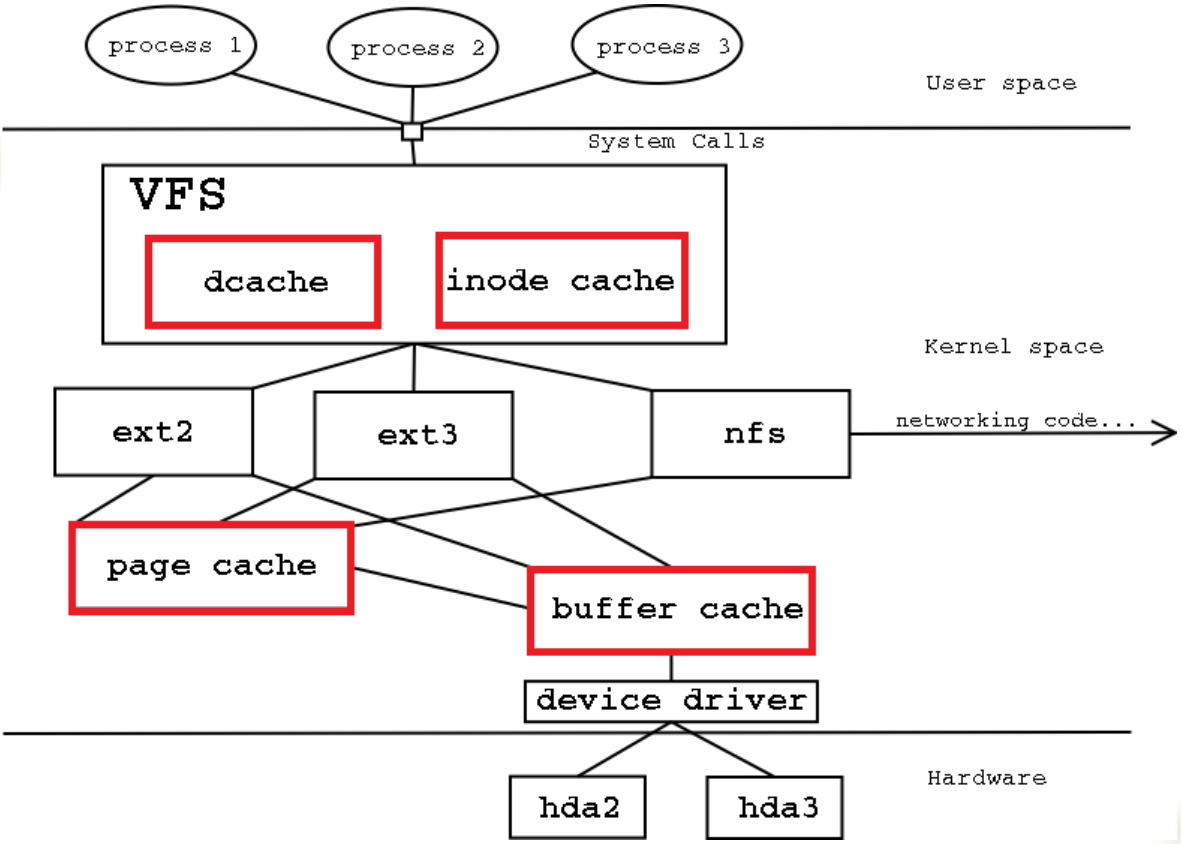
8.3 文件系统的缓冲区

西安邮电大学

文件系统中的缓冲区



缓冲区(buffer)，它是内存空间的一部分。也就是说，在内存空间中预留了一定的存储空间，这些存储空间用来缓冲输入或输出的数据，这部分预留的空间就叫做缓冲区，如图中的红色框，dcache，inode cache，page cache，buffer cache，它们有什么作用，有什么区别？

buffer和cache有何不同？

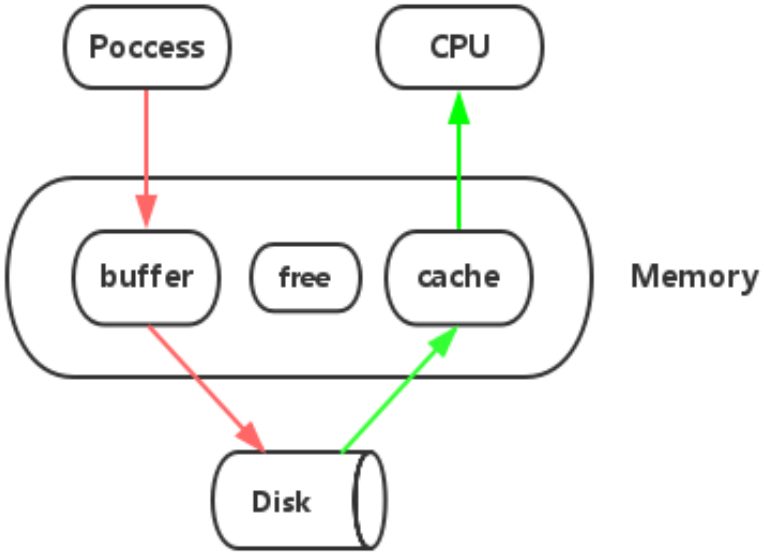
（1）buffer，内存缓冲区，是各进程产生的文件临时存放区，一定时间段内统一写入磁盘，减少磁盘碎片和 硬盘的反复寻道，从而提高系统性能；简单来说，buffer就是存放要写入磁盘的数据。

（2）cache，内存缓存区，经常被用在磁盘的I/O请求上，如果有文件频繁被访问到，系统会将文件缓存在cache区，供CPU、进程等访问；简单来说cache中的数据就是存放磁盘中读出来的数据。

可以查看proc目录下的meminfo文件，看到你机子上buffer和cache的大小。

buffer和cache有何不同？

```
[clj@localhost ~]$ cat /proc/meminfo
MemTotal:        1882772 kB
MemFree:          122280 kB
MemAvailable:     998148 kB
Buffers:          145728 kB
Cached:           729688 kB
SwapCached:            0 kB
```

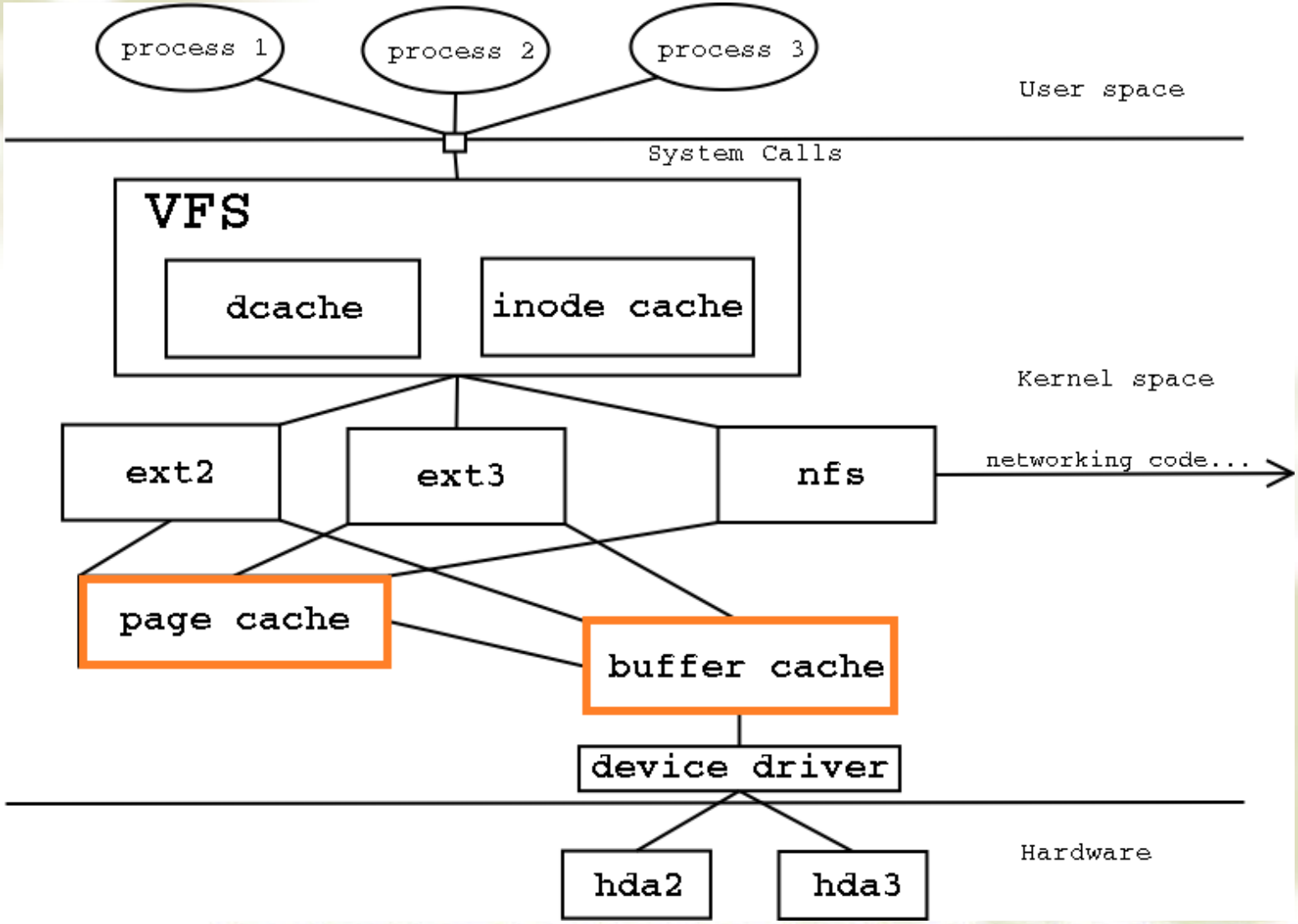


Buffer Cache和 Page Cache有何不同？

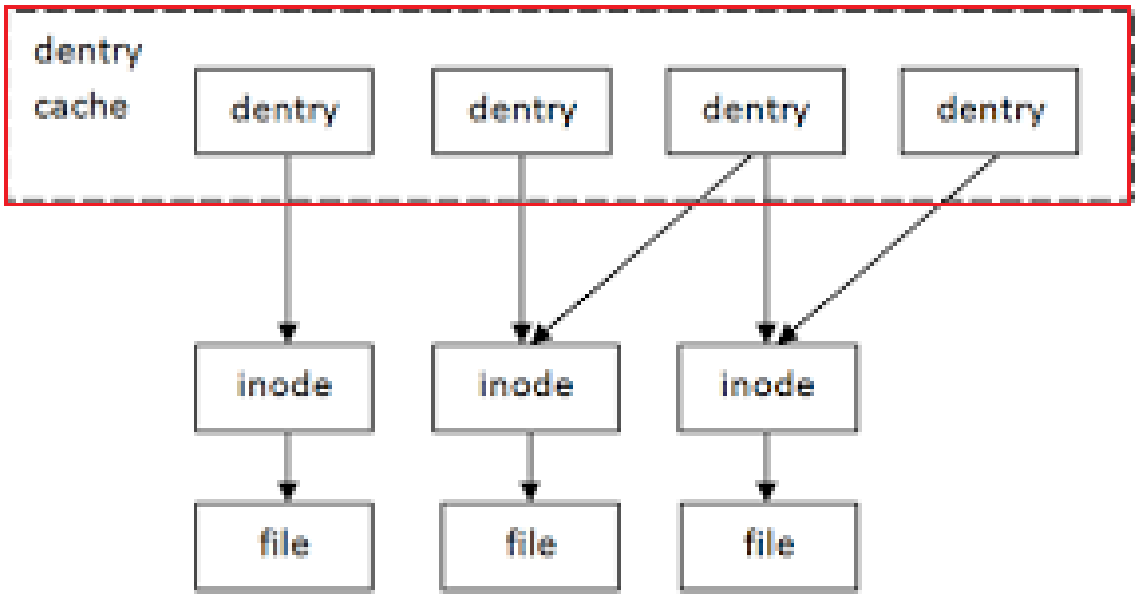
Page cache实际上是针对文件系统的，是文件的缓存，在文件层面上的数据会缓存到page cache。文件的逻辑层需要映射到实际的物理磁盘，这种映射关系由文件系统来完成。当page cache的数据需要刷新时，page cache中的数据交给buffer cache。

Buffer cache是针对磁盘块的缓存，也就是在没有文件系统的情况下，直接对磁盘进行操作的数据会缓存到buffer cache中，例如，文件系统的元数据都会缓存到buffer cache中，如图中的橘色框。

Buffer Cache和 Page Cache有何不同？



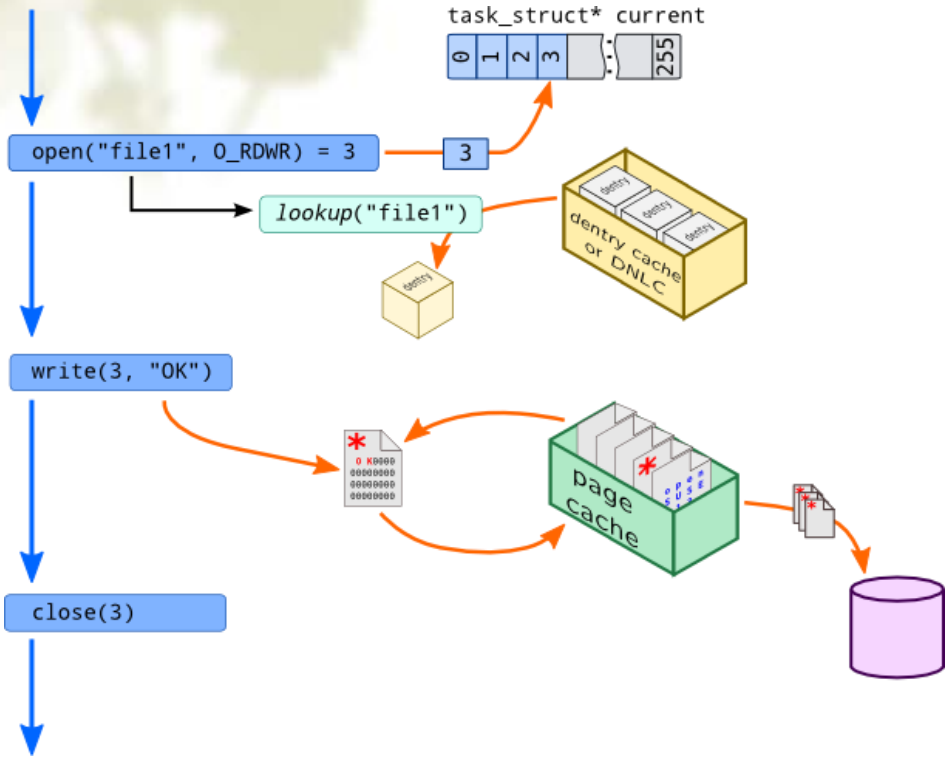
dentry cache 和inode cache有什么作用？



dcache –也就是 "dentry" 对象的cache，用于把路径转换为索引节点。

inode cache – 也就是“inode” 对象的cache，用于表示文件系统中的文件或者目录。

打开文件到底做什么？



打开文件的核心为查找。通常内核将查找过程分为两部分：

- 查找根目录信息
  - 主要是判断是系统根目录还是当前工作目录，以获取后面循环查找的起始位置（这里的位置指的是：具体的文件系统挂载位置以及从哪个目录开始）。
- 循环查找路径名后续分量
  - 以起始位置开始，循环查找后续每个路径分量。

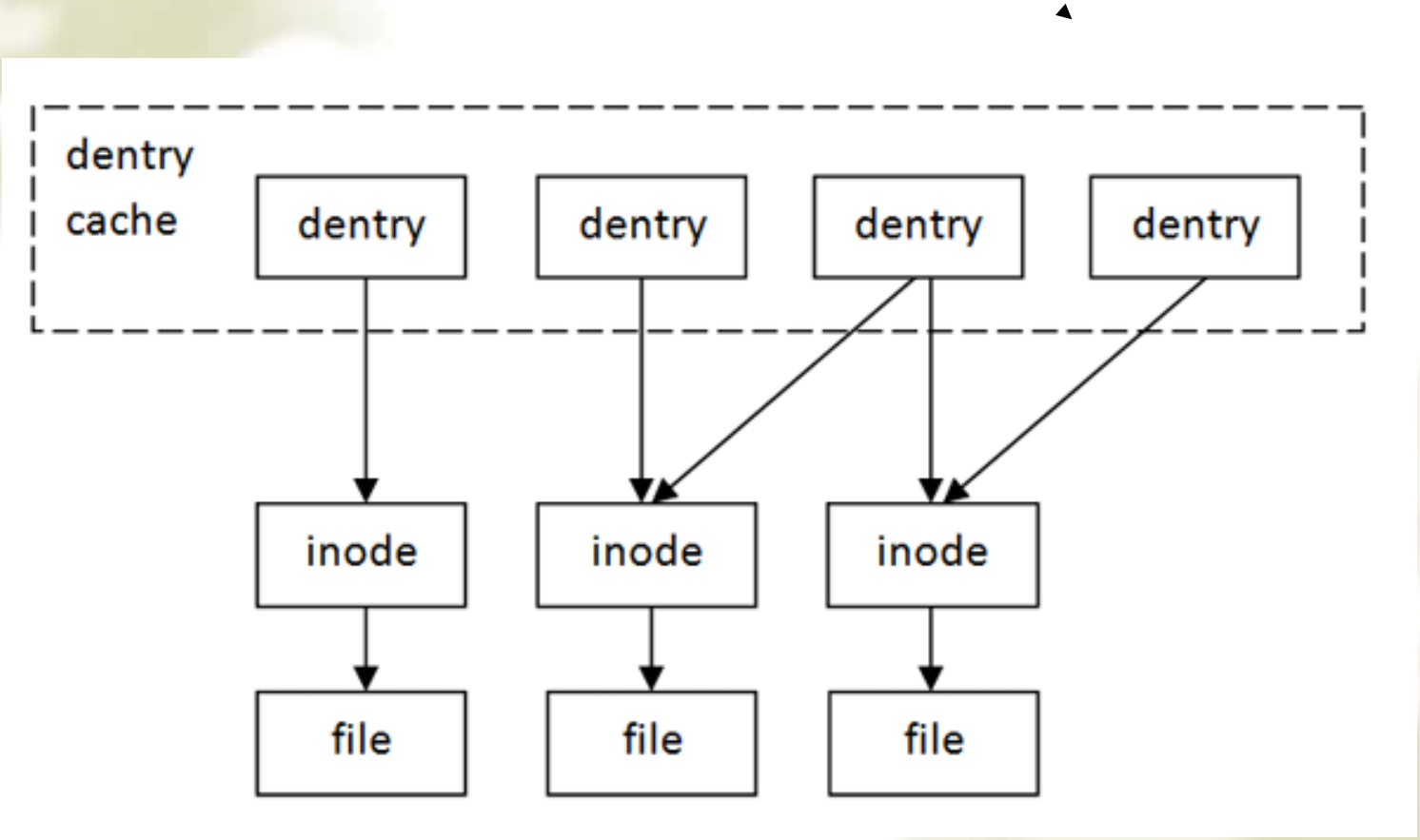
查找过程概述

查找过程看似简单，但内核实现复杂，涉及众多cache技术。查找的关键接口为do_lookup。其主要过程如下：

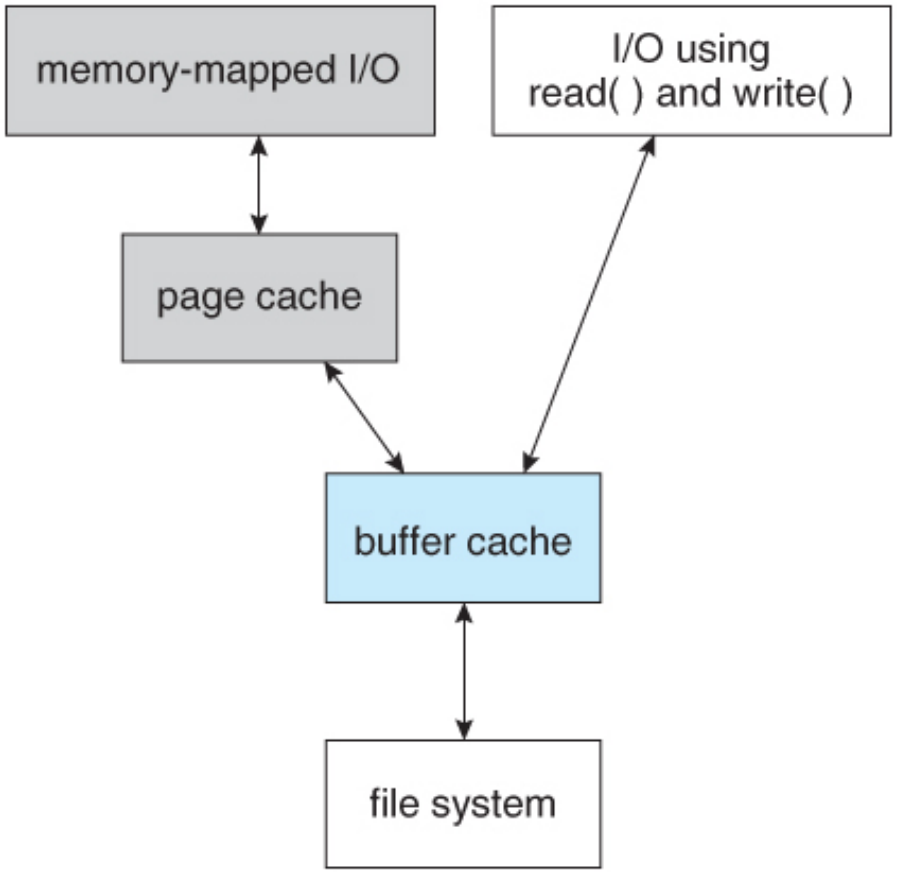
在dentry cache中查找相应的dentry，若找到则直接返回，若没有找到，则必须去底层文件系统查找对应的dentry。

调用底层文件系统对应的inode_operations操作集的lookup函数进行查找，首先在inode cache中查找是否存在对应的inode，如果有，则返回，如果没有，则必须去更底层的磁盘查找对应的inode信息。

查找过程概述

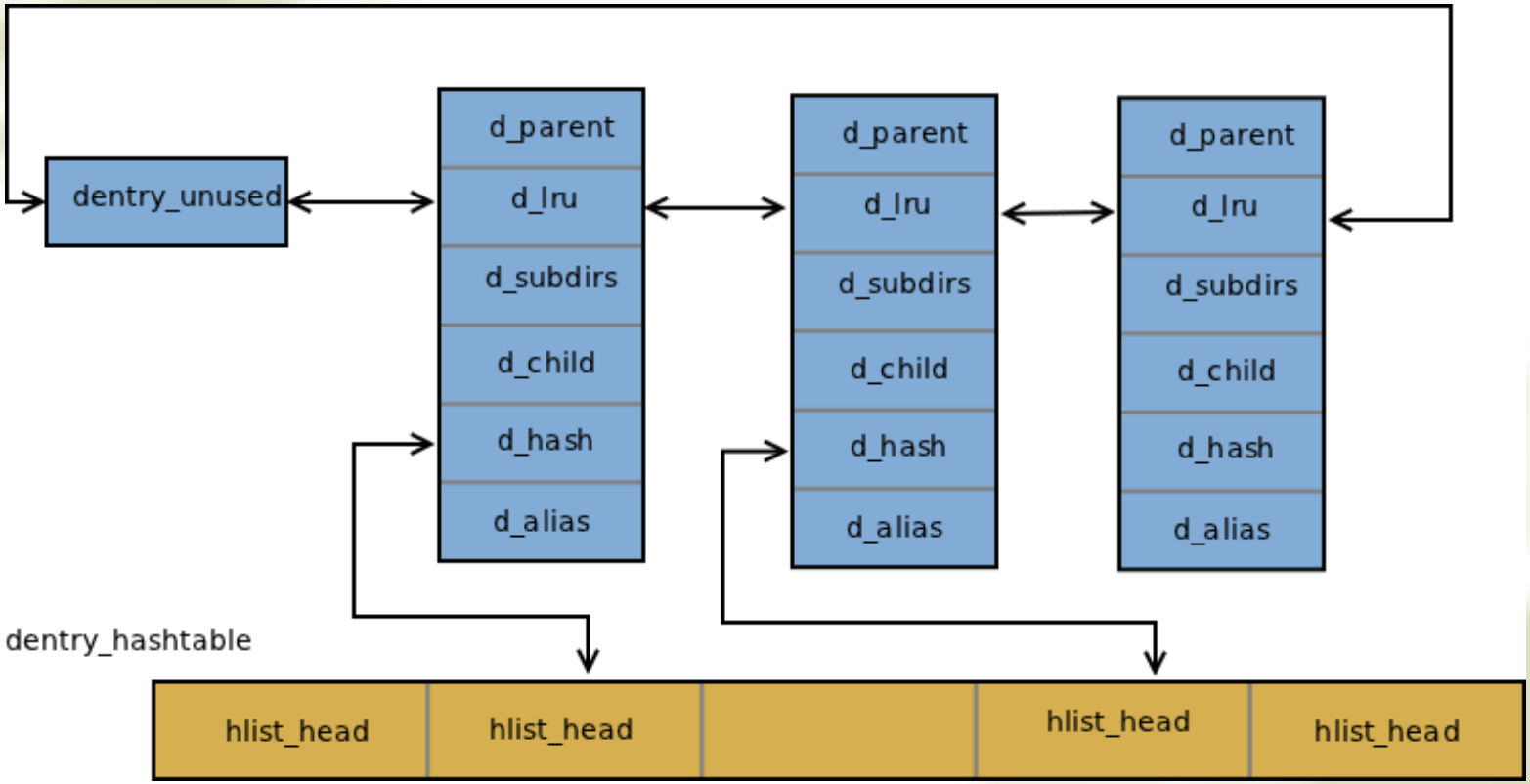

dentry
cache
dentry
dentry
dentry
dentry
inode
inode
inode
file
file
file

查找过程概述



去磁盘查找inode信息时，首先去buffer cache层查找相应的块，如果有相应的块存在，则从相应的buffer cache中提取inode信息，并将其转化为对应的文件系统的inode结构。

目录项缓存的组织和查找



目录项缓存的组织和查找

由于块设备速度比较慢，可能需要很长时间才能找到与一个文件名关联的inode信息，所以引入dentry cache。

缓存的组织包括：

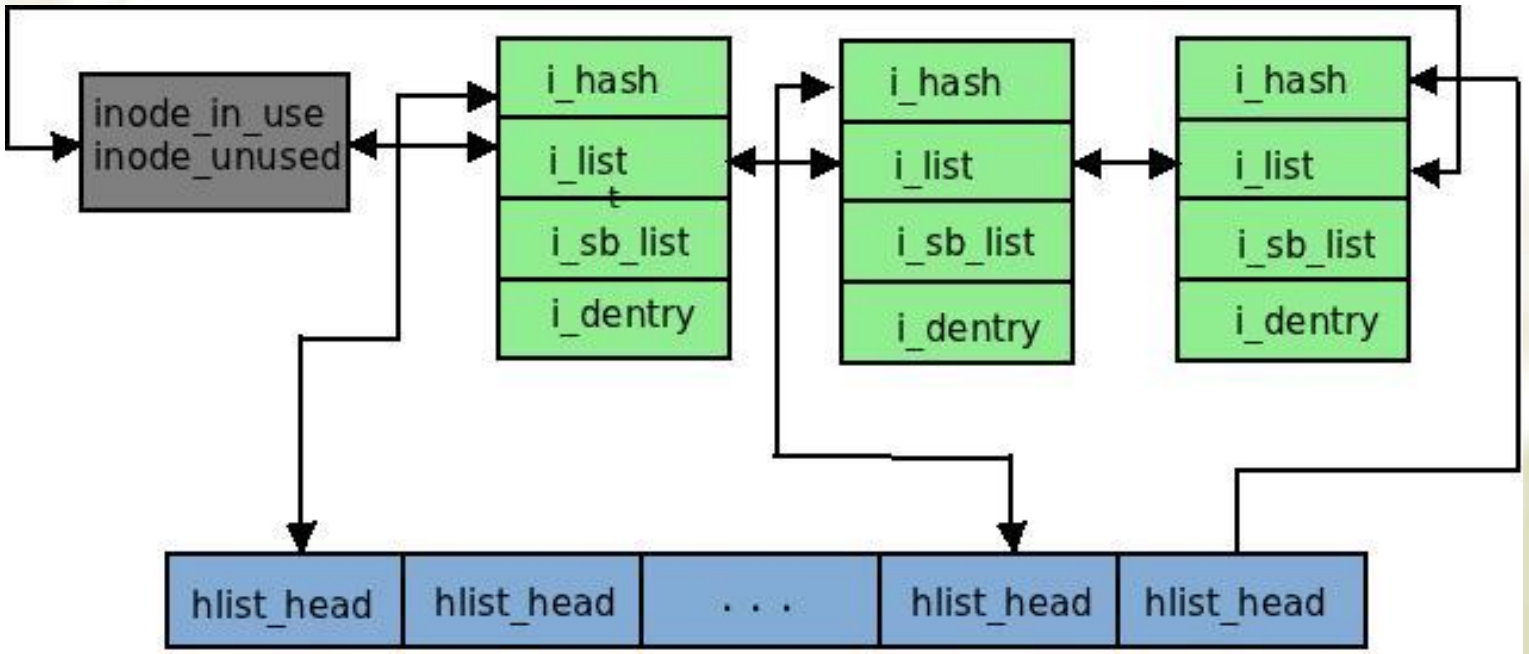
- 一个散列表，包含了所有活动的dentry对象
- 散列表由dentry_hashtable组织，dentry通过d_hash字段链入散列表中
- 一个LRU链表
  - Dentry结构体中由d_lru链表组织。

缓存中的查找

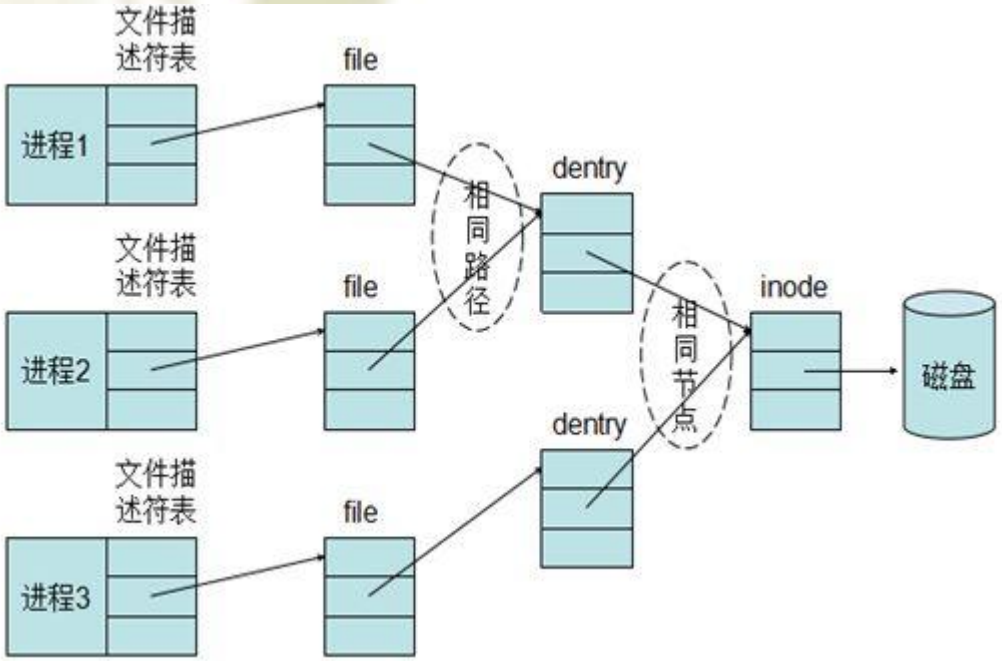
- 缓存由d_hash计算散列值，通过值对应的索引从dentry_hashtable中查找相应的队列，再从队列头循环查找对应的dentry（也就是先从哈希表中查找，然后从LRU表中查找）

索引节点缓存的组织和查找

同样为了加速查找，引入了索引节点缓存（Inode cache），索引节点缓存由inode_hashtable组织，如图所示。



buffer Cache技术



如果要查找的inode不在inode cache中，则需要从磁盘读取。这就涉及buffer cache技术。

buffer cache应用于经常按块读取的元数据。例如在查找过程中，为了获取inode的信息，需要首先从磁盘读取super block的信息。

buffer Cache的组织

Buffer cache 的组织：采用LRU链表（如图源码截图）

```
#define BH_LRU_SIZE     16

struct bh_lru {
        struct buffer_head *bhs[BH_LRU_SIZE];
};

static DEFINE_PER_CPU(struct bh_lru, bh_lrus) = {{ NULL }};
```

bhs是一个缓冲头指针的数组，是用作实现LRU算法的基础。内核使用DEFINE_PER_CPU为每个CPU都建立了一个LRU实例，以改进对CPU高速缓存的利用率。

LRU缓存操作接口：

- lookup_bh_lru：查找所需数据项是否在块缓存中
- lh_lru_install：将新的缓冲头添加到缓冲中

buffer Cache头数据结构

这个数据结构把内存中的页与磁盘中的块关联起来了，源码中对每个具体字段给予了解释。

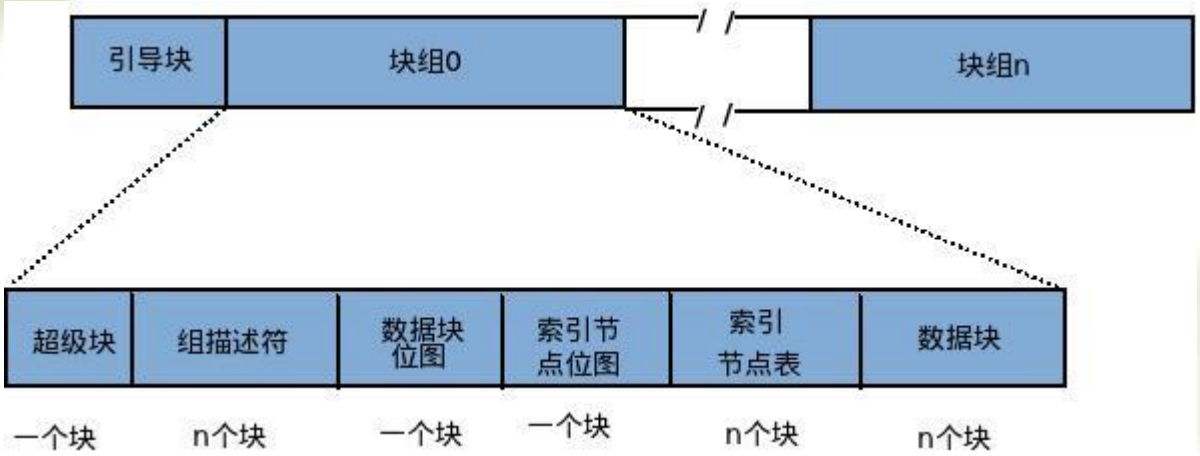
```
/
struct buffer_head {
        unsigned long b_state;          /* buffer state bitmap (see above) */
        struct buffer_head *b_this_page;/* circular list of page's buffers */
        struct page *b_page;            /* the page this bh is mapped to */

        sector_t b_blocknr;             /* start block number */
        size_t b_size;                  /* size of mapping */
        char *b_data;                   /* pointer to data within the page */

        struct block_device *b_bdev;
        bh_end_io_t *b_end_io;          /* I/O completion */
        void *b_private;                /* reserved for b_end_io */
        struct list_head b_assoc_buffers; /* associated with another mapping */
        struct address_space *b_assoc_map;      /* mapping this buffer is
                                                   associated with */
        atomic_t b_count;               /* users using this buffer_head */
};
```

Ext2文件系统超级块的组织形式

内核通过buffer cache技术，获取inode信息之前，我们首先看一下ext2文件系统超级块的组织形式：



任何Ext2分区中的第一个块不受Ext2文件系统的管理，因为这一块是为分区的引导扇区所保留的。Ext2分区的其余部分被分割成块组（block group），每个块组的分布图如图所示。在Ext2文件系统中的所有块组大小相同并被顺序存放，因此，内核可以从块组的整数索引很容易地得到磁盘中一个块组的位置。关于ext2的详细介绍请看维基百科：https://en.wikipedia.org/wiki/Ext2

内核如何从磁盘获取inode信息

```
static struct ext2_inode *ext2_get_inode(struct super_block *sb, ino_t ino,
                                         struct buffer_head **p)
{
        struct buffer_head * bh;
        unsigned long block_group;
        unsigned long block;
        unsigned long offset;
        struct ext2_group_desc * gdp;

        *p = NULL;
        if ((ino != EXT2_ROOT_INO && ino < EXT2_FIRST_INO(sb)) ||
            ino > le32_to_cpu(EXT2_SB(sb)->s_es->s_inodes_count))
                goto Einval;

        block_group = (ino - 1) / EXT2_INODES_PER_GROUP(sb);
        gdp = ext2_get_group_desc(sb, block_group, NULL);
        if (!gdp)
                goto Egdp;
        /*
         * Figure out the offset within the block group inode table
         */
        offset = ((ino - 1) % EXT2_INODES_PER_GROUP(sb)) * EXT2_INODE_SIZE(sb);
        block = le32_to_cpu(gdp->bg_inode_table) +
                (offset >> EXT2_BLOCK_SIZE_BITS(sb));
        if (!(bh = sb_bread(sb, block)))
                goto Eio;

        *p = bh;
        offset &= (EXT2_BLOCK_SIZE(sb) - 1);
        return (struct ext2_inode *) (bh->b_data + offset);
```

内核如何从磁盘获取inode信息

首先根据索引节点号计算出它所在的块组，并得到该块组的描述符。

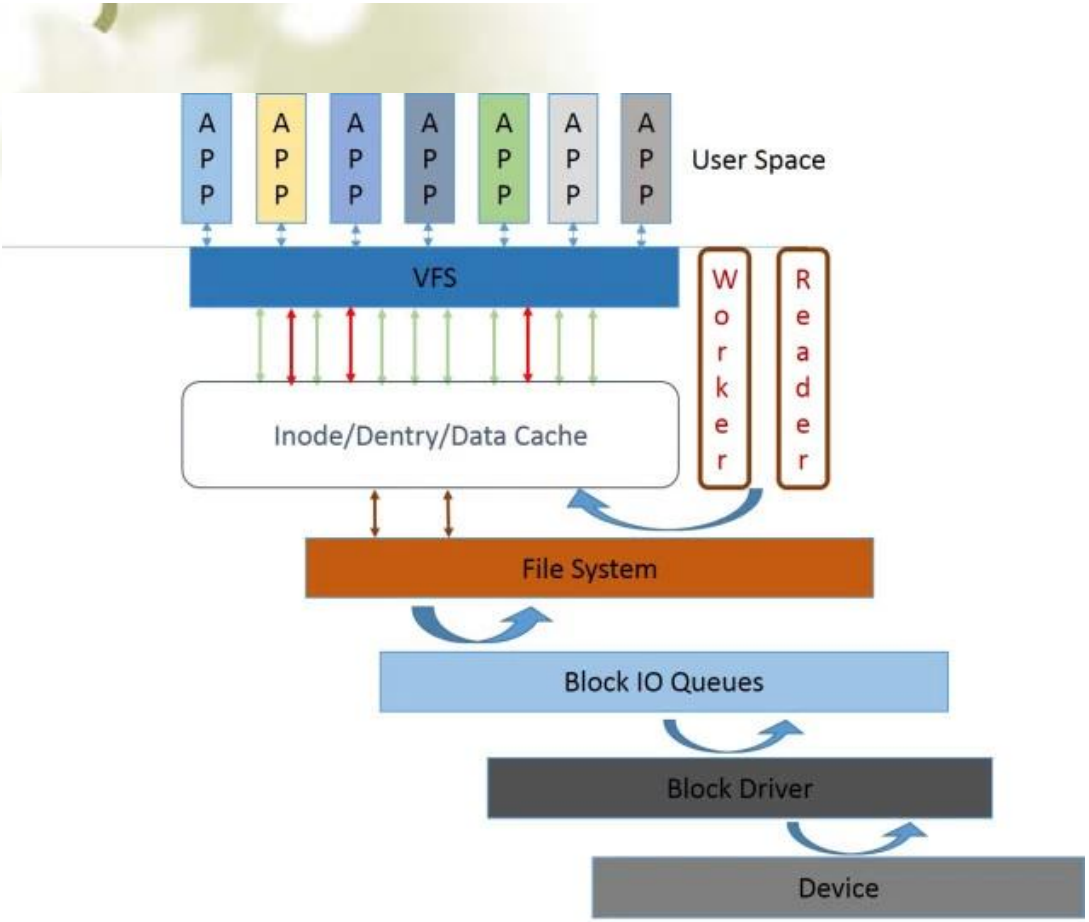
然后算出其在块组索引表中的偏移量，并算出对应的块号。

获取原始inode信息的过程，需要读取超级块信息，具体实现在sb_bread中。其实现过程如下：

通过参数，包括：块设备描述符、块号、以及索引去buffer cache组织的LRU链表中查找。

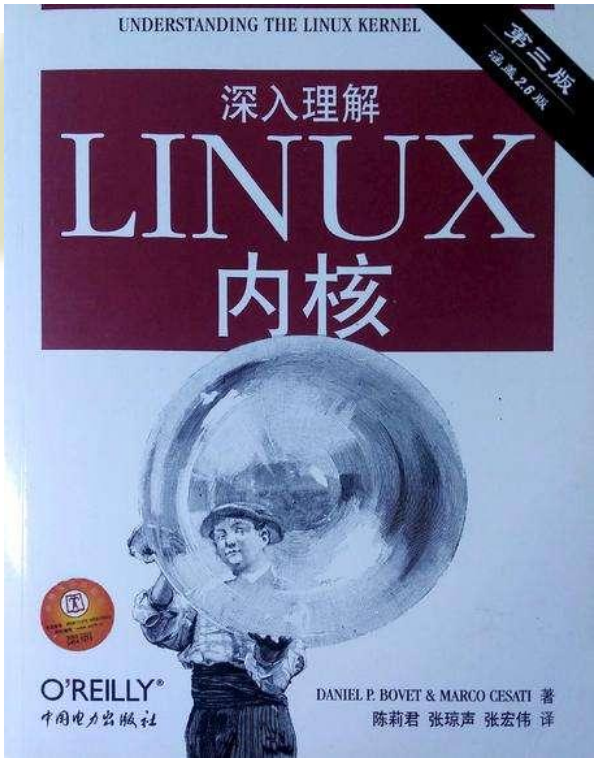
- 如果缓冲区首部在LRU块高速缓存中，则返回对应的buffer_head类型的缓冲区首部。
- 如果不存在，则需要去页高速缓冲中查找，看是否存在，如果存在，则返回页高速缓存中对应的块缓存区所对应的缓冲区首部
- 我们对基于缓冲区的查找过程给予简要概述，下面对本讲给予小结。

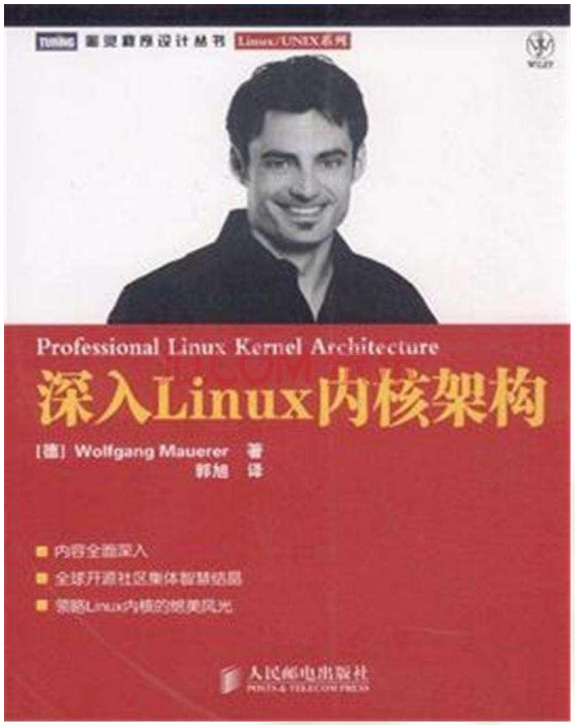
小结



当应用程序打开一个文件时，首先是要查找到这个文件，在查找的过程中，目录项缓存可以加速文件路径名的解析，索引节点缓存可以加速文件元数据的查找，而数据缓存（也就是页缓存）可以加速数据的查找，这些数据都通过文件系统传递给块I/O层，封装成I/O请求给驱动程序，驱动程序最终从设备上存取数据。

参考资料





这么多的内容，你可能说根本还没听懂，是的，肯定没听懂，那你听到了什么，这里相当于给你索引，需要你花数倍的时间和精力查看相关参考资料，深入钻研

深入理解Linux内核 第三版第十二章，第十八章

深入Linux内核架构第八章

以及一篇参考文献的链接，https://msreekan.com/2015/04/24/linux-storage-cache/

带着疑问上路

给定一个文件名，如何查找到相关的文件，请继续阅读open源代码，并说明缓冲区到底起什么作用？

谢谢大家！

THANK YOU