Linux铁三角之I/O(二)

vomocode阅码场

麦当劳喜欢您来，喜欢您再来

扫描关注

Linux阅码场

Vomocode阅码场

文件系统的架构

*一切都是文件：VFS
*字符设备文件、块设备文件
*超级块、目录、inode
*符号链接与硬链接
*目录的组织
*icache和dcache，slab shrink
*块映射
*发现并读取/usr/bin/xxx的全流程
*用户空间的文件系统:FUSE

vomocode阅码场

VFS

C++ virtual func() = 0

read write ioctl open...

VFS

char设备 Block RAW设备 Ext4中的file

各种字符设备驱动

```
static const struct file_operations lp_fops = {
        .owner          = THIS_MODULE,
        .write          = lp_write,
        .unlocked_ioctl = lp_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl   = lp_compat_ioctl,
#endif
        .open           = lp_open,
        .release        = lp_release,
```

fs/block_dev.c

```
const struct file_operations def_blk_fops = {
        .open           = blkdev_open,
        .release        = blkdev_close,
        .llseek         = block_llseek,
        .read           = new_sync_read,
        .write          = new_sync_write,
```

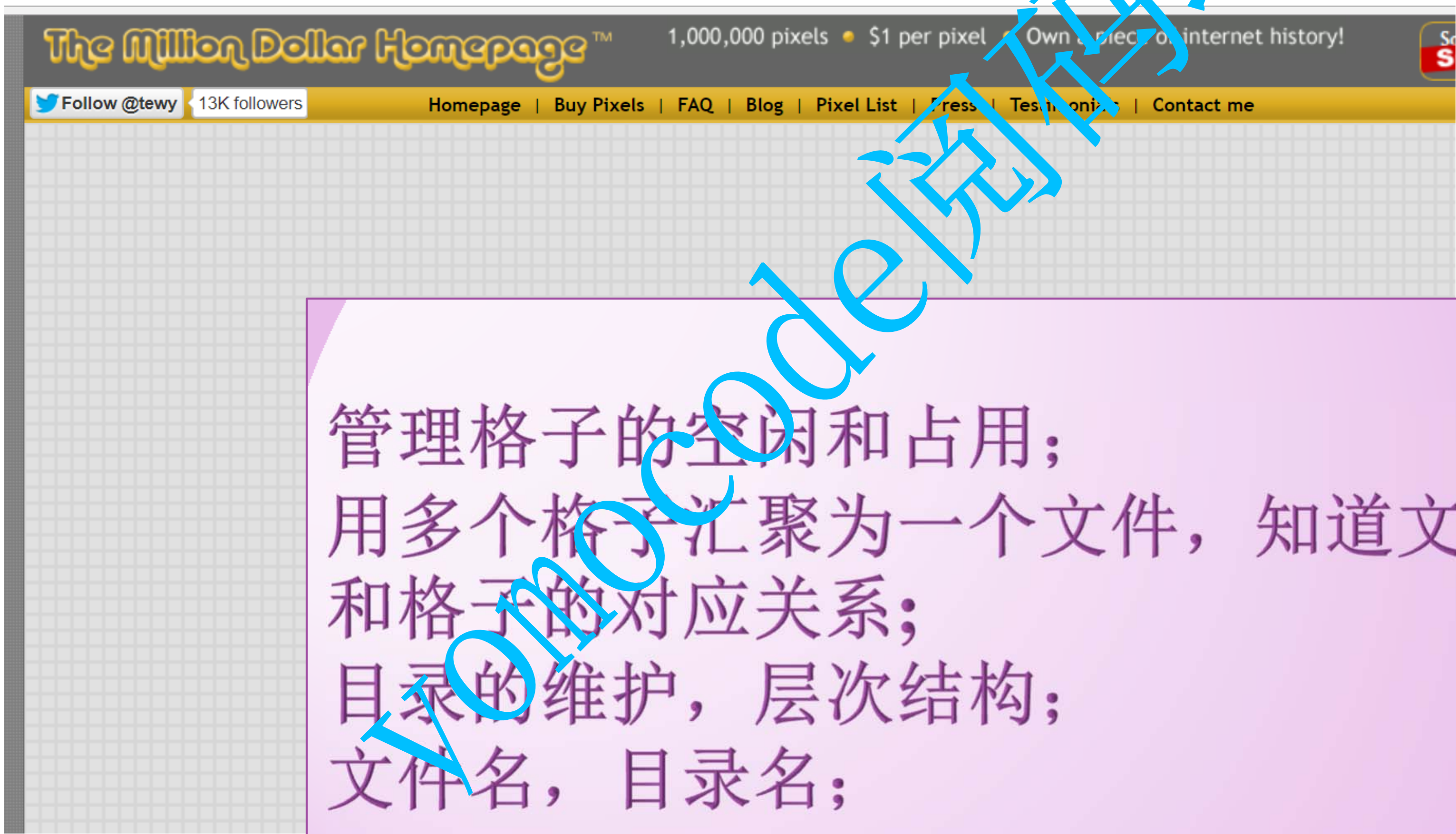
fs/ext4/file.c

```
const struct file_operations ext4_file_operations = {
        .llseek         = ext4_llseek,
        .read           = new_sync_read,
        .write          = new_sync_write,
        .read_iter      = generic_file_read_iter,
        .write_iter     = ext4_file_write_iter,
        .unlocked_ioctl = ext4_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl   = ext4_compat_ioctl,
#endif
        .mmap           = ext4_file_mmap,
```

Vomocode阅码场

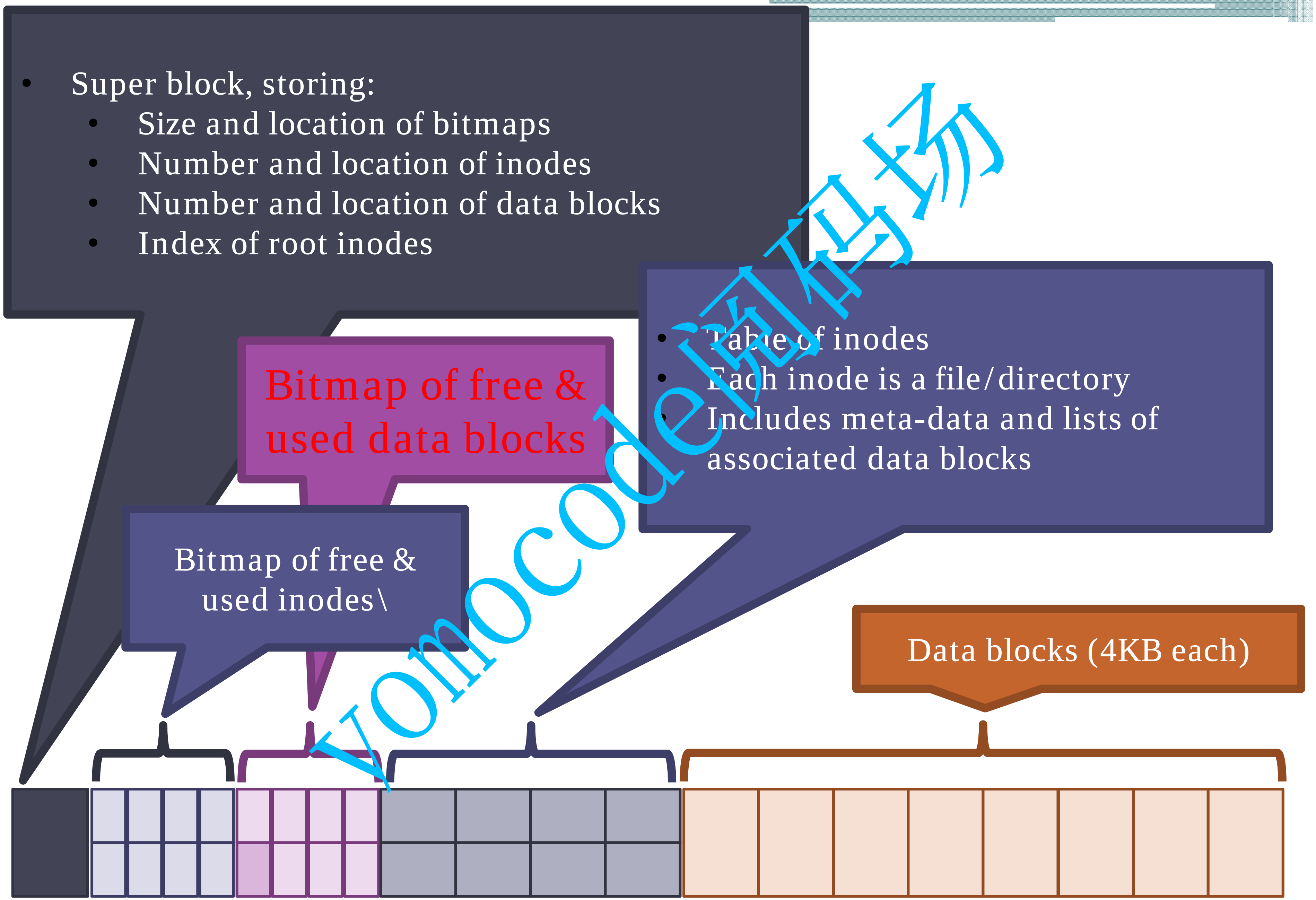
文件系统应该做什么？
——百万格子和文件系统的故事

milliondollarhomepage.com



管理格子的空闲和占用；
用多个格子汇聚为一个文件，知道文件和格子的对应关系；
目录的维护，层次结构；
文件名，目录名；
….

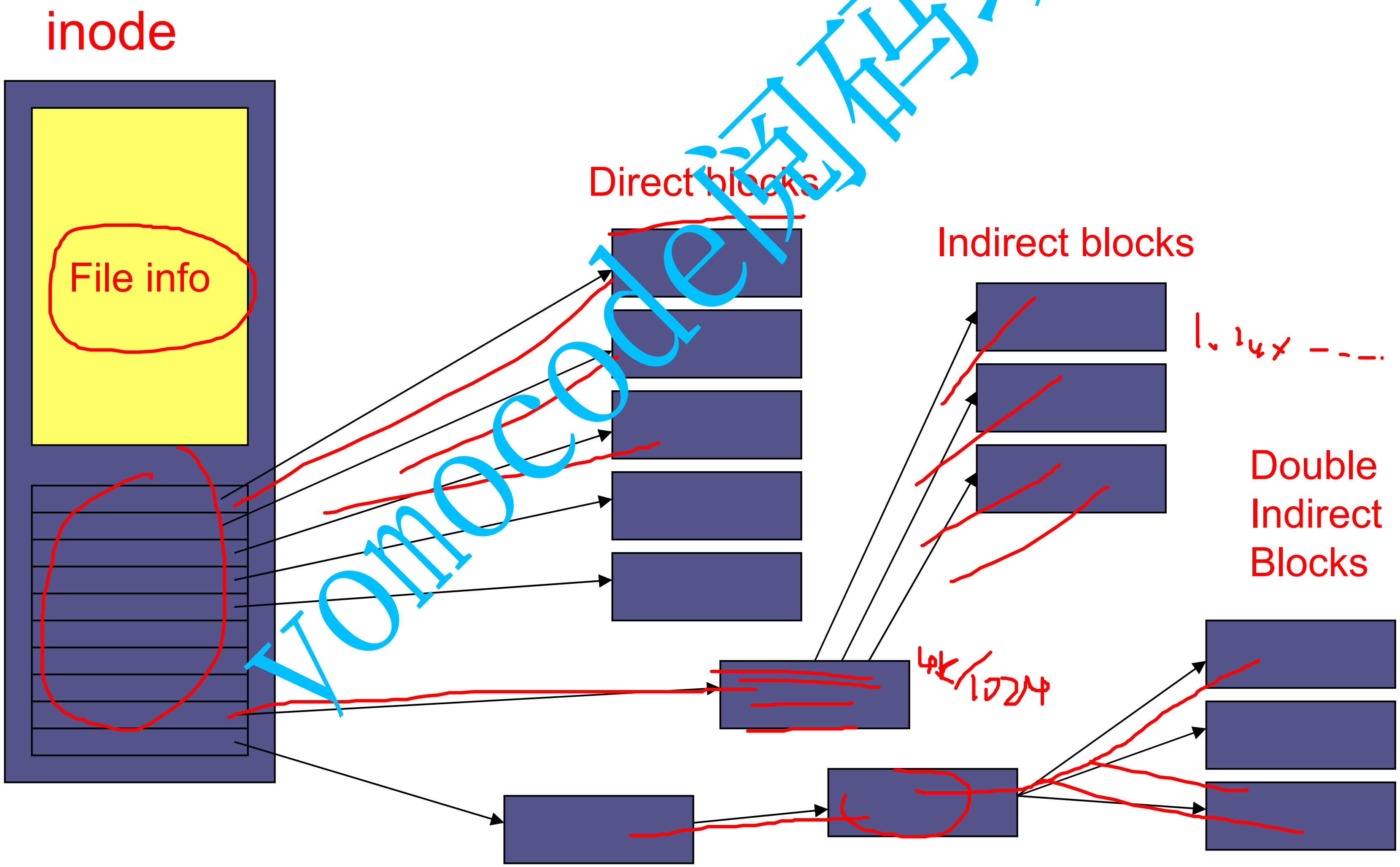
vonocode阅码场



vomocode阅码场

Inode diagram

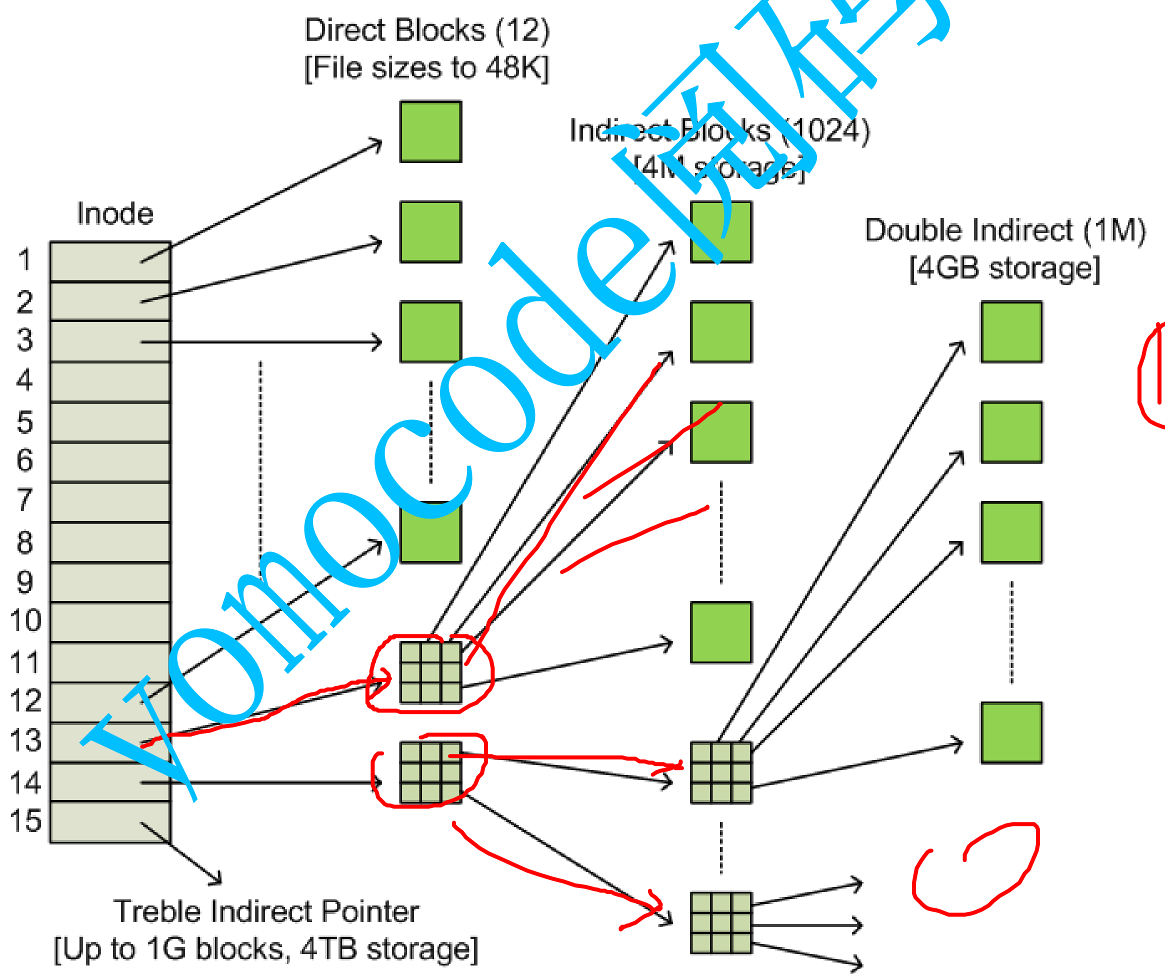
Inode diagram需要描述文件存放在磁盘的哪些块(block)



vomocode阅码场

Indirect blocks

Inode diagram不足以描述足够多的磁盘块，因此可以透过间接指向的方法，支持大文件。



1m

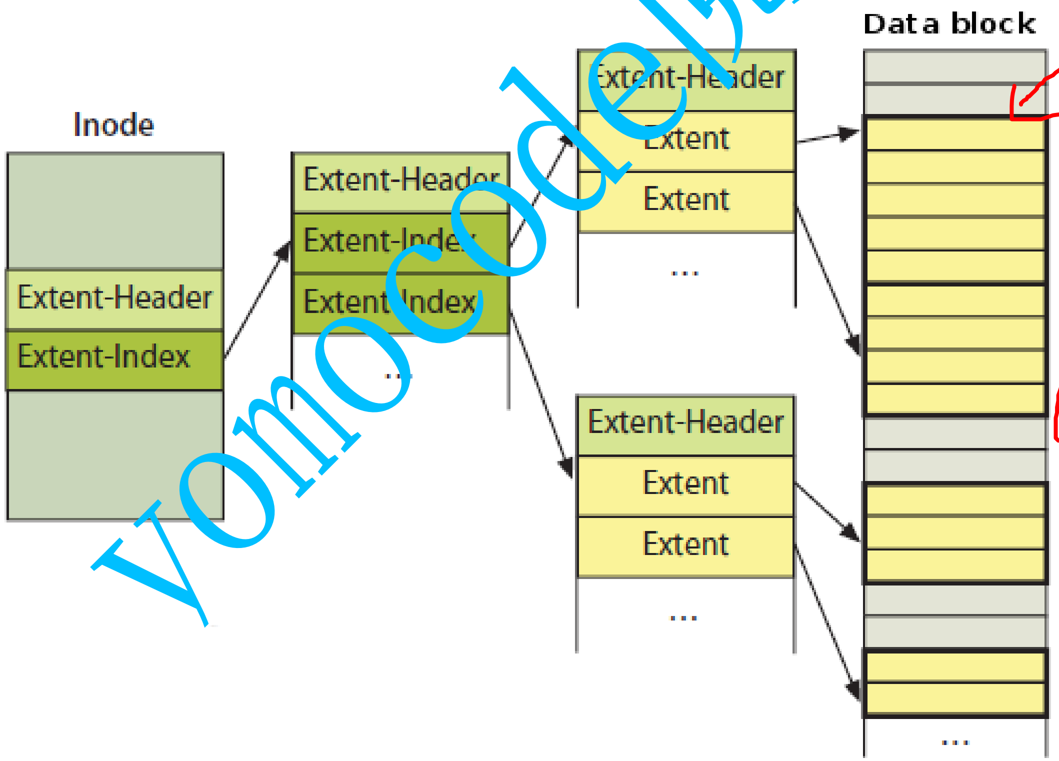
4K

vomocode阅码场

EXT4 Extents

一个Extents是一个地址连续的数据块(block)的集合。比如一个100MB的文件有可能被分配给一个单独的Extents，这样就不用像Ext3那样新增25600个数据块的记录（假设一个数据块是4KB）

vomocode阅码场



xxx_inode_cache

更低层次的cache，描述了inode diagram里面的东西，与硬盘联系更加紧密

```
static int __init init_inodecache(void)
{
        ext3_inode_cachep = kmem_cache_create("ext3_inode_cache",

        if (ext3_inode_c
                return -
        return 0;
}
```

```
/*
 * third extended file system inode data in memory
 */
struct ext3_inode_info {
        __le32  i_data[15];     /* unconverted */
        __u32   i_flags;
#ifdef EXT3_FRAGMENTS
        __u32   i_faddr;
        __u8    i_frag_no;
        __u8    i_frag_size;
#endif
        ext3_fsblk_t    i_file_acl;
        __u32   i_dir_acl;
        __u32   i_dtime;

        /*
         * i_block_group is the number of the block group which contains
         * this file's inode.  Constant across the lifetime of the inode,
         * it is ued for making block allocation decisions - we try to
         * place a file's data blocks near its inode block, and new inodes
         * near to their parent directory's inode.
         */
        __u32   i_block_group;
        unsigned long   i_state_flags;  /* Dynamic state flags for ext3 */
```

slab

~~kmalloc~~

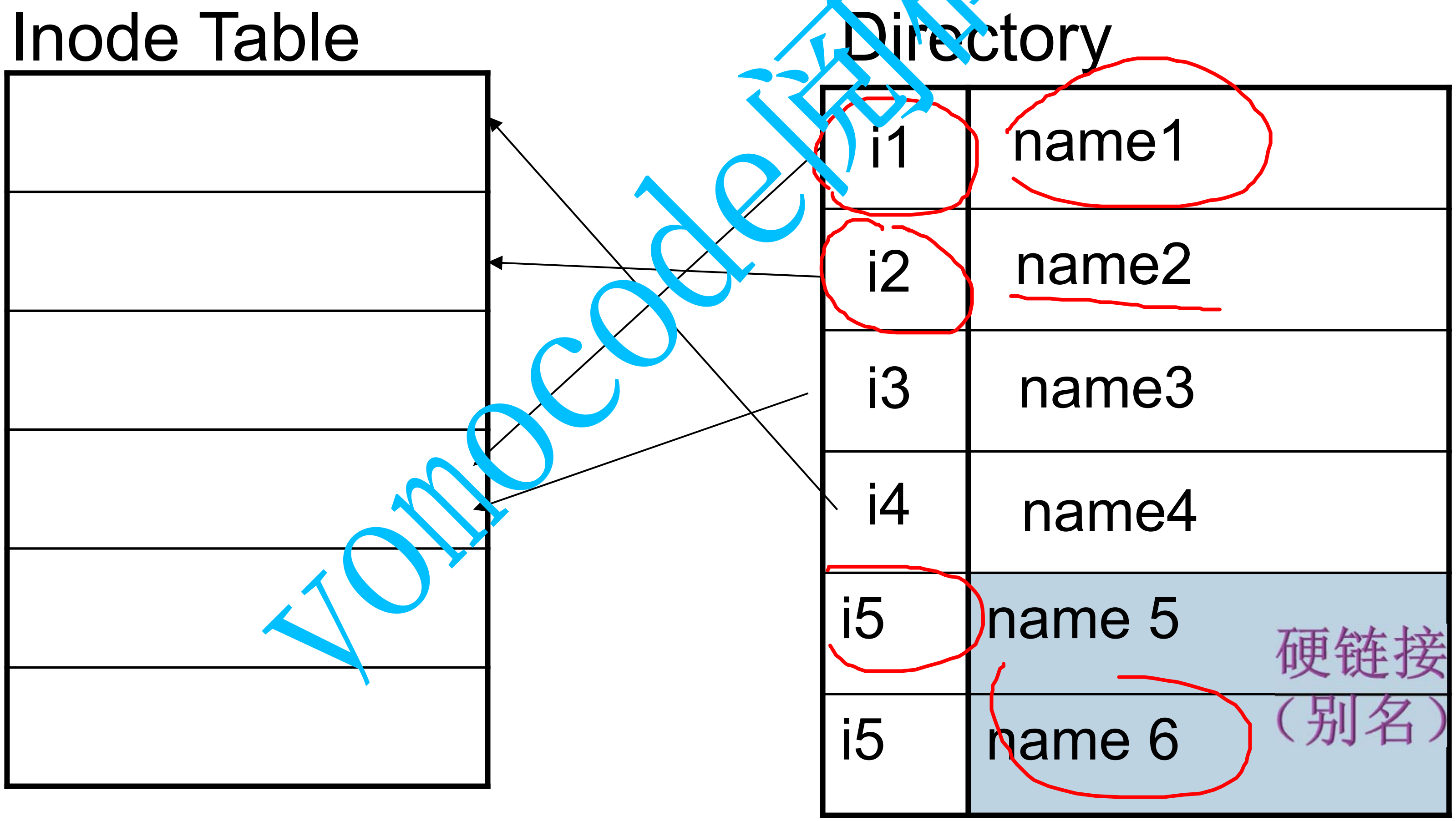
kfree

kmalloc

vomocode阅码场

Directory diagram

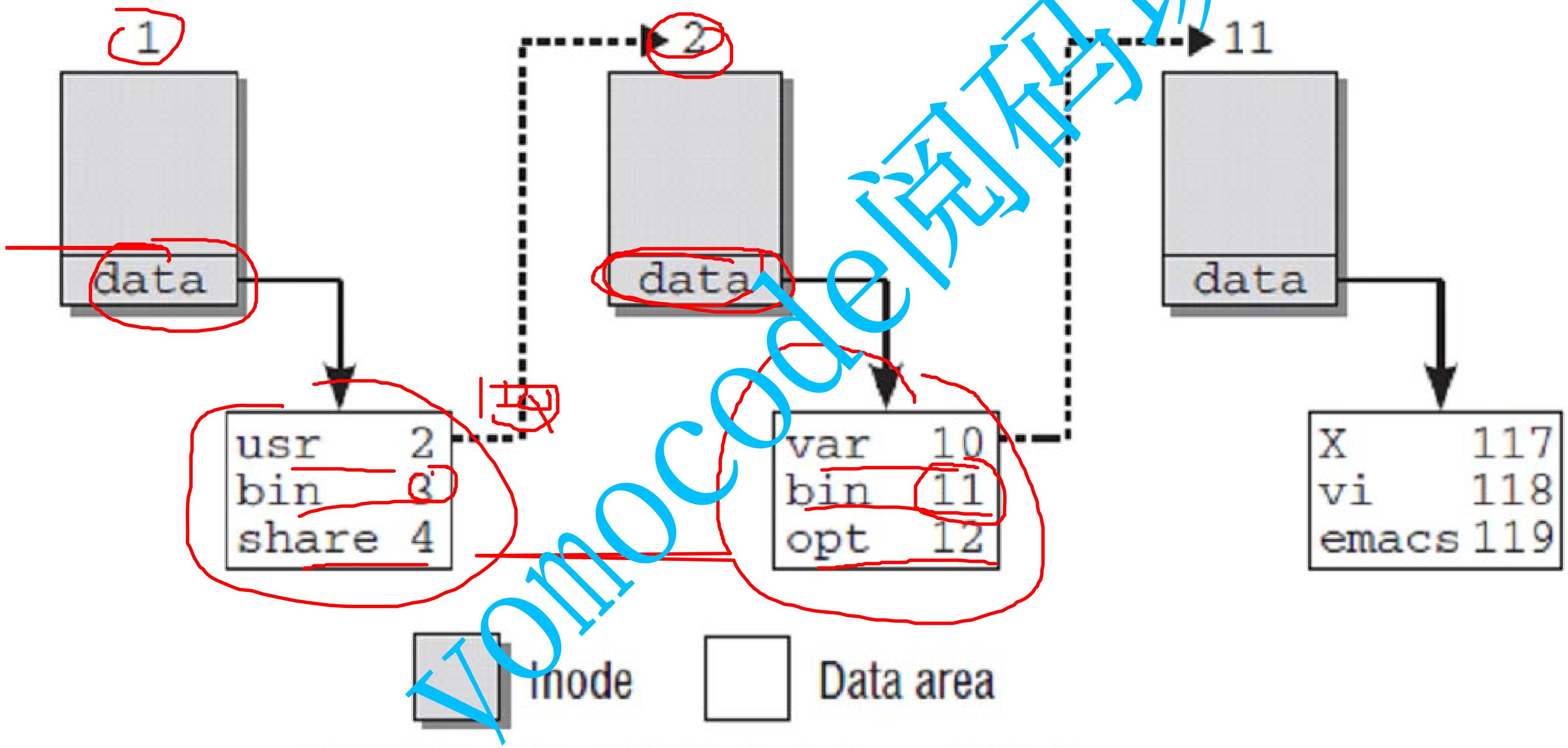
目录是一个特殊的文件，是inode号与名字的映射表



vomocode阅码场

Lookup /usr/bin/emacs

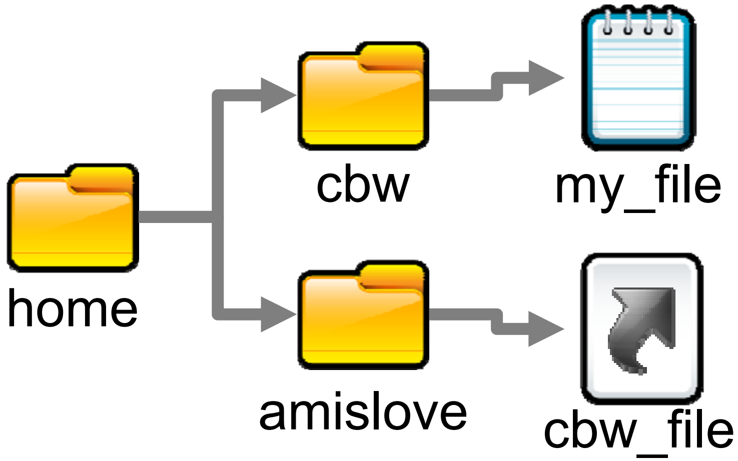
/usr/bin/emacs-ln



Lookup operation for /usr/bin/emacs.

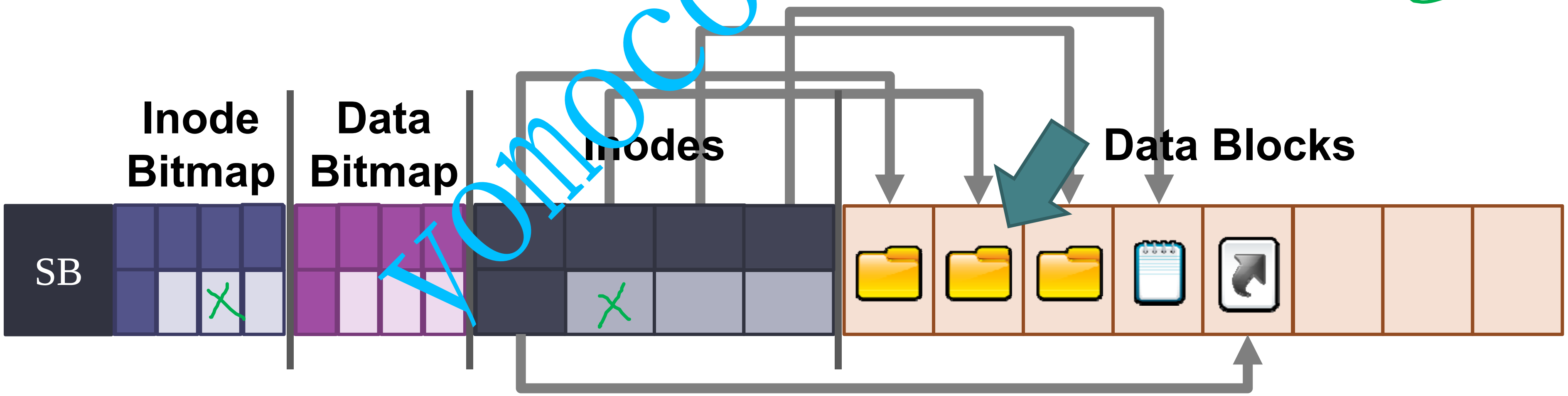
Vomocode阅码场

符号链接




```
[amislove@ativ9 ~] ln –s ../cbw/my_file
cbw_file
```

1. Create a soft link file
2. Add it to the current directory



vomocode闭码场

Icache和dcache

```
void __init inode_init(void)
{
        unsigned int loop;

        /* inode slab cache */
        inode_cachep = kmem_cache_create("inode_cache",
                                         sizeof(struct inode),
                                         0,
                                         (SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|
                                         SLAB_MEM_SPREAD),
                                         init_once);
```

```
static void __init dcache_init(void)
{
        unsigned int loop;

        /*
         * A constructor could be added for stable state like the lists,
         * but it is probably not worth it because of the cache nature
         * of the dcache.
         */
        dentry_cache = KMEM_CACHE(dentry,
                SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|SLAB_MEM_SPREAD);
```

Vomocode阅码场

Slab shrink

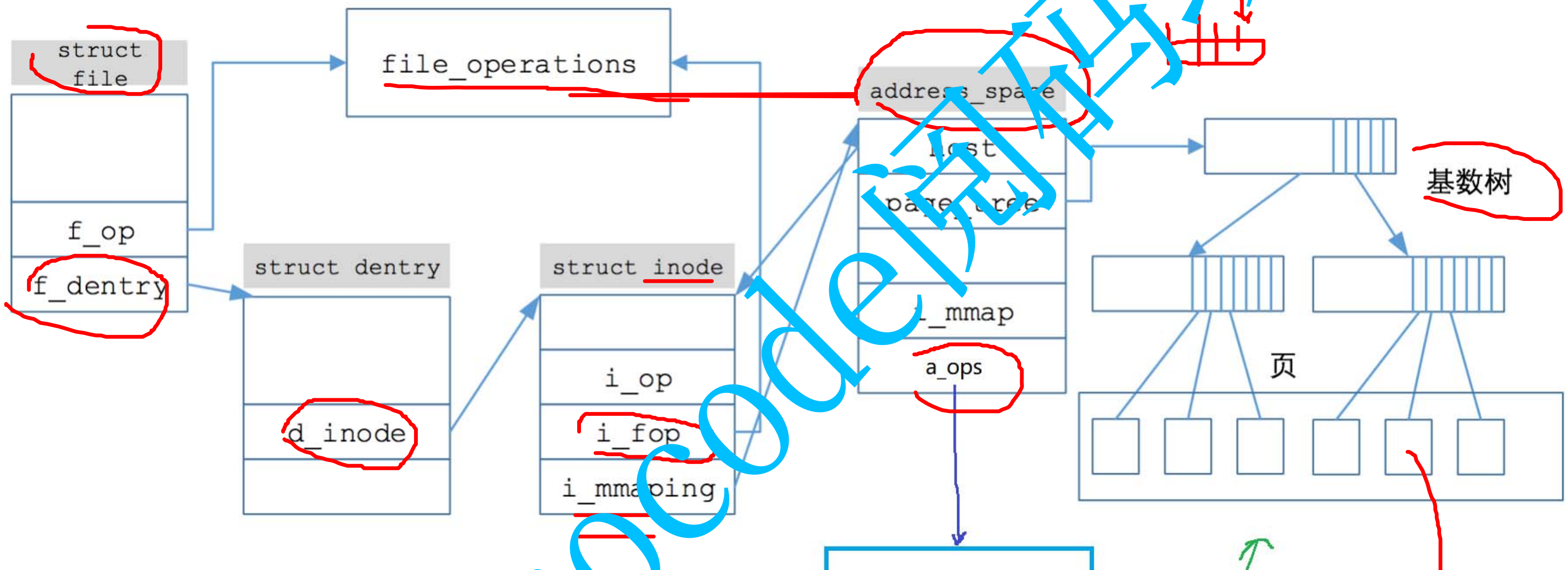
```
To free pagecache:
        echo 1 > /proc/sys/vm/drop_caches
To free reclaimable slab objects (includes dentries and inodes):
        echo 2 > /proc/sys/vm/drop_caches
To free slab objects and pagecache:
        echo 3 > /proc/sys/vm/drop_caches
```

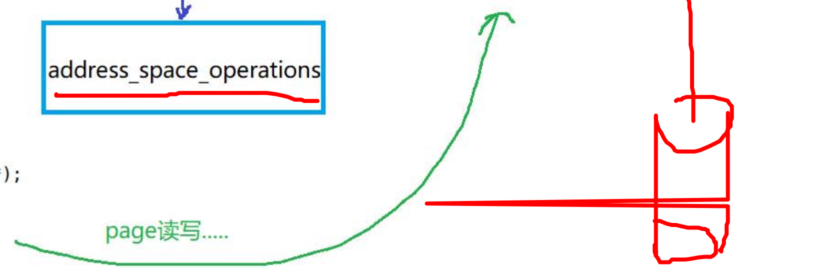
LRU Is Everywhere

```
 * shrink_dcache_sb - shrink dcache for a superblock
 * @sb: superblock
 *
 * Shrink the dcache for the specified super block. This is used to free
 * the dcache before unmounting a file system.
 */
void shrink_dcache_sb(struct super_block *sb)
{
        long freed;

        do {
                LIST_HEAD(dispose);

                freed = list_lru_walk(&sb->s_dentry_lru,
                        dentry_lru_isolate_shrink, &dispose, UINT_MAX);

                this_cpu_sub(nr_dentry_unused, freed);
                shrink_dentry_list(&dispose);
        } while (freed > 0);
}
EXPORT_SYMBOL(shrink_dcache_sb);
```

vomocode阅码场

File,inode, address_space






```
struct address_space_operations {
        int (*writepage)(struct page *page, struct writeback_control *wbc);
        int (*readpage)(struct file *, struct page *);

        /* Write back some dirty pages from this mapping. */
        int (*writepages)(struct address_space *, struct writeback_control *);

        /* Set a page dirty.  Return true if this dirtied it */
        int (*set_page_dirty)(struct page *page);

        int (*readpages)(struct file *filp, struct address_space *mapping,
                        struct list_head *pages, unsigned nr_pages);
```

vonocode阅码场

file_operations 与 address_space_operations关系

前者hook up到VFS，后者完成page cache访问(包括bio发起);

f_ops.read

```
if O_DIRECT
        a_ops. direct_IO()
else
        do_generic_file_read()
            如果page cache命中，读page cache;
            否则a_ops->readpage，发bio
```

vonocode阅码场

file_system_type
(register_filesystem)
	.mount -> fill_super
	{
	读硬盘的super_block;
	初始化super_clock: s_op
	Root inode初始化(i_ops和i_fop)
	根目录的dentry: sb->s_root
	}
	Umount:
	s_op.kill_sb()做清理	

super_block	文件系统的总体信息; super_operations.alloc_inode(), destroy_inode()	

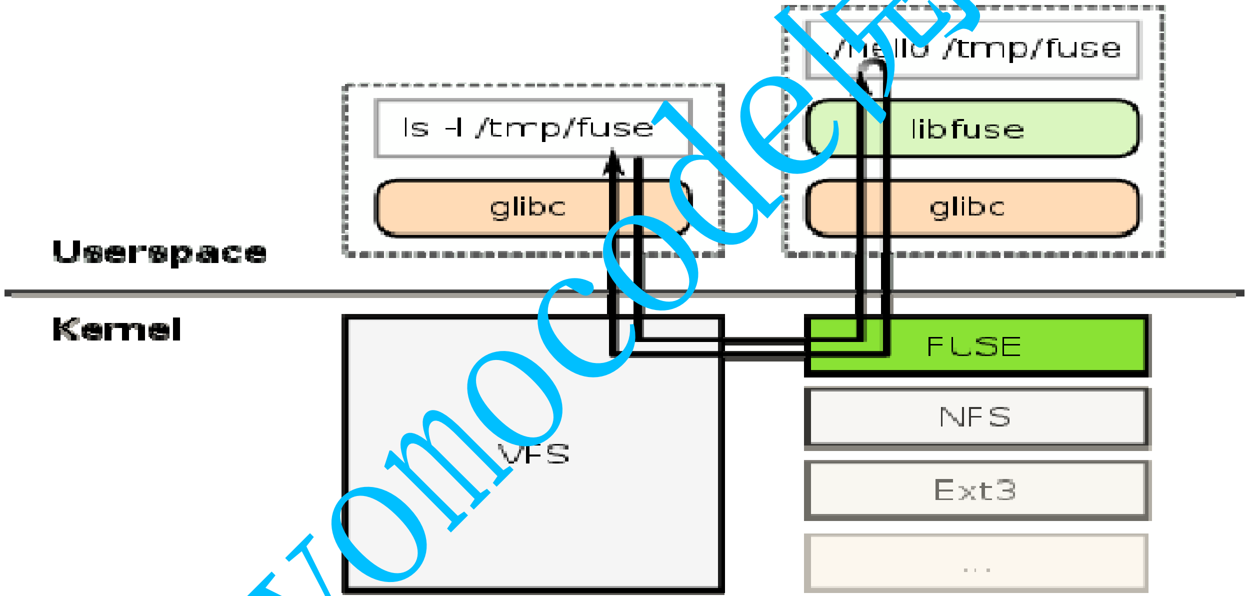
inode
(dir或者实体文件)
	i_ops:inode_operations
	.create
	.lookup
	{
	!strcmp(record->filename, child_dentry->d_name.name)
	分配inode,然后d_add()，绑定child dentry和inode:
	dentry->d_inode = inode;
	}
	.mkdir
		i_fops:
		file_operations

dentry	关于dentry父子关系的描述; dentry->d_inode指向相应的inode结构 dentry_operations	
file	进程级别的打开实例task_struct.file_struct.fd_array[] file.file_operations=inode.i_fops	

Vomocode阅码场

FUSE

FUSE需要把VFS层的请求传到用户态的fuse app，在用户态处理，然后再返回到内核态，把结果返回给VFS层；在用户态实现文件系统必然会引入额外的内核态/用户态切换带来的开销，对性能会产生一定影响



vomocode阅码场

fuse_loop():从/dev/fuse读取文件系统调用,调用fuse_operations结构中的处理函数，返回调用结果给/dev/fuse

谢 谢！

yomocode阅码场