北京郵電大學
Beijing University of Posts and Telecommunications

操作系统内核
-基于Linux

第9讲
文件系统

主讲：杨文川

内容

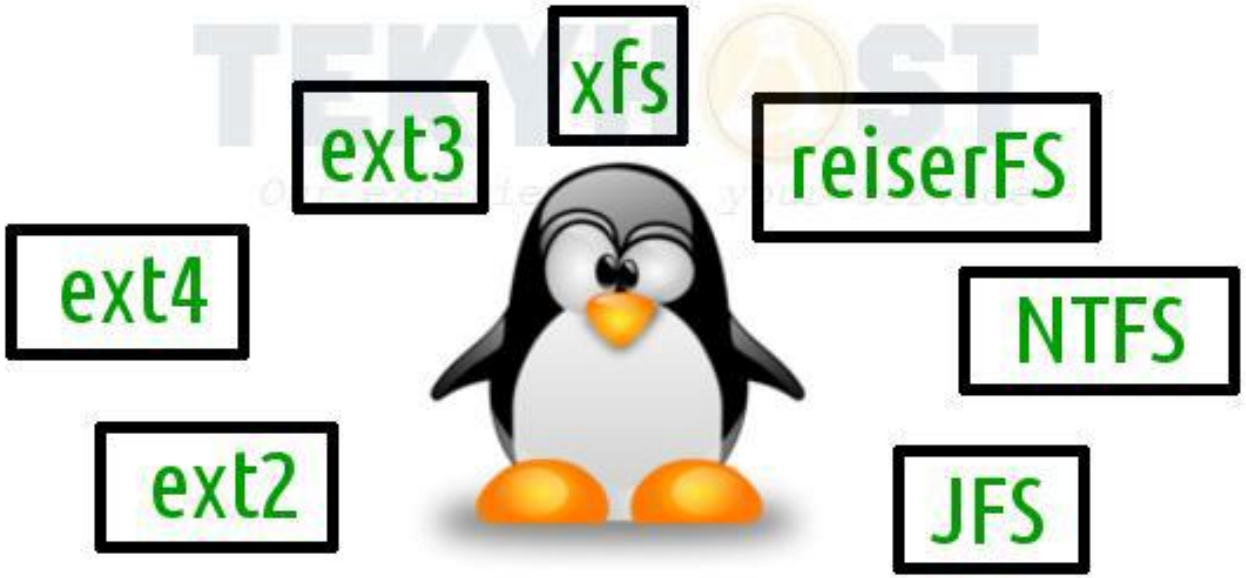
- 1 虚拟文件系统的引入
- 2 虚拟文件系统的主要数据结构
- 3 文件系统中的各种缓存
- 4 页高速缓存机制以及读写
- 5 实践-编写一个文件系统

内容导航：

1 虚拟文件系统(VFS)的引入

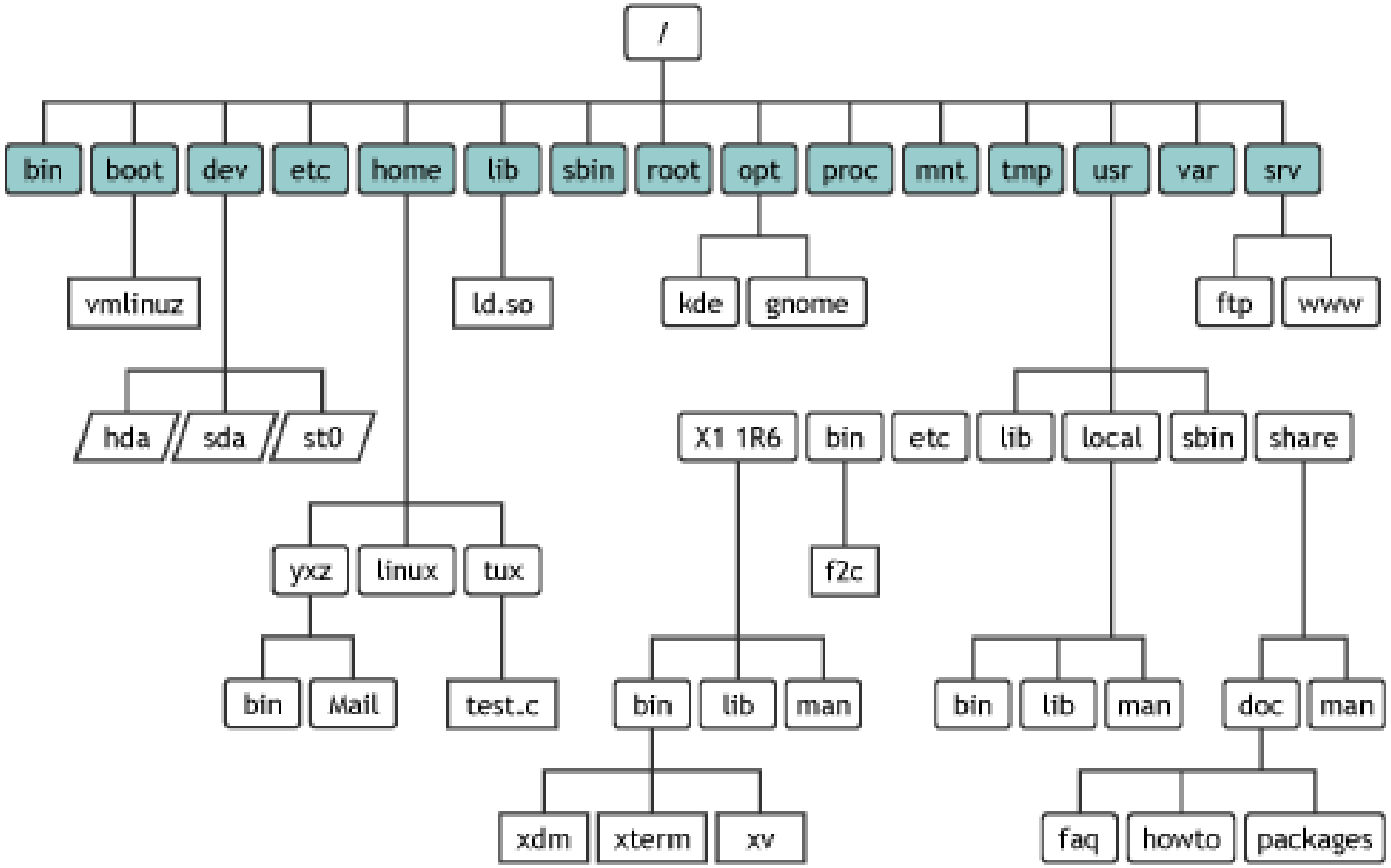
Linux文件系统

- 操作系统中，负责管理和存储文件信息的软件机构，称为**文件管理系统**，简称**文件系统**。也就是 在存储设备上组织文件的方法。
- Linux常见的文件系统有ext2,3,4，NTFS，JFS，XFS等



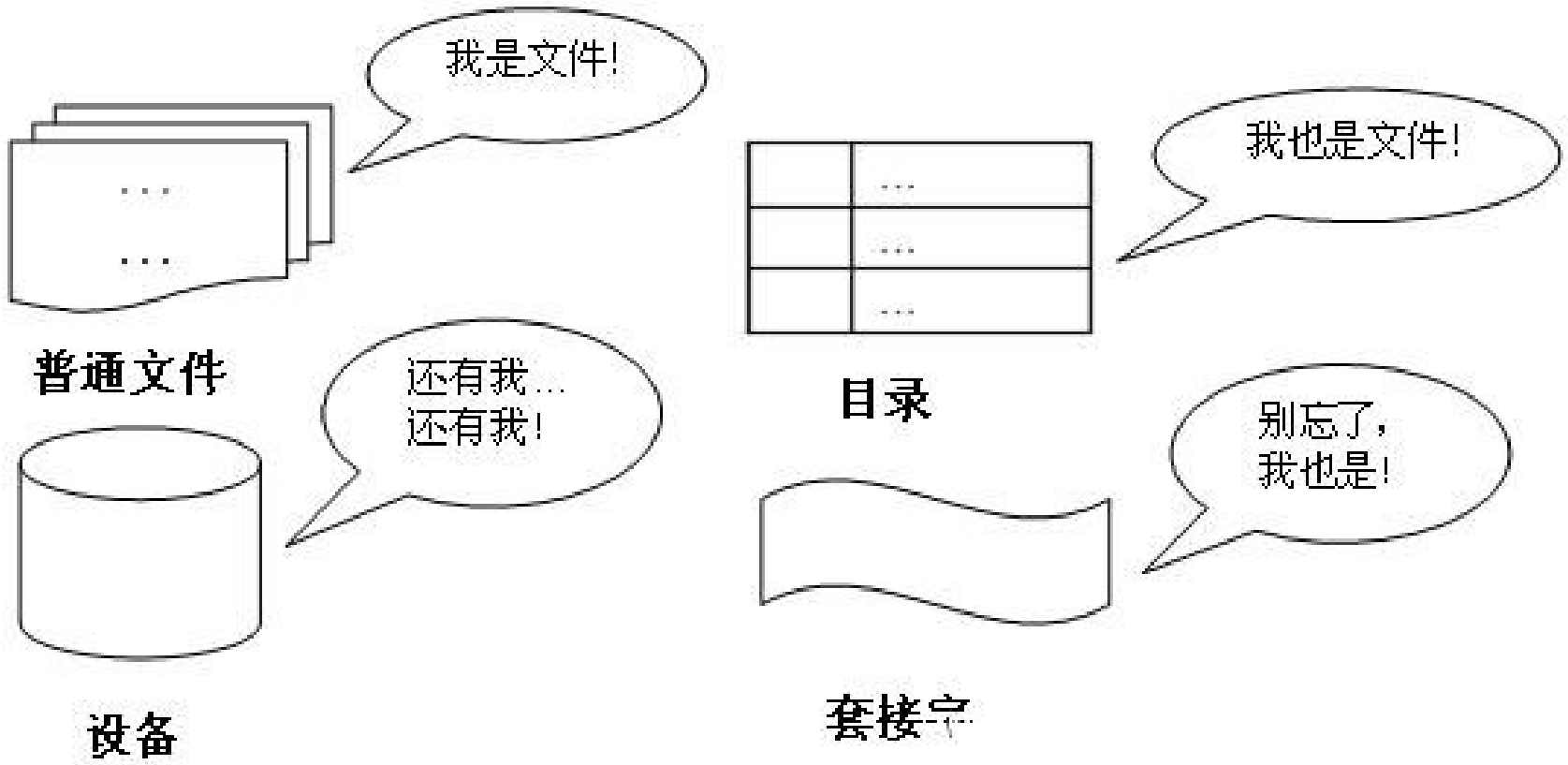
Linux的文件结构

- 文件结构是文件存放在磁盘存贮设备上的组织方式，主要体现在对文件和目录的组织上。
- Linux采用标准的目录结构---树型结构，无论操作系统管理几个磁盘分区，这样的目录树只有一个，如图所示。
- 这样的目录组织，有助于对文件系统，和不同用户文件进行统一管理。

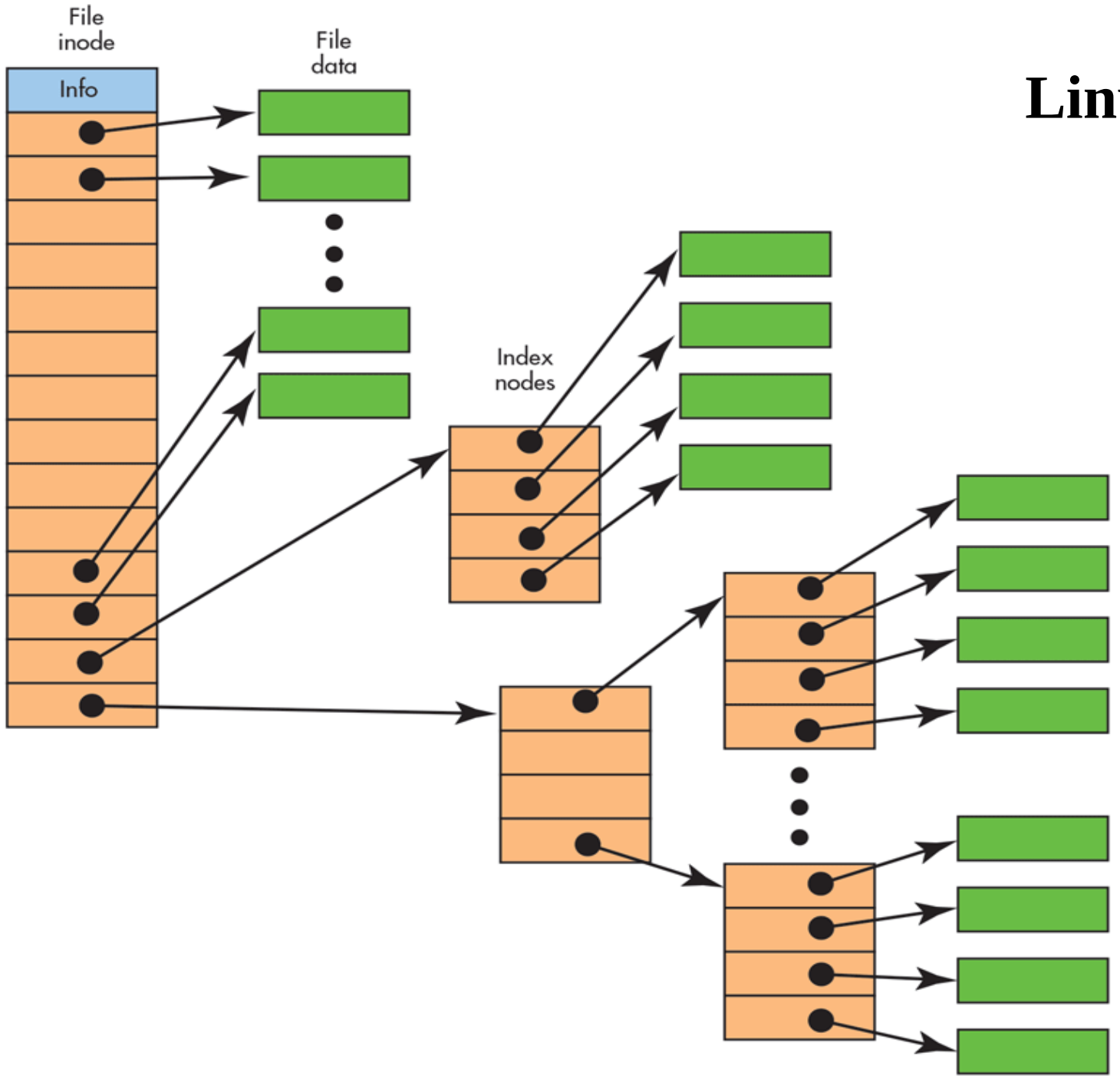


一切皆是文件

- “一切皆是文件”是Unix/Linux 的基本哲学之一。
- 普通的文件，目录、字符设备、块设备、套接字等在Unix / Linux 中，都是以文件被对待；
- 它们虽然类型不同，但是对其提供的却是同一套操作界面。



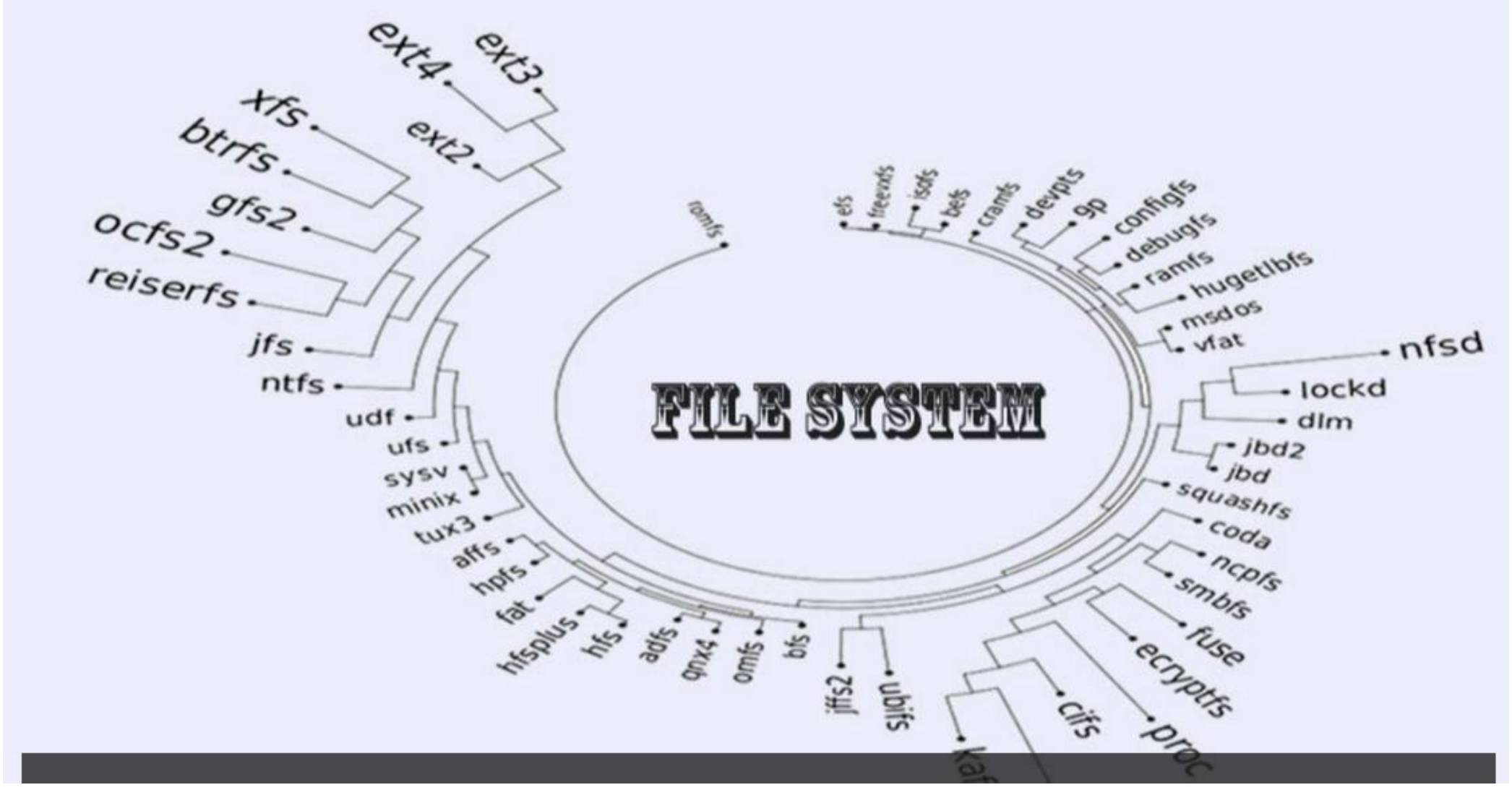
Linux文件系统的索引节点



- Linux文件系统使用索引节点来记录文件信息，
- 系统给每个索引节点分配了一个号码，称为索引节点号。
- 文件系统就是靠这个索引节点号，来识别一个文件.

文件系统类型

- Linux目前支持几十种文件系统类型，如ext2，ext3，ext4，xfs，brtfs等(man 5 fs 可以取得全部文件系统的介绍)。如图所示。



文件系统类型

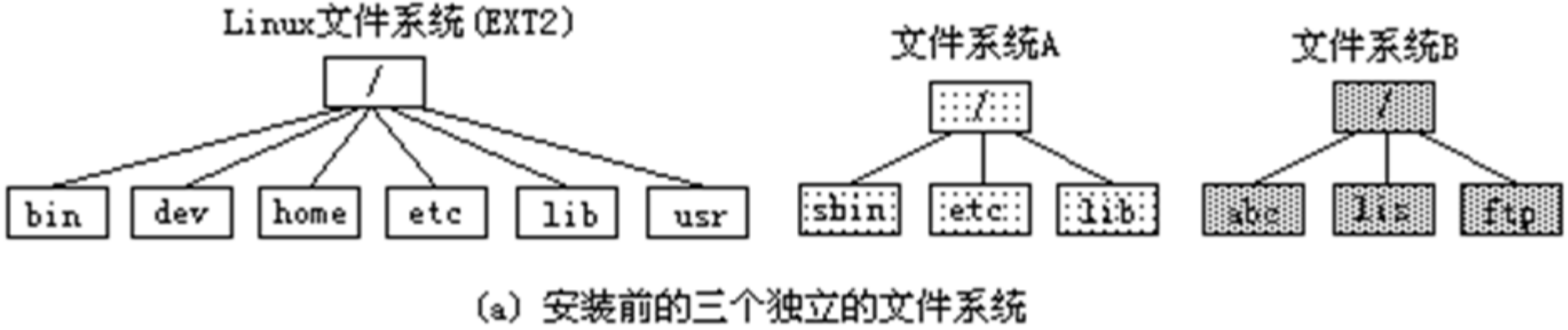
- 不同文件系统采用不同的方法来管理磁盘空间，各有优劣；
- 文件系统是具体到分区的，所以格式化针对的是分区，
- 分区格式化是指采用指定的文件系统类型，对分区空间进行登记、索引，并建立相应的管理表格的过程。
- 例如，常见的类型中：
- ext2具有极快的速度和极小的CPU占用率，可用于硬盘和移动存储设备
- ext3增加日志功能，可回溯追踪
- ext4日志式文件系统，支持1EB(1024*1024TB)，最大单文件16TB，支持连续写入可减少文件碎片。
- xfs可以管理500T的硬盘，brtfs文件系统针对固态盘做优化等等

```
[clj@localhost ~]$ cat /proc/filesystems
nodev   sysfs
nodev   rootfs
nodev   ramfs
nodev   bdev
nodev   proc
nodev   cgroup
nodev   cpuset
nodev   tmpfs
nodev   devtmpfs
nodev   debugfs
nodev   securityfs
nodev   sockfs
nodev   dax
nodev   pipefs
nodev   anon_inodefs
nodev   configfs
nodev   devpts
nodev   hugetlbfs
nodev   autofs
nodev   pstore
nodev   mqueue
        ext3
        ext2
        ext4
nodev   overlay
nodev   binfmt_misc
```

文件系统类型

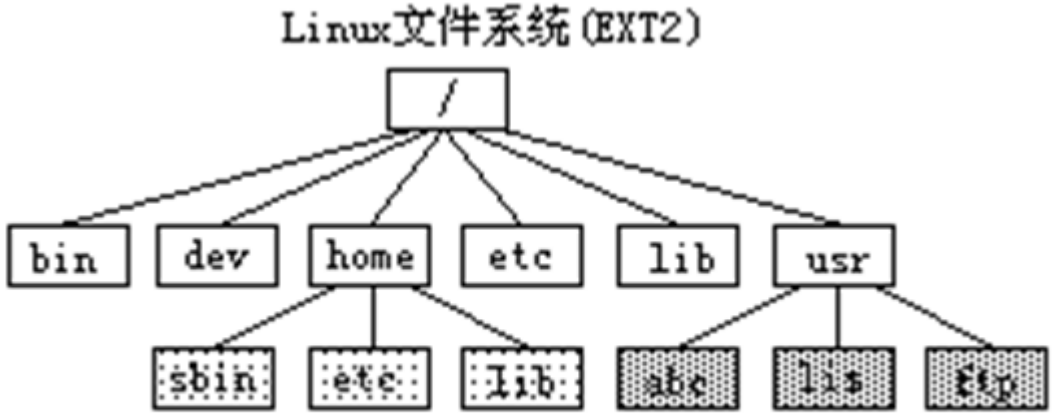
- Linux可以支持不同的文件系统可以在/proc目录下查看filesystems文件，
  - cat /proc/filesystems
- 看到你机子上已经安装的文件系统

文件系统的安装



(a) 安装前的三个独立的文件系统

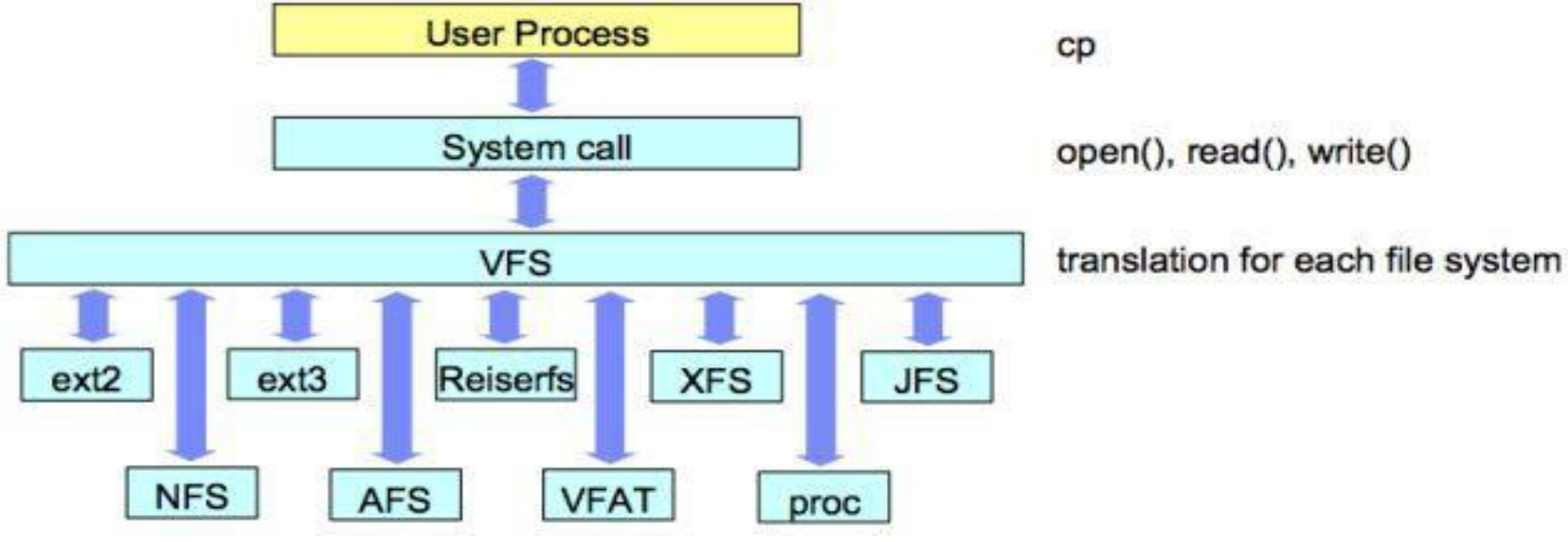
- 将一个文件系统的顶层目录，挂到另一个文件系统的子目录上，使他们成为一个整体，称为“安装(mount)”，把该子目录称为“安装点(mount point)”
- ext2/ext3/ext4是Linux的标准文件系统，
- 如图所示，系统把它的磁盘分区做为系统的根文件系统，
- ext2/ext3/ext4以外的文件系统，则安装在根文件系统下的某个目录，称为系统树形结构中的一个分支。



(b) 安装后的文件系统

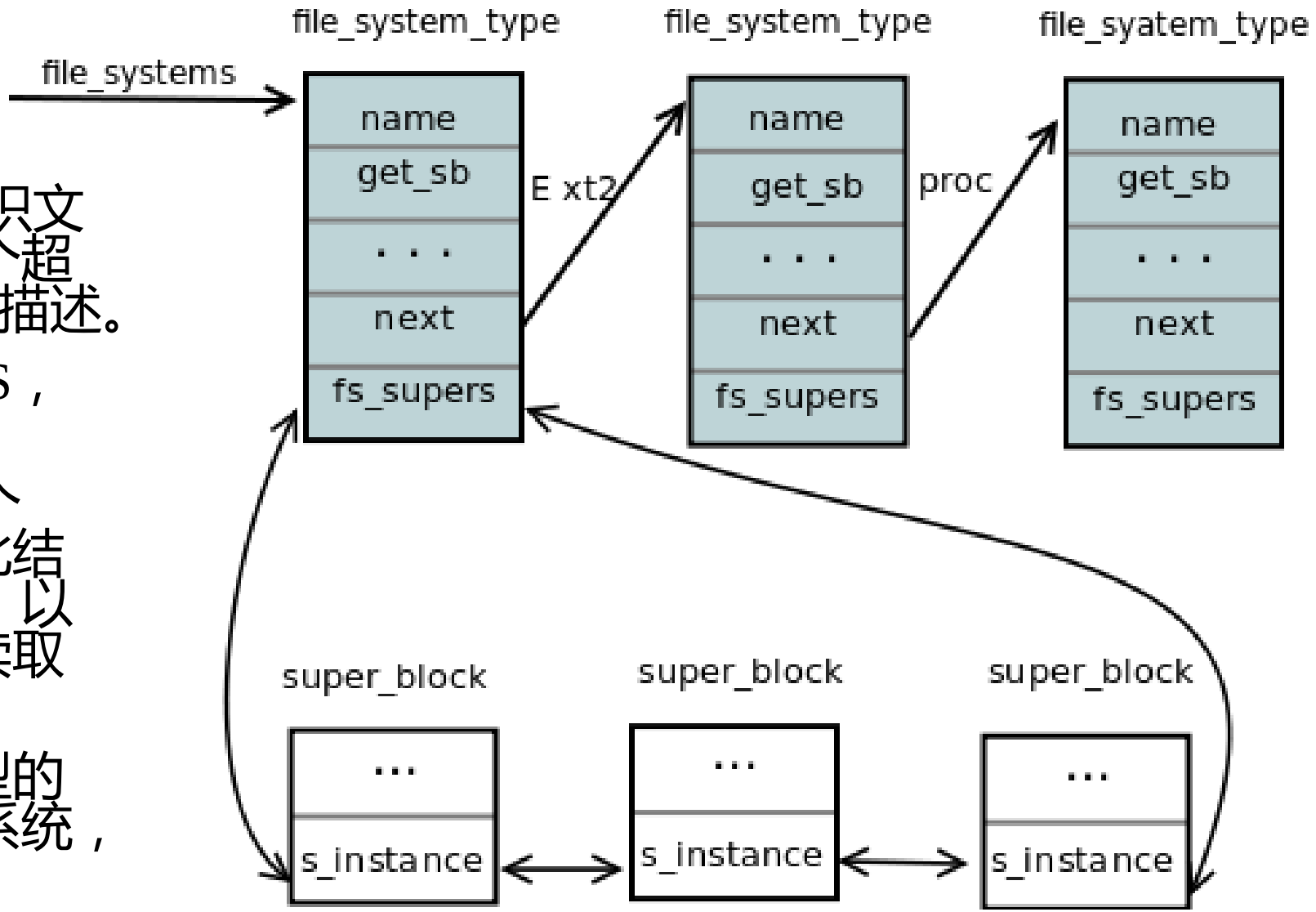
虚拟文件系统VFS框架

- Linux支持多种文件系统，它是如何进行统一、高效的组织以及管理？
- 先看Linux文件系统的统一框架VFS(Virtual Filesystem Switch)。
- 虚拟文件系统所提供的抽象界面，主要由一组标准的、抽象的操作构成，例如open()、read()、write()等，这些函数以系统调用的形式供用户程序调用。
- 这样，用户程序调用这些系统调用时，根本无需关心所操作的文件属于哪个文件系统，这个文件系统是怎样设计和实现的。



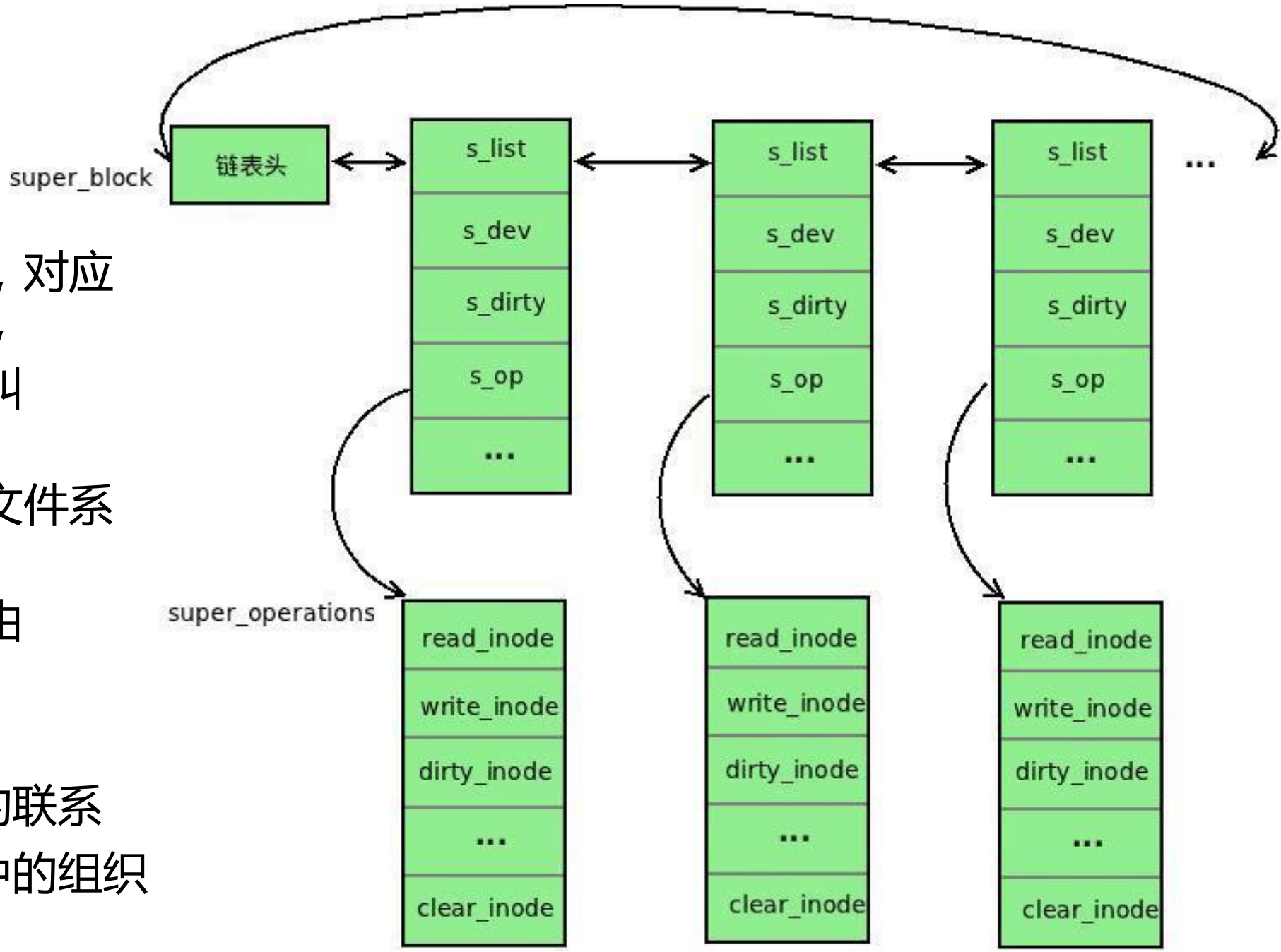
内核如何组织文件系统

- Linux内核采用VFS框架来组织文件系统，每个文件系统用一个超级块(super_block)数据结构来描述。
- 一个文件系统如果想加入VFS，必须先在VFS中进行注册
- 进行注册，实际上是填写一个file_system_type数据结构，此结构描述的是文件系统的名称，以及一个指向对应VFS超级块读取例程的地址。
- 内核采用file_system_type类型的链表，组织每个注册的文件系统，
- 右图为文件系统类型的组织。



超级块数据结构

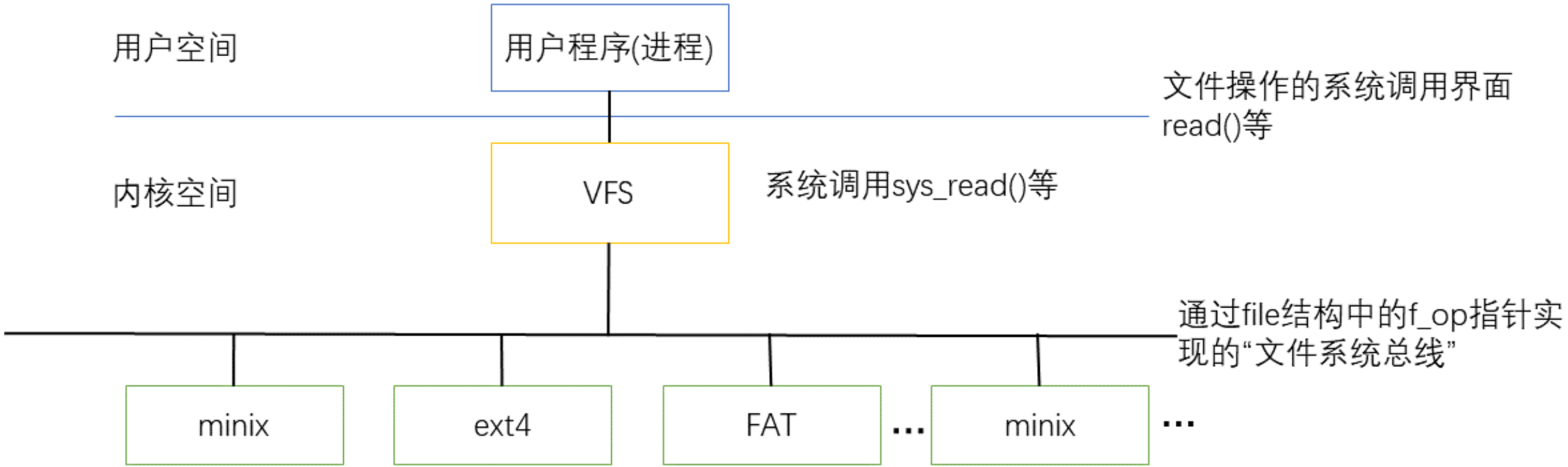
- 每个注册的文件系统，对应着相应的超级块对象，
- 在内核中的数据结构叫super_block。
- 该对象用于存储特定文件系统的信息。
- 系统中所有的超级块由super_blocks链表组织
- 右图描述了：
- 超级块和文件系统的联系
- 所有超级块在内核中的组织



2 VFS中的主要数据结构

虚拟文件系统中对象的引入

- 虚拟文件系统(VFS)的第一个词是“虚拟”，这就意味着，这样的文件系统在磁盘(或其他存储介质上)并没有对应的存储信息。



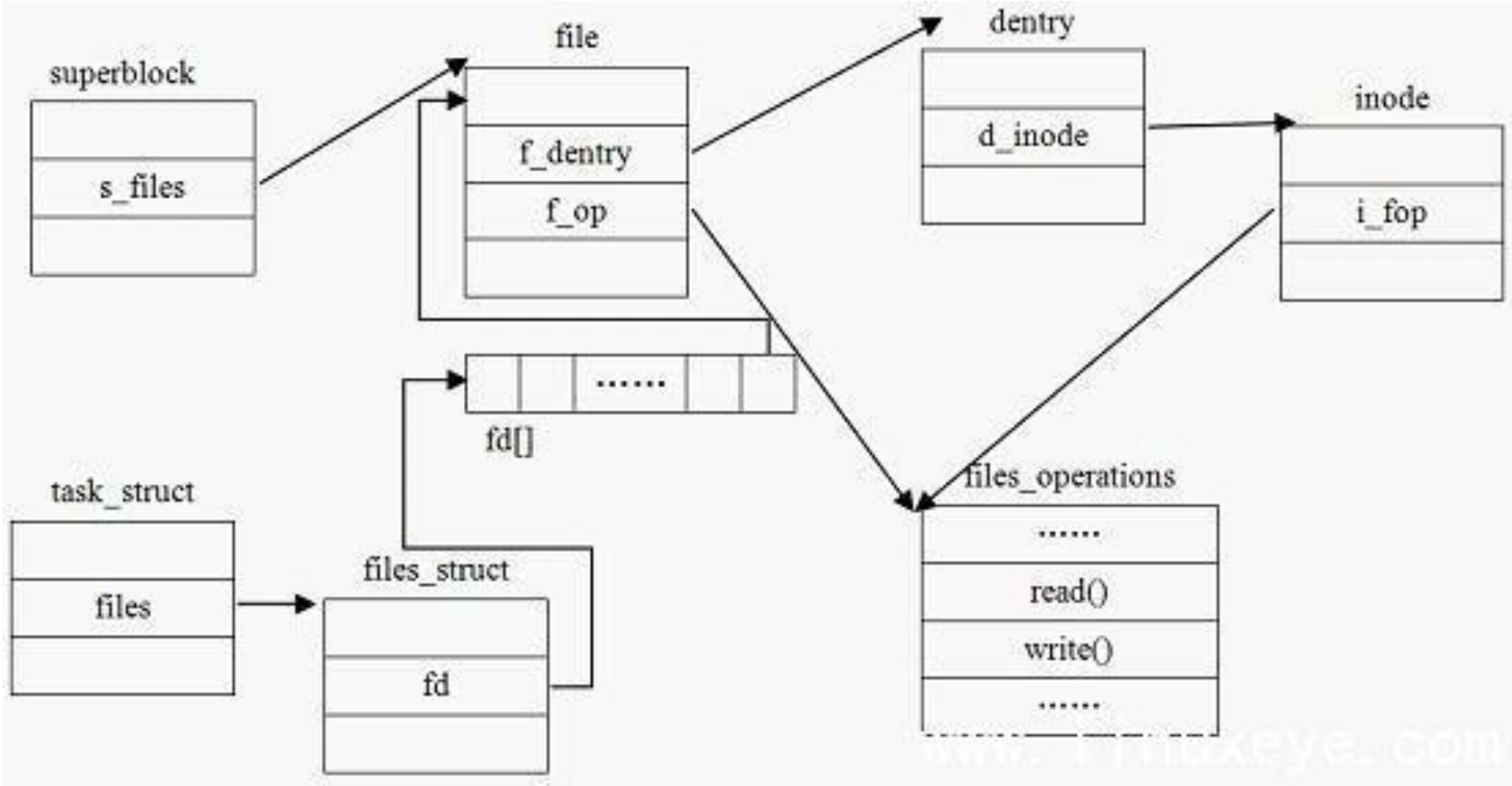
VFS中对象的引入

- 这样一个虚无的文件系统到底是怎样形成？
- 尽管Linux支持多达几十种文件系统，但这些真实的文件系统并不是一下子都挂在系统中的，他们实际上是**按需被挂载**的。
- 这个“虚”的VFS的信息都来源于“实”的文件系统，所以VFS必须承载各种文件系统的共有属性。
- 另外，这些实的文件系统只有安装到系统中，VFS才予以认可，也就是说，VFS只管理挂载到系统中的实际文件系统。

- 那么VFS到底要管哪些对象。
- Linux吸取了Unix的设计思想，在文件系统的设计中抽象出四个概念：
  - 文件，目录项，索引节点和超级块。

VFS中共性对象的抽象

- **超级块(superblock)对象:** 存放系统中已安装文件系统的有关信息
- **文件(file)对象:** 存放打开文件与进程之间进行交互的有关信息
- **索引节点(inode)对象:** 存放关于具体文件的一般信息
- **目录项(dentry)对象:** 文件目录树中一个路径的组成部分，不管是目录还是普通的文件，都是一个目录项对象。如图所示。



超级块对象-描述文件系统的属性

- 超级块用来描述整个文件系统的信息。每个具体的文件系统都有各自的超级块
- VFS超级块是各种具体文件系统在安装时建立的，并在卸载时被自动删除，其数据结构是super_block
- 所有超级块对象以双向环形链表的形式链接在一起
- 与超级块关联的方法就是超级块操作表。
- 这些操作是由数据结构super_operations来描述。
- 如图所示。

```
struct super_block { //超级块数据结构
        struct list_head s_list;                /*指向超级块链表的指针*/
        ……
        struct file_system_type  *s_type;       /*文件系统类型*/
       struct super_operations  *s_op;         /*超级块方法*/
        ……
        struct list_head         s_instances;   /*该类型文件系统*/
        ……
};

struct super_operations { //超级块方法
        ……
        //该函数在给定的超级块下创建并初始化一个新的索引节点对象
        struct inode *(*alloc_inode)(struct super_block *sb);
       ……
        //该函数从磁盘上读取索引节点，并动态填充内存中对应的索引节点对象的剩余部分
        void (*read_inode) (struct inode *);
       ……
};
```

在include/linux/fs.h

索引节点对象-描述文件属性

- 索引节点对象存储了文件的相关信息，代表了存储设备上的一个实际的物理文件。
- 当一个文件首次被访问时，内核会在内存中组装相应的索引节点对象，以便向内核提供对一个文件进行操作时，所必需的全部信息；
- 这些信息一部分存储在磁盘特定位置，另外一部分是在加载时动态填充的。
- 如图所示。

```
struct inode {//索引节点结构
      ……
      struct inode_operations  *i_op;      /*索引节点操作表*/
     struct file_operations    *i_fop;     /*该索引节点对应文件的文件操作集*/
     struct super_block        *i_sb;      /*相关的超级块*/
     ……
};

struct inode_operations { //索引节点方法
     ……
     //该函数为dentry对象所对应的文件创建一个新的索引节点，主要是由open()系统调用来调用
     int (*create) (struct inode *,struct dentry *,int, struct nameidata *);

     //在特定目录中寻找dentry对象所对应的索引节点
     struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
     ……
};
```

在include/linux/fs.h

目录项对象-描述文件的路径

- 引入目录项的概念，主要是出于方便查找文件的目的。
- 一个路径的各个组成部分，不管是目录还是普通的文件，都是一个目录项对象。
- 如在路径/home/source/test.c中，目录/, home, source和文件test.c都对应一个目录项对象。
- 不同于前面的两个对象，目录项对象没有对应的磁盘数据结构，
- VFS在遍历路径名的过程中，现场将它们逐个地解析成目录项对象。
- 如图所示。

```
struct dentry {//目录项结构
     ……
     struct inode *d_inode;           /*相关的索引节点*/
    struct dentry *d_parent;         /*父目录的目录项对象*/
    struct qstr d_name;              /*目录项的名字*/
    ……
     struct list_head d_subdirs;      /*子目录*/
    ……
     struct dentry_operations *d_op;  /*目录项操作表*/
    struct super_block *d_sb;        /*文件超级块*/
    ……
};

struct dentry_operations {
    //判断目录项是否有效;
    int (*d_revalidate)(struct dentry *, struct nameidata *);
    //为目录项生成散列值;
    int (*d_hash) (struct dentry *, struct qstr *);
    ……
};
```

在include/linux/dcache.h

文件对象

- 文件对象是已打开的文件在内存中的表示，主要用于建立进程和磁盘上的文件的对应关系。它由sys_open() 现场创建，由sys_close()销毁。文件对象和物理文件的关系有点像进程和程序的关系一样。
- 当我们站在用户空间来看待VFS，我们只需与文件对象打交道，而无须关心超级块，索引节点或目录项。
- 因为多个进程可以同时打开和操作同一个文件，所以同一个文件也可能存在多个对应的文件对象。
- 文件对象仅仅在进程观点上代表已经打开的文件，它反过来指向目录项对象。
- 一个文件对应的文件对象可能不是惟一的，但是其对应的索引节点和目录项对象无疑是惟一的。
- 如后图所示。

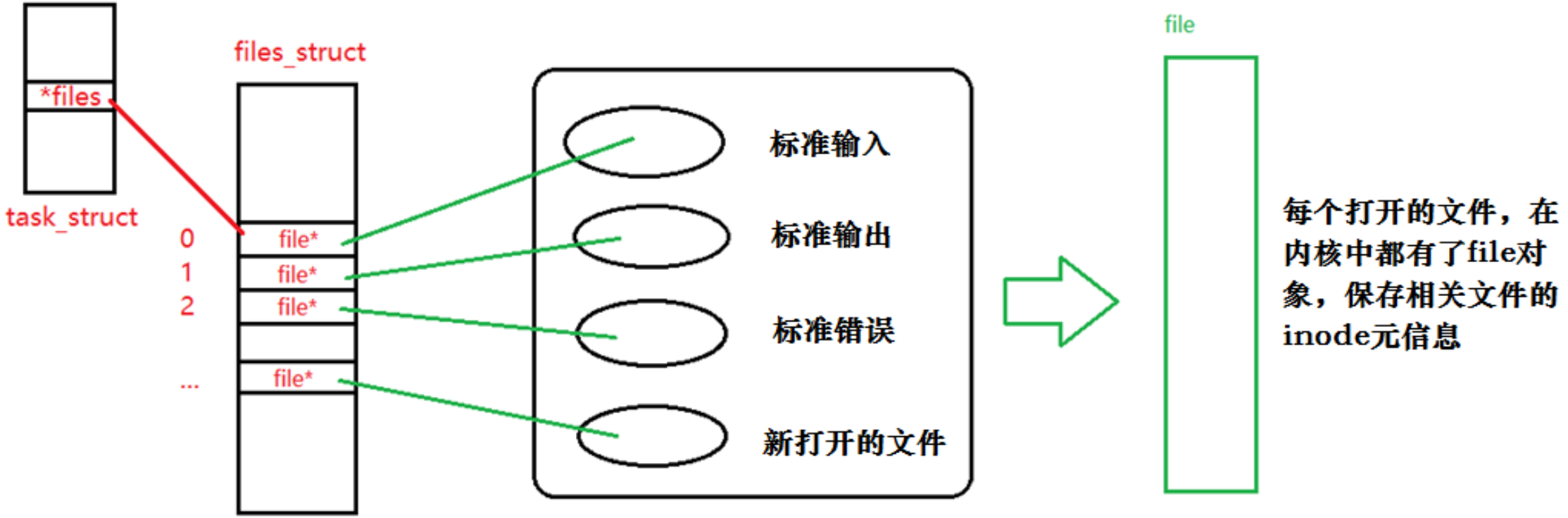
文件对象-描述进程打开的文件

```
struct file {
    ……
     struct list_head        f_list;        /*文件对象链表*/
    struct dentry           *f_dentry;      /*相关目录项对象*/
    struct vfsmount         *f_vfsmnt;      /*相关的安装文件系统*/
    struct file_operations *f_op;           /*文件操作表*/
    ……
};

struct file_operations {
    ……
    //文件读操作
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ……
    //文件写操作
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ……
    int (*readdir) (struct file *, void *, filldir_t);
    ……
    //文件打开操作
    int (*open) (struct inode *, struct file *);
    ……
};
```

- 在include/linux/fs.h

与进程相关的文件结构 - 用户打开文件表

- 文件描述符是用来描述打开的文件的。
- 每个进程用一个files_struct结构，来记录文件描述符的使用情况，这个files_struct结构称为进程打开文件表，它是进程的私有数据。
- 注意，每个打开的文件，在内核中都有file对象



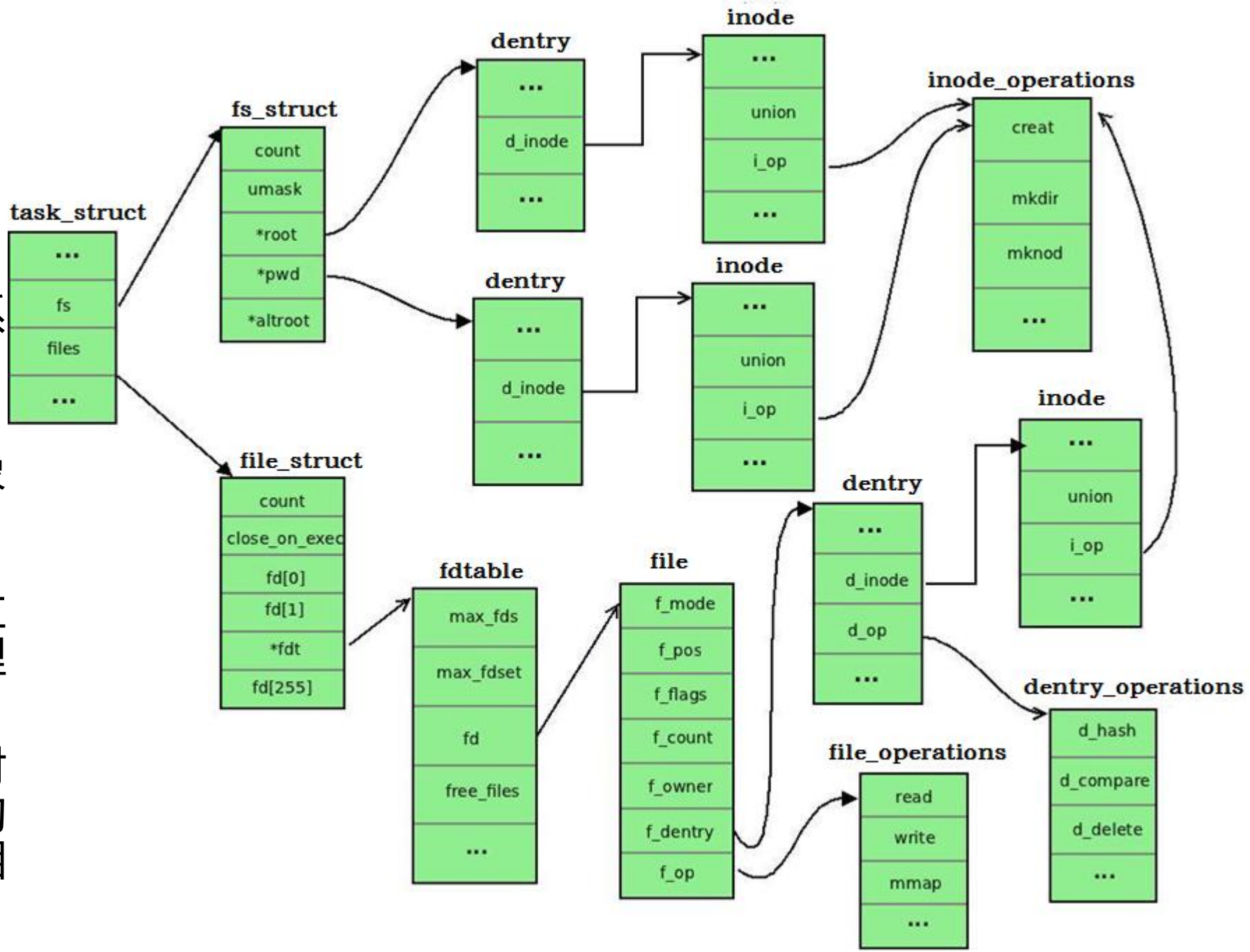
与进程相关的文件结构

```
struct files_struct {//打开的文件集
         atomic_t count;                /*结构的使用计数*/
        ……
         int max_fds;                   /*文件对象数的上限*/
        int max_fdset;                 /*文件描述符的上限*/
        int next_fd;                   /*下一个文件描述符*/
        struct file ** fd;             /*全部文件对象数组*/
        ……
 };

struct fs_struct {//建立进程与文件系统的关系
         atomic_t count;                /*结构的使用计数*/
        rwlock_t lock;                 /*保护该结构体的锁*/
        int umask;                     /*默认的文件访问权限*/
        struct dentry * root;          /*根目录的目录项对象*/
        struct dentry * pwd;           /*当前工作目录的目录项对象*/
        struct dentry * altroot;       /*可供选择的根目录的目录项对象*/
        struct vfsmount * rootmnt;     /*根目录的安装点对象*/
        struct vfsmount * pwdmnt;      /*pwd的安装点对象*/
        struct vfsmount * altrootmnt;/*可供选择的根目录的安装点对象*/
};
```

- file_struct结构用来记录进程打开的文件表，
  - 在include/linux/fdtable.h
- 而fs_sturct结构描述进程与文件系统之间的关系。
  - 在include/linux/fs_struct.h
  - 注意，Linux版本不同，内容有差异

数据结构关系图

- 进程通过task_struct中的files域，来了解它当前所打开的文件对象；通过fs域了解进程所在的文件系统。
- 文件对象通过域f_dentry找到它对应的目录项对象再由目录项对象的d_inode域，找到它对应的索引结点，这样就建立了文件对象与实际的物理文件的关联。
- 索引节点对象，目录项对象以及文件对象所对应的操作函数列表，是通过相应的操作域得到的。



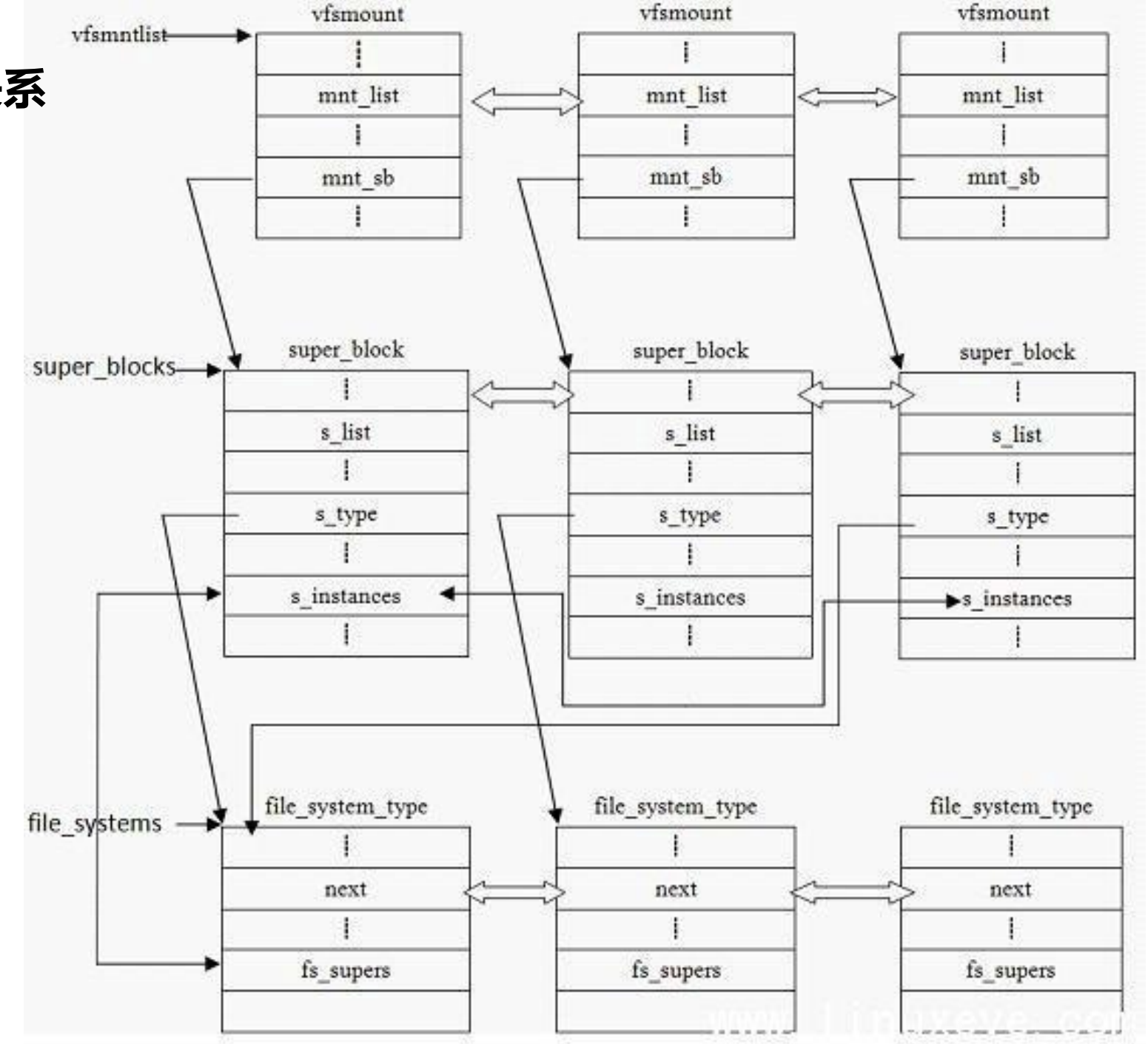
与文件系统相关的数据结构

```
struct vfsmount
{
        struct list_head mnt_hash;              /*散列表*/
        struct vfsmount *mnt_parent;            /*父文件系统*/
        struct dentry *mnt_mountpoint;          /*安装点的目录项对象*/
        struct dentry *mnt_root;                /*该文件系统的根目录项对象*/
        struct super_block *mnt_sb;             /*该文件系统的超级块*/
        struct list_head mnt_mounts;            /*子文件系统链表*/
        struct list_head mnt_child;             /*子文件系统链表*/
        atomic_t mnt_count;                     /*使用计数*/
        int mnt_flags;                          /*安装标志*/
        char *mnt_devname;                      /*设备文件名*/
        struct list_head mnt_list;              /*描述符链表*/
        struct list_head mnt_fslink;            /*具体文件系统的到期列表*/
        struct namespace *mnt_namespace;        /*相关的名字空间*/
};
```

- 之前介绍了与文件系统相关结构：
- 文件系统类型(file_system_type)和超级块(supe_block),
- 这里给出安装点数据结构VFSmount。
  - 在include/linux/mount.h
  - 注意，Linux版本不同，内容有差异

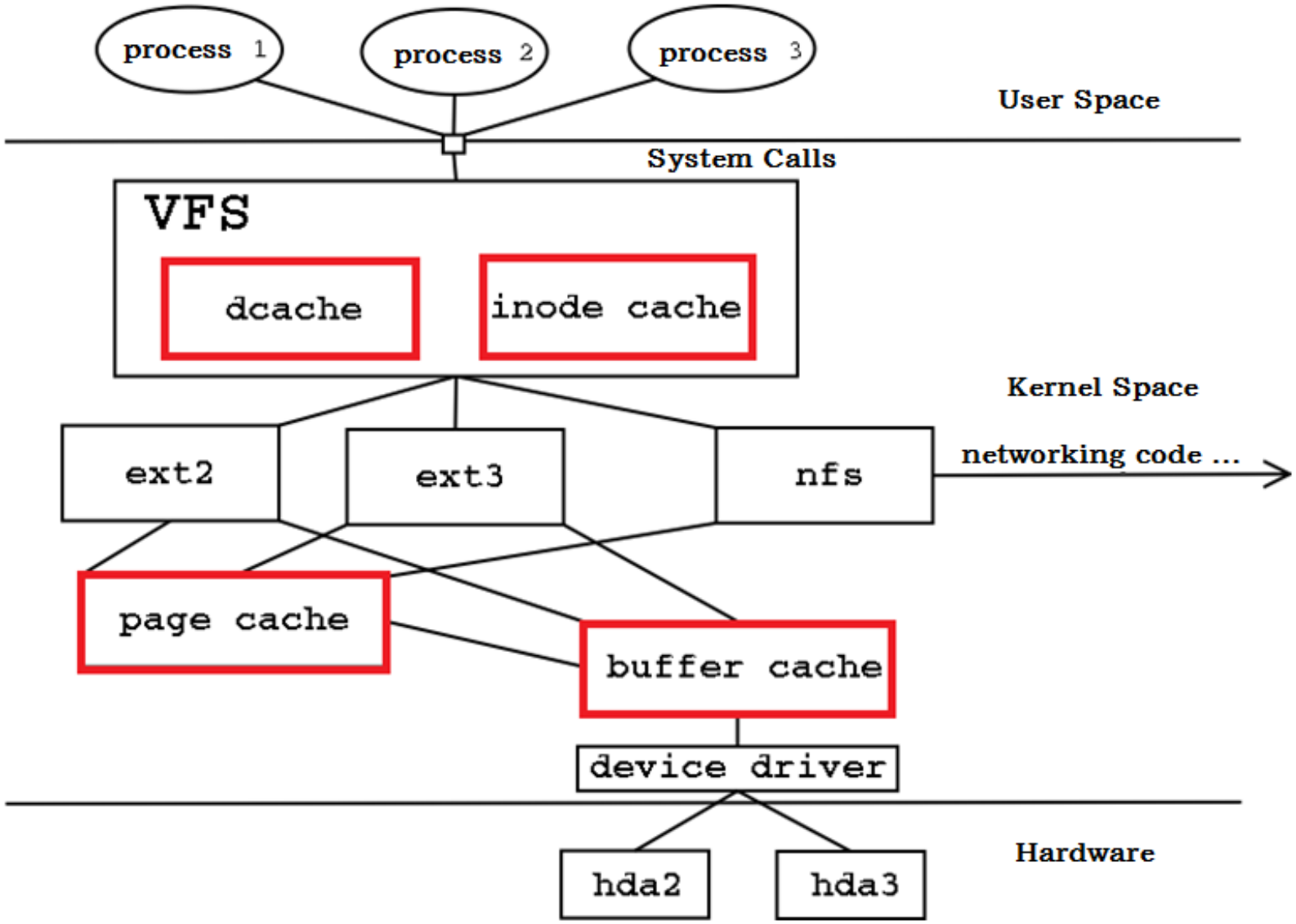
超级块、安装点和具体文件系统的关系

- 被Linux支持的文件系统，无论它有零个或多个实例被安装到系统中，file_system_type结构只有一个。
- 每安装一个文件系统，就对应有一个超级块和安装点。超级块通过它的一个域s_type指向其对应的具体的文件系统类型。
- 具体的文件系统通过file_system_type中的一个域fs_supers，把具有同一种文件类型的超级块链接起来。
- 同一种文件系统类型的超级块通过域s_instances链接。



3 文件系统的缓冲区

文件系统中的缓冲区



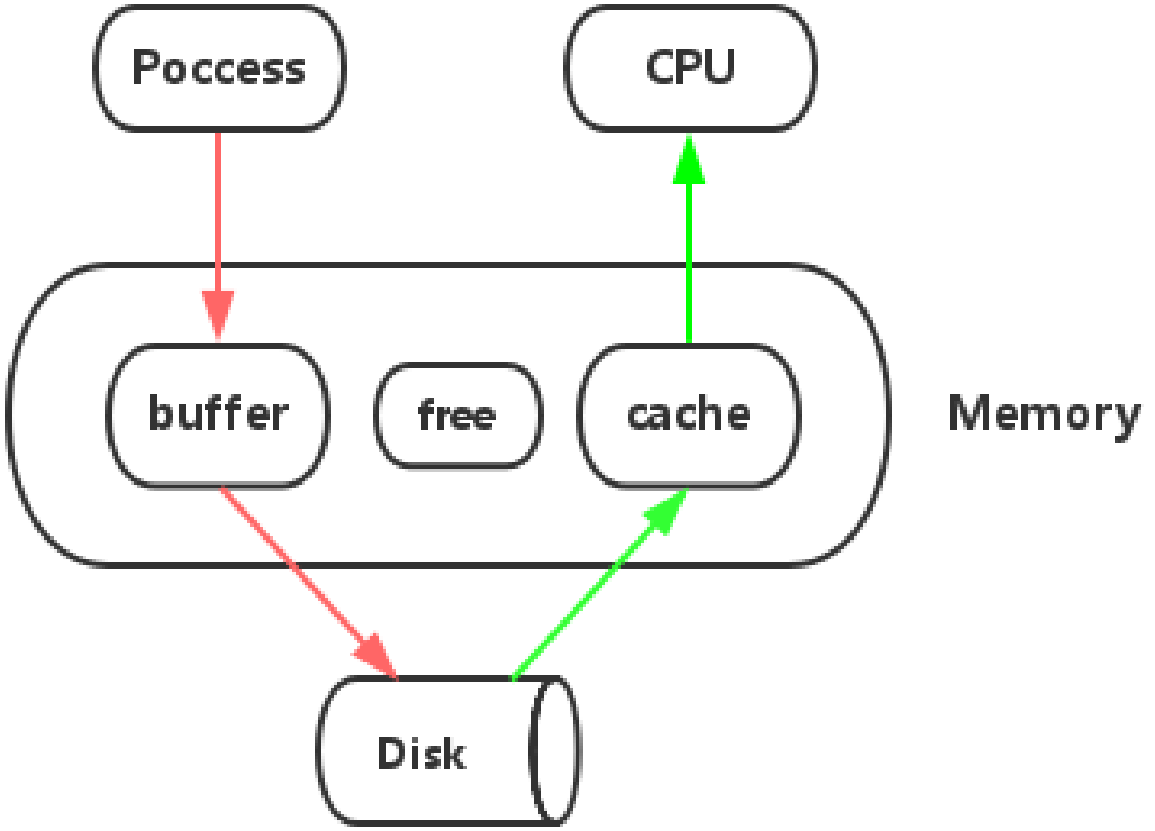
- 缓冲区(buffer)，是内存空间的一部分。也就是说，在内存空间中预留了一定的存储空间
- 这些存储空间用来缓冲输入或输出的数据，这部分预留的空间就叫做缓冲区，
- 如图中的红色框，dcache, inode cache, page cache, buffer cache，它们有什么作用，有什么区别？

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；
- buffer 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；
- cache 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。
  - cat /proc/meminfo

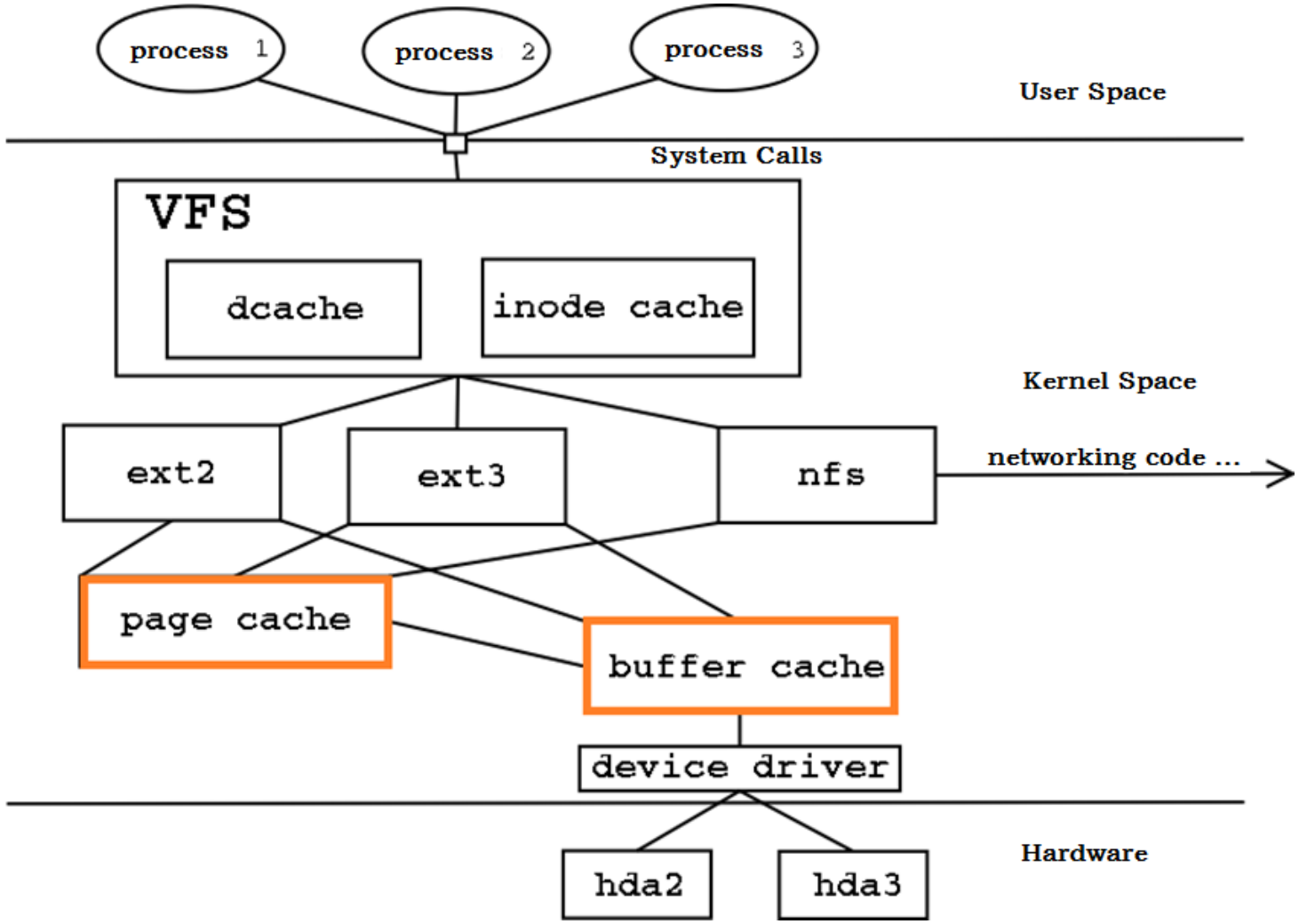
buffer和cache有何不同？

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



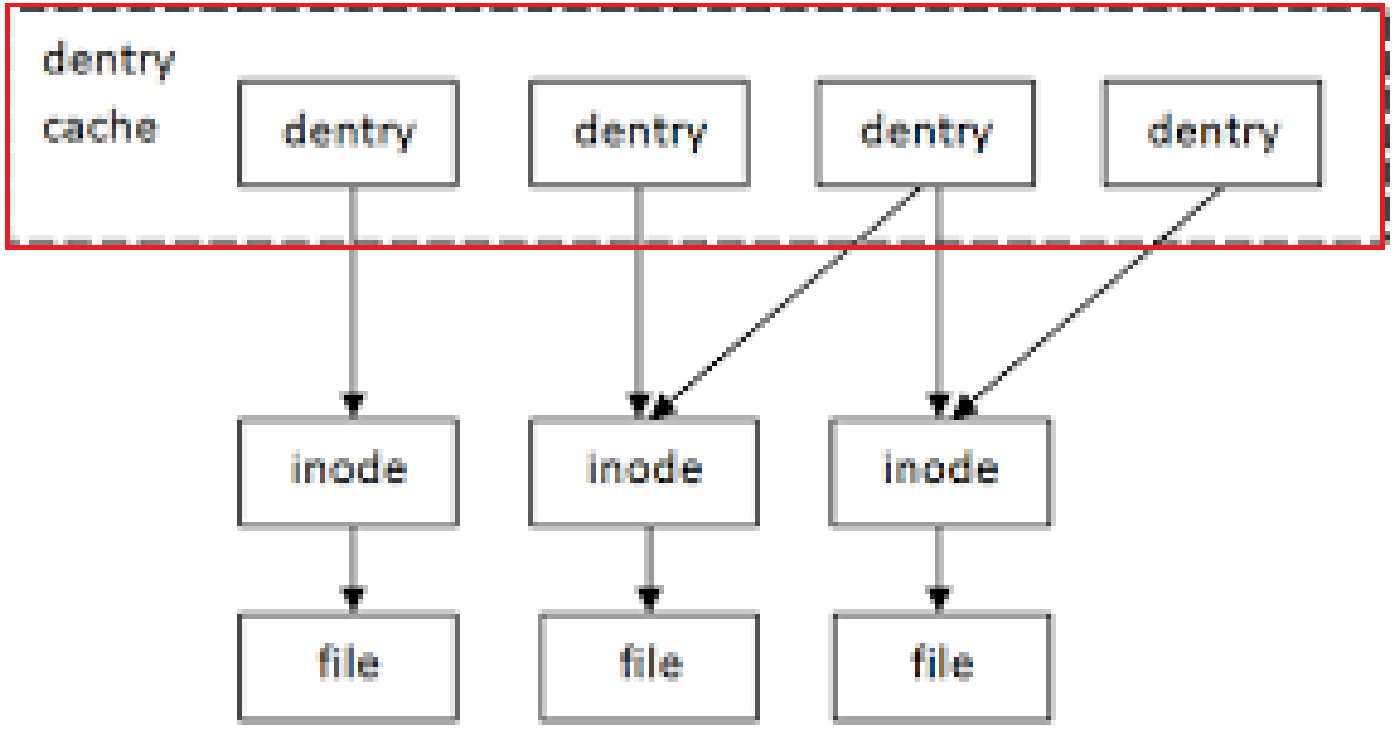
Buffer Cache和 Page Cache有何不同？

- 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。



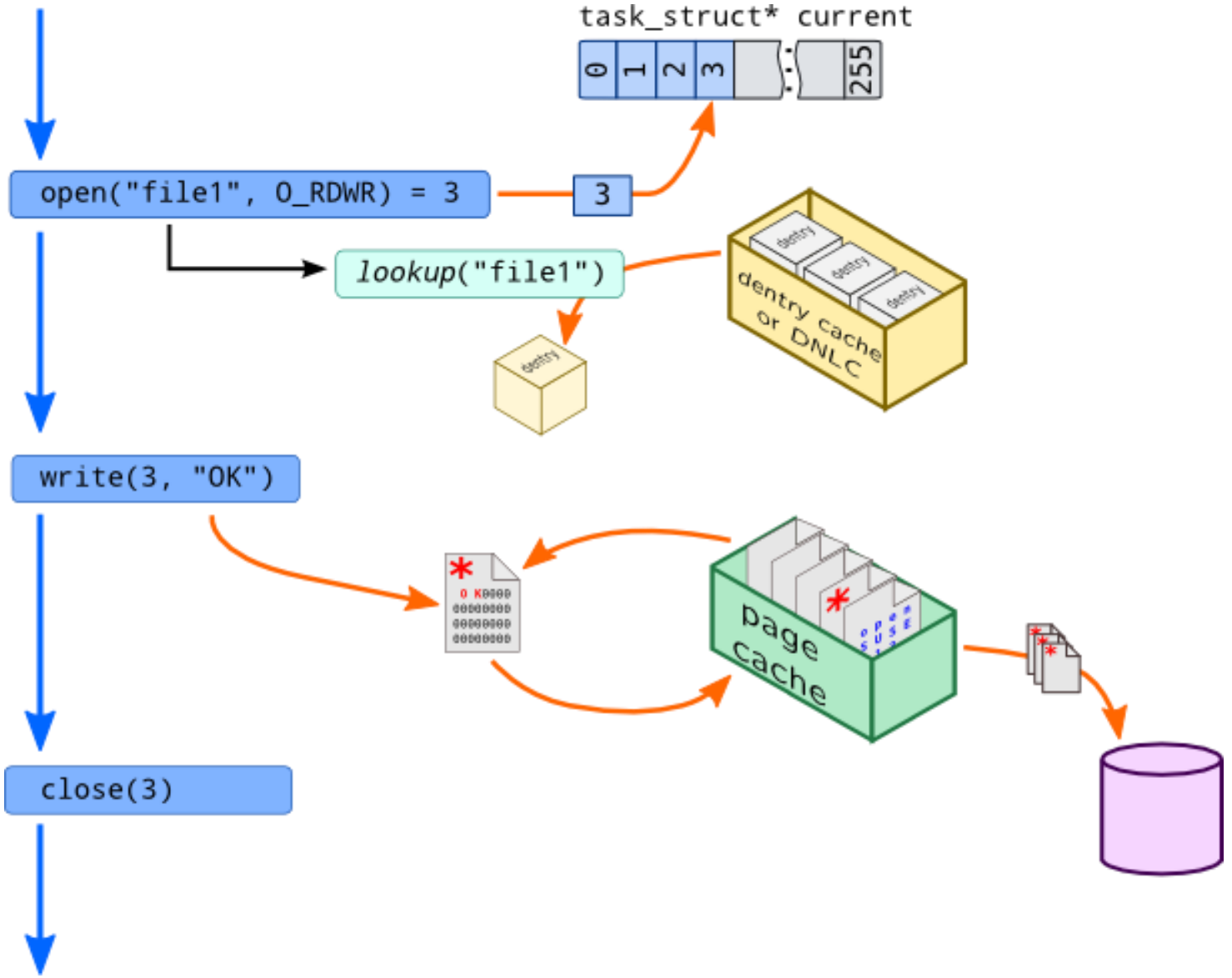
dentry cache 和inode cache有什么作用？

- dcache -也就是"dentry" 对象的cache, 用于把路径转换为索引节点。
- inodecache -也就是 “inode” 对象的cache, 用于表示文件系统中的文件或者目录。



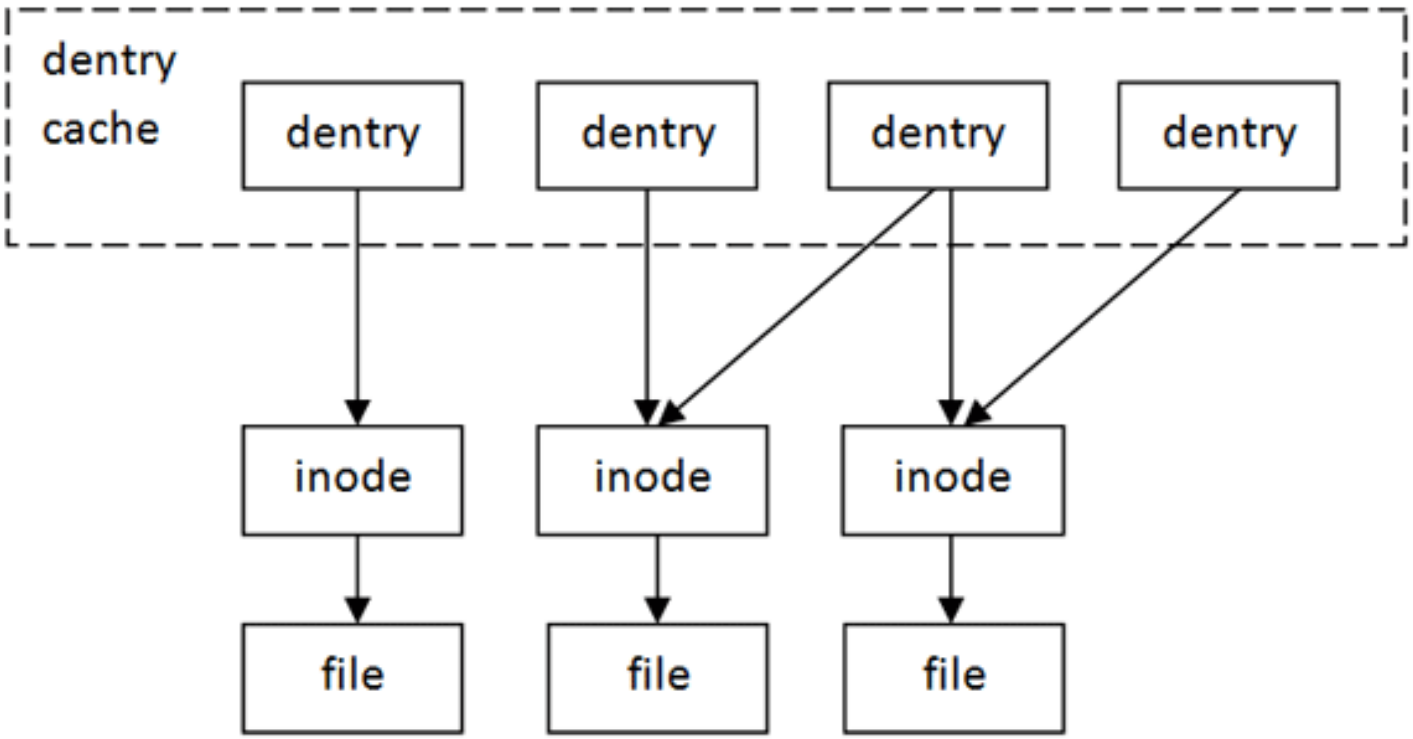
打开文件到底做什么？

- 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。
- 如图所示。

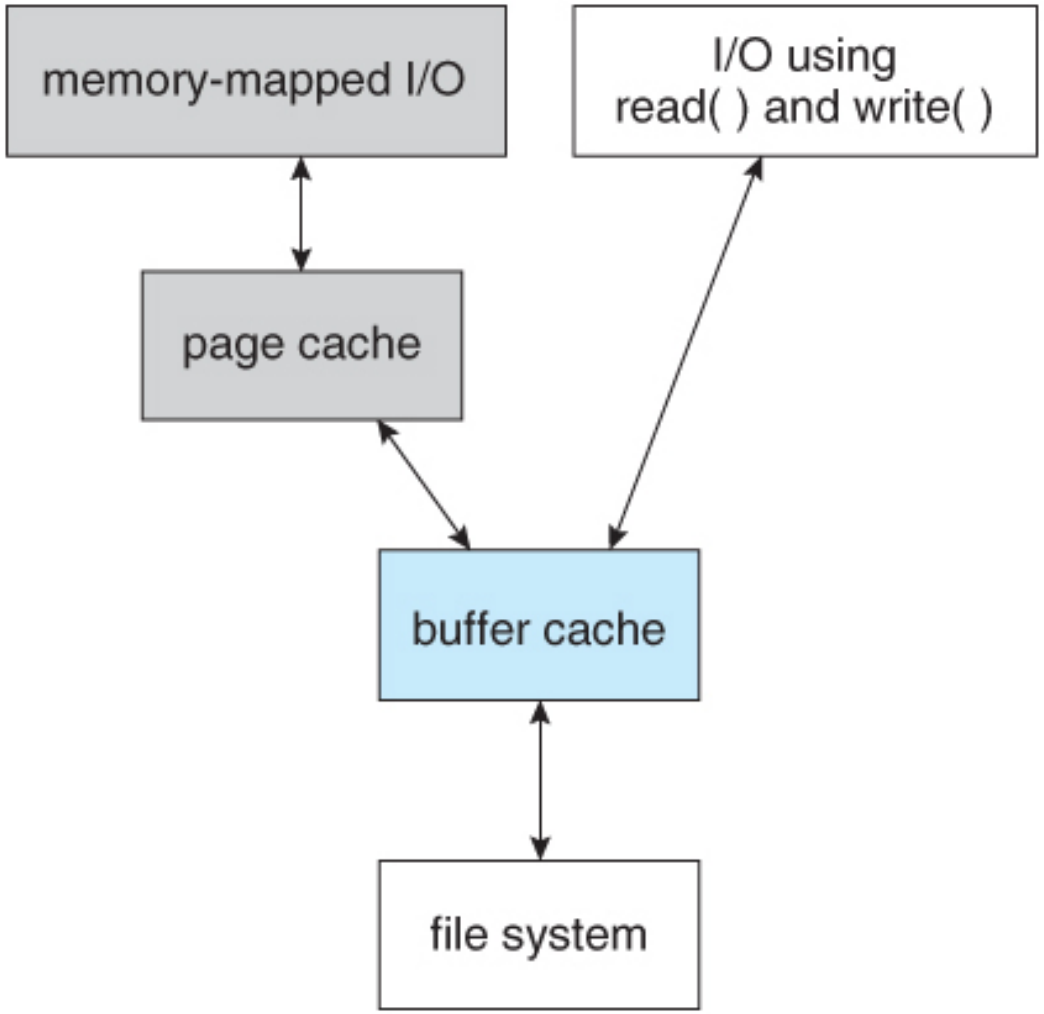


查找过程概述

- 查找过程内核实现复杂，涉及众多cache技术，如图所示。
- 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。



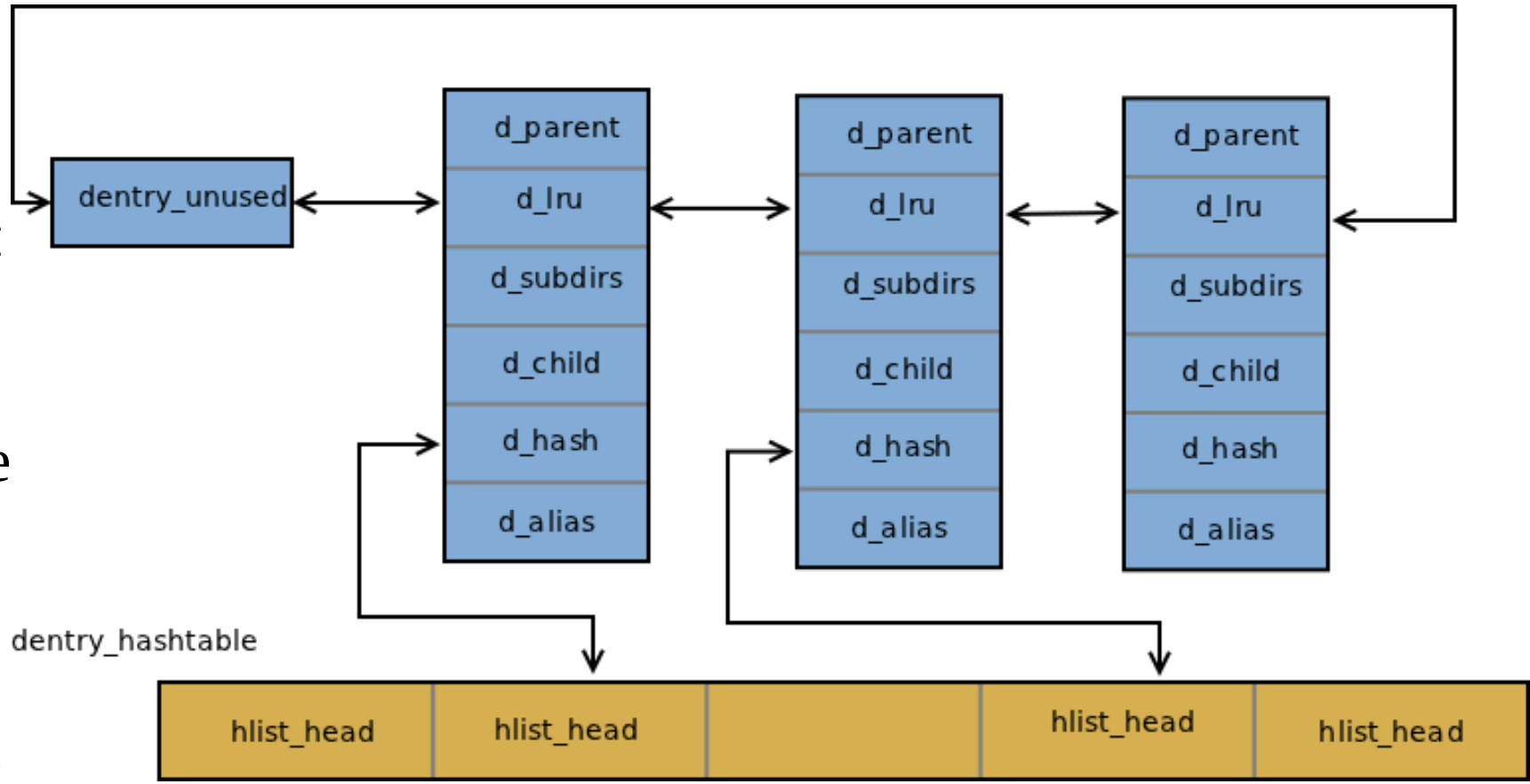
查找过程概述



- 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。

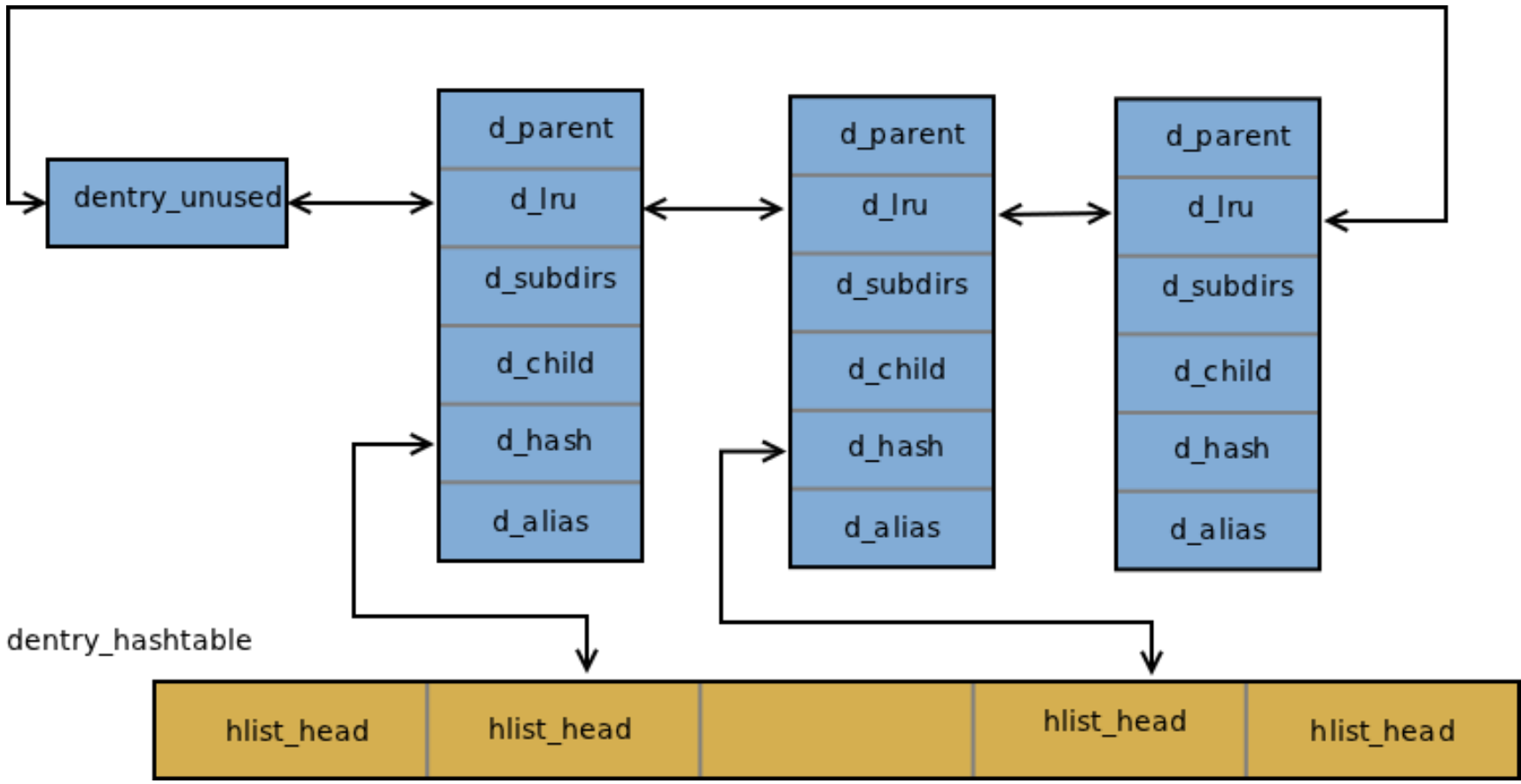
目录项缓存的组织

- 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dentrycache。
- 如图示，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。



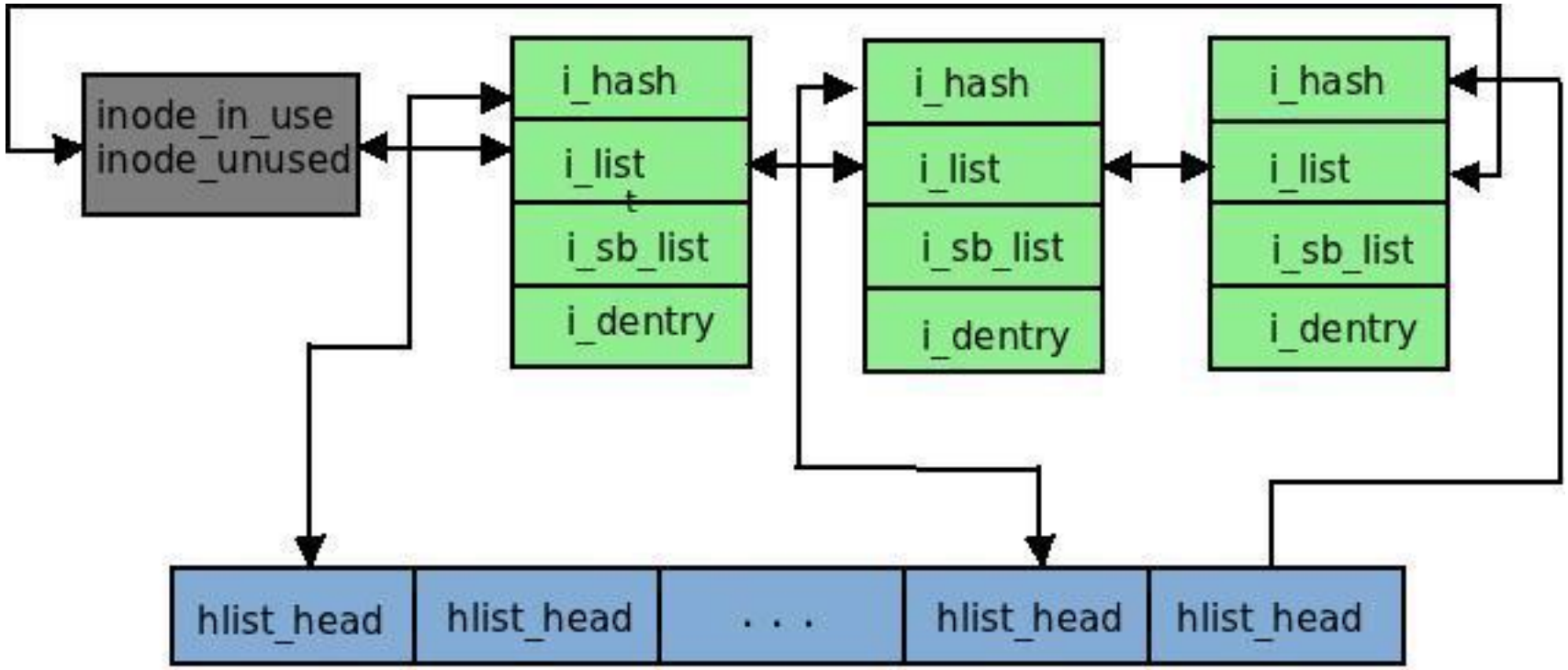
目录项缓存的查找

- 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(先从哈希表中查找，然后从LRU表中查找)



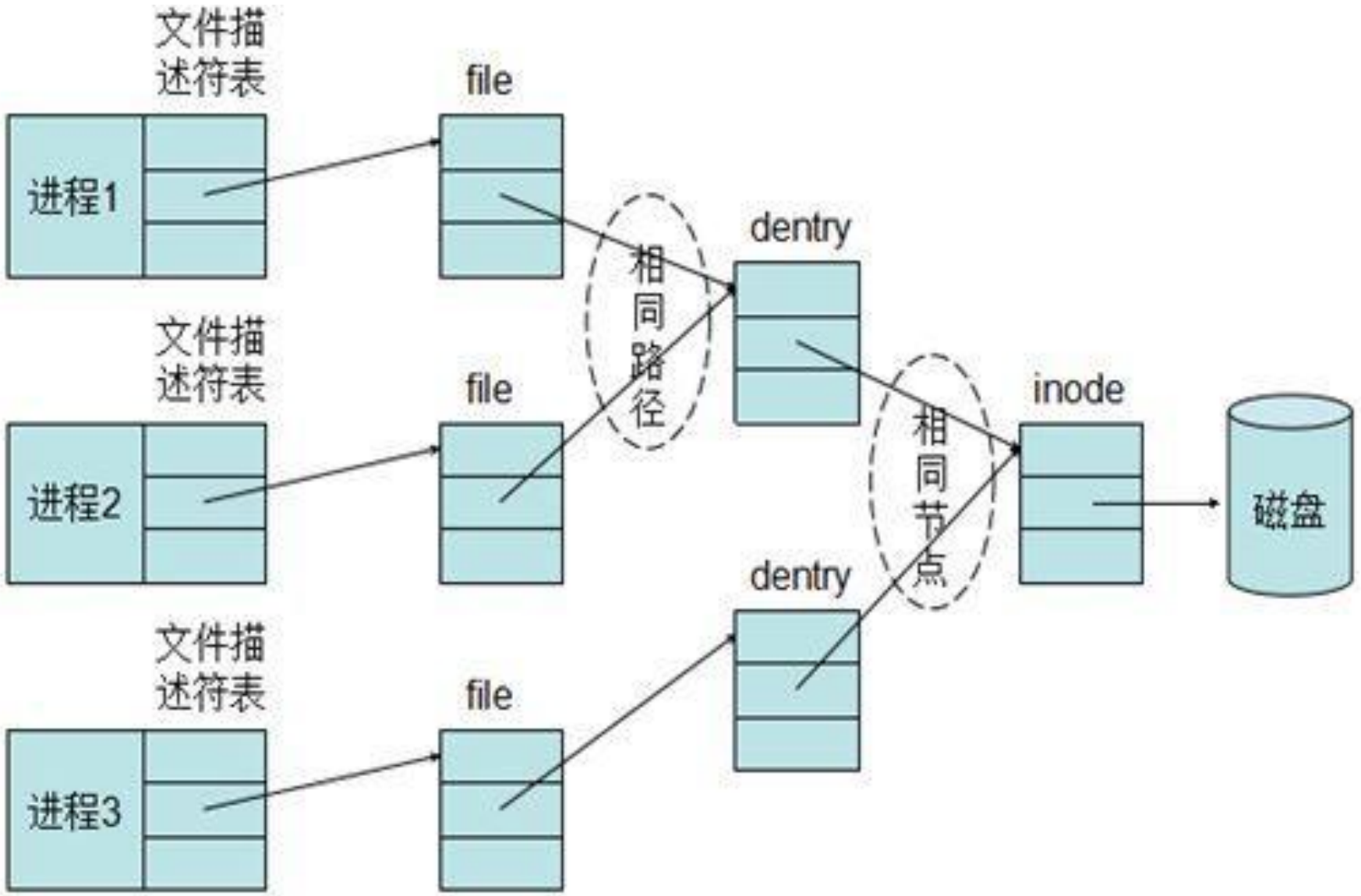
索引节点缓存的组织和查找

- 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。
- 如图所示。



buffer Cache组织

- Buffer cache 的组织：采用LRU链表(如图源码截图)
  - 在fs/buffer.c

```
#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

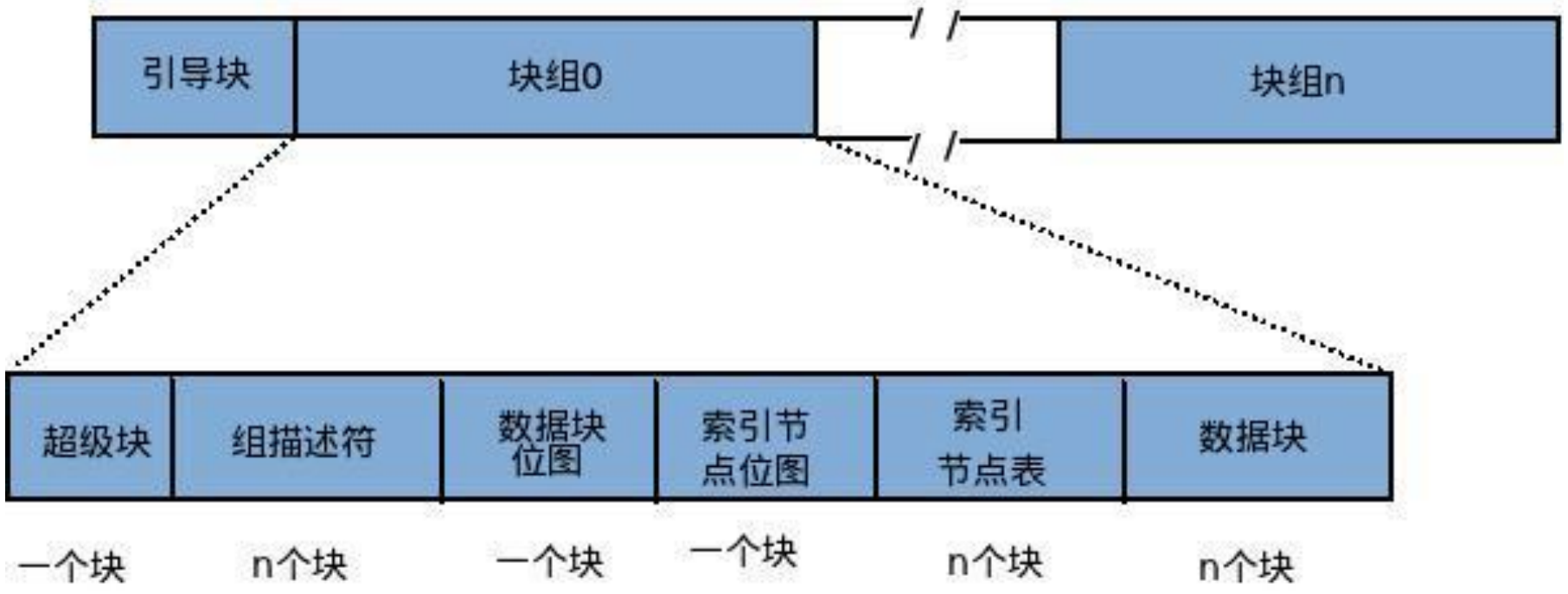
- 这个数据结构把内存中的页与磁盘中的块关联起来了，源码中对每个具体字段给予了解释。

```
/
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 */
};
```

在include/linux/buffer_head.h

Ext2文件系统超级块的组织形式

- 内核通过buffer cache获取inode之前，先看一下ext2文件系统超级块的组织：
- 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。



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。

源码在fs/ext2linux/inode.c

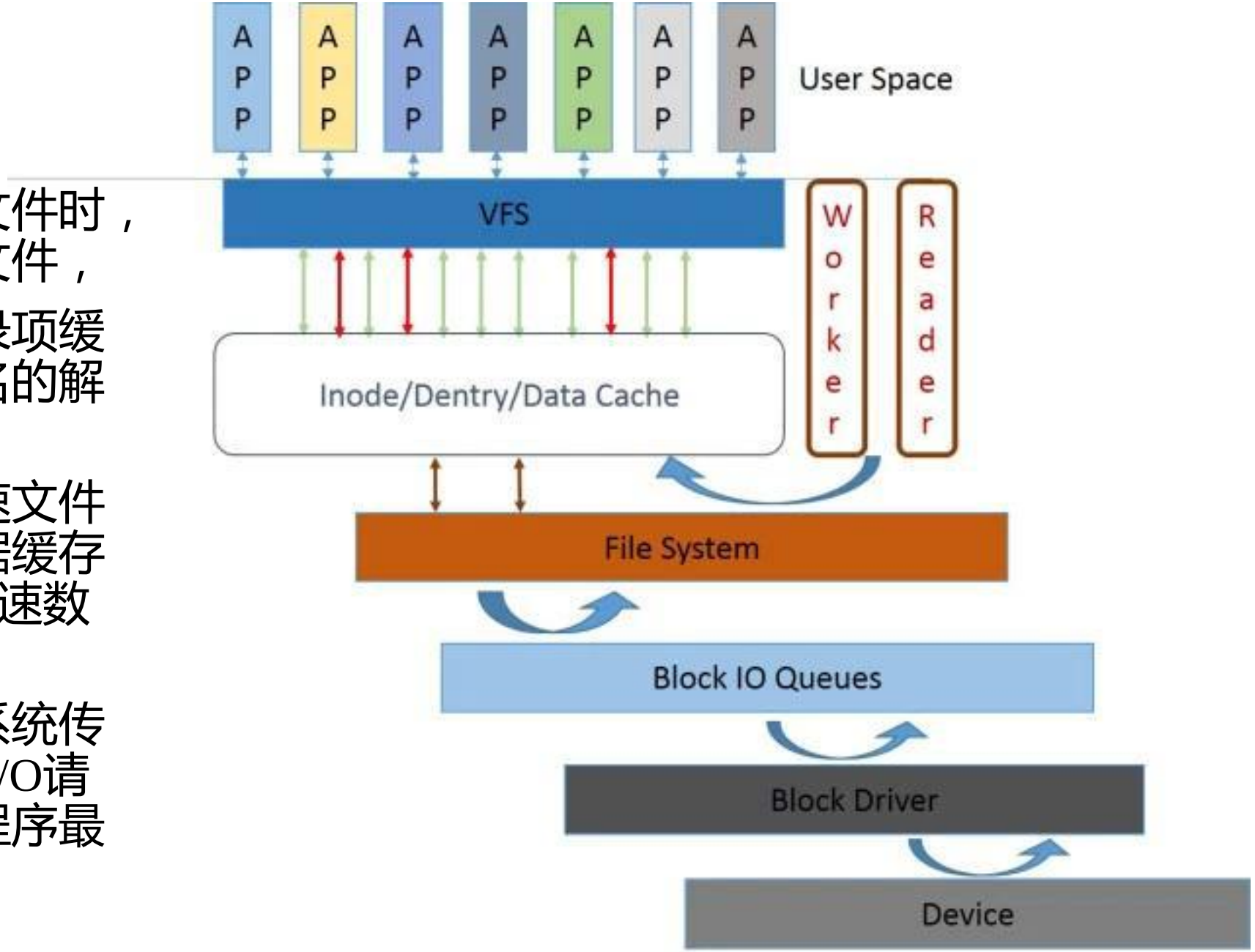
```
static struct ext2_inode *ext2_get_inode(struct super_block *sb, ino_t ino,
                                         struct buffer_head **p)
{
        struct buffer_head * bh;
        unsigned long block_group;
        unsigned long block;
        unsigned long offset;
        struct ext2_group_desc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

- 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

- 以上是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

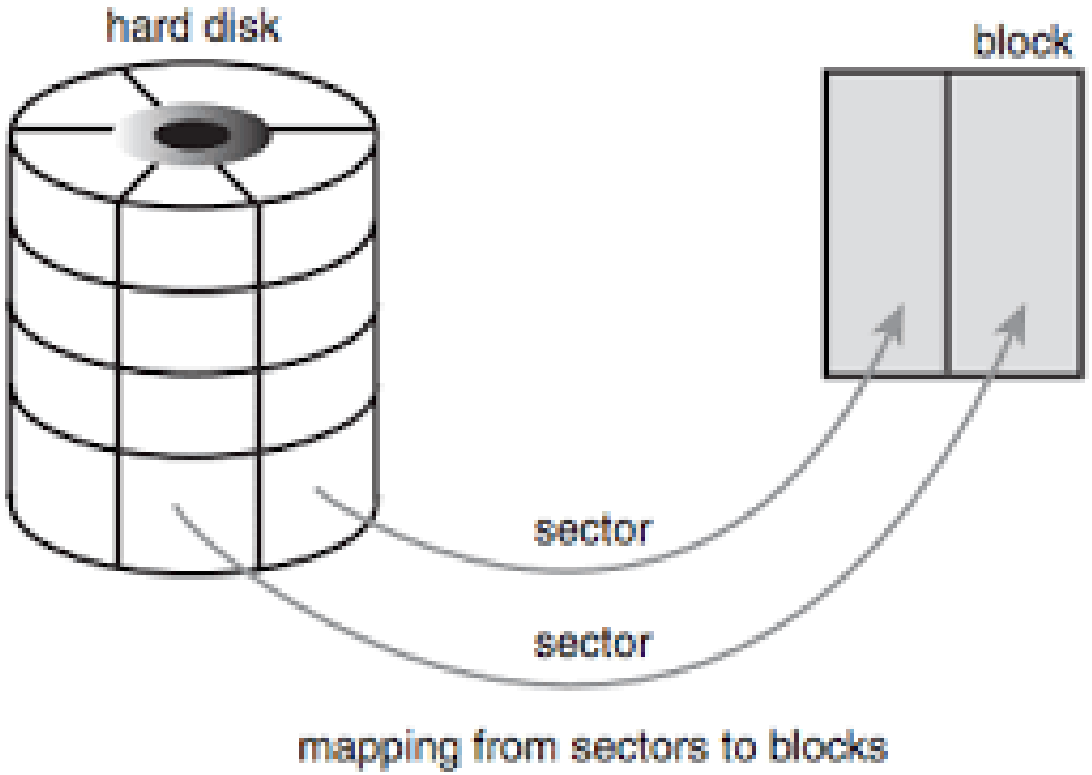
- 当应用程序打开一个文件时，首先是要查找到这个文件，
- 在查找的过程中，目录项缓存可以加速文件路径名的解析，
- 索引节点缓存可以加速文件元数据的查找，而数据缓存(也就是页缓存)可以加速数据的查找，
- 这些数据都通过文件系统传递给块I/O层，封装成I/O请求给驱动程序，驱动程序最终从设备上存取数据。



内容导航：

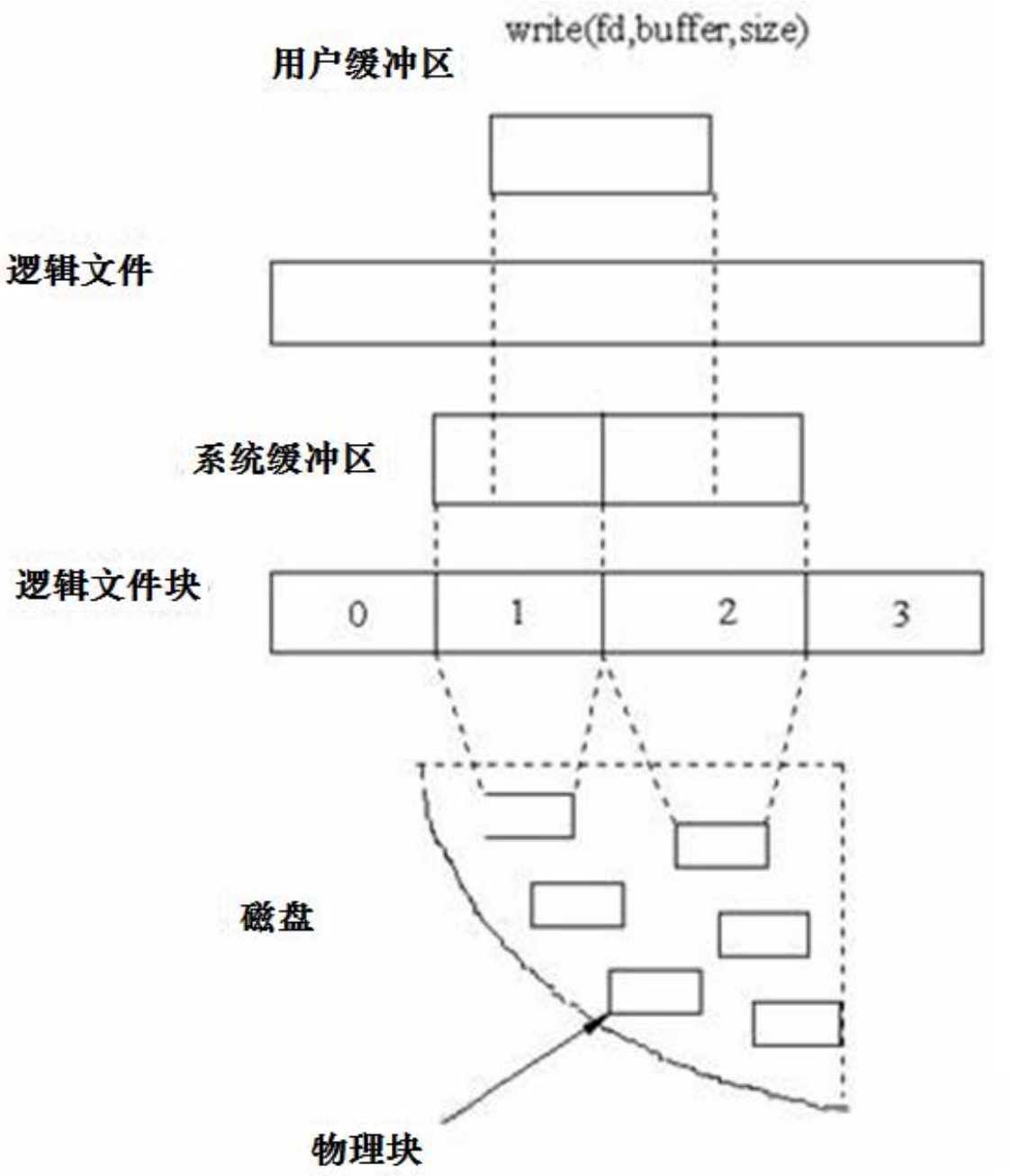
4 页高速缓存以及读写

文件系统的读写单位是什么



mapping from sectors to blocks

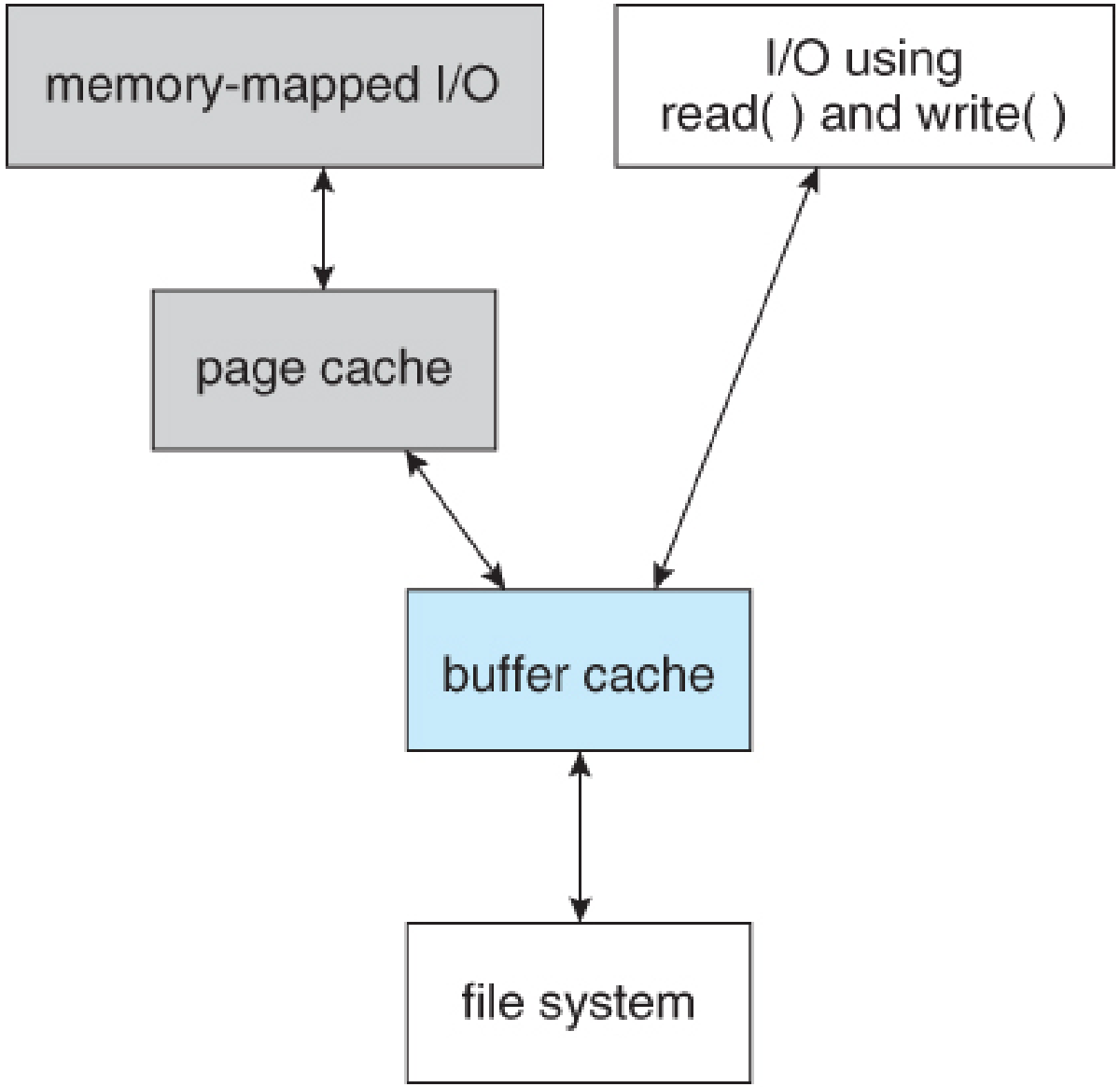
- 文件系统的读写单位是块，
- 一个块的大小是2的n次方个扇区，比如1k，2k, 4k, 4M 等



一个进程发出读写请求到读到数据

- 如图，当用户通过write系统调用给文件中写数据时，首先与用户态的I/O缓冲区打交道，然后逻辑文件中的写指针进行移动，真正的写入还没有发生，
- 然后陷入内核态，用户态缓冲区的数据被搬到内核缓冲区，这个缓冲区就是page cache，
- 如何把数据真正写入磁盘，我们继续讲解。

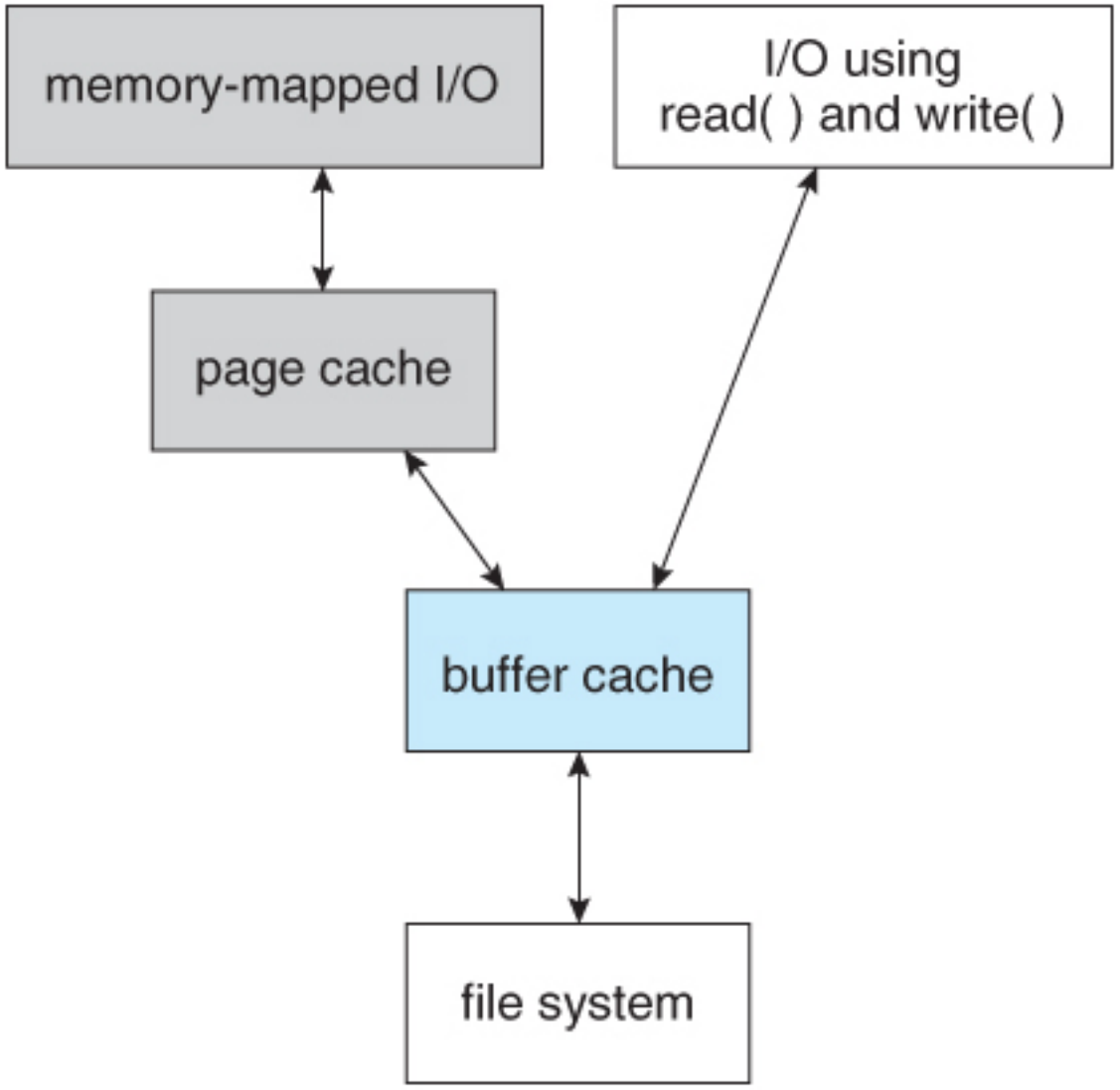
从Buffer Cache到 Page Cache



- buffer cache是面向底层块设备的，所以它的粒度是文件系统的**块**，块设备和系统采用块进行交互。
- 块再转换成磁盘的基本物理结构**扇区**。
- 扇区和块之间是可以快速转换的，这就涉及**页高速缓存**。

从Buffer Cache到 Page Cache

- 随着内核的功能越来越完善，块粒度的缓存已经不能满足性能的需要。
- 内核的内存管理组件采用了比文件系统的块更高级别的抽象-**页(page)**，页的大小一般从4KB到2MB，粒度更大，处理的性能更高。
- 所以缓存组件为了和内存管理组件更好地交互，，创建了页缓存page cache来代替原来的buffer cache。
- **页缓存是面向文件，面向内存的。**
- 通过一系列的数据结构，比如inode, address_space, page，将一个文件映射到页的级别，通过page + offset就可以定位到一个文件的具体位置。如图所示。



描述页缓存的address_space对象

- 其中，第一个字段是host, 每一个所有者可以理解为一个具体的文件，也就是一个inode指向的文件，它对应着一个address_space对象，页高速缓存的多个页可能属于一个所有者，从而可以链接到一个address_space对象。
- 那么一个页(page)怎么和一个address_space产生关联的呢？

```
struct address_space {
        struct inode            *host;          /* 所有者：inode，或块设备 */
        struct radix_tree_root  page_tree;      /* 所有页的基数树 */
        unsigned int            i_mmap_writable;/* VM_SHARED映射的计数 */
        struct prio_tree_root   i_mmap;         /* 私有和共享映射的树 */
        struct list_head        i_mmap_nonlinear;/* VM_NONLINEAR映射的链表元素 */
        unsigned long       wbri_banrpages;     /* 页的总数 */
        pgoff_t                 writeback_index;/* 回写由此开始 */
        struct address_space_operations *a_ops; /* 方法，即地址空间操作 */
        unsigned long           flags;          /* 错误标志位/gfp掩码 */
        struct backing_dev_info *backing_dev_info; /* 设备预读 */
        struct list_head        private_list;
```

在include/linux/fs.h

page对象

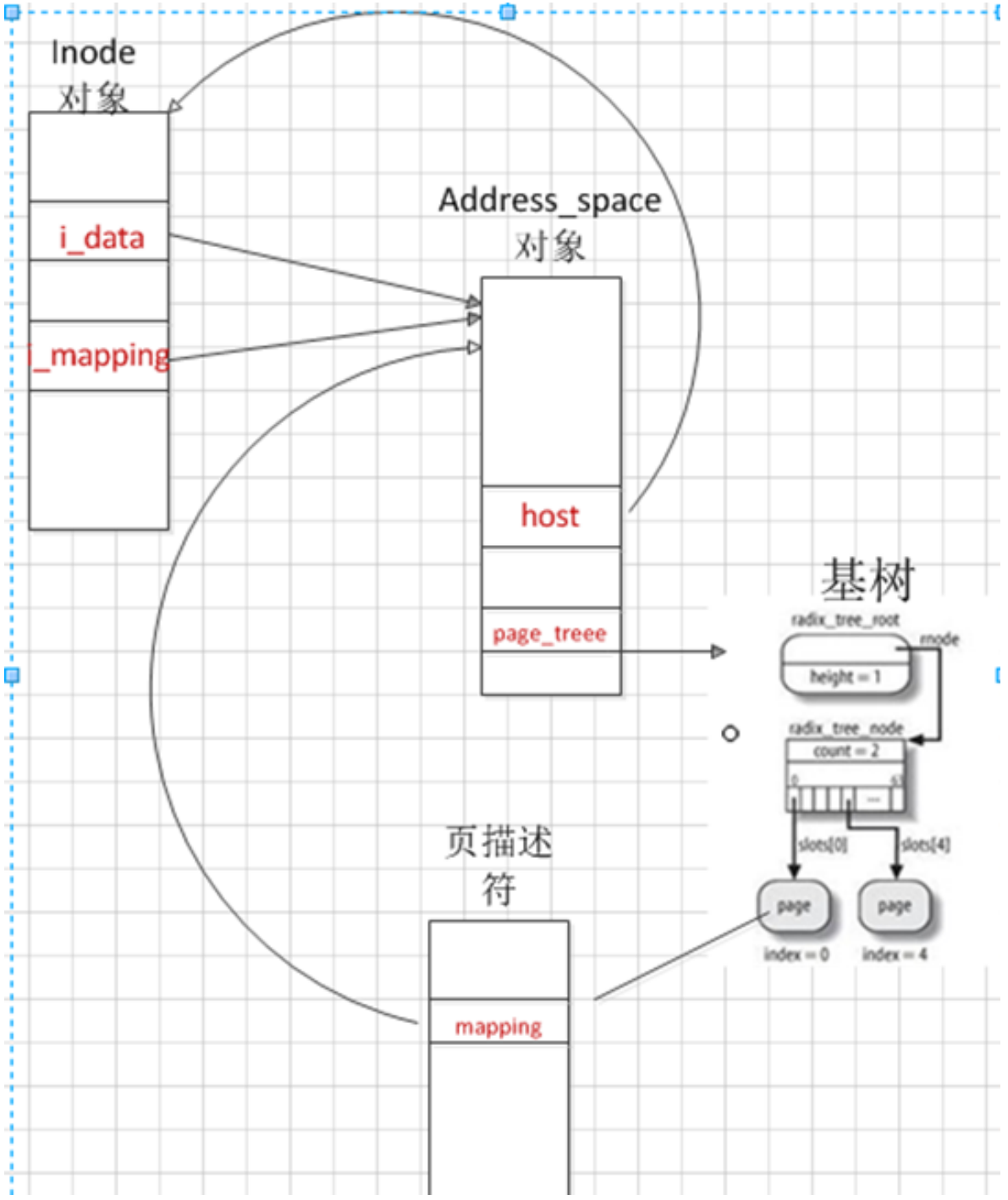
- page中有两个字段：mapping和index。其中mapping指向该页所有者的address_space，index字段表示所有者地址空间中，以页大小为单位的偏移量。
- 用这两个字段就能在页高速缓存中查找。

```
struct page {
        unsigned long          flags;
        atomic_t               _count;
        atomic_t               _mapcount;
        unsigned long          private;
        struct address_space   *mapping;
        pgoff_t                index;
        struct list_head       lru;
        void                   *virtual;
};
```

在include/linux/mm_types.h

索引节点、页和页缓存之间的关系

- (1)一个inode节点对象对应一个address_space对象。其中inode节点对象的i_mapping和i_data字段，指向相应的address_space对象，而address_space对象的host字段，指向对应的inode节点对象。
- (2)每个address_space对象对应一颗基树。他们之间的联系是通过address_space对象中的page_tree字段，指向该address_space对象对应的基树。
- (3)一般情况下一个inode节点对象对应的文件，或者是块设备都会包含多个页面的内容，所以一个inode对象对应多个page描述符。同一个文件拥有的所有page描述符，都可以在该文件对应的基树中找到。它们之间的关系如图所示。

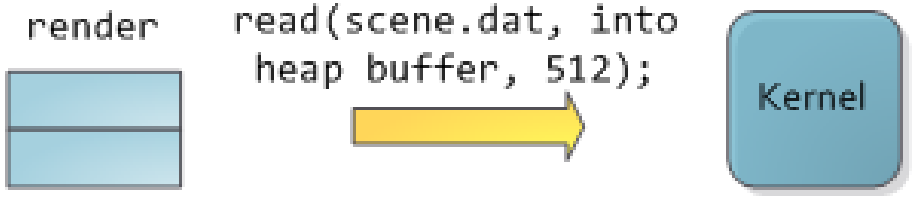


如何读取一个文件？

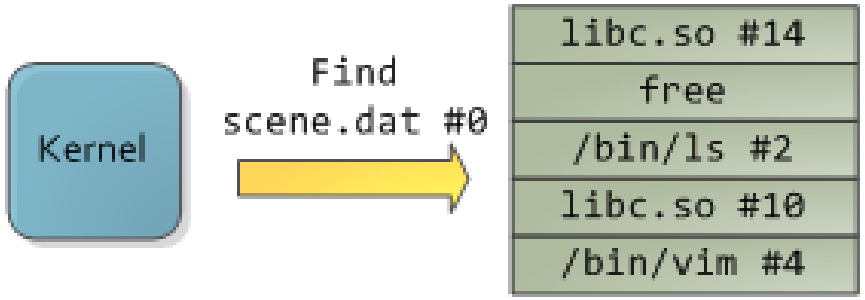
- 如后图所示，假设一个进程reader要读取一个my.dat文件，实际发生的步骤如下：
- 1. reader进程向内核发起读my.dat文件的请求
- 2. 内核根据my.dat的inode找到对应的address_space，在address_space中查找页缓存，如果没有找到，那么分配一个内存页page加入到页缓存
- 3. 从磁盘中读取my.dat文件相应的页，填充页缓存中的页，也就是第一次复制
- 4. 从页缓存的页，复制内容到reader进程的堆空间的内存中，也就是第二次复制

如何读取一个文件？

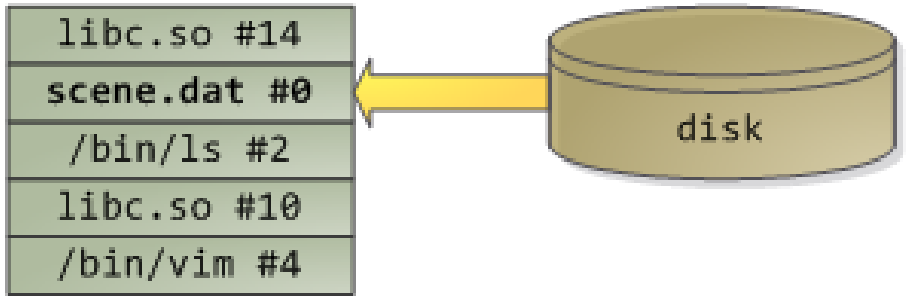
1. Render asks for 512 bytes of scene.dat starting at offset 0.



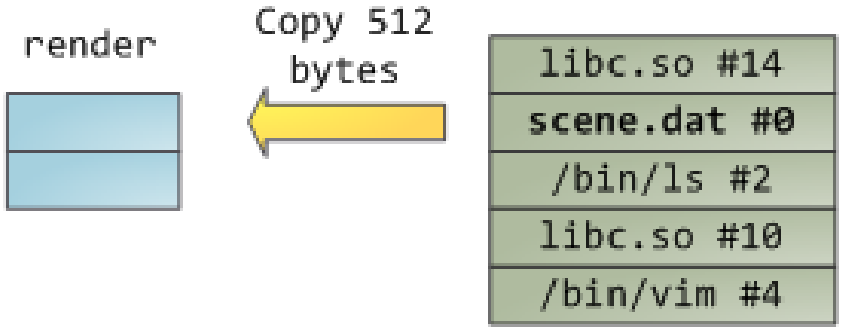
2. Kernel searches the page cache for the 4KB chunk of scene.dat satisfying the request. Suppose the data is not cached.



3. Kernel allocates page frame, initiates I/O requests for 4KB of scene.dat starting at offset 0 to be copied to allocated page frame

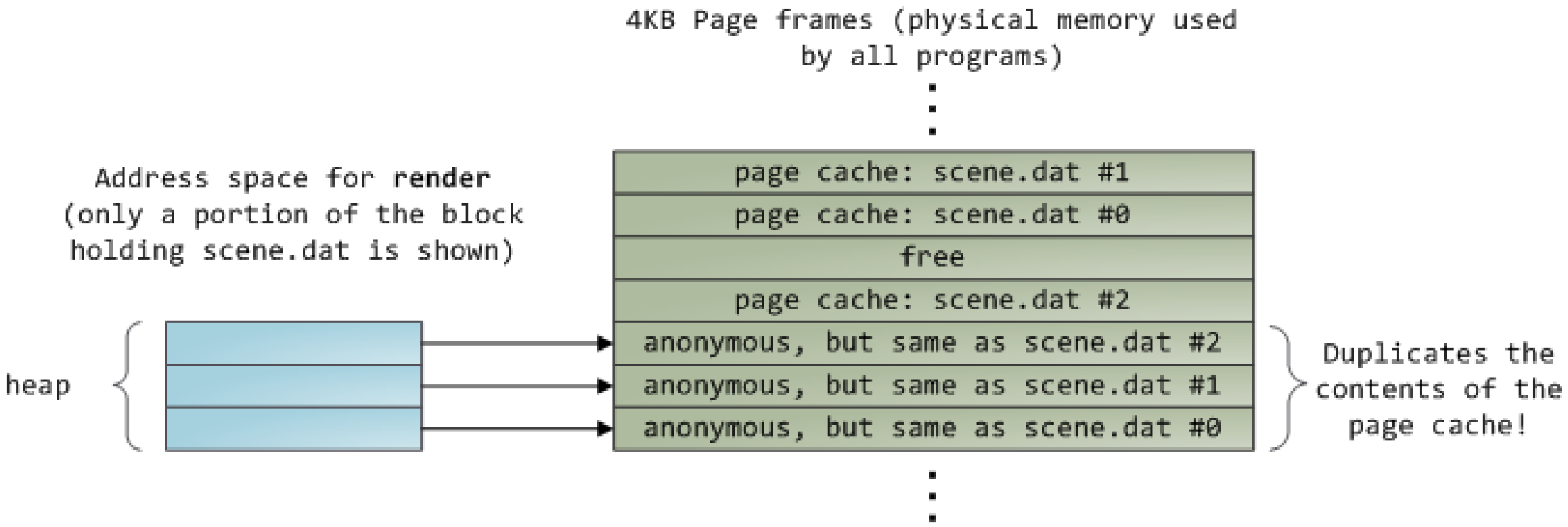


4. Kernel copies the requested 512 bytes from page cache to user buffer, read() system call ends.



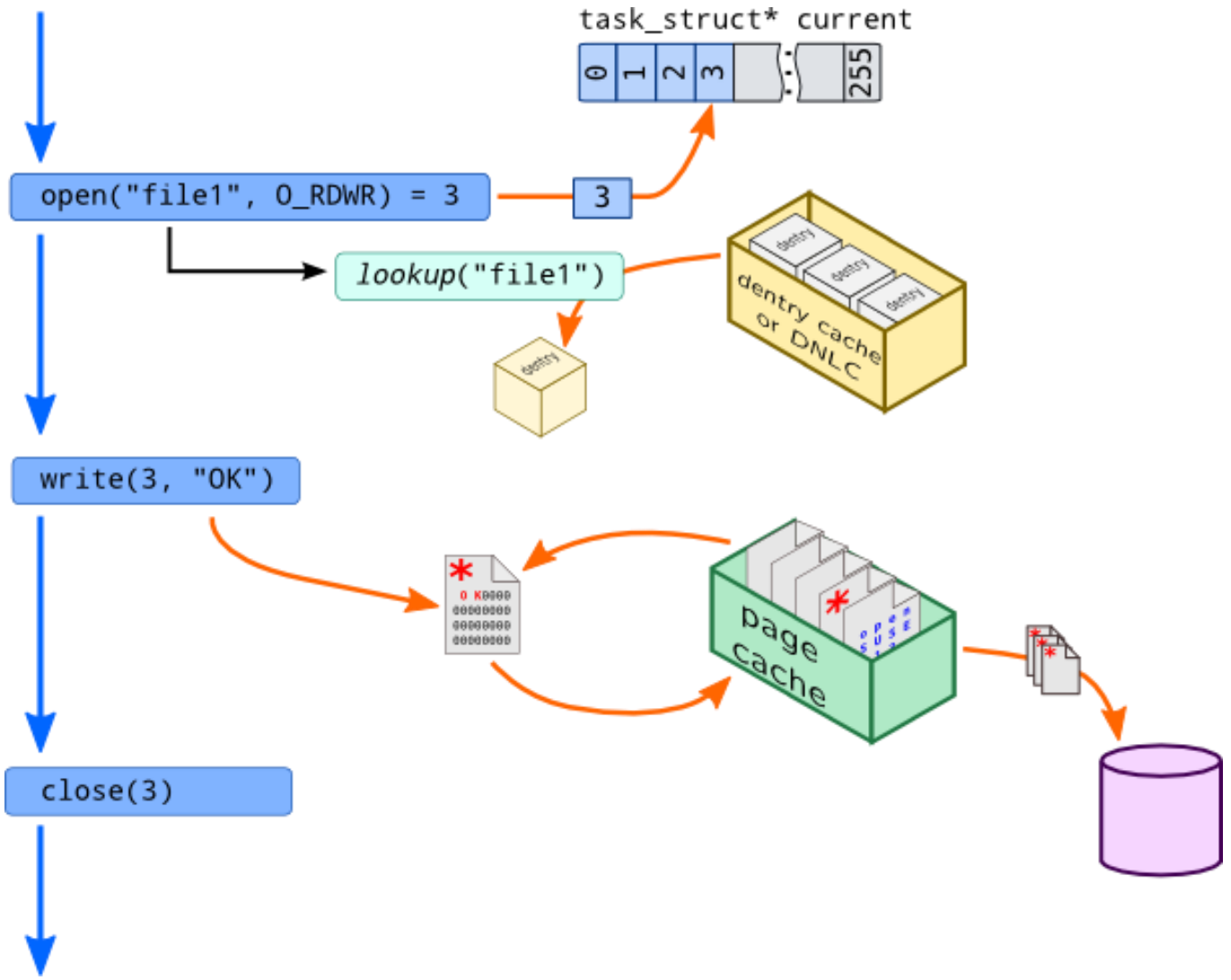
如何读取一个文件？

- 最后物理内存的内容是这样的，同一个文件my.dat的内容存在了两份拷贝，一份是页缓存，一份是用户进程的堆空间对应的物理内存空间。



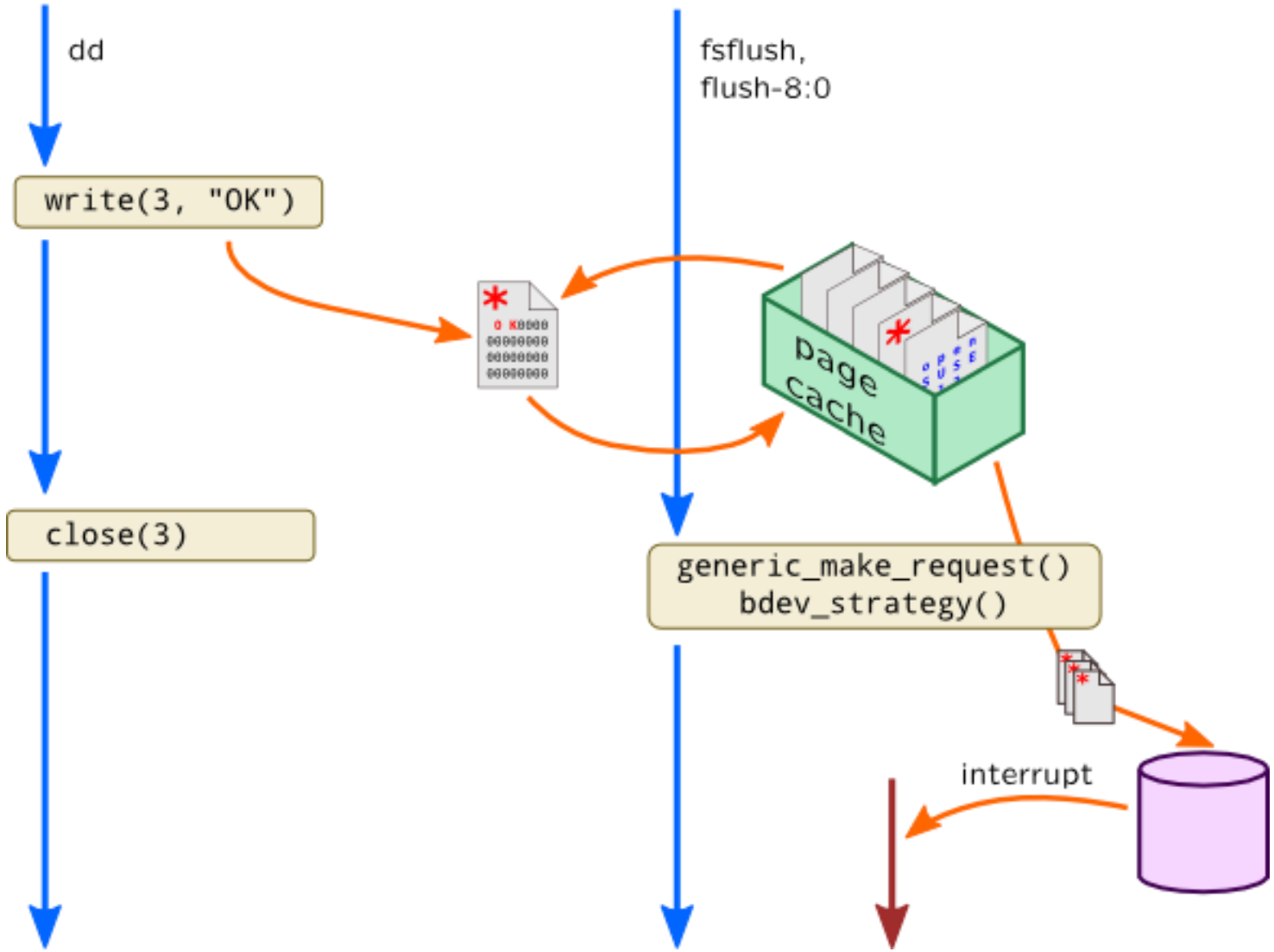
如何写入文件

- 从内核的角度来看，写文件对应的系统调用服务例程为sys_write。
- 与读取的函数处理流程相似，只是将相应函数中的read改为了write。
- 对于写操作，Linux同样采用了缓存技术。
- 但这里的缓存跟文件读取用到的有些不一样。



脏页写回

- 对于文件的写入，考虑到一页数据页中，可能只是某一块数据发生改变，
- 这里写的过程结合了块缓冲技术，也就是如果发现了脏页，只是发生写入的脏块回写到磁盘即可。
- 如图所示。



课程思政

课程思政

增强集体意识和团队精神

国内Linux系统的发展与国际上Linux系统的市场占有情况密切相关。

随着大数据与云计算等前沿技术的快速发展，越来越多的互联网公司开始构建自主控制与维护的云计算平台。具有开源与跨平台等属性的Linux系统，搭配采用Arm64芯片的计算平台，成为了这些互联网公司的首选技术方案。与此同时，Linux服务器端解决方案通过互联网企业迅速应用到了大数据与云计算的市场环境中。

但是，互联网企业使用Linux服务器端时，并未因采用Linux系统而形成典型的操作系统销售市场，专业的Linux系统厂商在服务器市场中还未形成较大的市场影响力。

增强集体意识和团队精神

目前，国内海量的应用软件都是基于Windows系统的，因为该系统用户学习成本低、熟练程度高；而针对Linux系统，存在用户熟练程度低、对专业技术支持团队的依赖程度高、使用和维护成本高等问题。

为了更好、更快地解决上述问题，亟须建立顺畅的产品服务情况与用户使用预期的沟通渠道，通过了解并满足用户针对操作系统在使用、维护等方面的多种需求，提升国产（基于Linux系统进行二次开发的）操作系统的整体性能。同时，亟须确定一个兼具稳定性和一致性的开发接口，以使开发Linux系统应用软件的代码可以跨平台落地，形成基于较为成熟的国产操作系统的产业氛围。

操作系统的自主可控是网络强国的关键基石，在操作系统国产化的过程中，我们一定要增强集体意识和团队精神，努力奋进。

课后练习题

1、虚拟文件系统（Virtual File System，VFS）有什么作用，请简述。
2、Linux目前支持几十种文件系统类型，请举例说明几种最常见的文件系统类型。
3、请简述扇区（sector）、块（block）的概念，以及它们之间的关系。
4、 Linux文件系统的索引节点是什么，请简述。
5、请对超级块数据结构进行简述。
6、Linux文件系统中引入目录项的概念主要出于什么目的？
7、请简述现代操作系统中的缓冲区（buffer）技术。
8、请简述buffer和cache有何不同？
9、请对一个文件读取的过程做一个简述？

谢 谢！

THANKS