继续使用了。因为read之后要求用户buffer立即可用，write之后要求用户buffer立即可销毁。

除了内核提供的AIO外，glibc还提供了一套AIO机制：aio_read(3)/aio_write(3)。Glibc的这套AIO机制并非对内核io_submit的封装。二者根本没有任何关系。Glibc的AIO直接通过线程做到异步，是纯用户态的机制——它专门设置一个或多个专门的IO线程负责实际的IO（调用read或write系统调用），其他线程调用aio_read/aio_write把IO提交给这些线程之后，等待IO线程的异步通知。和内核的AIO相比，glibc的AIO的异步效果要明显的多。

---

13 需和io_setup(2) , io_getevents(2), io_cancel(2), io_destroy(2)配合使用。

14 需要等待内核态cache和用户态buffer交换数据完毕，有时还需要等待磁盘IO完成。

AIO特别适合于那些计算密集型但偶尔又需要做一点IO的应用，这种情况下IO阻塞了线程，不能继续执行计算代码了，而CPU由于有DMA也不参与写磁盘，白白浪费了CPU; 如果采用AIO就可以解决这个问题。——实际上省去了自己写建立线程专门做IO异步处理的代码的麻烦。很多大型应用未必用到了AIO，但通常会把业务处理线程和IO处理线程分开，本质思想是一样的。

4.4 逻辑卷

逻辑卷的使用内核的device-mapper机制实现。device-mapper对外提供了一个从逻辑设备到物理设备的映射框架。

逻辑卷利用device-mapper把多个物理块设备映射为一个逻辑块设备：
逻辑设备上的连续地址空间（扇区）的背后可能是不连续的空间，甚至可以分布在多个不同的物理设备上。增加物理设备，并不破坏逻辑卷已有的地址空间，只是增加新的地址空间，并且保持连续性。逻辑卷可以方便扩展的玄机就在于此。一个**分层**和**映射**就轻松解决了令人头疼的难题！这就是**设计**！

device-mapper的代码定义在"drivers/md/dm.c"内。
逻辑卷作为块设备，自然也会有一个request_queue，并注册自己request_queue.request_fn函数接收块层下发的IO请求。 然后把这个请求转发给其映射的真实的物理设备。把对逻辑块设备上地址的访问映射为对应物理设备实际地址的访问，这就是device-mapper的核心工作。

4.5 free 命令的 buffers

我们前边提到free命令的cached字段就是文件[15]的page cache。但是buffers呢？buffers其实也是page cache，也是文件映射的page。只不过buffer统计的是块设备文件映射的page。（si_meminfo-> nr_blockdev_pages）

[15] 这些文件包括虚拟的文件，比如shmget获取的共享内存在内核内部是表示为一个文件的。所以共享内存也被统计到cached字段了。

所谓块设备文件，就是形如/dev/sda 这样的文件。

块设备文件的page实际上直接连续的映射到了每一个块，page 0对应block 0，page 1对应block 1 ——没有文件系统嘛，肯定是这样组织的了。

那么什么时候，块设备文件映射的page数即free的buffers值会增长？当然是读取这些page的时候，有两种情况会读取到这些page：

1. 直接越过文件系统访问设备文件时。比如通过dd命令。
2. 文件系统获取元数据时。

   我们知道文件内容是有专属于文件的单独page映射的，不使用设备文件映射的page。但元数据并没有，当文件系统需要读取位于某个block上的元数据时，就直接读取设备文件映射的对应page。

   在读写文件的时候，需要确定读写位置所对应的block，这个信息就是一个元数据，文件系统就是通过读取元数据所对应设备文件映射的page获取这个信息。

   （ext4_get_block-> ..-> sb_getblk）

   当一个设备文件映射的page被访问时其和普通文件的操作是一样的，同样是把page序号转化为block位置，然后调用submit_bio提交到块层。只不过page序号和block号是连续对应的，可直接计算获得。

   虽然元数据的访问，会导致buffers增加，但要知道这个buffers并非指inode cache inode cache是inode这个内核结构体的cache，位于slab中。

   通过meminfo文件的SReclaimable字段统计了这一部分内容，其中也包括了dentry cache。

   ```
   # cat /proc/meminfo | grep SReclaimable
   SReclaimable:          21520 kB
   ```

5 参考资料

Linux Source Code 2.6.32

Understanding.the.Linux.Kernel.3rd.Edition